Report

Università di Bologna · Campus di Cesena

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Alessandro Becci	0001125114	alessandro.becci@studio.unibo.it
Luca Tonelli	0001126379	luca.tonelli11@studio.unibo.it
Ludovico Nollino	0001148295	ludovico.nollino@studio.unibo.it

Table of Contents

Chapter 1. Domain Driven Development

Il sistema è stato progettato seguendo il paradigma DDD (Domain Driven Design), partendo dalla definizione dell' Ubiquitous Language e dall'analisi delle user story prodotte e dei casi d'uso, proseguendo con la costruzione del domain-model individuato. Con questi elementi è stato possibile definire i bounded context, con i relativi subdomains, e la context map.

1.1. Ubiquitous Language

L'uqibuitous language è stato definito in modo da rendere il sistema comprensibile e utilizzato per la definizione degli artefatti utili per definire: funzionalità, attributi di qualità, bounded context e context map.

Termine	Descrizione
Admin	Un individuo che può registrarsi e accedere con privilegi speciali, gestire biciclette, visualizzare tutti gli utenti registrati e monitorare lo stato e la posizione di tutte le bici sulla mappa.
User	Un individuo che può registrarsi, accedere, visualizzare le biciclette disponibili, avviare e interrompere le corse, ricaricare il credito del proprio account e visualizzare il saldo del credito.
Credit	Il saldo di un utente che può essere utilizzato per avviare una corsa.
E-Bike	Una bici elettrica che può avere stati diversi a seconda del livello della batteria e dell'utilizzo.
Ride	Una sessione in cui un utente utilizza una bicicletta per un periodo specifico.
Map	Una rappresentazione dell'area in cui si trovano tutte le biciclette. La mappa mostra sia bici disponibili e che quelle non disponibili per gli amministratori ma solo bici disponibili per gli utenti.

1.2. Requisiti del sistema

Di seguito l'elenco dei requisiti funzionali identificati che deve avere il sistema:

1. Il sistema deve permettere agli utenti di registrarsi e accedere.
2. Il sistema deve permettere agli utenti di visualizzare le ebike disponibili.
3. Il sistema deve permettere agli utenti di avviare, monitorare e fermare una ride.
4. Il sistema deve permettere agli utenti di monitorare la propria ride.
5. Il sistema deve permettere agli utenti di ricaricare il proprio credito.
6. Il sistema deve permettere agli utenti di visualizzare il saldo del proprio credito.
7. Il sistema deve permettere agli amministratori di visualizzare tutte le ebike sulla mappa.
8. Il sistema deve permettere agli amministratori di aggiungere nuove ebike.
9. Il sistema deve permettere agli amministratori di ricaricare le ebike.
10. Il sistema deve permettere agli amministratori di visualizzare tutti gli utenti registrati.
11. Il sistema deve permettere agli amministratori di monitorare tutte le eventuali ride in corso.

1.3. User Story

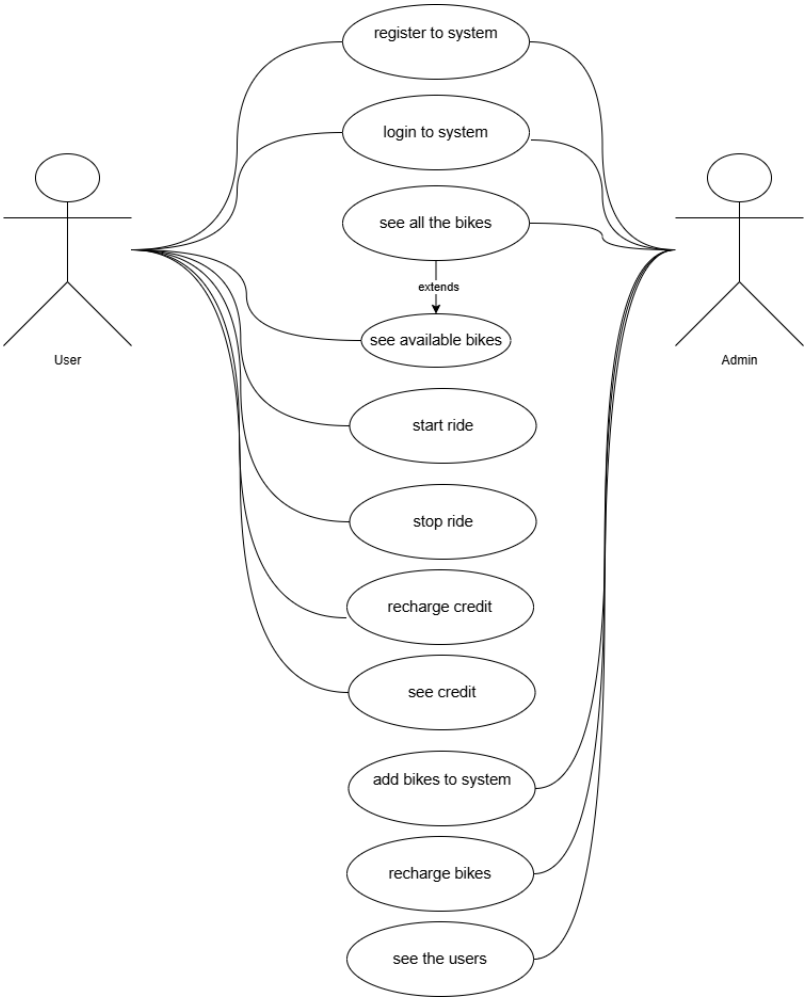
Le user story sono state definite in modo da rappresentare le funzionalità principali del sistema. Quelle individuate sono descritte di seguito:

User Story	Descrizione
US1: Sign Up	**Given** a user and an admin **When** a user or an admin sign up **Then** they are saved into the system and the user credit is set to 100
US2: Login	**Given** a user and an admin **When** a user or an admin sign in **Then** they can access to the system

User Story	Descrizione
US3: See All Bikes	**Given** an admin **When** is logged into the system **Then** he can view all the bikes on the map, also the riding ones
US4: Add E-Bike	**Given** an admin and the map and an e-bike **When** the admin adds a new e-bike **Then** the e-bike is saved and assigned to the map with 100% of charge level and is set to AVAILABLE state
US5: Recharge E-Bike	**Given** an admin and the map and an e-bike **When** the admin recharges the e-bike that is in MAINTENANCE state **Then** the e-bike's charging level is raised to 100% and is set to AVAILABLE state
US6: See All Users	**Given** an admin and the list of the users registered **When** the admin is logged into the system **Then** all the registered users are shown with their credit
US7: See All Available Bikes	**Given** a user **When** is logged into the system **Then** he can view all the bikes on the map that are in AVAILABLE state
US8: Start Ride	**Given** a user and the map and an e-bike **When** the user starts a new ride with the e-bike **Then** the e-bike is set to IN_USE state and the user's credit is decreased and the e-bike's battery level is decreased and the e-bike position is updated on the map following the ride and the map is updated following the e-bike position
US9: Stop Ride	**Given** a user and the map and an e-bike and an ongoing ride **When** the user stops the ride **Then** the e-bike change its state into AVAILABLE or MAINTENANCE depending on battery level and the map is updated following the bike position
US10: Observe Ride	**Given** a user and the map and a ride **When** the ride is ongoing **Then** the user can see the e-bike of the ride on the map
US11: Recharge Credit	**Given** a user **When** the user recharges his credit of an amount **Then** the user's credit is increased of the amount

1.4. Use Case

I casi d'uso sono stati definiti in modo da rappresentare le interazioni tra gli attori e il sistema. Quelli individuati sono raffigurati di seguito:



1.4.1. Descrizione dei casi d’uso

Register to system

Campo	Descrizione
Attori	User, Admin
Scope	E-Bike System
Livello	Alto
Precondizioni	L’utente non deve essere già registrato al sistema.
Postcondizioni	L’utente è registrato e può effettuare il login.
Trigger	L’utente desidera utilizzare il sistema e avvia la registrazione.
Flusso	1. L’utente inserisce i propri dati 2. Il sistema verifica i dati. 3. Il sistema crea un nuovo account per l’utente. 4. L’utente riceve una conferma della registrazione.

Login to system

Campo	Descrizione
Attori	User, Admin
Scope	E-Bike System
Livello	Alto
Precondizioni	L’utente deve essere registrato al sistema.
Postcondizioni	L’utente ha effettuato l’accesso e può utilizzare le funzionalità.
Trigger	L’utente desidera accedere al sistema.
Flusso	1. L’utente inserisce le proprie credenziali 2. Il sistema verifica le credenziali. 3. Il sistema consente l’accesso e mostra la dashboard.

See all bikes

Campo	Descrizione
Attori	Admin
Scope	E-Bike System
Livello	Medio
Precondizioni	L’attore deve essere autenticato.
Postcondizioni	L’attore visualizza tutte le e-bike.
Trigger	L’attore desidera vedere tutte le e-bike.
Flusso	1. L’attore esegue l’autenticazione. 2. Il sistema recupera le e-bike. 3. Il sistema mostra tutte le e-bike sulla dashboard.

See available bikes

Campo	Descrizione
Attori	User
Scope	E-Bike System
Livello	Medio
Precondizioni	L’attore deve essere autenticato.
Postcondizioni	L’attore visualizza solo le e-bike disponibili.
Trigger	L’attore desidera vedere le e-bike disponibili.
Flusso	1. L’attore effettua l’autenticazione. 2. Il sistema filtra le e-bike attualmente disponibili. 3. Il sistema mostra le e-bike disponibili sulla dashboard.

Start ride

Campo	Descrizione
Attori	User
Scope	E-Bike System
Livello	Alto
Precondizioni	L’utente deve avere credito sufficiente e selezionare una bicicletta disponibile.
Postcondizioni	La ride viene avviata e il sistema registra l’utilizzo.
Trigger	L’utente decide di iniziare una ride.
Flusso	1. L’utente seleziona una e-bike disponibile. 2. Il sistema verifica il credito dell’utente. 3. Il sistema sblocca l’e-bike. 4. La ride inizia e viene registrata nel sistema. 5. Il sistema aggiorna la posizione della bicicletta sulla mappa. 6. Il sistema aggiorna il livello di carica della bicicletta. 7. Il sistema scala il costo dal credito dell’utente.

Stop ride

Campo	Descrizione
Attori	User
Scope	E-Bike System
Livello	Alto
Precondizioni	Una ride deve essere in corso.
Postcondizioni	La ride viene terminata e il sistema calcola il costo.
Trigger	L’utente decide di terminare la ride.
Flusso	1. L’utente segnala il termine della ride. 2. Il sistema termina di scalare il credito dell’utente. 3. Il sistema mostra l’ultimo aggiornamento della posizione dell’e-bike sulla mappa. 4. Il sistema mostra l’ultimo aggiornamento della carica dell’e-bike: 4.1 Se la carica è 0%, l’e-bike' viene messa in manutenzione. 4.2 Altrimenti, l’e-bike' viene rimessa disponibile e mostrata sulla mappa.

Recharge credit

Campo	Descrizione
Attori	User
Scope	E-Bike System
Livello	Alto
Precondizioni	L’utente deve essere autenticato.
Postcondizioni	Il credito dell’utente viene incrementato.
Trigger	L’utente desidera ricaricare il proprio credito.
Flusso	1. L’utente sceglie l’opzione per ricaricare il credito. 2. L’utente specifica l’importo da ricaricare. 3. Il sistema aggiunge l’importo specificato dall’utente al credito corrente. 4. Il sistema aggiorna il saldo dell’utente.

See credit

Campo	Descrizione
Attori	User
Scope	E-Bike System
Livello	Basso
Precondizioni	L’utente deve essere autenticato.
Postcondizioni	L’utente visualizza il saldo del proprio account.
Trigger	L’utente desidera vedere il proprio credito.
Flusso	1. L’utente accede effettua l’autenticazione. 2. Il sistema recupera il saldo dal database. 3. Il sistema mostra il saldo all’utente sulla dashboard.

Add bikes to system

Campo	Descrizione
Attori	Admin
Scope	E-Bike System
Livello	Alto
Precondizioni	L’amministratore deve essere autenticato.
Postcondizioni	Nuove e-bike sono aggiunte al sistema.
Trigger	L’amministratore desidera aggiungere delle nuove e-bike.
Flusso	1. L’amministratore seleziona l’opzione per aggiungere le e-bike. 2. L’amministratore inserisce i dettagli delle e-bike. 3. Il sistema registra le e-bike nel database. 4. Le e-bike diventano disponibili nel sistema.

Recharge bikes

Campo	Descrizione
Attori	Admin
Scope	Bike Sharing System
Livello	Basso
Precondizioni	Le e-bike devono essere in stato di manutenzione.
Postcondizioni	Le e-bike sono ricaricate e rimesse in stato disponibile.
Trigger	L’amministratore rileva e-bike scariche in stato di manutenzione.
Flusso	1. L’amministratore individua e-bike scariche in stato di manutenzione. 2. L’amministratore seleziona l’opzione di rivarica dell’e e-bike. 3. L’amministratore specifica l’e-bike da ricaricare. 4. Il sistema aggiorna il livello di carica dell’e-bike e la rende disponibile.

See the users

Campo	Descrizione
Attori	Admin
Scope	E-Bike System

Campo	Descrizione
Livello	Basso
Precondizioni	L’amministratore deve essere autenticato.
Postcondizioni	L’amministratore visualizza l’elenco degli utenti.
Trigger	L’amministratore desidera vedere gli utenti registrati.
Flusso basico	1. L’amministratore effettua l’autenticazione. 2. Il sistema recupera l’elenco degli utenti e le loro relative informazioni dal database. 3. L’elenco degli utenti con le loro relative informazioni viene mostrato all’amministratore nella dashboard.

