

Relazione Assignment #02

E-Bike Application - Microservices

Software Architecture and Platforms

Ravaioli Michele - michele.ravaioli3@studio.unibo.it

4 novembre 2024

Indice

- 1. Requisiti**
- 2. QAS**
- 3. Use cases**
 - 3.1. User stories**
 - 3.2. Use cases and scenarios**
- 4. Bounded Contexts**
- 5. Microservices**
 - 5.1. Bike Manager**
 - 5.2. Account Manager**
 - 5.3. Ride Manager**
 - 5.4. Authentication Server**
 - 5.5. API Gateway**
 - 5.6. Metric Server**
- 6. Testing**

1. Requisiti

Il progetto "E-Bike Application" prevede lo sviluppo di sistema di noleggio di e-bike con un'architettura distribuita basata su microservizi e diversi pattern architetturali. Per progettare l'applicazione si richiede di utilizzare un approccio Domain-Driven.

Adottando il metodo DDD, si chiede di identificare nel problema:

- Quality Attributes;
- Use Cases;
- Bounded Contexts e Ubiquitous Languages.

Una volta trovati i bounded contexts, si chiede di realizzare i vari microservizi, utilizzando i seguenti pattern architetturali:

- API Gateway, Service discovery;
- Service-as-Container per l'installazione;
- Health Check API, Application Metrics per l'osservabilità;
- Externalized Configuration;
- Testing per ogni layer della piramide di test (Unit, Integration, Component, End-to-End);

Inoltre, è richiesto l'utilizzo di un framework disponibile per lo sviluppo di almeno un microservizio.

2. QAS

Per garantire la qualità del servizio, il sistema deve soddisfare almeno due requisiti fondamentali: **disponibilità** e **reattività**. Queste qualità architetturali sono essenziali affinché il sistema sia idoneo al proprio scopo; la loro mancanza comprometterebbe l'efficacia del sistema.

Per verificare il rispetto di questi requisiti, sono stati definiti e analizzati specifici scenari di test, descritti di seguito.

QAS: Availability	
Source	Server hosting a service
Stimulus	Server fails
Artifact	Server
Environment	Normal operation
Response	Server restarts
Response measure	Service downtime less than 1 min

QAS: Responsiveness	
Source	Server hosting a service
Stimulus	Server receives a request
Artifact	Server
Environment	Overloaded operation
Response	Server replies (with correct values or with error)
Response measure	Request is managed in less than 1 sec

3. Use cases

Per descrivere e analizzare il dominio con approccio Domain-Driven, si è deciso di identificare tutti i possibili casi d'uso, e di derivare così i componenti necessari del sistema.

3.1. User stories

Per poter sviluppare i casi d'uso, si sono raccolte le user stories del sistema, riportate di seguito:

As a user without account
I want to create a new user account
So that i can start using the service

As a user
I want to log into my account
So that i can start using the service

As a user
I want to connect to a ebike
So that i can rent that ebike

As a user
I want to disconnect from a ebike
So that i can stop the rent of that ebike
And i can rent another ebike again

As a user
I want to recharge my credit
So that i can continue using the service

As an administrator
I want to add an ebike
So that i can make that ebike available to the users

As an administrator
I want to monitor the state of the system
So that i can run analysis on the system's state

As an ebike
I want to periodically notify the service about my state
So that the service knows my current state

3.2. User stories

A partire dalle user stories, si sono individuati i casi d'uso principali e i vari scenari collegati:

User account creation

Main Success Scenario:

1. User requests to create a new account
2. User provides a unique username and a password
3. System creates a new account with 0 credits
4. System automatically logs the user in
5. System sends to the user a session token related to the new account

Extensions:

- 2a: User provides a username already used and a password
1. System cancels the operation and informs the user

User login

Main Success Scenario:

1. User requests to log into their account
2. User provides a valid username and password
3. System authenticates the credentials
4. System sends the user a session token and grants access to the account

Extensions:

- 2a: User provides an incorrect username or password
1. System denies access and displays an error message

User connection to an Ebike

Main Success Scenario:

1. User requests to connect to an ebike
2. User provides ID
3. System verifies the user has enough credits
4. System verifies the ebike is not connected to another user
5. System registers the connection between the user and the ebike
6. User starts using the ebike

Extensions:

- 2a: User provides a wrong ebike ID
1. System denies the connection and displays an error message
- 3a: User has not enough credits
1. System denies the connection and informs the user
- 4a: Ebike is already used by another user
1. System denies the connection and informs the user

User disconnection from an Ebike

Main Success Scenario:

1. User requests to disconnect from the currently connected ebike
2. System verifies the user's current connection to the ebike
3. System computes the price for the rental and deducts it from the user credits
4. System registers the disconnection and ends the rental session
5. System makes the ebike available for other users to rent

Extensions:

- 2a: User is not connected to this or any ebike
1. System denies the operation and informs the user

User credits recharge

Main Success Scenario:

1. User requests to recharge their account credit
2. User provides a recharge amount
3. System processes the payment
4. System adds the selected amount to the user's account balance

Extensions:

- 2a: User provides an invalid recharge amount
1. System denies the operation and displays an error message

Add ebike

Main Success Scenario:

1. Administrator requests to add a new ebike
2. Administrator provides the required details for the new ebike (ID, model, battery capacity)
3. System makes the ebike available for users to connect to and rent

Extensions:

- 2a: Administrator provides an ebike already present in the system
1. System denies the operation and displays an error message

Monitor the state of the system

Main Success Scenario:

1. Administrator requests to see the system's state
2. System collects data from all ebikes and user activity logs and sends them to the administrator
3. Administrator reviews the state and conducts analysis

Periodic notification of ebike state

Main Success Scenario:

1. Ebike periodically sends its current state to the system (location, battery level)
2. System receives the data
3. System updates the ebike's status for administrators

Extensions:

- 1a: Ebike sends data in an unexpected format
1. System ignores message

Di seguito viene riportato il diagramma dei casi d'uso, che li comprende tutti:

