SAPIENZA
UNIVERSITÀ DI ROMA

Progettazione e implementazione di un sistema di raccolta dati per l'allenamento di un modello di Machine Learning

Facoltà di Ingegneria dell'informazione,informatica e statistica
Corso di Laurea in Informatica

Federico Raponi
Matricola 1963339

Relatore
Dott. Emanuele Panizzi

Anno Accademico 2023/2024

Progettazione e implementazione di un sistema di raccolta dati per l'allenamento di un modello di Machine Learning
Relazione di Tirocinio. Sapienza Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: fraponi98@gmail.com

Sommario

Questa relazione descrive la progettazione e l'implementazione di un sistema per memorizzare i dati provenienti da diversi sensori presenti su i dispositivi mobili, con lo scopo di raccogliere quante più informazioni possibili per creare un dataset consistente, utile per l'addestramento di un modello di Machine Learning. Il sistema è stato progettato e implementato seguendo le regole REST per garantire un'interazione efficiente tra il software e i dispositivi. Per assicurare la robustezza del sistema, sono stati creati appositi Unit Test, migliorando l'affidabilità e la qualità del software implementato.

Capitolo 1 Si introduce il problema del traffico e della difficoltà nel trovare parcheggio nelle città italiane, evidenziando come questi aspetti influenzino negativamente la qualità della vita urbana. Viene presentata "GeneroCity", un'applicazione mobile che consente agli utenti di condividere informazioni sui parcheggi disponibili attraverso un'interazione implicita, sfruttando sensori e tecnologie come Bluetooth Low Energy (BLE) e il Machine Learning.

Capitolo 2 Si illustra l'architettura del progetto, suddividendola in moduli backend e frontend. Si analizzano i protocolli utilizzati per il trasferimento dei dati, come HTTP e HTTPS, e si discute l'importanza delle REST API. Inoltre, vengono descritte le tecnologie principali dello *stack* di "GeneroCity", tra cui GoLang e MariaDB.

Capitolo 3 In questo capitolo l'attenzione è rivolta al testing del software, sottolineando l'importanza dei test unitari per garantire la qualità e la manutenzione del codice. Si esamina l'uso di strumenti come GO-SQLMock e la creazione di un sistema MockFS per facilitare i test.

Capitolo 4 Questo capitolo si concentra sull'implementazione di un meccanismo per la raccolta e la gestione dei dati provenienti dai sensori. Questo include la progettazione del database e la creazione di endpoint per l'interazione con i dati. Viene inoltre trattata l'ottimizzazione degli endpoint per migliorare le prestazioni durante l'invio dei dati.

Capitolo 5 In questo ultimo capitolo vengono descritte le contribuzioni minori al progetto, con particolare attenzione al sistema "GeneroCity BLE" e alla raccolta di dati relativi ai dispositivi Bluetooth nelle vicinanze dell'utente.

Indice

Capitolo 1

Introduzione

1.1 Traffico e Parcheggi

Secondo le analisi ISTAT[(1)], l'aumento continuo e costante del traffico nelle città italiane ha portato a un significativo incremento della congestione stradale. La crescita del numero di automobili e del traffico ha accentuato il problema della ricerca del parcheggio, con conseguenze negative sia per le persone che per l'ambiente, tra cui alti livelli di stress e inquinamento.

Non solo aumentano i tempi necessari per trovare un posto libero, ma anche i costi sostenuti dagli automobilisti. Questi costi possono derivare, ad esempio, dall'aumento del consumo di carburante dovuto alla ricerca di un parcheggio, o dall'acquisto di un box privato, il cui prezzo nella capitale può raggiungere fino a 170€ a settimana.

Almeno il 30% del traffico nelle aree del centro è costituito da automobili in cerca di parcheggio. È una quantità enorme, se si considera che la ricerca di un posto costituisce solo l'ultimo tratto di un viaggio. Le infrastrutture del centro rappresentano spesso un collo di bottiglia a causa della difficoltà di reperire parcheggio, scatenando un effetto a catena che si diffonde anche alle altre aree della città.