1.5. Quality Attributes Scenarios

I casi d’uso individuati sono stati analizzati per identificare i rispettivi attributi di qualità. Di seguito sono riportati i principali attributi di qualità individuati:

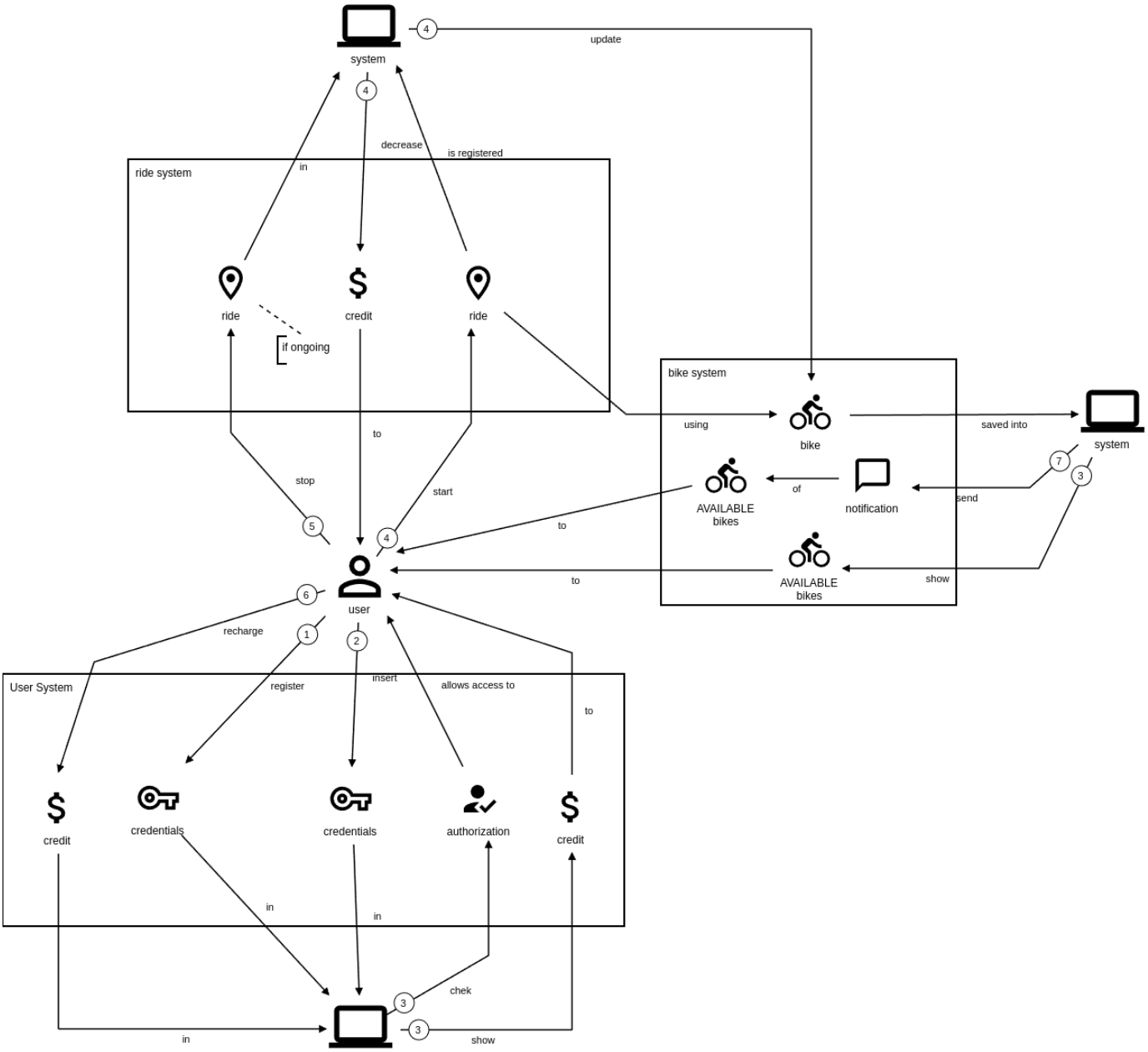
Quality Attribute	Source	Stimulus	Artifact	Environment	Response	Response Measure
Estendibilità	Developer	Aggiungere un nuovo servizio	Codebase di un microservizio	Situazione normale	Il nuovo servizio viene aggiunto senza impattare gli altri servizi	Nessuna modifica necessaria ai microservizi esistenti
Scalabilità	DevOps	Aumento del traffico	Microservizi esistenti	Ambiente di produzione	I microservizi scalano	La latenza rimane sotto il valore soglia
Scalabilità	Stakeholder	Aggiunta di un nuovo microservizio	Sistema di microservizi	Ambiente di progettazionee sviluppo	L’intero sistema scala, permettendo l’aggiunta del nuov microservizio adattato alle tecnologie esistenti	La codebase dei microservizi esistenti non viene modificata
Modificabilità	Stakeholder	Cambio nei requisiti di business	Microservizi esistenti	Situazione normale	Le modifiche si applicano solo ai servizi interessati	Nessuna interruzione del sistema complessivo
Testabilità	Tester	Esecuzione di test end-to-end	Microservizi distribuiti	Ambiente di test	I servizi sono testabili indipendentemente e insieme	Tutti i test passano con successo (e.g., test di integrazione)
Interoperabilità	System Integrator	Integrazione con un nuovo sistema esterno	API Gateway	Ambiente di produzione	Il sistema comunica correttamente con il nuovo sistema	Tempo di integrazione accettabile

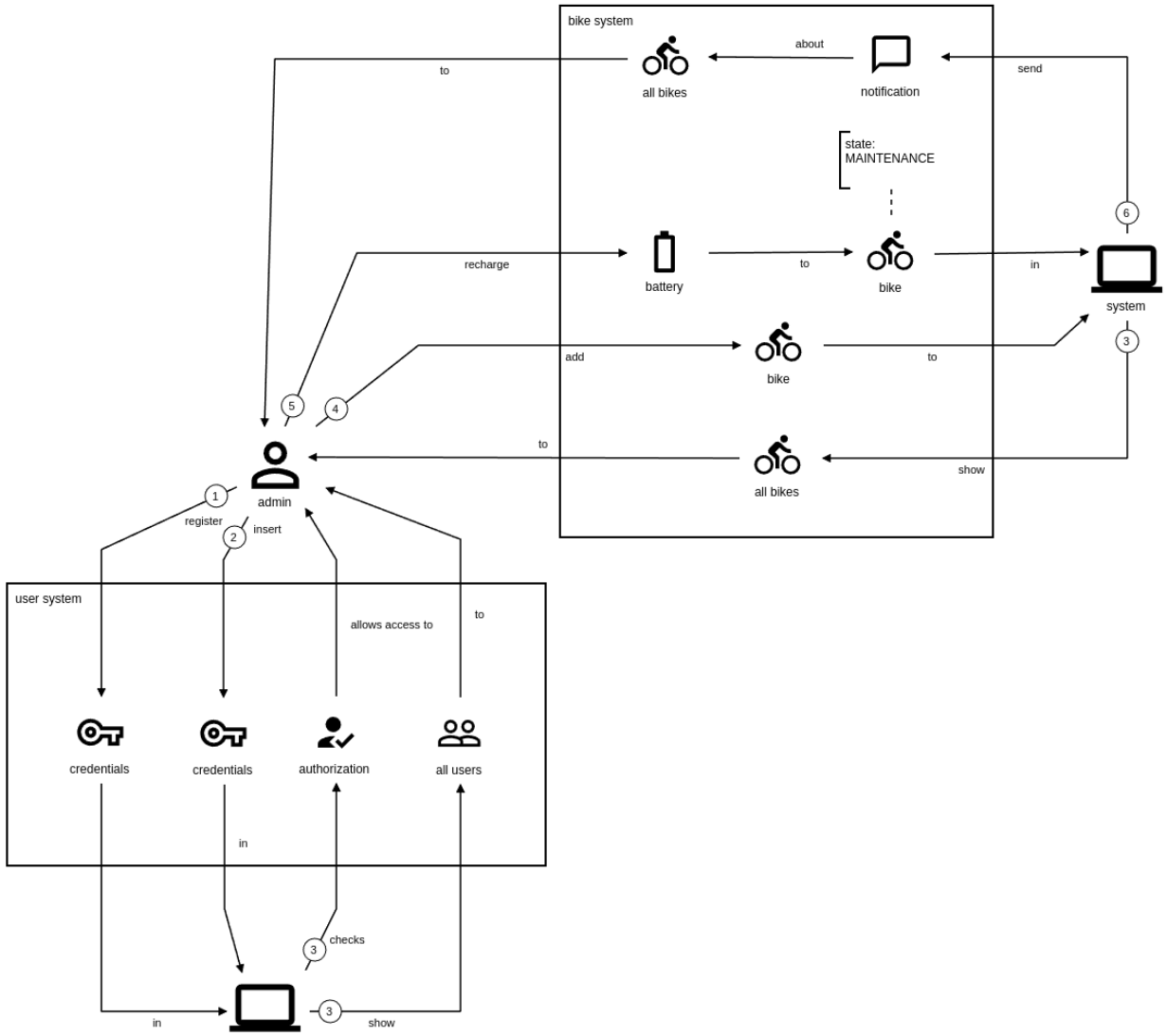
1.6. Domain Storytelling

In aiuto alla definizione del domain-model, sono stati realizzati dei Domain Storytelling sia per lo user sia per l’admin.

User storytelling

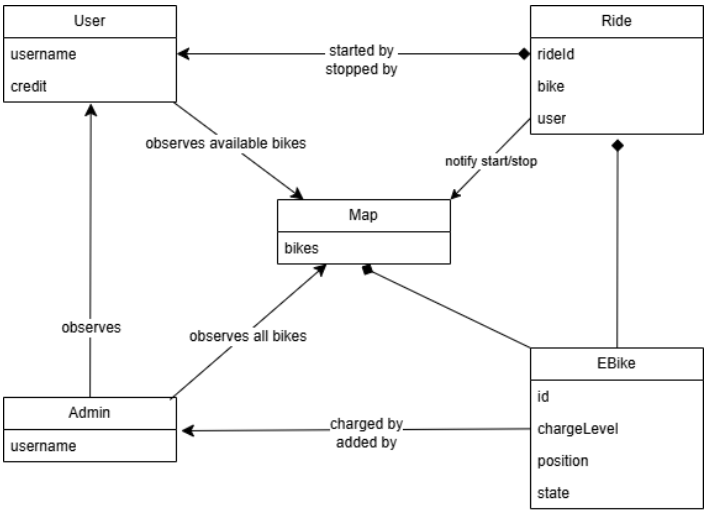
undefined





1.7. Domain Model

Grazie all'analisi delle user story, dei casi d'uso e dei rispettivi Domain Storytelling è stato possibile individuare facilmente il domain-model del sistema, definito in modo da rappresentare le entità principali del sistema e le relazioni tra di esse. Esso è rappresentato nel seguente diagramma:



Entità: User

Campo	Descrizione
Nome	User
Attributi	username, credit
Ruolo	Utente normale che può osservare le e-bike, ricaricare il proprio credito e effettuare le ride.

Entità: Admin

Campo	Descrizione
Nome	Admin
Attributi	username
Ruolo	Amministratore che gestisce le e-bike, aggiungendone nuove o ricaricando quelle esistenti, e che osserva gli utenti.

Entità: Ride

Campo	Descrizione
Nome	Ride
Attributi	rideId, bike, user
Ruolo	Rappresenta una corsa di un utente su una bicicletta.

Entità: Map

Campo	Descrizione
Nome	Map
Attributi	bikes
Ruolo	Contiene informazioni sulle e-bike disponibili per gli utenti e di tutte le e-bike per l'admin.

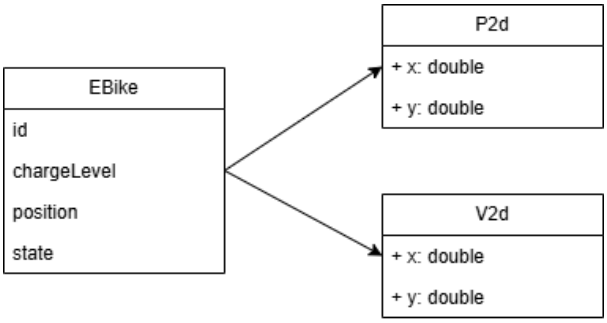
Entità: EBike

Campo	Descrizione
Nome	EBike
Attributi	id, chargeLevel, position, state
Ruolo	Rappresenta una e-bike con variabili di stato.

Interazioni tra le entità

- **Admin** può visualizzare tutti gli **User** registrati.
- **Admin** può aggiungere o modificare una **EBike**.
- **Admin** può visualizzare tutte le **EBike** sulla **Map**.
- **Admin** può ricaricare una **EBike**.
- **User** può avviare e fermare una **Ride**.
- **User** può osservare le biciclette disponibili nella **Map**.
- **User** può ricaricare il proprio credito.
- **User** può visualizzare il proprio credito.
- **User** può visualizzare la **Bike** della **Ride** che ha iniziato tramite la **Map**.
- **Ride** notifica lo stato di avvio e fermo delle ride alla **Map**.
- **Ride** notifica il decremento del credito dell'**User**.
- **Ride** notifica il decremento del livello di carica della **EBike**.

Value Objects



- **P2d**: rappresenta una posizione bidimensionale e contiene i valori x e y.
- **V2d**: rappresenta un vettore bidimensionale e contiene i valori x e y.

Aggregates

- **Ride**: contiene le informazioni relative alla ride e include **User** e **EBike**.
- **EBike**: contiene le informazioni relative alla e-bike e include i value objects **P2d** e **V2d**.

Domain Events

- **RideStarted**: notifica l'avvio di una ride e contiene l'aggregato **Ride**.
- **RideStopped**: notifica la fine di una ride e contiene l'aggregato **Ride**.
- **EBikeAdded**: notifica l'aggiunta di una e-bike e contiene l'aggregato **Map**.

Domain Services

- **RideSimulation**: simula una ride e contiene l'aggregato **Ride**.

Factory

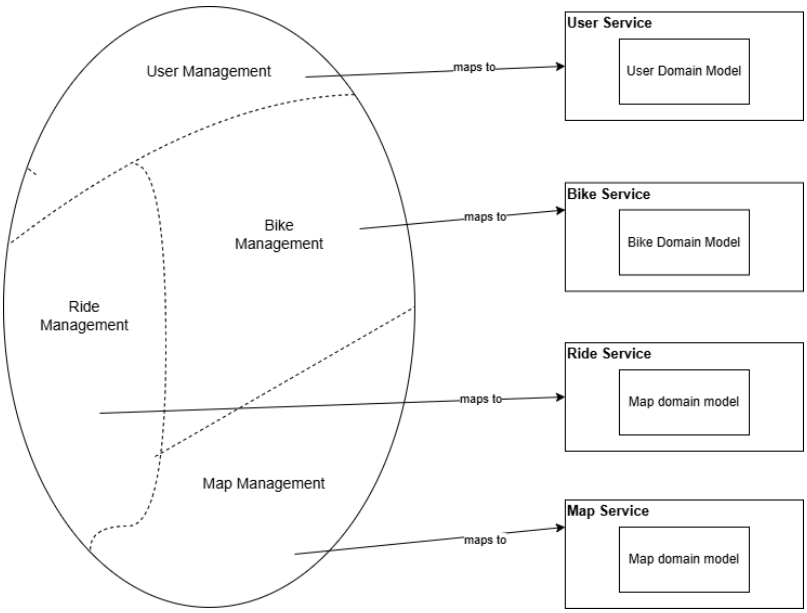
1. **EBikeFactory**: crea una nuova e-bike.