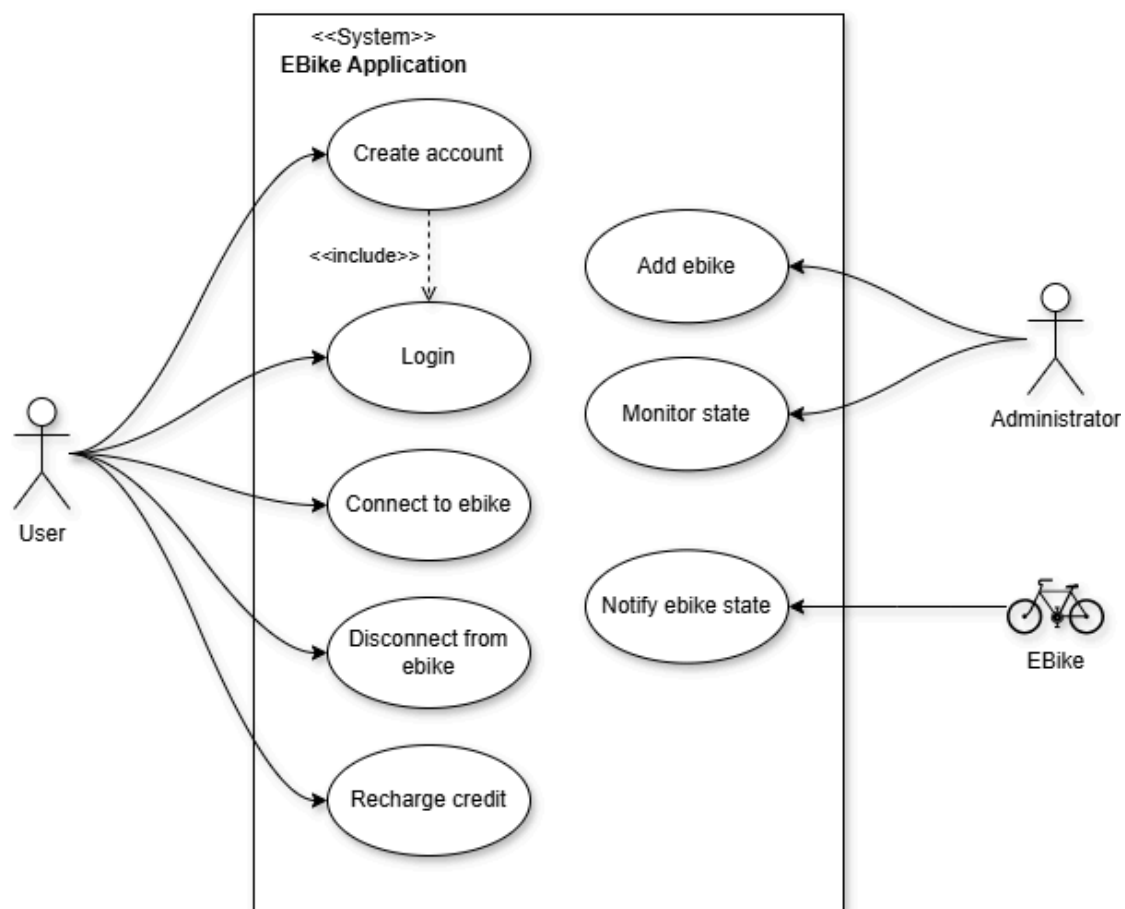


Fig. 1: Diagramma dei casi d'uso

4. Bounded Contexts

Dai casi d'uso identificati, sono stati derivati i **bounded contexts** del dominio, suddivisi in quattro principali aree funzionali:

1. **Gestione delle biciclette (Bike Management):**

Questo contesto si occupa di tutte le operazioni relative alle biciclette, come l'aggiunta di nuove bici, l'aggiornamento del loro stato e le query sui dettagli delle biciclette.

2. **Gestione degli account (Account Management):**

Qui vengono gestiti i dati degli utenti, con particolare attenzione alla gestione del credito e alle operazioni di ricarica.

3. **Gestione dell'autenticazione (Authentication):**

Si occupa della generazione dei token di sessione e verifica l'autenticità dell'utente basandosi sulla password. Questo contesto dipende dal **Account Management**.

4. **Gestione delle corse (Ride Management):**

Riguarda il servizio di noleggio delle biciclette, consentendo agli utenti di connettersi e disconnettersi da una bici. Questo contesto dipende sia dal **Bike Management** che dall'**Account Management**.

In seguito è presentata la **context map** (Fig. 2), che illustra le collaborazioni tra i vari contesti.