Nel 2016 i problemi maggiormente sentiti dalle famiglie relativamente alla zona in cui abitano sono stati l'inquinamento dell'aria (38,0%), il traffico (37,9%) e la difficoltà di parcheggio (37,2%).

1.1.1 Applicazione per lo Smart Parking: GeneroCity

GeneroCity è un progetto nato con l'obiettivo di facilitare gli utenti nella ricerca di parcheggi disponibili nelle loro vicinanze. Questa applicazione, ancora in fase di sviluppo, sarà disponibile sia per sistemi Android che iOS.

Figura 1.1 Logo di GeneroCity

L'idea alla base di GeneroCity è lo scambio di parcheggi tra utenti. L'applicazione sarà in grado di rilevare quando un utente sta lasciando il proprio parcheggio, registrando automaticamente nel sistema la disponibilità di quel posto per altri utenti.

1.1.2 Interazione implicita con l'applicazione

Un elemento cruciale dell'applicazione è la quasi totale assenza di interazione diretta da parte dell'utente. Questo approccio è adottato per garantire la sicurezza durante la guida e per migliorare l'esperienza dell'utente, evitando operazioni ripetitive. Per raggiungere questo livello di autonomia, l'applicazione dovrà sfruttare i sensori del dispositivo su cui è installata, analizzando i dati raccolti.

Ad esempio, si farà uso della tecnologia BLE *(Bluetooth Low Energy)*, che rispetto al Bluetooth classico offre un consumo energetico e un costo significativamente ridotti, mantenendo tuttavia un intervallo di comunicazione simile.

Sensori e Machine Learning

I sensori utilizzati da quest'applicazione sono:

- Sensore GPS
- Sensore WiFi
- Sensore Bluetooth
- Sensore di stato di carica della batteria

I dati rilevati dai suddetti sensori vengono elaborati da un modello di Machine Learning che determina se l'utente è attualmente alla guida e in cerca di un parcheggio. Qualora sia confermato, l'applicazione invia una notifica all'utente contenente la posizione del parcheggio più vicino alla sua ubicazione attuale. Analogamente, il modello riconosce quando un utente sta lasciando un parcheggio, aggiungendolo automaticamente alla lista dei parcheggi disponibili nel sistema e facilitando eventuali scambi con altri utenti in cerca di parcheggio.

1.1.3 Il mio contributo al progetto

Durante il mio tirocinio, ho focalizzato i miei sforzi sul miglioramento e sull'introduzione di nuove funzionalità nel backend dell'applicazione. Le attività svolte includono:

- Revisione e miglioramento della struttura REST dell'API esistente.
- Implementazione di UnitTest per gli endpoint privi di questa copertura.
- Introduzione di un sistema MockFS per facilitare i test, simulando un file system fittizio.

- Creazione di una nuova sezione nel backend dedicata alla raccolta dati provenienti dai vari sensori.

Queste attività hanno contribuito significativamente alla robustezza e alla funzionalità complessiva del sistema backend dell'applicazione.

Capitolo 2

GeneroCity

2.1 Architettura del progetto

GeneroCity è un'applicazione composta da un modulo backend e un modulo frontend che comunicano tra di loro utilizzando il protocollo HTTP per fornire il servizio all'utente.

2.1.1 Modulo Backend

Il backend è un software che ha come principale funzione quella di elaborare le richieste che il *frontend* effettua tramite l'API REST composta dai vari endpoint che il backend fornisce. Una volta ricevuta la richiesta da parte del client questo software implementa la logica con la quale i dati inviati dall'utente devono interagire, o essere salvati sul database. Questo software è perennemente in esecuzione su uno o più server pronto per ricevere e gestire le chiamate da parte degli utenti.

Il backend dell'applicazione inoltre gestisce i permessi dei vari utenti sull'accesso in lettura e scrittura dei dati.

2.1.2 Modulo Frontend

Il frontend invece è la componente grafica dell'applicazione, quella che effettivamente viene scaricate sui vari dispositivi dagli utenti. In questa parte del software c'è tutta la logica delle richieste che vengono effettuate all'API e la gestione grafica dei dati che vengono ritornati dalle relative risposte. Questa componente ha una code-base in *Swift* per iOS e una in *Java* per Android dato che, i due sistemi hanno dei requisti grafici e funzionali diversi.