Repository

- **RideRepository**: gestisce la persistenza delle ride.
- **EBikeRepository**: gestisce la persistenza delle e-bike nella **Map**.

1.8. Bounded Context

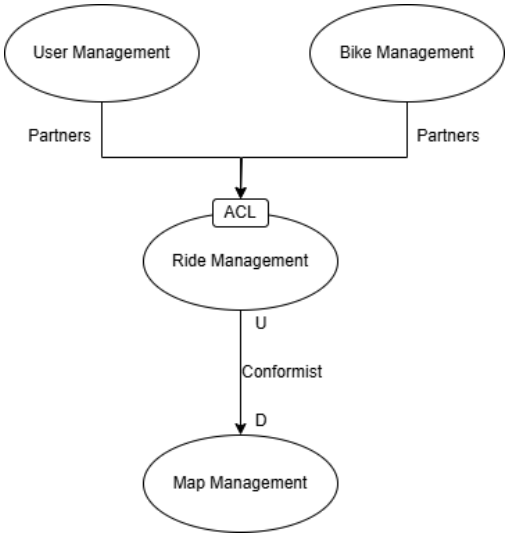
Dall'analisi del domain-model si è potuto definire i bounded context del sistema, individuando quattro contesti ben distinti:



- **User Management Context**: di tipo ***core***, contiene le entità e i servizi relativi agli utenti, sia admin che user.
- **Ride Management Context**: di tipo ***core***, contiene le entità e i servizi relativi alle ride.
- **EBike Management Context**: di tipo ***core***, contiene le entità e i servizi relativi alle e-bike.
- **Map Management Context**: di tipo ***core***, contiene le entità e i servizi relativi alla mappa.

1.9. Context Map

Dall'analisi dei bounded context si è potuto definire il context map del sistema, individuando le relazioni tra i contesti:



Di seguito una descrizione delle relazioni tra i contesti:

- **User Management Ride Management**: Questi due contesti sono in partnership in quanto collaborano in modo paritario, scambiandosi dati senza una relazione gerarchica o dipendenze.
- **Bike Management Ride Management**: Stesso discorso del precedente.
- **Ride Management → Map Management**: `Ride Management` si adatta completamente alle regole e all'interfaccia imposta dal `Map Management`, quindi il pattern utilizzato è il conformist. Non tenta di modificarne il comportamento, accettando le sue dipendenze così come sono.
  - `Map Management` è **upstream**, in quanto detta le regole e fornisce i dati richiesti.
  - `Ride Management` è **downstream**, in quanto dipende dai dati e dalle regole di `Map Management`.
- **Anti-Corruption Layer (ACL) su Ride Management**: Serve come livello di astrazione tra `Ride Management` e i suoi partner, proteggendo il servizio principale da eventuali cambiamenti o complessità delle interfacce dei partner.

Chapter 2. Dal Domain Driven Development ai Microservizi

Seguendo i bounded context individuati e, in generale, l'analisi del dominio, è stato possibile identificare le system operations, effettuare la suddivisioni in subdomains per identificare i servizi ed infine identificare le API di ogni servizio.

2.1. System operations

In questa fase sono stati prima identificati i comandi:

Actor	Story	Command	Description
Admin/User	Register	signUp()	Creazione profilo
Admin/User	Login	signIn()	Accesso al sistema
Admin	Add a New Bike	addBike()	Aggiunta di una nuova ebike nel sistema
Admin	Recharge Bike	rechargeBike()	Ricarica la batteria di una ebike
User	Start a Ride	startRide()	Lo User inizia una ride
User	Stop a Ride	stopRide()	Lo User termina una ride
User	Recharge credit	rechargeCredit()	Lo User ricarica il proprio credito di una certa quantità

In seguito sono stati identificati le query:

Query	Descrizione
getUser(userName)	Recupera i dettagli di un utente specifico in base al suo nome utente. Questa funzione è utile per verificare se un utente è registrato e accedere alle informazioni del suo profilo.
observeAllEBikes()	Restituisce un elenco di tutte le EBike nel sistema, comprese quelle che potrebbero non essere disponibili per le corse. Questa funzione sarebbe tipicamente utilizzata da un Admin per visualizzare e gestire tutte le biciclette, indipendentemente dal loro stato attuale.
getEBike(bikeId)	Recupera informazioni dettagliate su una specifica EBike tramite il suo bikeId unico. Questo potrebbe includere attributi come livello di carica, posizione e stato. Gli Admin potrebbero utilizzare questa funzione per ispezionare o gestire biciclette singole.
getEBikeState(bikeId)	Restituisce lo stato attuale di una specifica EBike identificata dal bikeId. Lo stato potrebbe indicare se la bicicletta è disponibile, in uso o in manutenzione. Questo è utile sia per gli Admin (per gestire la disponibilità delle biciclette) che per gli utenti (per vedere se una bicicletta è pronta per l'uso).
observeAllUsers()	Recupera un elenco di tutti gli utenti registrati nel sistema. Questa funzione è destinata agli Admin che devono visualizzare e gestire gli account utente o analizzare i modelli di utilizzo.
observeUserEBikes()	Restituisce un osservabile delle EBike attualmente disponibili per le corse.
getUserCredit(userId)	Recupera il saldo attuale del credito di un utente specifico tramite il suo userId. Questa funzione è utile per gli utenti per controllare il loro saldo disponibile prima di iniziare una corsa e per gli Admin per assistere gli utenti con domande relative all'account.

2.2. Subdomains

I subdomains identificati sono i seguenti:

User Management

Capacità	Descrizione
Registrazione e Autenticazione	Gestisce la registrazione e il login, assicurando che utenti e admin abbiano credenziali valide per accedere al sistema.
Gestione del Credito (User)	Gestisce il credito degli utenti, compresa l'allocazione iniziale e la ricarica.
Visualizzazione Profilo e Account	Consente agli utenti di visualizzare il saldo del credito e le informazioni del profilo.
Supervisione Utenti (Admin)	Consente agli admin di visualizzare tutti gli utenti registrati e i dettagli sul loro credito.

Bike Management

Capacità	Descrizione
Gestione Inventario E-Bike	Gestisce l'inventario delle e-bike, consentendo agli admin di registrare nuove e-bike.
Stato e Manutenzione Biciclette	Consente agli admin di verificare e aggiornare lo stato delle e-bike, comprese le operazioni ricarica.

Capacità	Descrizione
Tracciamento E-Bike	Abilita gli admin a visualizzare tutte le e-bike sulla mappa, indipendentemente dalla loro disponibilità, fornendo una visione completa della distribuzione e dell'uso.
Disponibilità e-bike (Utente)	Mostra solo le e-bike disponibili per gli utenti.

Ride Management

Capacità	Descrizione
Inizio ride	Gestisce il processo di avvio di una ride, aggiornando lo stato della bicicletta a IN_USE, modificando il credito dell'utente e iniziando a tracciare il movimento della e-bike sulla mappa.
Tracciamento della ride e Aggiornamenti della Map	Assicura che la posizione della e-bike venga aggiornata sulla mappa mentre l'utente sta effettuando una ride, riflettendo i cambiamenti di posizione in tempo reale.
Fine ride	Gestisce la fine di una ride, impostando lo stato dell'e-bike su AVAIABLE o MAINTENANCE in base al livello della batteria e salvando la posizione finale sulla mappa.
Monitoraggio della Ride	Consente agli utenti di vedere la posizione in tempo reale della propria e-bike sulla mappa durante la ride.

Map Management

Capacità	Descrizione
Rendering della Map e Disponibilità e-bike	Visualizza una mappa con solo le e-bike attualmente disponibili per l'accesso degli utenti.
Panoramica della Map (Admin)	Fornisce agli admin una vista completa di tutte le e-bike.

2.3. Identificazione dei servizi dalle operazioni di sistema

Servizio	Operazione
User Service	signUp(), signIn(), rechargeCredit(), decreaseCredit(), getUser(), observeAllUsers()
EBike Service	addBike(), rechargeBike(), getEBike(), getEBikeState(), updateEBike()
Ride Service	startRide(), stopRide()
Map Service	updateBikes, observeAllBikes(), observeAvaiableBikes(), notifyStartRide(), notifyStopRide()

Servizio	Operazione	Collaboratori
User Service	signUp(), signIn(), rechargeCredit(), decreaseCredit(), getUser(), observeAllUsers(), observeUser()	□–□
EBike Service	addBike(), rechargeBike(), getEBike(), getEBikeState(), updateEBike()	- Map Service: updateBikes - Map Service: updateBikes □–□ □–□ -Map Service: updateBikes
Ride Service	startRide(), stopRide()	- EBike Service: getEBike(), updateEBike() - User Service: observeUser(), getUser() - Map Service: notifyStartRide - Map Service: notifyStopRide
Map Service	updateBikes, observeAllBikes(), observeAvaiableBikes(), notifyStartRide(), notifyStopRide()	□–□

2.4. Identificazione delle API dei servizi

Seguendo quanto detto sopra, sono state identificate le API di tipo REST per i servizi. Di seguito sono riportati gli endpoint di ogni servizio:

2.4.1. USER SERVICE

- `/signIn` (username) ⇒ POST
  - Ritorna `statusCode OK` se login avvenuto con successo
- `/signUp` (username) ⇒ POST
  - Ritorna `status OK` se registrazione avvenuta con successo
- `/rechargeCredit` (username, creditToAdd) ⇒ PATCH
  - Ritorna `status OK` e JSON se cambio avvenuto con successo
- `/decreaseCredit` (username, creditToDecrease) ⇒ PATCH
  - Ritorna `status OK` e JSON se cambio avvenuto con successo
- `/getUser` (username) ⇒ GET
  - Ritorna `status` e JSON con informazioni utente
- `/observeAllUsers` ⇒ WebSocket
  - Emette JSON degli utenti mano a mano che cambiano (ad ogni refresh)
- `/observeUser` (username) ⇒ WebSocket
  - Emette JSON di tutti i cambiamenti dell'utente

2.4.2. BIKE SERVICE

- **Endpoint:**
- `/addBike` (bikeId, position) ⇒ POST
  - Ritorna `status OK` se avvenuto con successo
- `/rechargeBike` (bikeId) ⇒ PATCH
  - Ritorna `status OK` e un JSON Object di ebike con il batteryLevel aggiornato
- `/getEBike` (bikeId) ⇒ GET
  - Ritorna `status OK` e un JSON Object di ebike
- `/getEBikeState` (bikeId) ⇒ GET
  - Ritorna `status OK` e JSON Object di ebike con lo stato
- `/updateEBike` (bikeid, JSON bike) ⇒ PUT
  - Ritorna `status OK` e JSON di ebike

2.4.3. MAP SERVICE

- **Endpoint:**
- `/updateEBike` (JSON bike) ⇒ PUT
  - Ritorna `status OK` se avvenuto con successo
- `/notifyStartRide` (username, bikeName) ⇒ POST
  - Emette `status OK`
- `/notifyStopRide` (username, bikeName) ⇒ POST
  - Emette `status OK`
- `/observeAllBikes` ⇒ WebSocket

    - Emette JSON Object di tutte le ebike ad ogni cambiamento di ogni singola ebike
- `/observeUserBikes` (username) ⇒ WebSocket
    - Ritorna JSON Object di tutte le ebike disponibili o quelle legate all'utente

2.4.4. RIDE SERVICE

- **Endpoint:**
- `/startRide` (user, bike) ⇒ POST
    - Ritorna `status OK` se avvenuto con successo
- `/stopRide` (username) ⇒ POST
    - Ritorna `status OK` se avvenuto con successo

Chapter 3. Descrizione dei microservizi

Di seguito verrà riportata una descrizione di tutti i microservizi e le configurazioni utilizzate nel sistema. Vedremo nel dettaglio le singole parti, mentre nel prossimo capitolo parleremo di come esse sono integrate una con l'altra.

3.1. Configurazione Docker per i Microservizi

Ogni microservizio del sistema è containerizzato utilizzando Docker per garantire portabilità, isolamento e facilità di distribuzione. La configurazione Docker per ogni microservizio segue una struttura comune, progettata per funzionare nel Docker Compose globale.

Il Dockerfile utilizza una configurazione multistage per ottimizzare il processo di build e runtime di un'applicazione Java basata su Gradle.

3.1.1. Stage 1: Build

```
FROM gradle:8.11-jdk21 AS build
COPY app/build.gradle.kts settings.gradle.kts gradle.properties ./
COPY app/src src
RUN --mount=type=cache,target=/home/gradle/.gradle/caches gradle jar --no-daemon --parallel --build-cache
```

- **Immagine Gradle con JDK 21** per la compilazione del progetto con tutte le dipendenze.
- **Copia dei file Gradle** di configurazione e del codice sorgente.
- **Build del file JAR** utilizzando la cache per velocizzare le compilazioni successive.

Allo scopo di generare il jar con tutte le dipendenze (fatjar) è stato definito un task specifico nel `build.gradle.kts`:

```
tasks.jar {
    archiveFileName.set("app.jar")
    manifest {
        attributes["Main-Class"] = application.mainClass.get() // or specify your main class directly
    }

    // Include all runtime dependencies into the JAR file
    from(configurations.runtimeClasspath.get().filter { it.name.endsWith("jar") }.map { zipTree(it) })

    // Optionally, include your compiled classes (if not already included by default)
    from(sourceSets.main.get().output)

    // Ensure the JAR is built as a single fat JAR
    duplicatesStrategy = DuplicatesStrategy.EXCLUDE
}
```

3.1.2. Stage 2: Runtime

```
FROM openjdk:21-jdk-slim
RUN apt-get update && apt-get install -y curl
WORKDIR /app
COPY --from=build /home/gradle/build/libs/app.jar app.jar
EXPOSE 8087
ENTRYPOINT ["java", "-jar", "app.jar"]
```

- **Immagine Java runtime minima** per ridurre la dimensione del container finale.
- **Installazione di `curl`** per eseguire health check dal `docker-compose`.
- **Copia del file JAR** dalla fase di build al filesystem del container.
- **Esposizione della porta 8087** e avvio dell'app con `java -jar app.jar`.