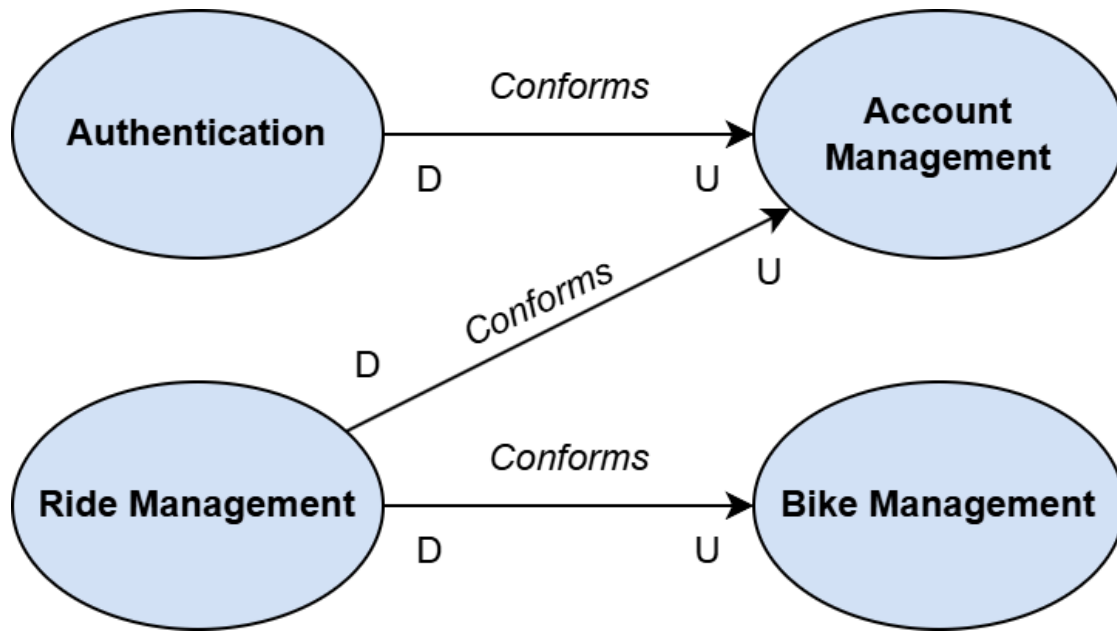


Fig. 2: Context map

5. Microservices

Dall'analisi dei bounded context, sono stati individuati i microservizi da implementare. Oltre a quelli derivati direttamente dai bounded context, sono stati creati ulteriori microservizi per soddisfare specifici requisiti del progetto. I microservizi individuati sono:

1. **Bike Manager:** gestisce tutte le operazioni relative alle biciclette.
2. **Account Manager:** si occupa della gestione degli account degli utenti.
3. **Ride Manager:** gestisce il flusso delle corse, inclusa la connessione e disconnessione dalle biciclette.
4. **Authentication Server:** fornisce il servizio di autenticazione degli utenti.
5. **Metric Server:** raccoglie e centralizza le metriche dai vari servizi, seguendo il pattern di observability.
6. **API Gateway:** agisce come proxy per orchestrare e semplificare le chiamate alle API.

Nella Fig. 3 è riportato lo schema dei microservizi e delle interazioni tra di essi.

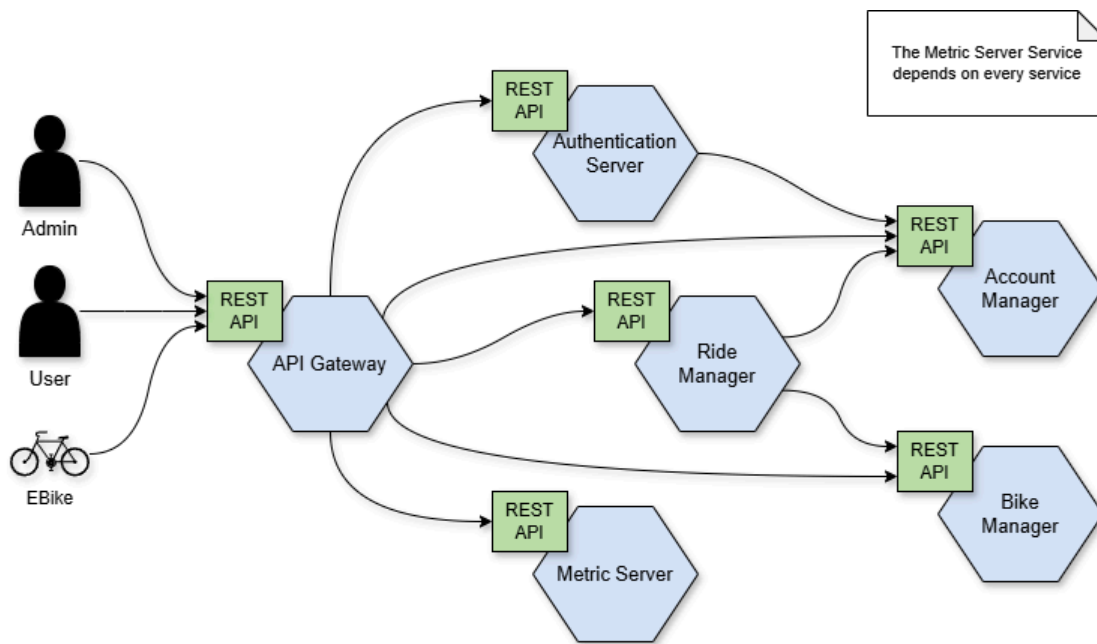


Fig. 3: Schema dei microservizi

Implementazioni pattern architetturali

Per soddisfare i requisiti del progetto, sono stati adottati specifici pattern architetturali supportati da tecnologie dedicate, utili per lo sviluppo e la gestione dei servizi:

- **Docker:** utilizzato per il *deployment* dei servizi come container e per la loro *discoverability*.
- **Prometheus:** impiegato per gestire l'observability delle metriche dei servizi e per implementare automaticamente l'*health check pattern*.

In aggiunta:

- L'*externalized configuration* è stata sviluppata ex novo all'interno del servizio **Bike Manager**.
- Per il *testability pattern*, sono stati implementati semplici test sempre nel contesto del **Bike Manager** (per i dettagli sui test eseguiti, consultare il capitolo dedicato).

5.1. Bike Manager

Bike Manager è il servizio responsabile della gestione delle e-bikes. Deve mantenere in memoria le informazioni riguardo le biciclette del sistema e deve essere continuamente aggiornato dalle stesse biciclette, che autonomamente inviano i propri dati al servizio. Il servizio deve consentire inoltre di poter eseguire interrogazioni sulle e-bikes presenti e di aggiungerne di nuove.