Di fondamentale importanza è la *User Interface*, cioè il modo in cui è progettata e sviluppata l'interfaccia che comporta la conseguente *User Experience*, l'esperienza di uso dell'applicazione da parte dell'utente.

2.2 Protocolli per il trasferimento dati

Per il trasferimento dei dati tra app o browser e server si possono utilizzare il protocollo HTTP o HTTPS. Per entrambi il paradigma utilizzato è quello del *client-server* che permette di inoltrare richieste da parte del client verso il server che a sua volta risponderà con i dati richiesti.

2.2.1 HTTP (HyperText Transfer Protocol)

Questo protocollo utilizza la porta *80* per il trasferimento dei dati ma, non implementa nessun metodo di crittografia, infatti i dati che vengono inviati sono vulnerabili ad intercettazioni esterne.

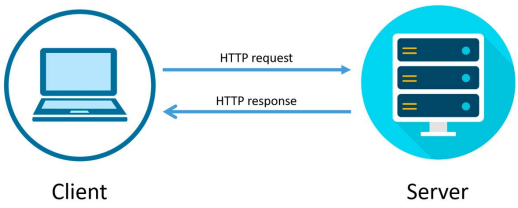

Figura 2.1 Rappresentazione grafica del funzionamento del protocollo HTTP

2.2.2 HTTPS (HyperText Transfer Protocol Secure)

Questo protocollo implementa tutte le funzionalità del precedente, aggiungendo però la crittografia di tipo SSL/TLS che protegge i dati durante il trasferimento. Il passaggio dei dati avviene sulla porta *443*.

L'utilizzo di questo nuovo standard è fortemente consigliato per tutti i servizi che gestiscono dati sensibili, come banche o qualsiasi servizio che richiede un accesso tramite credenziali.

Differenza tra PUT e POST

- **POST**: Utilizzato per inviare dati al server per l'elaborazione, creare nuove risorse senza specificare un URI (Uniform Resource Identifier) o eseguire operazioni non idempotenti.
  - Esempio: `POST /users/` - Crea un nuovo utente nella collection "users".
- **PUT**: Utilizzato per aggiornare o creare una risorsa specifica in un URI noto in modo idempotente.
  - Esempio: `PUT /users/{id}` - Aggiorna l'utente identificato dall'ID presente nell'URI.

2.2.3 Concetto di idempotenza

Nel contesto del protocollo HTTP, l'idempotenza si riferisce alla proprietà di determinati metodi HTTP per cui l'esecuzione ripetuta della stessa richiesta con identici dati produce lo stesso effetto sul server come se fosse stata eseguita una sola volta.

Ad esempio, l'invio ripetuto di richieste POST *(non idempotenti)* con gli stessi dati genererà molteplici risorse identiche nel database. Al contrario, l'invio ripetuto di richieste PUT *(idempotenti)* con gli stessi dati risulterà nella creazione di una singola risorsa nel database.

Idempotenza nei Database

Nel contesto delle migrazioni dei database, l'idempotenza si riferisce alla proprietà di uno script di migrazione di poter essere eseguito più volte senza causare effetti collaterali o cambiamenti indesiderati oltre alla prima esecuzione. In altre parole, l'applicazione ripetuta dello stesso script di migrazione deve produrre lo stesso stato del database come se fosse stato applicato una sola volta.

Per assicurarsi ad esempio che una tabella venga creata una sola volta, anche se lo script di creazione viene eseguito più volte si utilizzano, durante la dichiarazione della tabella in SQL, le parole chiave ***IF NOT EXIST*** che permettono la creazione della tabella solo nel caso non sia già presente.

Esempio

```
1 CREATE TABLE IF NOT EXISTS users (
2     id INT PRIMARY KEY,
3     name VARCHAR(100)
4 );
```

2.2.4 Risposte HTTP

Quando il client effettua una richiesta, il protocollo HTTP prevede una risposta corrispondente. Le risposte HTTP possono contenere i dati richiesti oppure non contenerne affatto.