3.2. Prometheus

Il microservizio è responsabile del monitoraggio e della raccolta delle metriche di sistema per i vari microservizi all'interno dell'architettura.

Utilizza Prometheus per raccogliere e memorizzare le metriche, che possono poi essere utilizzate per analisi e monitoraggio in tempo reale.

Per capire a quali microservizi raccogliere le metriche, esso usa il file di configurazione `prometheus.yml`.

```
global:
  scrape_interval: 15s  # Scrape targets every 15 seconds

scrape_configs:
  - job_name: 'api-gateway'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['api-gateway:8081']

  - job_name: 'service-discovery'
    static_configs:
      - targets: ['eureka-server:8761']
    metrics_path: '/actuator/prometheus'

  # Map microservice
  - job_name: 'map-microservice'
    metrics_path: '/metrics'
    static_configs:
      - targets: ['map-microservice:8087']

  ...

  - job_name: 'user-rideadapter'
    metrics_path: '/metrics'
    static_configs:
      - targets: ['user-microservice:8083']
```

Per gestire le metriche all'interno del microservizio viene utilizzata la classe `MetricsManager`.

Essa utilizza la libreria Micrometer per registrare e gestire le metriche, e specificamente il `PrometheusMeterRegistry` per l'integrazione con Prometheus.

Ecco una descrizione delle principali funzionalità della classe MetricsManager:

- Singleton Pattern: La classe utilizza il pattern Singleton per garantire che ci sia una sola istanza di MetricsManager in esecuzione.
- Registrazione delle Metriche: Utilizza PrometheusMeterRegistry per registrare le metriche.
- Temporizzazione dei Metodi: Fornisce metodi per avviare e registrare il tempo di esecuzione dei metodi.
- Contatori di Metodi: Incrementa i contatori per il numero di chiamate ai metodi.
- Registrazione degli Errori: Registra gli errori che si verificano durante l'esecuzione dei metodi.

3.3. API Gateway

L'API Gateway funge da punto di ingresso unico per le richieste client verso i microservizi di un sistema distribuito. Questo Gateway è configurato per scoprire automaticamente i servizi registrati in **Eureka**, instradare dinamicamente le richieste e monitorare lo stato del sistema tramite metriche e health check.

3.3.1. Funzionamento

Il gateway utilizza Spring Boot e Spring Cloud Gateway per la gestione delle rotte, l'integrazione con **Eureka** per il service discovery e l'esposizione di metriche tramite **Prometheus**.

3.3.2. Configurazione `application.yml`

La configurazione del file `application.yml` definisce:

- La porta su cui il Gateway ascolta.
- Il nome del Gateway per la registrazione in Eureka.
- Il discovery dinamico delle rotte basato sui servizi registrati.
- La configurazione per l'export delle metriche e gli endpoint di health.

```yaml
server:
  port: 8081  # Il Gateway sarà in ascolto sulla porta 8081

spring:
  application:
    name: api-gateway  # Nome del Gateway per la registrazione in Eureka

  cloud:
    gateway:
      discovery:
        locator:
          enabled: true  # Abilita il routing dinamico basato sui servizi registrati in Eureka

eureka:
  client:
    serviceUrl:
      defaultZone: http://eureka-server:8761/eureka/  # URL di Eureka per risoluzione tramite rete Docker
    registerWithEureka: true  # Il Gateway si registra su Eureka
    fetchRegistry: true       # Recupera il registro dei servizi da Eureka

management:
  endpoints:
    web:
      exposure:
        include: health, prometheus  # Espone gli endpoint di health e metriche Prometheus

  health:
    livenessstate:
      enabled: true
    readinessstate:
      enabled: true
  prometheus:
    metrics:
      export:
        enabled: true  # Esporta le metriche per Prometheus

logging:
  level:
    org.springframework.cloud.gateway: DEBUG
```

Questa configurazione dell'API Gateway con Spring Boot e Spring Cloud offre una soluzione scalabile e monitorabile per la gestione delle richieste in un'architettura a microservizi, semplificando la comunicazione tra i componenti e migliorando la resilienza del sistema.

3.4. Service Discovery

Il Service Discovery è un meccanismo che consente ai microservizi di localizzarsi e comunicare tra loro in modo dinamico in un sistema distribuito.

In un'architettura basata su microservizi, i servizi sono registrati in un registro centrale, come Eureka, che consente ad altri microservizi di trovare e interagire con essi senza dover conoscere a priori il loro indirizzo o posizione.

3.4.1. Funzionamento

Il Service Discovery utilizza Spring Cloud Eureka per consentire la registrazione e la scoperta dei servizi.

I microservizi possono registrarsi in Eureka al momento del loro avvio, mentre altri servizi possono utilizzare Eureka per scoprire e comunicare con i servizi disponibili. Questo approccio dinamico riduce la necessità di configurazioni statiche e rende l'architettura più flessibile e scalabile.

3.4.2. Configurazione `application.yml`

La configurazione del file application.yml definisce:

- La porta su cui il server Eureka è in ascolto.
- La disabilitazione della registrazione e della raccolta del registro da parte del servizio di discovery stesso.
- L'esposizione di endpoint per il monitoraggio del sistema, come health check e metriche Prometheus.

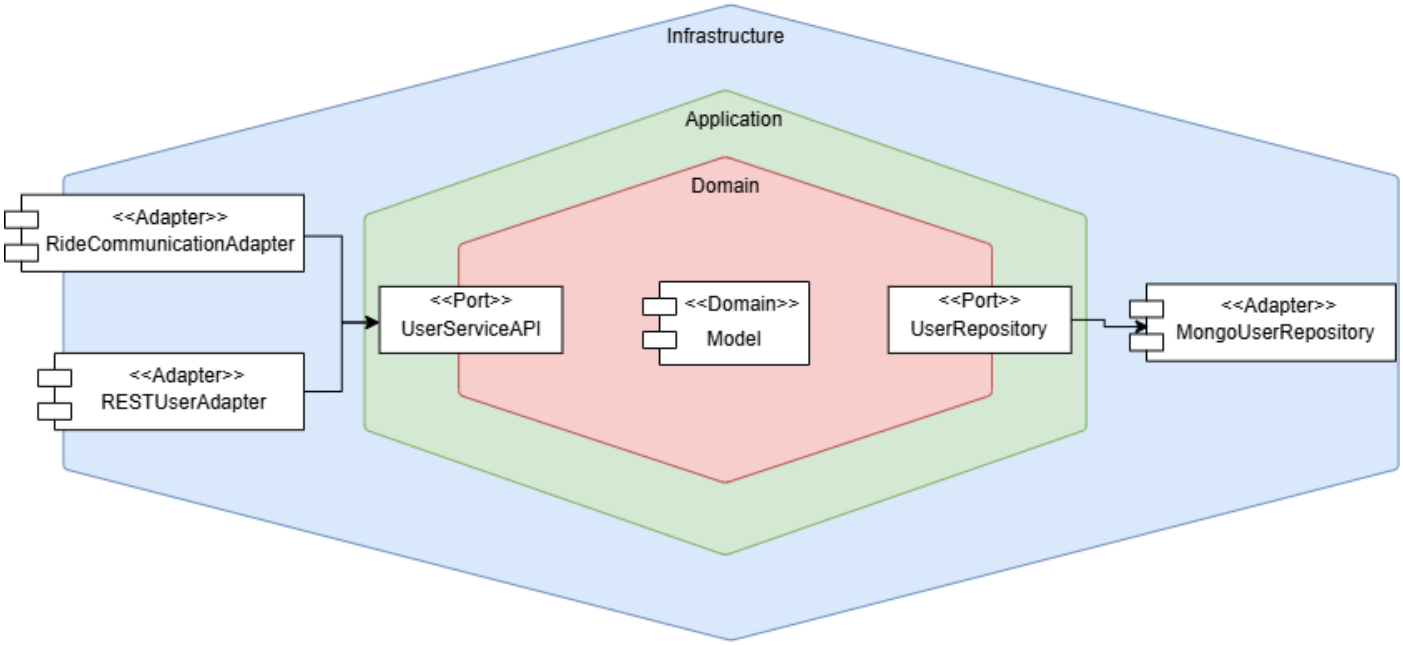
```
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false

management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics,prometheus
  endpoint:
    health:
      show-details: always

server:
  port: 8761
```

3.5. User Microservice

Il microservizio, organizzato in architettura esagonale, definisce tre porte per interfacciarsi al layer di applicazione. Queste porte sono utilizzate nel layer infrastrutturale, che fornisce le implementazioni concrete per interagire con la repository, pubblicare eventi, e gestire le richieste REST. Di seguito un immagine che ne descrive la struttura (Si specifica che il sistema di event publisher è stato non rappresentato per dare maggior enfasi all'interazione tra porte e adattatori riferiti al funzionamento del microservizio degli utenti).



3.5.1. Domain Layer

L'unico oggetto del dominio è l'entità `User`, che rappresenta un utente all'interno del sistema. L'utente ha un `username`, un `credit` e un `type` (user o admin). L'entità include metodi per accedere e modificare i dati dell'utente, in particolare il credito.

3.5.2. Application Layer

UserRepository

La porta è responsabile della gestione dei dati degli utenti all'interno del sistema. Fornisce metodi per salvare, aggiornare e trovare un utente tramite il suo username. Inoltre, permette di recuperare tutti gli utenti presenti all'interno del sistema.

Questa porta assicura che il sistema possa gestire in modo efficiente i dati degli utenti.

UserServiceAPI

La porta definisce il contratto per l'API REST che interagisce con il sistema degli utenti.

Include metodi per registrare un nuovo utente, autenticare un utente e recuperare i dati di un utente tramite il suo username. Inoltre, consente di aggiornare i dati di un utente, la ricarica del credito, il decremento del credito durante, ad esempio, una corsa, e fornisce un metodo per recuperare tutti gli utenti presenti nel sistema.

Questa interfaccia facilita la comunicazione tra il sistema degli utenti e i client esterni, consentendo un'interazione fluida e uno scambio di dati.

UserEventPublisher

La porta è progettata per gestire la pubblicazione di eventi relativi agli utenti e alle attività degli utenti.

Include metodi per pubblicare aggiornamenti su un singolo utente e su tutti gli utenti. Questi eventi sono fondamentali per mantenere il sistema e gli utenti informati sugli aggiornamenti in tempo reale.

UserServiceImpl

La classe implementa la logica di business del microservizio, utilizzando le porte definite nel layer di applicazione. In particolare, gestisce le richieste provenienti dall'API REST e pubblica gli eventi relativi agli utenti.

Fornisce i metodi per registrare un nuovo utente, autenticare un utente, aggiornare i dati di un utente, recuperare i dati di un utente tramite il suo username, ricaricare il credito di un utente, decrementare il credito di un utente e recuperare tutti gli utenti presenti nel sistema. Utilizza il repository per accedere ai dati degli utenti e l'event publisher per notificare gli aggiornamenti.

3.5.3. Infrastructure Layer

Le tre porte sono quindi implementate nel layer infrastrutturale, che fornisce le implementazioni concrete per interagire con la repository, pubblicare eventi, registrare il microservizio al Service Discovery e gestire le richieste REST.

RideCommunicationAdapter

Gestisce la comunicazione con il microservizio delle corse. Configura un server HTTP con diverse routes per gestire le richieste in arrivo. In particolare, espone un endpoint per recuperare i dettagli di un utente e un endpoint per dli aggiornamenti di un utente.

RESTUserAdapter

Gestisce le richieste REST per il microservizio degli utenti. Configura le rotte HTTP e definisce i metodi per gestire le operazioni sugli utenti. In particolare, espone un endpoint per registrare un nuovo utente, un endpoint per autenticare un utente, un endpoint per ricaricare il credito di un utente, un endpoint per osservare gli utenti e un endpoint per osservare un singolo utenti. Gli ultimi due endpoint sono gestiti utilizzando una connessione WebSocket per l'invio degli aggiornamenti in tempo reale.

UserVerticle

Rappresenta il punto di ingresso del microservizio degli utenti. Configura le rotte HTTP usando il `RESTUserAdapter` e gestisce la registrazione del microservizio al Service Discovery, l'inizio e l'invio degli `heartBeat` per mantenere l'istanza registrata. Include, inoltre, i metodi per la configurazione del server HTTP e per il deploy del verticle.

MongoUserRepository

Implementa la porta `UserRepository`e gestisce i dati degli utenti in un database MongoDB. Utilizza il MongoClient di Vert.x per eseguire operazioni sulla collezione users.

MetricsManager

Rappresenta un singleton che gestisce le metriche dell'applicazione utilizzando Micrometer e Prometheus. Fornisce metodi per avviare e

registrare timer, incrementare contatori e registrare errori per vari metodi nel servizio utenti.

UserEventPublisherImpl

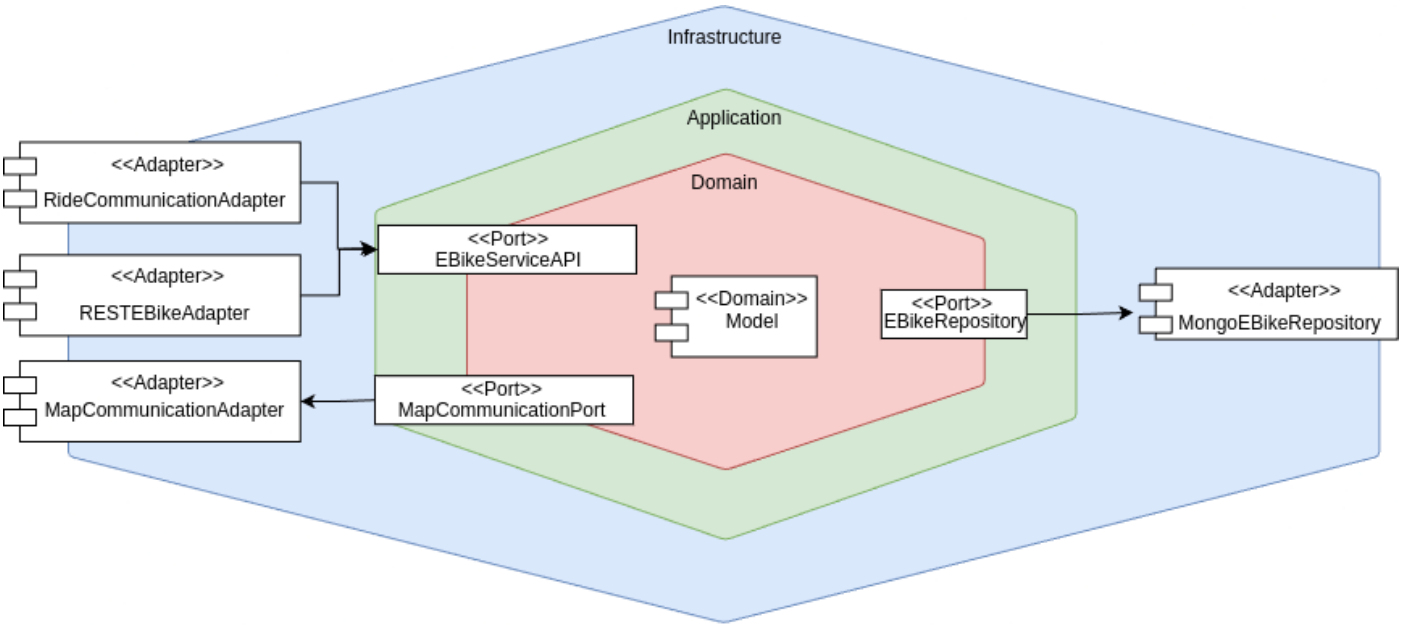
Implementa l'interfaccia UserEventPublisher e gestisce la pubblicazione degli eventi relativi agli utenti utilizzando il bus degli eventi di Vert.x.