Ubiquitous Language

Per lavorare in questo contesto, è necessario definire alcuni termini per evitare ambiguità, presentati nella seguente tabella:

Termine	Definizione	Sinonimi
<i>Bike</i>	Entità che rappresenta una bicicletta elettrica	<i>EBike</i>

Design

L'architettura è basata sul modello port-adapter (Fig. 4), ed espone delle Rest API per integrarsi con gli altri servizi del sistema:

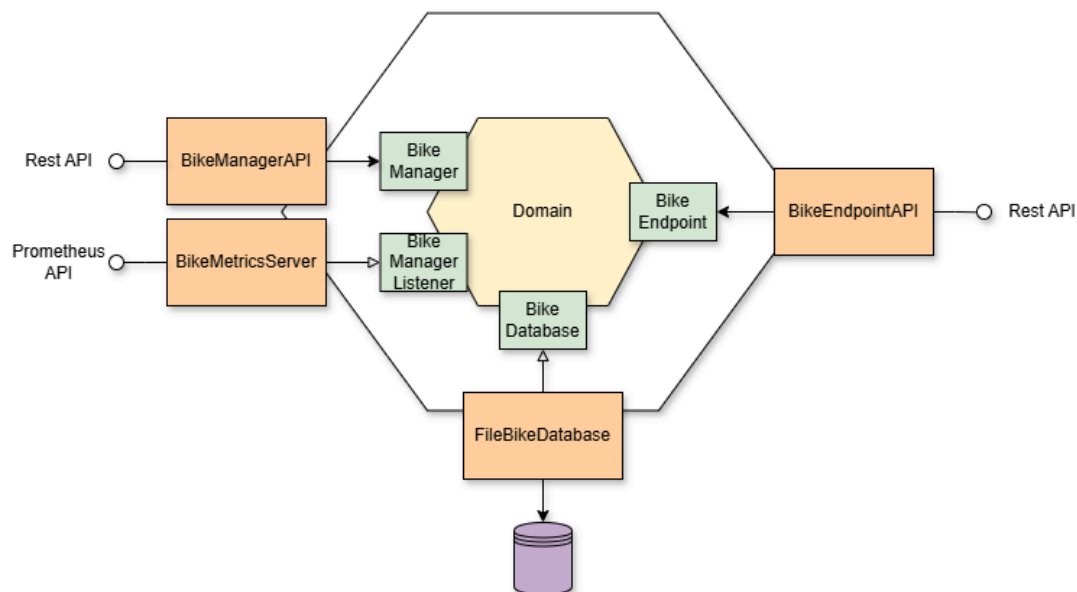


Fig. 4: Schema dell'architettura del servizio Bike Manager

Sono stati individuati 3 layer distinti, secondo l'architettura port-adapter:

- **Dominio**

Contiene l'entità *EBike*, che rappresenta la bicicletta elettrica con ID, livello di batteria e posizione.

- **Applicazione**

Contiene la logica del servizio *BikeManagerLogic*, e definisce outbound e inbound ports: *BikeManager* (per la gestione delle ebike), *BikeEndpoint* (per l'aggiornamento degli stati delle ebike), *BikeDatabase* (per la persistenza dei dati) e *BikeManagerListener* (per rendere il dominio osservabile).

- **Infrastruttura**

Contiene gli adapter che implementano le port del livello applicazione. Qui vengono definite le Rest API.

Implementazione

La port *BikeDatabase* è stata implementata dall'adapter *FileBikeDatabase*, che consente di memorizzare i dati sulle ebike in un file su disco rigido.

Per poter esporre delle API utilizzando HTTP si è utilizzato **Micronaut** come framework che consente di creare Rest API in maniera semplice e controllata. Gli adapter *BikeManagerAPI* e *BikeEndpointAPI* definiscono le API del servizio e utilizzano questo framework per realizzarle.

Nota: Queste API richiedono un token generato dal servizio di autenticazione per poter autorizzare un'operazione. Per semplicità, si è deciso di controllare in maniera semplice e poco sicura questo token, usando la classe *SimpleAuthChecker*. Cambiando l'implementazione dell' *AuthChecker*, è possibile migliorare la sicurezza del servizio.

La port *BikeManagerListener* è implementata dall'adapter *BikeMetricsServer*, che esegue misurazioni sul sistema utilizzando le informazioni notificate tramite il pattern observer. I risultati calcolati vengono poi esposti su una HTTP route “/metrics”, e sono in un formato compatibile con Prometheus, così da poter essere elaborati dal Metric Server.

5.2. Account Manager

Account Manager è il servizio predisposto a gestire gli utenti del sistema e i crediti a loro associati. Tramite questo servizio si possono aggiungere nuovi user o fare interrogazioni su quelli presenti. Permette inoltre di aumentare o diminuire il credito di ogni utente.

Ubiquitous Language

Per lavorare in questo contesto, è necessario definire alcuni termini per evitare ambiguità, presentati nella seguente tabella:

Termine	Definizione	Sinonimi
<i>User</i>	Entità che rappresenta un utente del sistema	<i>Account</i>
<i>Credit</i>	Il credito di un utente, ovvero la valuta adottata dal servizio. Può avere valori negativi	

Design

L'architettura è basata sul modello port-adapter (Fig. 5), ed espone delle Rest API per integrarsi con gli altri servizi del sistema:

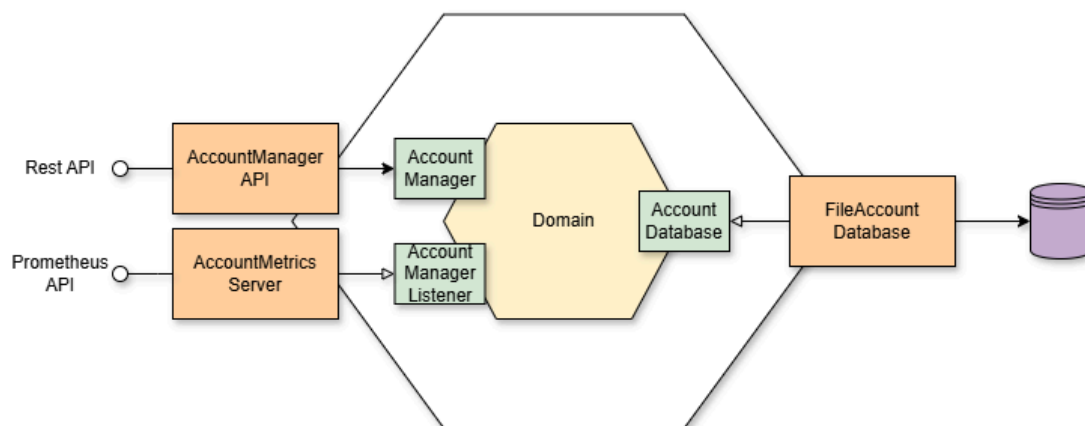


Fig. 5: Schema dell'architettura del servizio Account Manager

Sono stati individuati 3 layer distinti, secondo l'architettura port-adapter:

- **Dominio**

Contiene l'entità *User*, che rappresenta la bicicletta elettrica con username univoco e credito modificabile.

- **Applicazione**

Contiene la logica del servizio *AccountManagerLogic*, e definisce outbound e inbound ports: *AccountManager* (per la gestione degli account), *AccountDatabase* (per la persistenza dei dati) e *AccountManagerListener* (per rendere il dominio osservabile).

- **Infrastruttura**

Contiene gli adapter che implementano le port del livello applicazione. Qui vengono definite le Rest API.

Implementazione

La port *AccountDatabase* è stata implementata dall'adapter *FileAccountDatabase*, che consente di memorizzare i dati sugli utenti in un file su disco rigido (utilizza la stessa implementazione presente nel servizio Bike Manager).

Per poter esporre delle API utilizzando HTTP si è utilizzato **Micronaut** come framework. L'adapter *AccountManagerAPI* definisce le API del servizio e utilizzano questo framework per realizzarle.

Nota: Queste API richiedono un token generato dal servizio di autenticazione per poter autorizzare un'operazione. Per semplicità, si è deciso di controllare in maniera semplice e poco sicura questo token, usando la classe *SimpleAuthChecker*. Cambiando l'implementazione dell' *AuthChecker*, è possibile migliorare la sicurezza del servizio.

La port *AccountManagerListener* è implementata dall'adapter *AccountMetricsServer*, che esegue misurazioni sul sistema utilizzando le informazioni notificate tramite il pattern observer. I risultati calcolati vengono poi esposti su una HTTP route *"/metrics"*, e sono in un formato compatibile con Prometheus, così da poter essere elaborati dal Metric Server.

5.3. Ride Manager

Ride Manager è il servizio che gestisce tutti i noleggi effettuati dagli utenti. Si occupa di associare gli utenti alle bici che vogliono noleggiare, di disaccoppiarli quando invece terminano di utilizzarle. Inoltre, al termine del rental, ha il compito di detrarre il credito all'utente in base alla business policy adottata.

Questo servizio per poter funzionare necessita di un Bike Manager e di un Account Manager, i quali sono forniti tramite Rest API.

Ubiquitous Language

Per lavorare in questo contesto, è necessario definire alcuni termini per evitare ambiguità, presentati nella seguente tabella:

Termine	Definizione	Sinonimi
<i>Ride</i>	Entità che rappresenta un noleggio, composto da user e bike	<i>Rental</i>
<i>Credit</i>	Il credito di un utente, ovvero la valuta adottata dal servizio. Può avere valori negativi	
<i>Fee</i>	Il costo del servizio, calcolato in base alla durata del noleggio	<i>Price, Cost</i>
<i>Connection</i>	Azione che lo user esegue su una bici che vuole iniziare a utilizzare	
<i>Disconnection</i>	Azione che lo user esegue su una bici che vuole smettere di utilizzare	

Design

L'architettura è basata sul modello port-adapter (Fig. 6), ed espone delle Rest API per integrarsi con gli altri servizi del sistema:

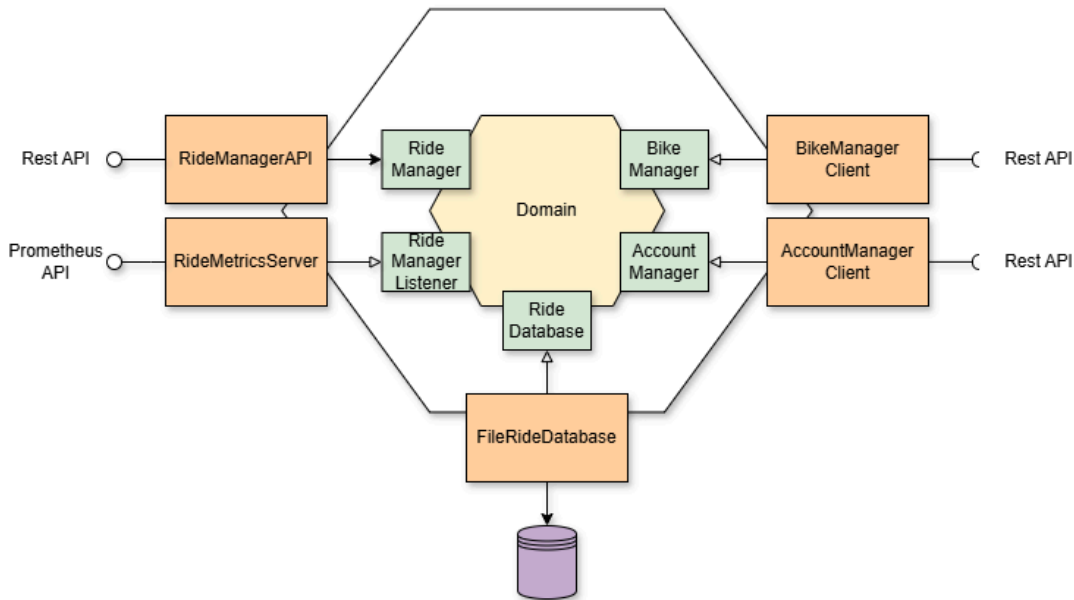


Fig. 6: Schema dell'architettura del servizio Ride Manager

Sono stati individuati 3 layer distinti, secondo l'architettura port-adapter:

- **Dominio**

Contiene l'entità *Ride*, che rappresenta il noleggio di una bicicletta elettrica che memorizza username dello user, id della bici e data di inizio noleggio.