ServiceConfiguration

Rappresenta un singleton che gestisce la configurazione del microservizio dello User. Fornisce metodi per caricare la configurazione del microservizio dalle variabili d'ambiente e per accedere ai valori di configurazione dei vari adapter e verticle.

3.6. EBike Microservice

Il microservizio, organizzato in architettura esagonale, definisce tre porte per interfacciarsi al layer di applicazione. Queste porte sono utilizzate nel layer infrastrutturale, che fornisce le implementazioni concrete per interagire con la repository, pubblicare eventi, registrare il microservizio al Service Discovery e gestire le richieste REST. Di seguito un immagine che ne descrive la struttura.



3.6.1. Domain Layer

- **EBike**: Rappresenta una bicicletta elettrica nel sistema. Contiene attributi come `id`, `state`, `location` e `batteryLevel`. Include metodi per accedere e modificare questi attributi, nonché per gestire la ricarica e il decremento della batteria.
- **EBikeState**: È un *enum* che definisce i possibili stati di una bicicletta elettrica: AVAILABLE (disponibile), IN_USE (in uso) e MAINTENANCE (in manutenzione).
- **P2d**: Rappresenta un punto bidimensionale nello spazio. Contiene coordinate x e y e include metodi per operazioni vettoriali come somma, sottrazione e normalizzazione.

3.6.2. Application Layer

EBikeRepository

La porta è responsabile della gestione dei dati delle ebike all'interno del sistema. Fornisce metodi per salvare, aggiornare, trovare una ebike tramite il suo id e trovare tutte le ebike presenti nel sistema.

Questa porta assicura che il sistema possa gestire in modo efficiente i dati delle ebike.

EBikeServiceAPI

La porta definisce il contratto per l'API REST che interagisce con il sistema delle ebike. Include metodi per creare una nuova ebike, recuperare una ebike per ID, ricaricare la batteria di una ebike, aggiornare i dettagli di una ebike e recuperare tutte le ebike.

Questa interfaccia facilita la comunicazione tra il sistema delle ebike e i client esterni, consentendo un'interazione fluida e uno scambio di dati.

MapCommunicationPort

La porta è progettata per comunicare con il Map Microservice.

Include metodi per inviare aggiornamenti su una singola ebike e inviare aggiornamenti su tutte le ebike.

Questi metodi sono fondamentali per mantenere il sistema e le ebike informate sugli aggiornamenti in tempo reale.

EBikeServiceImpl

La classe implementa la logica di business del microservizio, utilizzando le porte definite nel layer di applicazione. Utilizza l'EBikeRepository per accedere ai dati delle ebike ed eseguire operazioni su di esse, e il `MapCommunicationAdapter` per comunicare con il Map Microservice.

Fornisce i metodi per creare una nuova ebike, recuperare una ebike per ID, ricaricare la batteria di una ebike, aggiornare i dettagli di una ebike, ricaricare la batteria di una ebike e recuperare tutte le ebike presenti nel sistema.

3.6.3. Infrastructure Layer

RESTEBikeAdapter

Gestisce le richieste REST per il microservizio delle ebike. Configura le rotte HTTP e definisce i metodi per gestire le operazioni sulle ebike, in particolare, espone un endpoint per creare una nuova ebike, un endpoint per ricaricare la batteria dell'ebike, e un endpoint per recuperare tutte le ebike.

MapCommunicationAdapter

Gestisce la comunicazione con il Map Microservice e implementa l'interfaccia MapCommunicationPort. Utilizza un HttpClient per inviare aggiornamenti sulle ebike al microservizio delle mappe. Configura un istanza di HttpClient per inviare richieste HTTP al Map Microservice. Include metodi per inviare aggiornamenti su una singola ebike e inviare aggiornamenti su tutte le ebike.

RideCommunicationAdapter

Gestisce la comunicazione con il Ride Microservice e utilizza `EBikeServiceAPI` per eseguire operazioni sulle ebike e inviare aggiornamenti. Configura un server HTTP con diverse rotte per gestire le richieste in arrivo. Include metodi per recuperare i dettagli di una ebike e aggiornare i dettagli di una ebike.

EBikeVerticle

Rappresenta il punto di ingresso del microservizio degli utenti. Configura le rotte HTTP usando il `RESTEbikeAdapter` e gestisce la registrazione del microservizio al Service Discovery, l'inizio e l'invio degli `heartBeat` per mantenere l'istanza registrata. Include, inoltre, i metodi per la configurazione del server HTTP e per il deploy del verticle.

MongoEBikeRepository

Implementa l'interfaccia `EBikeRepository` e fornisce un'implementazione concreta per l'accesso ai dati delle ebike tramite un database MongoDB per garantire la persistenza. Utilizza MongoClient di Vert.x per eseguire operazioni sulla collezione ebikes.

MetricsManager

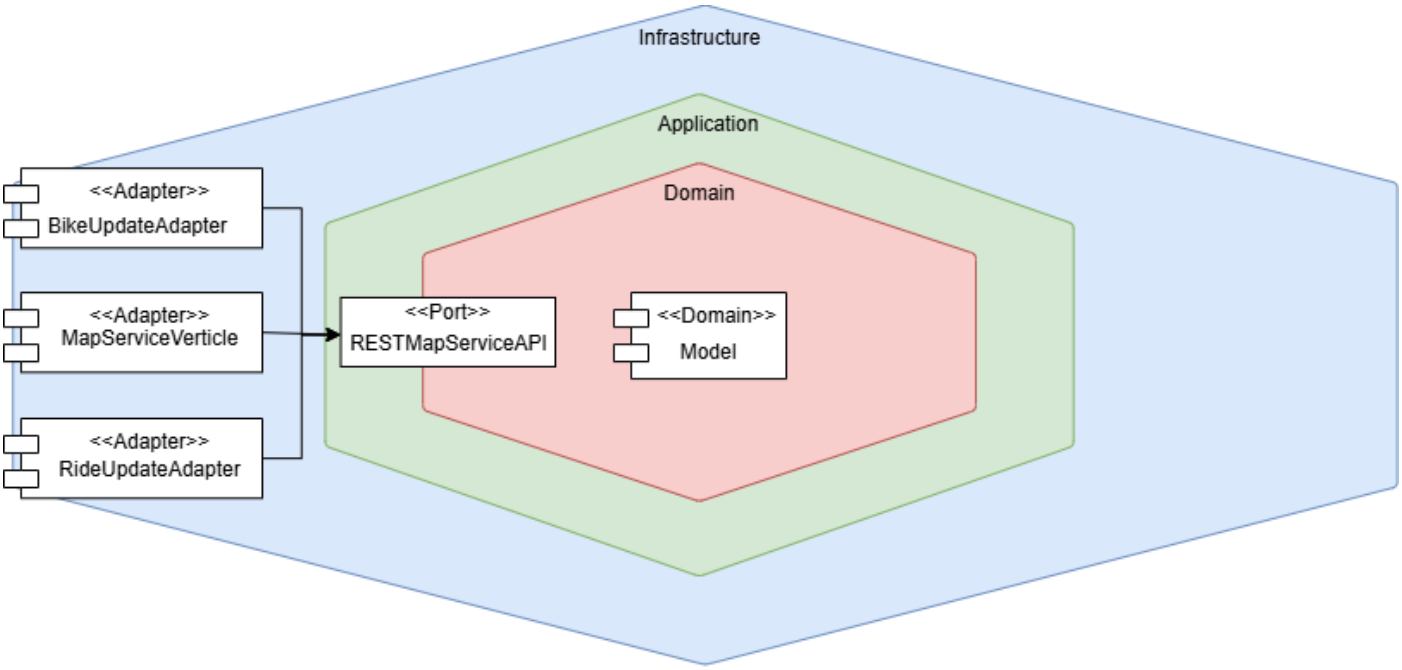
Rappresenta un singleton che gestisce le metriche dell'applicazione utilizzando Micrometer e Prometheus. Fornisce metodi per avviare e registrare timer, incrementare contatori e registrare errori per vari metodi nell'ebike microservice.

ServiceConfiguration

Rappresenta un singleton che gestisce la configurazione del microservizio delle ebike. Fornisce metodi per caricare la configurazione del microservizio dalle variabili d'ambiente e per accedere ai valori di configurazione dei vari adapter e verticle.

3.7. Map Microservice

Il microservizio, organizzato in architettura esagonale, definisce due porte per interfacciarsi al layer di applicazione. Tali porte sono utilizzate nel layer infrastrutturale, che fornisce le implementazioni concrete per pubblicare eventi e gestire le richieste REST. Di seguito un immagine che ne descrive la struttura (Si specifica che il sistema di event publisher è stato non rappresentato per dare maggior enfasi all'interazione tra porte e adattatori riferiti al funzionamento della mappa):



3.7.1. Domain Layer

Il domain layer di Map Microservice è composto dalle classi `Ebike` e `User` che rappresentano rispettivamente una ebike e un utente all'interno del sistema ma adattate al dominio:

- **Ebike**: a differenza della classe Ebike del microservizio delle ebike, questa classe è stata adattata al dominio , pertanto contiene attributi come `bikeName` per identificare la ebike, `position` per indicare la posizione della ebike, `batteryLevel` per indicare il livello di carica della batteria, `state` per indicare lo stato della ebike.
- **User**: a differenza della classe User del microservizio degli user, questa classe è stata adattata al dominio, pertanto contiene attributi come `username` per identificare l'utente e `userBikes` che ospiterà le ebike assegnate all'utente.
- **EBikeFactory**: È una classe *singleton* che fornisce un metodo per creare istanze di EBike.
- **EBikeRepository**: L'interfaccia è responsabile della gestione dei dati delle ebike all'interno del sistema.
- **EbikeRepositoryImpl**: La classe implementa l'interfaccia EBikeRepository e fornisce metodi per salvare, recuperare e gestire le ebike e i loro assegnamenti agli utenti.

A supporto di esse sono state introdotte, inoltre, le classi `EbikeState` e `P2d`, viste in precedenza.

3.7.2. Application Layer

EventPublisher

La porta è progettata per gestire la pubblicazione di eventi relativi alle e-bike e alle attività degli utenti. Include metodi per pubblicare aggiornamenti su tutte le e-bike, aggiornamenti sulle e-bike assegnate a specifici utenti e aggiornamenti sulle e-bike disponibili per gli utenti. Dispone anche di un metodo per pubblicare un evento quando un utente interrompe una corsa. Questa porta è fondamentale per mantenere il sistema e gli utenti informati sullo stato attuale delle e-bike e delle attività degli utenti, garantendo aggiornamenti e notifiche in tempo reale.

RestMapServiceAPI

La porta definisce il contratto per l'API REST che interagisce con il sistema delle e-bike. Include metodi per aggiornare le e-bike, notificare il sistema riguardo agli eventi di corsa e recuperare i dati delle e-bike. Ad esempio, consente di aggiornare una singola e-bike o più e-bike, notificare il sistema quando un utente inizia o interrompe una corsa, e recuperare tutte le e-bike o quelle assegnate a uno specifico

utente. Questa interfaccia facilita la comunicazione tra il sistema delle e-bike e i client esterni, consentendo un'interazione fluida e uno scambio di dati.

3.7.3. Infrastructure Layer

BikeUpdateAdapter

L'adapter utilizza Vert.x per creare un server HTTP e configurare le rotte per gestire le richieste di aggiornamento delle ebike. Espone due endpoint principali: uno per aggiornare una singola ebike e uno per aggiornare più ebike contemporaneamente. Utilizza il RestMapServiceAPI per inviare aggiornamenti al Map Microservice e il MetricsManager per monitorare le metriche delle operazioni eseguite.

RideUpdateAdapter

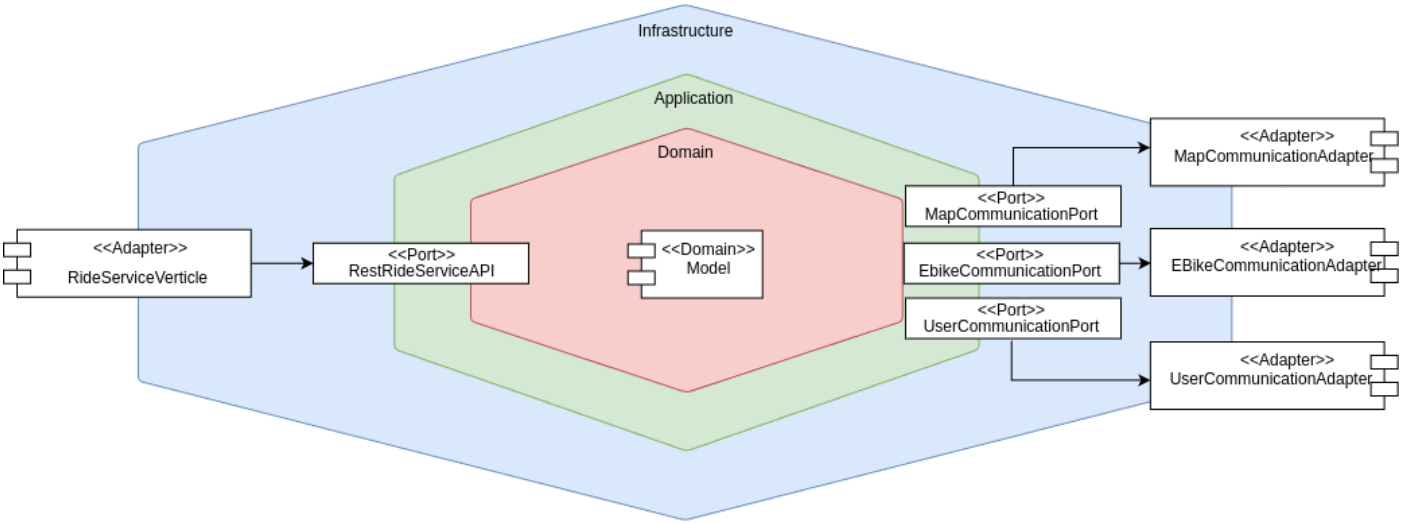
Gestisce le notifiche di inizio e fine corsa delle ebike. Configura un server HTTP con Vert.x e definisce le rotte per ricevere le notifiche di inizio e fine corsa. Utilizza il RestMapServiceAPI per inviare queste notifiche al servizio delle mappe e il MetricsManager per registrare le metriche delle operazioni. Questo adapter assicura che il sistema delle mappe sia sempre aggiornato con lo stato corrente delle corse.