- **Applicazione**

Contiene la logica del servizio *RideManagerLogic*, e definisce outbound e inbound ports: *RideManager* (per eseguire le connessioni/disconnessioni) *AccountManager* e *BikeManager* (per eseguire query su utenti e bici), *RideDatabase* (per la persistenza dei dati) e *RideManagerListener* (per rendere il dominio osservabile). Presenta inoltre *FeeCalculator* per il calcolo del costo del servizio.

- **Infrastruttura**

Contiene gli adapter che implementano le port del livello applicazione. Qui vengono definite le Rest API.

Implementazione

La port *RideDatabase* è stata implementata dall'adapter *FileRideDatabase*, che consente di memorizzare i dati sui noleggi in un file su disco rigido (utilizza la stessa implementazione presente nel servizio Bike Manager).

Per poter esporre delle API utilizzando HTTP si è utilizzato **Micronaut** come framework. L'adapter *RideManagerAPI* definisce le API del servizio e utilizzano questo framework per realizzarle.

Nota: Queste API richiedono un token generato dal servizio di autenticazione per poter autorizzare un'operazione. Per semplicità, si è deciso di controllare in maniera semplice e poco sicura questo token, usando la classe *SimpleAuthChecker*. Cambiando l'implementazione dell' *AuthChecker*, è possibile migliorare la sicurezza del servizio.

Le port *AccountManager* e *BikeManager* sono implementate rispettivamente da *AccountManagerClient* e *BikeManagerClient*. Questi non sono altro che dei client che eseguono chiamate HTTP verso i corrispettivi servizi.

La port *RideManagerListener* è implementata dall'adapter *RideMetricsServer*, che esegue misurazioni sul sistema utilizzando le informazioni notificate tramite il pattern observer. I risultati calcolati vengono poi esposti su una HTTP route “/metrics”, e sono in un formato compatibile con Prometheus, così da poter essere elaborati dal Metric Server.

5.4. Authentication Server

L'Authentication Server è il server adibito alla registrazione di nuovi utenti e al login di quelli già registrati nel sistema. Il suo compito è quello di generare dei token per la sessione degli utenti che eseguono un login. Comunica anche con il servizio Account Manager, per verificare che gli utenti effettivamente esistano nel sistema.

Ubiquitous Language

Per lavorare in questo contesto, è necessario definire alcuni termini per evitare ambiguità, presentati nella seguente tabella:

Termine	Definizione	Sinonimi
---------	-------------	----------

<i>User</i>	Entità che rappresenta un utente, con username e password
<i>Token</i>	Oggetto generato a partire dalle informazioni sull'utente, che autorizza l'utente stesso ad effettuare operazioni

Design

L'architettura è basata sul modello port-adapter (Fig. 7), ed espone delle Rest API per integrarsi con gli altri servizi del sistema:

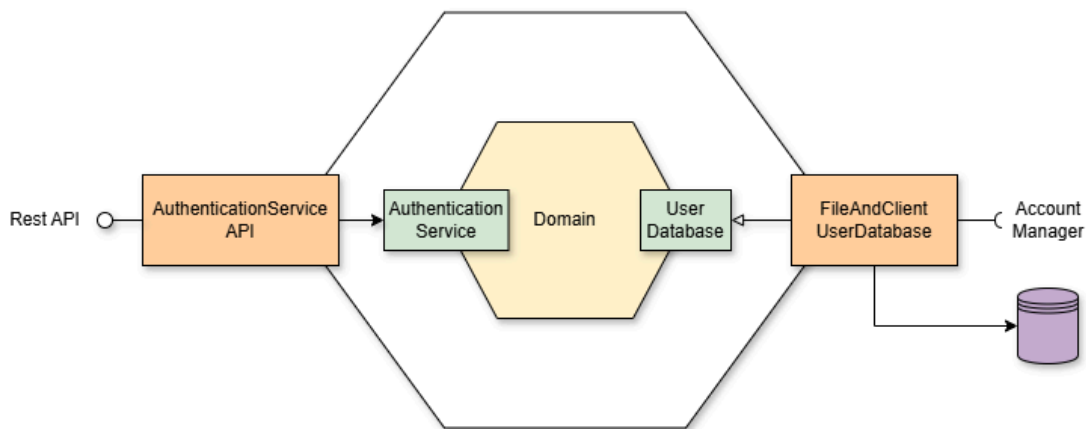


Fig. 7: Schema dell'architettura del servizio AuthenticationServer

Sono stati individuati 3 layer distinti, secondo l'architettura port-adapter:

- **Dominio**

Contiene l'entità *User*, che rappresenta l'utente.

- **Applicazione**

Contiene la logica del servizio *AuthenticationServiceLogic*, e definisce outbound e inbound ports: *AuthenticationService* (per la gestione dei login), *TokenGenerator* (per la generazione dei token), e *UserDatabase* (per la persistenza dei dati).

- **Infrastruttura**

Contiene gli adapter che implementano le port del livello applicazione. Qui vengono definite le Rest API.

Implementazione

La port *UserDatabase* è stata implementata dall'adapter *FileAndClientUserDatabase*, che consente di memorizzare i dati sugli user in un file su disco rigido, e inoltre comunica attraverso un client HTTP con il servizio Account Manager, così da sincronizzare i dati sugli utenti.

Per poter esporre delle API utilizzando HTTP si è utilizzato **Micronaut** come framework. L'adapter *AuthenticationServiceAPI* definisce le API del servizio e utilizza questo framework per realizzarle.

Nota: Queste API restituiscono il token che permette agli utenti di utilizzare il servizio. Per semplicità, si è deciso di implementare un semplicissimo token testuale, ovvero AUTHORIZED per gli user e ADMIN per l'amministratore. Il token può essere migliorato cambiando l'implementazione di *TokenGenerator*.

AuthenticationMetricsServer espone delle API per il metric server Prometheus.

5.5. API Gateway

L'API Gateway ha il compito di smistare le richieste del sistema ai servizi adibiti all'elaborazione di quella richiesta. Questo servizio risulta quindi stateless e sfrutta il DNS integrato di Docker per trovare i servizi presenti nella rete.

Il servizio è stato implementato utilizzando **Go** come linguaggio, così da avere prestazioni elevate in termini di elaborazione delle richieste e di deployment.

5.6. Metric Server

Il Metric Server ha il compito di ricevere le informazioni riguardanti lo stato fisico dei vari servizi, che espongono le proprie metriche, a fini di analisi. In questo caso, si è utilizzato come metric server **Prometheus**, che si basa su un modello pull-based e offre delle metriche predefinite già impostate. Permette inoltre di eseguire, oltre alle query, delle health check sui servizi in maniera completamente automatica.

6. Testing

Le qualità architetturali del sistema sono state confermate dai test eseguiti:

- **Reattività:** il sistema, in condizioni operative normali, presenta una latenza di risposta molto bassa, nell'ordine dei 200 ms.
- **Disponibilità:** il sistema risulta sempre operativo, anche in caso di eccezioni interne. Ad esempio, è stata configurata una route specifica */bikes/crash* per simulare un crash del servizio **Bike Manager**. In questi casi, il framework **Micronaut** gestisce le eccezioni in modo trasparente, garantendo la continuità del servizio.

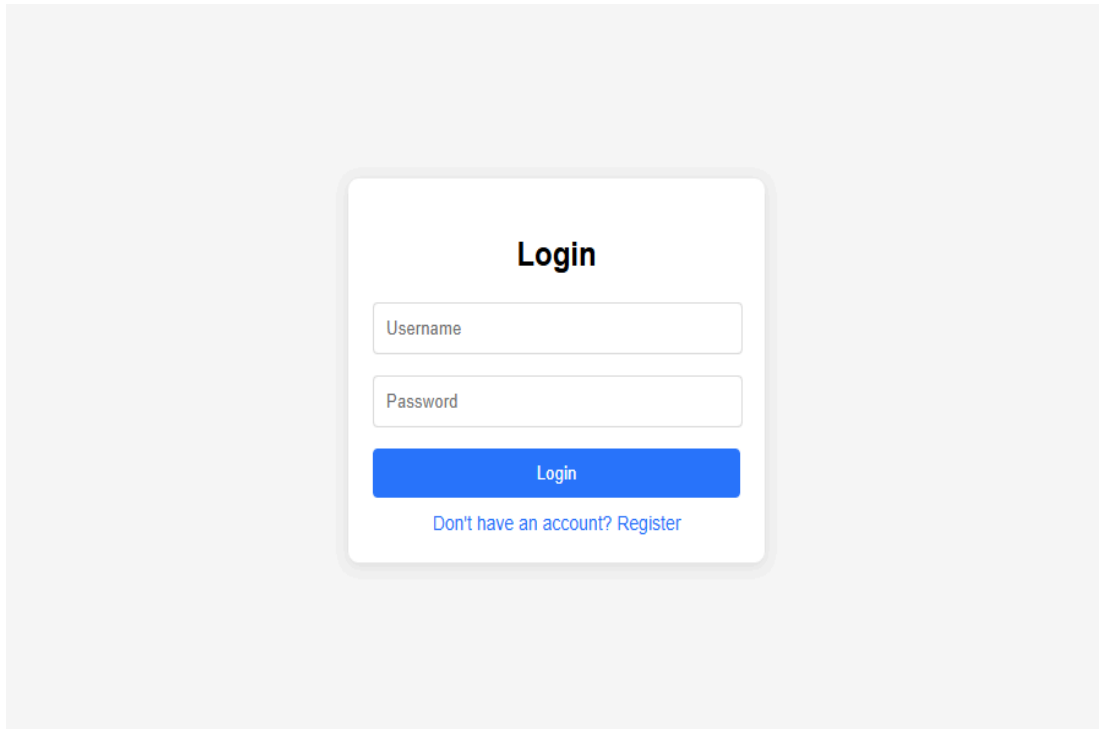
Inoltre, grazie all'uso di **Docker**, i container sono configurati con policy di *restart automatico* in caso di failure, assicurando ulteriore robustezza e resilienza del sistema.

Sono stati implementati dei test all'interno del servizio **Bike Manager** per verificare le funzionalità a ogni livello della piramide dei test. I test eseguiti sono i seguenti:

- **Unit testing:** eseguito nel file *FileBikeDatabaseTests*, per testare il componente di implementazione *BikeDatabase*.
- **Integration testing:** eseguito in *BikeManagerTests*, per verificare il componente *BikeManager*.
- **Component testing:** eseguito in *BikeManagerAPITests*, per testare le API REST della classe *BikeManagerAPI*.
- **End-to-end testing:** effettuato in un progetto separato chiamato *TestService*, esterno rispetto all'intero servizio. Questi test verificano il sistema nel suo complesso simulando l'esperienza di un utente reale.