MapServiceVerticle

Questa classe rappresenta il verticle principale del microservizio. Configura un server HTTP e definisce diverse rotte per gestire le richieste, tra cui endpoint per monitorare tutte le e-bike e quelle di un utente specifico. Utilizza il WebClient di Vert.x per registrarsi con Eureka e inviare heartbeat periodici per mantenere l'istanza registrata. Anch'esso utilizza il MetricsManager per monitorare le metriche delle operazioni e garantire che il sistema sia sempre operativo e monitorato.

3.8. Ride Microservice

Il microservizio, organizzato in architettura esagonale, definisce quattro porte per interfacciarsi al layer di applicazione. Queste porte sono utilizzate nel layer infrastrutturale, che fornisce le implementazioni concrete per interagire con la repository, pubblicare eventi, registrare il microservizio al Service Discovery e gestire le richieste REST. Di seguito un immagine che ne descrive la struttura (Si specifica che il sistema di event publisher è stato non rappresentato per dare maggior enfasi all'interazione tra porte e adattatori riferiti al funzionamento della ride).



3.8.1. Domain Layer

Il domain layer di Ride Microservice, essendo un aggregato come visto nel capitolo 1, è composto dalle entità di e-bike e di user, già descritte in precedenza, più le seguenti:

- **Ride**: Rappresenta una corsa all'interno del sistema. Contiene attributi come `id` per identificare la corsa, `user` per identificare l'utente, `ebike` per identificare l'e-bike, `startTime` per indicare il tempo di inizio, `endTime` per indicare il tempo di fine e `ongoing`, un booleano per indicare lo stato della ride. Include i metodi per accedere a tali attributi e per gestire l'inizio e la fine della corsa, modificando gli attributi.
- **RideRepository**: interfaccia che definisce i metodi per gestire le operazioni di persistenza delle corse. Questi metodi includono l'aggiunta, la rimozione e il recupero delle corse, nonché il recupero delle simulazioni di corse tramite l'identificativo della corsa o

dell'utente.

- **RideRepositoryImpl**: implementa l'interfaccia RideRepository utilizzando una struttura dati concorrente per gestire le corse. Essa utilizza una collezione concorrente di simulazioni di corse e un'istanza di Vertx per gestire il ciclo di vita delle simulazioni. I metodi implementati includono l'aggiunta, la rimozione e il recupero delle corse e delle simulazioni di corse.
- **RideSimulation**: gestisce la simulazione di una corsa, aggiornando periodicamente lo stato dell'ebike e dell'utente. Essa contiene un'istanza di Ride, un EventPublisher per pubblicare aggiornamenti, e un'istanza di Vertx per gestire il ciclo di vita della simulazione. La classe fornisce metodi per avviare, aggiornare e completare la simulazione, oltre a metodi per fermare la simulazione manualmente o automaticamente quando la batteria dell'ebike o il credito dell'utente si esauriscono.

Inoltre sono definite le classi `Ebike` e `User` che rappresentano rispettivamente una ebike e un utente all'interno del sistema, ma adattate al dominio specifico della ride. A supporto di esse sono state introdotte inoltre le classi `EbikeState` e `P2d` e `V2d`.

3.8.2. Application Layer

RestRideServiceAPI

La porta definisce i metodi principali per gestire le corse. Include metodi per avviare una corsa e per fermare una corsa, entrambi implementati in modo asincrono. Questa interfaccia facilita la comunicazione tra il sistema delle corse e i client esterni, consentendo un'interazione fluida e uno scambio di dati.

EBikeCommunicationPort

La porta definisce i metodi per comunicare con l'ebike microservice. Include metodi per inviare aggiornamenti sulle ebike, recuperare informazioni su una specifica ebike, e inizializzare la comunicazione. Questi metodi sono fondamentali per mantenere il sistema e il microservizio delle ebike informati sugli aggiornamenti in tempo reale.

MapCommunicationPort

La porta gestisce la comunicazione con il Map Microservice. Fornisce metodi per notificare l'inizio e la fine di una corsa, oltre a un metodo per inizializzare la comunicazione. Questi metodi sono fondamentali per mantenere il sistema e il microservizio delle mappe informati sugli aggiornamenti in tempo reale.

UserCommunicationPort

La porta definisce i metodi per comunicare con l'user microservice. Include metodi per inviare aggiornamenti sugli utenti, recuperare informazioni su un utente specifico, e inizializzare la comunicazione. Questi metodi sono fondamentali per mantenere il sistema e il microservizio degli utenti informati sugli aggiornamenti in tempo reale.

EventPublisher

La porta è progettata per gestire la pubblicazione di aggiornamenti relativi agli utenti e alle ebike coinvolti in una `RideSimulation`.

RestRideServiceAPIImpl

La classe implementa l'interfaccia RestRideServiceAPI e fornisce l'implementazione concreta dei metodi per gestire le corse. Questa classe utilizza le porte definite sopra per interagire con i microservizi delle ebike, delle mappe e degli utenti. Tale classe è il cuore dell'application layer del microservizio delle corse, orchestrando le interazioni tra i vari microservizi e gestendo la logica di business relativa alle corse.

3.8.3. Infrastructure Layer

EBikeCommunicationAdapter

L'adapter implementa l'interfaccia EbikeCommunicationPort e gestisce la comunicazione con il microservizio delle ebike. Utilizza un WebClient per inviare richieste HTTP al microservizio delle ebike e include metodi per la registrazione per ricevere aggiornamenti sulle ebike tramite EventBus, l'invio degli aggiornamenti sulle ebike e il recupero delle informazioni di una specifica ebike inviando una richiesta HTTP.

MapCommunicationAdapter

L'adapter implementa l'interfaccia MapCommunicationPort e gestisce la comunicazione con il microservizio delle mappe. Include metodi

per la registrazione per ricevere aggiornamenti sulle mappe e la notifica dell'inizio e della fine di una corsa.

UserCommunicationAdapter

L'adapter implementa l'interfaccia UserCommunicationPort e gestisce la comunicazione con il microservizio degli utenti. Include metodi per la registrazione per ricevere aggiornamenti sugli utenti tramite EventBus, l'invio degli aggiornamenti sugli utenti e il recupero delle informazioni di uno specifico utente inviando una richiesta HTTP.

RideServiceVerticle

Il verticle rappresenta il punto di ingresso del Ride Microservice. Configura un server HTTP utilizzando Vert.x e definisce le rotte per gestire le richieste di avvio e terminazione delle corse. Inoltre, gestisce la registrazione del microservizio al Service Discovery, l'inizio e l'invio degli `heartBeat` per mantenere l'istanza registrata.

EventPublisherImpl Si tratta dell'implementazione dell'interfaccia `EventPublisher`.

MetricsManager

Anche qui è stato configurato un gestore delle metriche che ha la stessa funzionalità descritta negli altri microservizi.

ServiceConfiguration

Rappresenta un singleton che gestisce la configurazione del microservizio della Ride. Fornisce metodi per caricare la configurazione del microservizio dalle variabili d'ambiente e per accedere ai valori di configurazione dei vari adapter e verticle.

Chapter 4. Descrizione dell'architettura dei microservizi

In questo capitolo viene riportata una descrizione dettagliata di come i microservizi sono integrati tra di loro e come interagiscono, sia all'interno delll'architettura proposta che con il mondo esterno.

È giusto parlare all'inizio di come vengono deployati i microservizi. Come anticipato in precedenza, ognuno ha il suo `Dockerfile` e quindi è già pronto per essere eseguito tramite `docker compose`.

```
services:
  eureka-server:
    build:
      context: ./service-discovery
    ports:
      - 8761:${EUREKA_INSTANCE_PORT}
    networks:
      - eureka-network
    environment:
      - EUREKA_CLIENT_REGISTER_WITH_EUREKA=${EUREKA_CLIENT_REGISTER_WITH_EUREKA}
      - EUREKA_CLIENT_FETCH_REGISTRY=${EUREKA_CLIENT_FETCH_REGISTRY}
      - EUREKA_INSTANCE_HOSTNAME=${EUREKA_INSTANCE_HOSTNAME}
      - EUREKA_INSTANCE_PORT=${EUREKA_INSTANCE_PORT}
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:${EUREKA_INSTANCE_PORT}/actuator/health"]
      interval: 10s
      timeout: 5s
      retries: 3
      start_period: 10s

  map-microservice:
      build:
        context: ./map-microservice
      ports:
        - 8081:${MAP_INSTANCE_PORT}
      networks:
        - eureka-network
      depends_on:
        eureka-server:
          condition: service_healthy
      environment:
        - EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=${EUREKA_CLIENT_SERVICEURL_DEFAULTZONE}
        - EUREKA_HOST=${EUREKA_INSTANCE_HOSTNAME}
        - EUREKA_PORT=${EUREKA_INSTANCE_PORT}
        - SERVICE_NAME=${MAP_INSTANCE_HOSTNAME}
        - SERVICE_PORT=${MAP_INSTANCE_PORT}
        - ADAPTER_RIDE_PORT=${MAP_ADAPTER_RIDE_PORT}
        - ADAPTER_EBIKE_PORT=${MAP_ADAPTER_EBIKE_PORT}
      healthcheck:
        test: ["CMD-SHELL", "curl -f http://localhost:${MAP_INSTANCE_PORT}/health || exit 1", "curl -f
http://localhost:${MAP_ADAPTER_EBIKE_PORT}/health || exit 1", "curl -f http://localhost:${MAP_ADAPTER_RIDE_PORT}/health || exit 1"]
        interval: 10s
        timeout: 5s
        retries: 3
        start_period: 10s

  mongodb:
    image: mongo:latest
    ports:
      - 27017:${MONGODB_INSTANCE_PORT}
    networks:
      - eureka-network
    environment:
      - GLIBC_TUNABLES=glibc.pthread.rseq=0
    volumes:
      - mongodb_data:/data/db
    command: --wiredTigerCacheSizeGB 1
    ulimits:
      memlock: -1
    deploy:
      resources:
        limits:
          memory: 1G
    healthcheck:
      test: mongosh --eval 'db.runCommand("ping").ok' localhost:${MONGODB_INSTANCE_PORT}/test --quiet
      interval: 5s
      timeout: 5s
      retries: 3
      start_period: 5s
```

```
  ebike-microservice:
    build:
      context: ./ebike-microservice
    ports:
      - 8082:${EBIKE_INSTANCE_PORT}
    networks:
      - eureka-network
    depends_on:
      eureka-server:
        condition: service_healthy
      mongodb:
        condition: service_healthy
      map-microservice:
        condition: service_healthy
    environment:
      - EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=${EUREKA_CLIENT_SERVICEURL_DEFAULTZONE}
      - EUREKA_HOST=${EUREKA_INSTANCE_HOSTNAME}
      - EUREKA_PORT=${EUREKA_INSTANCE_PORT}
      - SERVICE_NAME=${EBIKE_INSTANCE_HOSTNAME}
      - SERVICE_PORT=${EBIKE_INSTANCE_PORT}
      - ADAPTER_RIDE_PORT=${EBIKE_ADAPTER_RIDE_PORT}
      - MAP_HOST=${MAP_INSTANCE_HOSTNAME}
      - MAP_PORT=${MAP_ADAPTER_EBIKE_PORT}
      - MONGO_CONNECTION=${MONGODB_CONNECTION_STRING}
      - MONGO_DATABSE=${EBIKE_MONGODB_DATABASE}
    healthcheck:
      test: [ "CMD-SHELL", "curl -f http://localhost:${EBIKE_INSTANCE_PORT}/health", "curl -f
http://localhost:${EBIKE_ADAPTER_RIDE_PORT}/health" ]
      interval: 5s
      timeout: 5s
      retries: 3
      start_period: 5s

  ride-microservice:
    build:
      context: ./ride-microservice
    ports:
      - 8083:${RIDE_INSTANCE_PORT}
    networks:
      - eureka-network
    depends_on:
      eureka-server:
        condition: service_healthy
      ebike-microservice:
        condition: service_healthy
      user-microservice:
        condition: service_healthy
      map-microservice:
        condition: service_healthy
    environment:
      - EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=${EUREKA_CLIENT_SERVICEURL_DEFAULTZONE}
      - EUREKA_HOST=${EUREKA_INSTANCE_HOSTNAME}
      - EUREKA_PORT=${EUREKA_INSTANCE_PORT}
      - SERVICE_NAME=${RIDE_INSTANCE_HOSTNAME}
      - SERVICE_PORT=${RIDE_INSTANCE_PORT}
      - MAP_HOST=${MAP_INSTANCE_HOSTNAME}
      - MAP_PORT=${MAP_ADAPTER_RIDE_PORT}
      - EBIKE_HOST=${EBIKE_INSTANCE_HOSTNAME}
      - EBIKE_PORT=${EBIKE_ADAPTER_RIDE_PORT}
      - USER_HOST=${USER_INSTANCE_HOSTNAME}
      - USER_PORT=${USER_ADAPTER_RIDE_PORT}
    healthcheck:
      test: [ "CMD", "curl", "-f", "http://localhost:${RIDE_INSTANCE_PORT}/health" ]
      interval: 5s
      timeout: 5s
      retries: 3
      start_period: 5s

  user-microservice:
    build:
      context: ./user-microservice
    ports:
      - 8084:${USER_INSTANCE_PORT}
    networks:
      - eureka-network
    depends_on:
      eureka-server:
        condition: service_healthy
      mongodb:
        condition: service_healthy
    environment:
```

```
      - EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=${EUREKA_CLIENT_SERVICEURL_DEFAULTZONE}
      - EUREKA_HOST=${EUREKA_INSTANCE_HOSTNAME}
      - EUREKA_PORT=${EUREKA_INSTANCE_PORT}
      - SERVICE_NAME=${USER_INSTANCE_HOSTNAME}
      - SERVICE_PORT=${USER_INSTANCE_PORT}
      - ADAPTER_RIDE_PORT=${USER_ADAPTER_RIDE_PORT}
      - MONGO_CONNECTION=${MONGODB_CONNECTION_STRING}
      - MONGO_DATABSE=${USER_MONGODB_DATABASE}
    healthcheck:
      test: [ "CMD-SHELL", "curl -f http://localhost:${USER_INSTANCE_PORT}/health", "curl -f
http://localhost:${USER_ADAPTER_RIDE_PORT}/health" ]
      interval: 5s
      timeout: 5s
      retries: 3
      start_period: 5s

  api-gateway:
    build:
      context: ./api-gateway
    ports:
      - 8080:${API_GATEWAY_EUREKA_INSTANCE_PORT}
    networks:
      - eureka-network
    depends_on:
      eureka-server:
        condition: service_healthy
      ebike-microservice:
        condition: service_healthy
      user-microservice:
        condition: service_healthy
      map-microservice:
        condition: service_healthy
      ride-microservice:
        condition: service_healthy
    environment:
      - EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=${EUREKA_CLIENT_SERVICEURL_DEFAULTZONE}
      - API_GATEWAY_EUREKA_INSTANCE_HOSTNAME=${API_GATEWAY_EUREKA_INSTANCE_HOSTNAME}
      - API_GATEWAY_EUREKA_INSTANCE_PORT=${API_GATEWAY_EUREKA_INSTANCE_PORT}
    healthcheck:
      test: [ "CMD", "curl", "-f", "http://localhost:${API_GATEWAY_EUREKA_INSTANCE_PORT}/actuator/health" ]
      interval: 5s
      timeout: 5s
      retries: 3
      start_period: 10s

  prometheus:
    image: prom/prometheus:latest
    ports:
      - 9090:${PROMETHEUS_INSTANCE_PORT}
    networks:
      - eureka-network
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml  # Mount Prometheus configuration file
    depends_on:
      api-gateway:
        condition: service_healthy


networks:
  eureka-network:
    driver: bridge

volumes:
  mongodb_data:
```