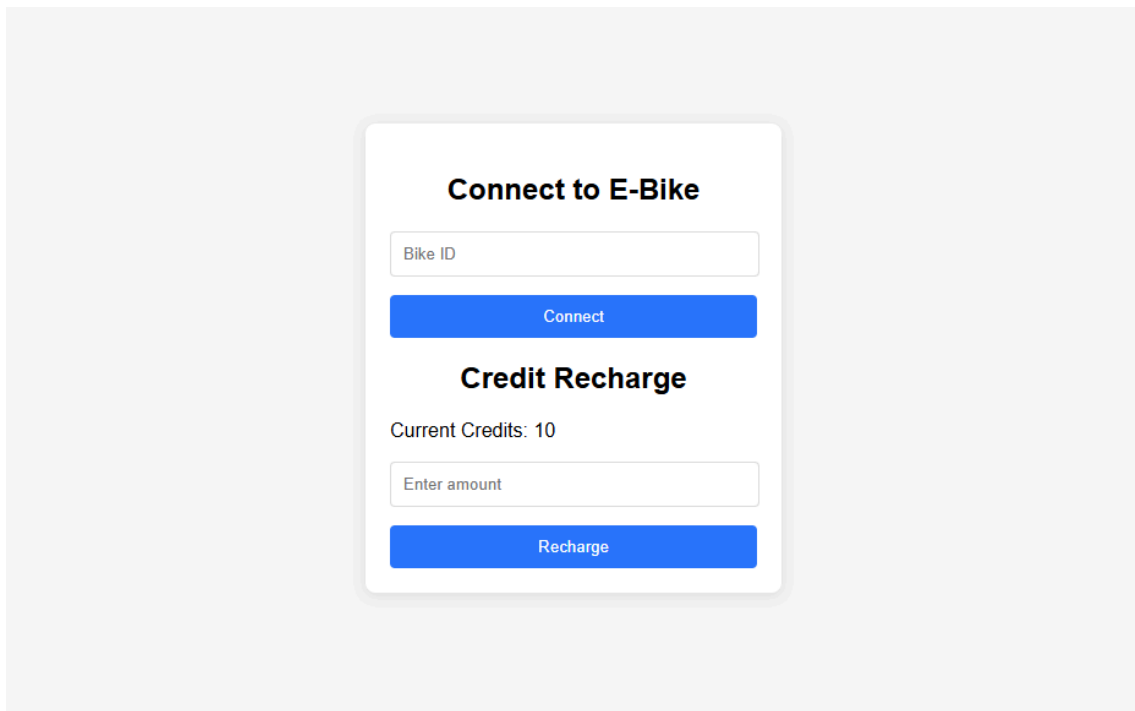
Per avviare tutti i microservizi, è fornito uno script chiamato *build-all.sh*, che aiuta a creare correttamente tutte le immagini Docker, e un file *docker-compose* per avviare tutti i container.

Per utilizzare il servizio di noleggio, il progetto include una semplice applicazione client, basata su JavaScript, chiamata *UserClient*. Questa applicazione consente agli utenti di accedere o registrarsi al servizio di noleggio, connettersi a una bicicletta elettrica o ricaricare i propri crediti. Di seguito sono riportati alcuni screenshot dell'applicazione.



The image shows a login screen with a white rounded rectangle centered on a light gray background. At the top of the rectangle is the title "Login" in bold black text. Below the title are two input fields: the first is labeled "Username" and the second is labeled "Password". Both labels are in a small, light gray font. Below the password field is a blue button with the text "Login" in white. At the bottom of the rectangle is a link that says "Don't have an account? Register" in a small, blue font.

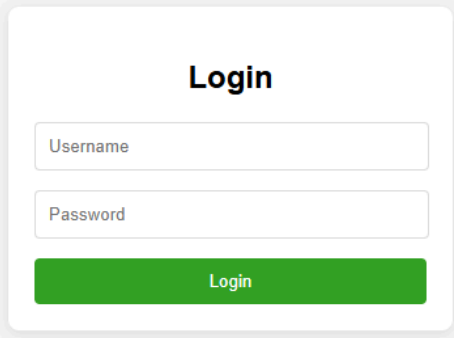
Fig. 8: Schermata d'accesso per il client dell'utente



The image shows a user account management screen with a white rounded rectangle centered on a light gray background. The screen is divided into two sections. The top section is titled "Connect to E-Bike" in bold black text. Below the title is an input field labeled "Bike ID" in a small, light gray font. Below the input field is a blue button with the text "Connect" in white. The bottom section is titled "Credit Recharge" in bold black text. Below the title is the text "Current Credits: 10" in a small, black font. Below this text is an input field labeled "Enter amount" in a small, light gray font. Below the input field is a blue button with the text "Recharge" in white.

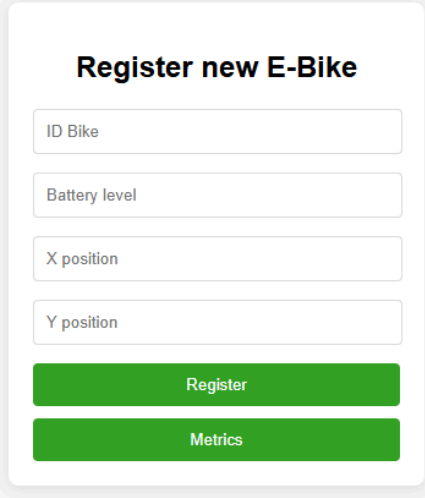
Fig. 9: Schermata della gestione dell'account dell'utente

È inoltre disponibile un client dedicato agli amministratori, che permette di aggiungere nuove biciclette elettriche al sistema e monitorare le metriche del server Prometheus. Anche per questa applicazione sono presentati degli screenshot.



The screenshot shows a login interface for administrators. It features a white card with rounded corners on a light gray background. The card has a title "Login" in bold black text. Below the title are two input fields: "Username" and "Password", both with placeholder text. At the bottom of the card is a green button labeled "Login".

Fig. 10: Schermata d'accesso dell'amministratore



The screenshot shows a form for registering a new e-bike. It features a white card with rounded corners on a light gray background. The card has a title "Register new E-Bike" in bold black text. Below the title are four input fields: "ID Bike", "Battery level", "X position", and "Y position", all with placeholder text. At the bottom of the card are two green buttons: "Register" and "Metrics".

Fig. 11: Schermata per l'aggiunta di ebike

Prometheus				
Query Alerts Status > Target health				
Select scrape pool Filter by target health Filter by endpoint or labels				
service-metrics 4 / 4 up				
Endpoint	Labels		Last scrape	State
http://account-manager:8080/metrics	instance="account-manager:8080"	job="service-metrics"	4.204s ago 8 4ms	UP
http://ride-manager:8080/metrics	instance="ride-manager:8080"	job="service-metrics"	3.115s ago 8 4ms	UP
http://auth-service:8080/metrics	instance="auth-service:8080"	job="service-metrics"	1.16ms ago 8 6ms	UP
http://bike-manager:8080/metrics	instance="bike-manager:8080"	job="service-metrics"	3.614s ago 8 4ms	UP

Fig. 12: Schermata per il controllo delle metriche