Vengono sfruttate in maniera estensiva le variabili d'ambiente, che vengono recuperate da un file .env esterno.

```
#eureka configuration
EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=http://eureka-server:8761/eureka/
EUREKA_CLIENT_REGISTER_WITH_EUREKA=false
EUREKA_CLIENT_FETCH_REGISTRY=false
EUREKA_INSTANCE_HOSTNAME=eureka-server
EUREKA_INSTANCE_PORT=8761

...

RIDE_INSTANCE_HOSTNAME=ride-microservice
```

```
RIDE_INSTANCE_PORT=8080
```

Questo approccio consente una gestione dinamica delle configurazioni, come le porte, i nomi di host e altre impostazioni fondamentali per ogni servizio.

Utilizzando variabili d'ambiente, è possibile configurare la comunicazione tra i vari microservizi, definendo ad esempio gli URL di Eureka (come `EUREKA_CLIENT_SERVICEURL_DEFAULTZONE`), le porte per ciascun servizio (per esempio, `MAP_INSTANCE_PORT`), e i dettagli di connessione al database MongoDB (`MONGO_CONNECTION`). Grazie a questa flessibilità, le configurazioni possono essere facilmente modificate senza dover intervenire direttamente sul file `docker-compose.yml`, migliorando la portabilità e la facilità di adattamento tra ambienti diversi, come sviluppo, test e produzione. Ogni microservizio all'avvio carica la configurazione da utilizzare dalle variabili d'ambiente definite per il proprio container Docker, il caricamento è stato gestito sfruttando la libreria Config di Vert.x .

Per quanto riguarda la gestione della disponibilità dei servizi, ogni microservizio è dotato di un meccanismo di health check, che permette di monitorare lo stato di salute del contenitore. In caso di problemi, come un servizio che non risponde correttamente, Docker può tentare di riavviarlo o segnalarne il fallimento. Gli health check sono configurati in modo tale da assicurare che i microservizi vengano avviati e siano pronti per la comunicazione uno alla volta.

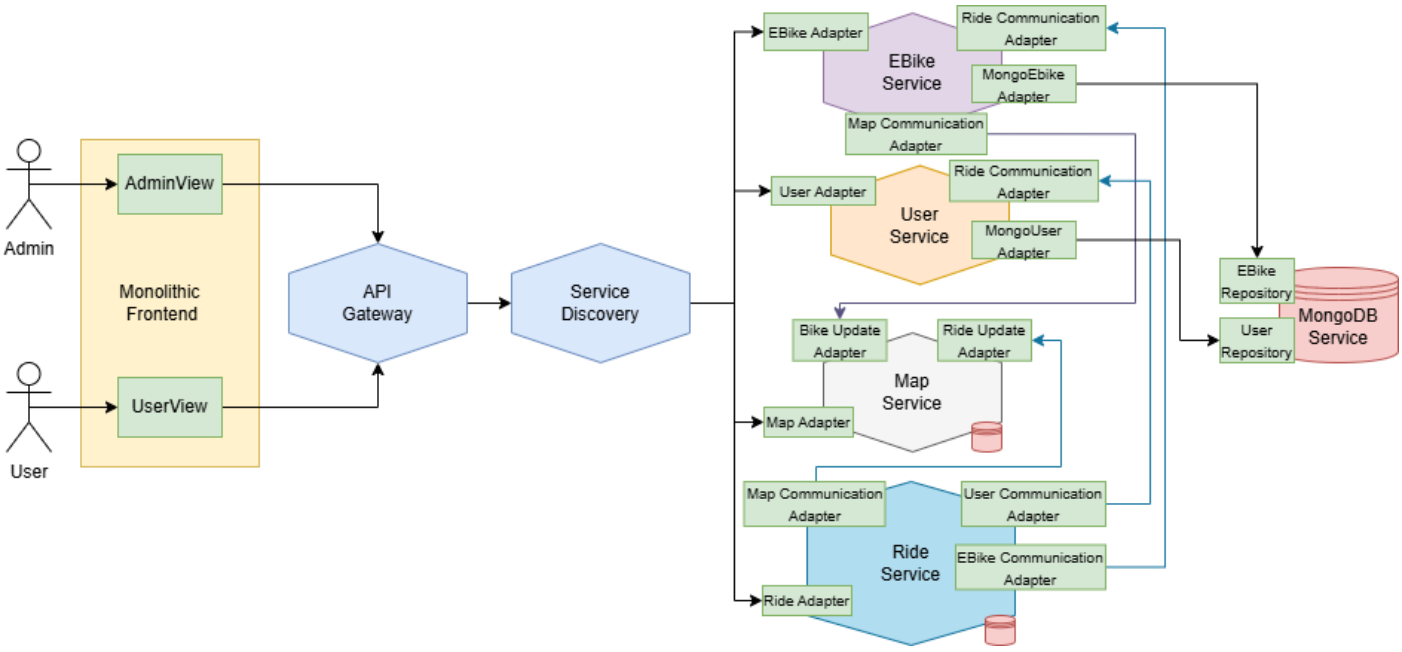
La sequenza di esecuzione è gestita tramite la direttiva `depends_on` e l'opzione `condition: service_healthy`, che garantisce che i servizi vengano avviati in ordine. Il server Eureka, che è fondamentale per la registrazione e la comunicazione dei microservizi, deve essere avviato per primo e deve essere pronto prima che gli altri microservizi possano iniziare a funzionare. I microservizi che dipendono da Eureka, come ad esempio `map-microservice`, `ebike-microservice`, `ride-microservice`, `user-microservice` e `api-gateway`, non verranno avviati finché il server Eureka non sarà attivo. Alcuni di questi microservizi, come `ebike-microservice` e `user-microservice`, dipendono anche dalla disponibilità di MongoDB, il che significa che l'accesso a tale servizio è monitorato tramite health check dedicati.

In generale la comunicazione tra i microservizi avviene tramite HTTP e chiamate REST: essi espongono le loro funzionalità tramite API RESTful. Ogni microservizio ha un set di endpoint HTTP che possono essere chiamati da altri microservizi o da client esterni. Ad esempio, il `RideServiceVerticle` espone endpoint come `/startRide` e `/stopRide` per gestire le ride.

Ogni microservizio si registra con Eureka all'avvio, fornendo informazioni come l'hostname, la porta e gli endpoint di health, questi ultimi usati per rispettare il pattern di `HealthCheckAPI`. Questo permette agli altri microservizi di trovare e comunicare con esso senza dover conoscere a priori il suo indirizzo. I microservizi utilizzano client HTTP, come WebClient di Vert.x, per inviare richieste ad altri microservizi. Ad esempio, il `EBikeCommunicationAdapter` utilizza WebClient per inviare aggiornamenti al microservizio delle ebike.

L'API Gateway è responsabile di instradare le richieste ai microservizi corretti, in base al path della richiesta. Ad esempio, una richiesta GET come `getAllBikes` viene instradata al `EbikeServiceVerticle`.

Di seguito, un grafico di come avviene la comunicazione dell'intero sistema e tra i microservizi:



Flussi di comunicazione

- Avvio di una ride:

1. Il client invia una richiesta HTTP al RideServiceVerticle tramite l'API Gateway per avviare una ride.
2. Il RideServiceVerticle verifica la disponibilità dell'ebike e dell'utente chiamando i microservizi delle ebike e degli utenti tramite i rispettivi adapter (EBikeCommunicationAdapter e UserCommunicationAdapter).
3. Una volta avviata la corsa, il RideServiceVerticle notifica il microservizio delle mappe tramite il MapCommunicationAdapter.
4. Il RideServiceVerticle invia una richiesta al EBikeCommunicationAdapter per iniziare la ride.
5. Il EBikeCommunicationAdapter invia una richiesta alla ebike per iniziare la ride.
6. La ebike conferma l'inizio della ride.
7. Il RideServiceVerticle restituisce la conferma al client.

- Stop di una ride:
  1. Il client invia una richiesta HTTP al RideServiceVerticle tramite l'API Gateway per fermare una ride.
  2. Il RideServiceVerticle recupera la simulazione della corsa associata all'utente.
  3. Il RideServiceVerticle ferma la simulazione manualmente e aggiorna lo stato dell'ebike.
  4. Il RideServiceVerticle notifica il microservizio delle mappe della fine della corsa tramite il MapCommunicationAdapter.
  5. Il RideServiceVerticle rimuove la corsa dal repository.
  6. Il RideServiceVerticle restituisce la conferma al client.
- Aggiunta di una nuova e-bike:
  1. L'amministratore invia una richiesta HTTP al EbikeServiceVerticle tramite l'API Gateway per aggiungere una nuova bici.
  2. Il EbikeServiceVerticle verifica i dettagli della nuova bici.
  3. Il EbikeServiceVerticle aggiunge la nuova bici al repository delle ebike.
  4. Il EbikeServiceVerticle notifica il microservizio delle mappe della nuova bici tramite il MapCommunicationAdapter.
  5. Il EbikeServiceVerticle restituisce la conferma all'amministratore.
- Ricarica di una e-bike:
  1. L'amministratore invia una richiesta HTTP al EbikeServiceVerticle tramite l'API Gateway per ricaricare una bici.
  2. Il EbikeServiceVerticle verifica l'identificativo della bici e la sua disponibilità.
  3. Il EbikeServiceVerticle invia una richiesta al EBikeCommunicationAdapter per ricaricare la bici.
  4. Il EBikeCommunicationAdapter invia una richiesta alla bici per iniziare la ricarica.
  5. La bici conferma l'inizio della ricarica.
  6. Il EbikeServiceVerticle aggiorna lo stato della bici nel repository.
  7. Il EbikeServiceVerticle restituisce la conferma all'amministratore.
- Ricarica del credito:
  1. L'utente invia una richiesta HTTP al UserServiceVerticle tramite l'API Gateway per ricaricare il proprio credito.
  2. Il UserServiceVerticle verifica l'identificativo dell'utente e l'importo della ricarica.
  3. Il UserServiceVerticle aggiorna il credito dell'utente nel repository degli utenti.
  4. Il UserServiceVerticle notifica il microservizio delle corse dell'aggiornamento del credito tramite il RideServiceVerticle.
  5. Il UserServiceVerticle restituisce la conferma all'utente

Chapter 5. Validazione dell'architettura proposta

Allo scopo di validare il sistema, sono stati definiti dei test, seppur limitati a una piccola parte di applicazione.

Inoltre, è stato eseguito un test di utilizzo del sistema, allo scopo di verificare che funzioni allo stesso modo della versione non a microservizi.

5.1. Test pyramid

Nel dettaglio, è stato definito un test per ogni layer dell'applicazione, seguendo la piramide dei test. Questi ultimi sono stati scritti utilizzando il framework JUnit, e sono stati eseguiti con successo. Di seguito vengono descritti brevemente i test eseguiti suddivisi per tipologia.

5.1.1. Unit Test

```
@Test
    public void testSaveAndGetBike() throws ExecutionException, InterruptedException {
        repository.saveBike(bike1).get();

        EBike retrievedBike = repository.getBike("Bike1").get();
        assertNotNull(retrievedBike);
        assertEquals("Bike1", retrievedBike.getBikeName());
        assertEquals(EBikeState.AVAILABLE, retrievedBike.getState());
    }

    @Test
    public void testGetBike_NotFound() {
        assertThrows(ExecutionException.class, () -> repository.getBike("NonExistentBike").get());
    }

    @Test
    public void testAssignBikeToUser() throws ExecutionException, InterruptedException {
        repository.saveBike(bike1).get();
        repository.assignBikeToUser("User1", bike1).get();

        String assignedUser = repository.isBikeAssigned(bike1).get();
        assertEquals("User1", assignedUser);
    }

    @Test
    public void testUnassignBikeFromUser() throws ExecutionException, InterruptedException {
        repository.saveBike(bike1).get();
        repository.assignBikeToUser("User1", bike1).get();

        repository.unassignBikeFromUser("User1", bike1).get();
        String assignedUser = repository.isBikeAssigned(bike1).get();
        assertNull(assignedUser);
    }

    @Test
    public void testGetAvailableBikes() throws ExecutionException, InterruptedException {
        repository.saveBike(bike1).get();
        repository.saveBike(bike2).get();

        List<EBike> availableBikes = repository.getAvailableBikes().get();
        assertEquals(1, availableBikes.size());
        assertEquals("Bike1", availableBikes.get(0).getBikeName());
    }
```

Trattandosi di un test Unit, viene testata la parte di codice interna al microservizio, senza interagire con altri.

Vengono testati i metodi di salvataggio e recupero di una bicicletta, l'assegnazione e la rimozione di una bicicletta da un utente, e il recupero delle biciclette disponibili.

5.1.2. Integration Test

```
@Test
    void testSignUpAndSignIn() {
        System.out.println("Testing sign Up and sign In");
```

```
        registerUser("testuser", "USER").join();
        signIn("testuser").join();
    }

    private static CompletableFuture<Void> signIn(String username) {
        CompletableFuture<Void> future = new CompletableFuture<>();
        JsonObject user = new JsonObject().put("username", username);

        client.post(8084, "localhost", "/api/users/signin")
                .sendJsonObject(user, ar -> {
                    if (ar.succeeded()) {
                        HttpResponse<Buffer> response = ar.result();
                        System.out.println("User sign in SUCCEEDED: " + response.bodyAsString());
                        assertEquals(200, response.statusCode());
                        future.complete(null);
                    } else {
                        System.err.println("User sign in FAILED: " + ar.cause().getMessage());
                        future.completeExceptionally(ar.cause());
                    }
                });

        return future;
    }
```

Allo scopo di testare il collegamento tra un microservizio e un servizio esterno come quello di database, è stato definito un docker compose ad hoc per testare esclusivamente questo aspetto.

```
services:

  mongodb:
    image: mongo:latest
    ports:
      - 27017:${MONGODB_INSTANCE_PORT}
    networks:
      - eureka-network
    environment:
      - GLIBC_TUNABLES=glibc.pthread.rseq=0
    volumes:
      - mongodb_data:/data/db
    command: --wiredTigerCacheSizeGB 1
    ulimits:
      memlock: -1
    deploy:
      resources:
        limits:
          memory: 1G
    healthcheck:
      test: mongosh --eval 'db.runCommand("ping").ok' localhost:${MONGODB_INSTANCE_PORT}/test --quiet
      interval: 5s
      timeout: 5s
      retries: 3
      start_period: 5s

  user-microservice:
    build:
      context: ../../../../../../../user-microservice
    ports:
      - 8084:${USER_INSTANCE_PORT}
    networks:
      - eureka-network
    depends_on:
      mongodb:
        condition: service_healthy
    environment:
      - EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=${EUREKA_CLIENT_SERVICEURL_DEFAULTZONE}
      - EUREKA_HOST=${EUREKA_INSTANCE_HOSTNAME}
      - EUREKA_PORT=${EUREKA_INSTANCE_PORT}
      - SERVICE_NAME=${USER_INSTANCE_HOSTNAME}
      - SERVICE_PORT=${USER_INSTANCE_PORT}
      - ADAPTER_RIDE_PORT=${USER_ADAPTER_RIDE_PORT}
      - MONGO_CONNECTION=${MONGODB_CONNECTION_STRING}
      - MONGO_DATABSE=${USER_MONGODB_DATABASE}
    healthcheck:
      test: [ "CMD-SHELL", "curl -f http://localhost:${USER_INSTANCE_PORT}/health", "curl -f
http://localhost:${USER_ADAPTER_RIDE_PORT}/health" ]
      interval: 5s
      timeout: 5s
      retries: 3
```

```
      start_period: 5s

networks:
  eureka-network:
    driver: bridge

volumes:
  mongodb_data:
```

Il test è stato eseguito con successo, dimostrando che il microservizio è in grado di comunicare con il database e di eseguire le operazioni di registrazione e accesso degli utenti.

5.1.3. Component Test

```
@BeforeEach
    void setUp(VertxTestContext testContext) {
        vertx = Vertx.vertx();
        client = vertx.createHttpClient();
        webClient = WebClient.create(vertx);

        // Initialize components
        EBikeRepositoryImpl repository = new EBikeRepositoryImpl();
        EventPublisher eventPublisher = new EventPublisherImpl(vertx);
        RestMapServiceAPI mapService = new RestMapServiceAPIImpl(repository, eventPublisher);

        ServiceConfiguration config = ServiceConfiguration.getInstance(vertx);
        config.load().onSuccess(conf -> {
            // Deploy verticles
            vertx.deployVerticle(new MapServiceVerticle(mapService, vertx))
                    .compose(id -> vertx.deployVerticle(new BikeUpdateAdapter(mapService, vertx)))
                    .onComplete(ar -> {
                        if (ar.succeeded()) {
                            vertx.setTimer(1000, id -> testContext.completeNow());
                        } else {
                            testContext.failNow(ar.cause());
                        }
                    });
        });

    }

    @AfterEach
    void tearDown(VertxTestContext testContext) {
        vertx.close().onComplete(testContext.succeeding(v -> testContext.completeNow()));
    }

    @Test
    void testUpdateEBikeAndObserveAllBikes(VertxTestContext testContext) {
        JsonObject bikeJson = new JsonObject()
                .put("id", "bike1")
                .put("location", new JsonObject()
                        .put("x", 10.0)
                        .put("y", 20.0))
                .put("state", "AVAILABLE")
                .put("batteryLevel", 100);

        // Send update request
        webClient.put(BIKE_UPDATE_PORT, "localhost", "/updateEBike")
                .putHeader("Content-Type", "application/json")
                .sendJsonObject(bikeJson)
                .onComplete(ar -> {
                    if (ar.failed()) {
                        testContext.failNow(ar.cause());
                    }
                });

        // Set up WebSocket client
        client.webSocket(8080, "localhost", "/observeAllBikes")
                .onComplete(testContext.succeeding(webSocket -> {
                    webSocket.handler(buffer -> {
                        JsonArray receivedBike = buffer.toJsonArray();
                        JsonObject bike = new JsonObject(receivedBike.getString(0));
                        testContext.verify(() -> {
                            assertEquals("bike1", bike.getString("bikeName"));
                            assertEquals(10.0, bike.getJsonObject("position").getDouble("x"));
                            assertEquals(20.0, bike.getJsonObject("position").getDouble("y"));
                            assertEquals("AVAILABLE", bike.getString("state"));
```

```
                        assertEquals(100, bike.getInteger("batteryLevel"));
                        testContext.completeNow();
                    });
                });
            }));
    }
```

Il test verifica due operazioni chiave:

- Aggiornamento dello stato di una eBike tramite una richiesta HTTP (PUT).
- Osservazione in tempo reale dello stato delle biciclette tramite WebSocket.

La logica è testata end-to-end all'interno del servizio, senza però coinvolgere infrastrutture esterne reali come database o network, che vengono simulate (grazie a Vertx). Questo rende il test rappresentativo di un comportamento reale del servizio, pur mantenendolo isolato e rapido.

Questo test dimostra che il servizio è in grado di gestire correttamente il flusso di dati dal momento in cui riceve una richiesta di aggiornamento fino alla notifica in tempo reale tramite WebSocket.

Coinvolge più componenti del servizio, verificandone il funzionamento interno senza coinvolgere l'intero sistema.

5.1.4. End-to-End Test

```
@BeforeAll
    static void setUp() {
        String command = "cd ../../ && docker compose up -d --build";
        ProcessBuilder processBuilder = new ProcessBuilder();
        processBuilder.command("bash", "-c", command);

            int exitCode = process.waitFor();
            System.out.println("Docker Compose Exit code: " + exitCode);

        } catch (InterruptedException | IOException e) {
            throw new RuntimeException(e);
        }

        vertx = Vertx.vertx();
        client = WebClient.create(vertx);

        // Wait for health check to succeed
        waitForHealthCheck();
    }

    private static void waitForHealthCheck() {
        CompletableFuture<Void> healthCheckFuture = new CompletableFuture<>();

        vertx.setPeriodic(1000, id -> { // Poll every 1 second
            client.get(8080, "localhost", "/actuator/health")
                    .send(ar -> {
                        if (ar.succeeded()) {
                            HttpResponse<Buffer> response = ar.result();
                            String body = response.bodyAsString();
                            if (response.statusCode() == 200 && body.contains("\"status\":\"UP\"")) {
                                System.out.println("Health check SUCCEEDED: " + body);
                                healthCheckFuture.complete(null); // Complete the future on success
                                vertx.cancelTimer(id); // Cancel the periodic task
                            } else {
                                System.err.println("Health check response but not ready: " + body);
                            }
                        } else {
                            System.err.println("Health check failed: " + ar.cause().getMessage());
                        }
                    });
        });

    try {
        // Wait for the health check to succeed or timeout after 60 seconds
        healthCheckFuture.get(100, TimeUnit.SECONDS);
        System.out.println("Health check succeeded in time");
    } catch (Exception e) {
        throw new RuntimeException("Health check did not succeed in time", e);
    }
}
```

```
@Test
void testEBikeCreationAndArrive() {
    var httpClient = vertx.createHttpClient();

    // Register users after health check succeeds
    CompletableFuture<Void> adminFuture = registerUser("admin", "ADMIN");
    CompletableFuture<Void> userFuture = registerUser("user", "USER");

    // Wait for user registration to complete
    CompletableFuture.allOf(adminFuture, userFuture).thenCompose(v -> {
        // Create an eBike after user registration
        return createEBike("bike1", 10.0, 20.0, "AVAILABLE", 100);
    }).thenAccept(v -> {
        // Connect to WebSocket and verify the received message
        httpClient.webSocket(8080, "localhost", "/MAP-MICROSERVICE/observeUserBikes?username=user")
                .onSuccess(ws -> {
                    ws.textMessageHandler(message -> {
                        System.out.print("Received WebSocket message: " + message);
                        JsonArray receivedArray = new JsonArray(message);
                        String bikeString = receivedArray.getString(0);
                        JsonObject receivedBike = new JsonObject(bikeString);

                        JsonObject expectedBike = new JsonObject()
                            .put("bikeName", "bike1")
                            .put("position", new JsonObject()
                                .put("x", 10.0)
                                .put("y", 20.0))
                            .put("state", "AVAILABLE")
                            .put("batteryLevel", 100);

                        assertEquals(expectedBike, receivedBike);
                    });
                });
    }).join(); // Wait for all operations to complete
}

private static CompletableFuture<Void> registerUser(String username, String type) {
    CompletableFuture<Void> future = new CompletableFuture<>();
    JsonObject user = new JsonObject()
            .put("username", username)
            .put("type", type);

    client.post(8080, "localhost", "/USER-MICROSERVICE/api/users/signup")
            .sendJsonObject(user, ar -> {
                if (ar.succeeded()) {
                    System.out.println("User registration SUCCEEDED: " + ar.result().bodyAsString());
                    future.complete(null);
                } else {
                    System.err.println("User registration failed: " + ar.cause().getMessage());
                    future.completeExceptionally(ar.cause());
                }
            });
    return future;
}

private static CompletableFuture<Void> createEBike(String id, double x, double y, String status, int batteryLevel) {
    CompletableFuture<Void> future = new CompletableFuture<>();
    JsonObject ebike = new JsonObject()
            .put("id", id)
            .put("x", x)
            .put("y", y)
            .put("status", status)
            .put("batteryLevel", batteryLevel);

    client.post(8080, "localhost", "/EBIKE-MICROSERVICE/api/ebikes/create")
            .sendJsonObject(ebike, ar -> {
                if (ar.succeeded()) {
                    System.out.println("EBike creation SUCCEEDED: " + ar.result().bodyAsString());
                    future.complete(null);
                } else {
                    System.err.println("EBike creation failed: " + ar.cause().getMessage());
                    future.completeExceptionally(ar.cause());
                }
            });
    return future;
}
```

Questo tipo di test verifica l'intero flusso operativo del sistema, assicurandosi che tutti i componenti interagiscano correttamente in un ambiente il più vicino possibile a quello di produzione.

L’obiettivo principale di tale test è simulare l’esperienza utente, dalla registrazione degli utenti alla creazione di una eBike e alla verifica delle notifiche in tempo reale tramite WebSocket. Questo assicura che tutte le parti del sistema funzionino bene insieme.

Esso coinvolge l’intero sistema, avviando tutti i container docker interessati tramite un comando shell.

La registrazione degli utenti è il primo passo: un amministratore e un utente comune vengono creati con i rispettivi ruoli. Successivamente, un utente comune registra una nuova eBike con attributi specifici, come l’ID, la posizione e lo stato della batteria. Una volta che la eBike è creata, il test prosegue con la connessione all’endpoint WebSocket per osservare le bici utente.

La fase successiva implica la validazione del messaggio ricevuto dal WebSocket. Una volta connesso, il sistema invia messaggi contenenti i dati della eBike appena creata, e il test verifica che questi dati corrispondano esattamente a quelli previsti.

Infine, una volta terminato il test, si passa alla fase di pulizia finale, dove tutti i container e i volumi associati ai microservizi vengono fermati e rimossi tramite Docker Compose. Questo assicura che non ci siano effetti collaterali tra diversi test e che l’ambiente venga sempre ripristinato allo stato iniziale, pronto per un nuovo ciclo di test.

5.2. Commento sulla validazione dell’architettura

In conclusione, l’analisi condotta nei precedenti capitoli e i test generati dimostrano come la strategia adottata per soddisfare i requisiti e gli attributi di qualità sia stata efficace. Si evince, infatti, come i quattro microservizi principali prodotti, ricordando che discendono da sottodomini di tipo core, racchiudano in maniera appropriata ogni funzionalità identificata, favorendone la gestione modulare e facilitando l’estendibilità, la modificabilità e la scalabilità del sistema. L’utilizzo di Docker e di API standard per la comunicazione ha permesso di isolare i servizi e di garantire l’interoperabilità tra di essi, mentre l’uso di Vert.x ha assicurato performance elevate e una gestione efficiente degli eventi. Inoltre, la testabilità dei microservizi è stata garantita grazie alla definizione di test a vari livelli, che hanno permesso di verificare il corretto funzionamento di alcune funzionalità offerte da essi, sia individualmente, sia nel loro funzionamento integrato. Tuttavia, l’insieme dei test potrebbe essere ulteriormente esteso per coprire scenari più complessi e garantire una maggiore robustezza, migliorando così l’affidabilità e l’efficacia del sistema in ambienti di produzione.