Codici di Stato HTTP

Una componente fondamentale delle risposte HTTP è costituita dai codici di stato, i quali indicano l'esito della richiesta. Questi codici forniscono informazioni sul risultato dell'operazione richiesta, aiutando a identificare se la richiesta è stata completata con successo, se c'è stato un errore, o se sono necessarie ulteriori azioni.

Questi codici di stato possono essere classificati in base al numero, ad esempio tutti quelli della forma `2xx` sono messaggi su successo, mentre quelli della forma `4xx` sono messaggi di errore.

2.3 REST API

API è l'acronimo di Application Programming Interface ed è l'insieme delle regole e delle procedure che permettono la corretta comunicazione tra i vari client e il database.

2.3.1 L'importanza della filosofia REST nella progettazione di un'API

La filosofia REST (Representational State Transfer) è fondamentale durante la progettazione e la creazione di un'API per vari motivi, tra cui l'efficienza e scalabilità, oppure la facilità di manutenzione e integrazione. Per potersi definire REST un'API deve rispettare dei punti fondamentali.

Client-server L'architettura client-server implementa la separazione delle responsabilità, migliorando la portabilità e la scalabilità delle applicazioni. Questa separazione consente al backend di operare indipendentemente dalla presenza di un frontend funzionante. Inoltre, permette a molteplici client, sviluppati da terze parti, di utilizzare l'API nelle loro applicazioni.

Stateless Ogni richiesta inviata dal client deve contenere tutte le informazioni necessarie per essere compresa, senza fare affidamento su alcun contesto memorizzato sul server. Lo stato della sessione è completamente gestito dal client, mentre il server si occupa di mantenere lo stato delle risorse.

Cacheable Il client ha la possibilità di riutilizzare in futuro una risorsa che può essere memorizzata nella cache. Il periodo di tempo per il quale la risorsa può essere riutilizzata è specificato nella risposta fornita dal server.

Interfaccia Uniforme Il vincolo dell'interfaccia uniforme è fondamentale per la progettazione di qualsiasi sistema REST. Semplifica e disaccoppia l'architettura, consentendo a ogni parte di evolvere in modo indipendente, promuovendo la standardizzazione.

Sistema a Strati In un'architettura a strati, diversi componenti possono essere coinvolti nella comunicazione tra client e server. Questi componenti includono il server di origine, il gateway dell'elemento, il proxy dell'elemento e altri strumenti necessari per gestire e dirigere le richieste e le risposte attraverso la rete.

2.4 Lo Stack di GeneroCity

Lo *Stack* di un'applicazione è l'insieme delle tecnologie utilizzate al suo interno. In questo caso GeneroCity sfrutta diverse tecnologie che permettono la comunicazione tra le varie componenti dell'applicazione. Per la creazione del *Backend* sono state utilizzate diverse teconologie.

GoLang Go è un linguaggio realizzato da Google progettato per affrontare le esigenze di programmazione del software moderno, puntando nello specifico sulla semplicità d'utilizzo, l'efficienza e il supporto per la concorrenza. La robusta libreria standard e il modello di concorrenza basato su *goroutines*, che permettono l'esecuzione di codice concorrente e parallelo, lo rendono una scelta eccellente per una vasta gamma di applicazioni.

> At the time, no single team member knew Go, but within a month, everyone was writing in Go and we were building out the endpoints. It was the flexibility, how easy it was to use, and the really cool concept behind Go (how Go handles native concurrency, garbage collection, and of course safety+speed.) that helped engage us during the build. Also, who can beat that cute mascot! (3)

MariaDB MariaDB è un sistema di gestione di database relazionale open source. È progettato per essere compatibile con MySQL, dato che nasce come *fork* di quest'ultimo, offrendo le stesse funzionalità di base, ma con miglioramenti significativi in termini di performance.

OpenAPI OpenAPI è uno strumento di fondamentale importanza per la creazione di una documentazione di alta qualità e per la gestione delle API REST. L'utilizzo di questa risorsa consente di aumentare l'efficienza e la produttività durante lo sviluppo, definendo in modo chiaro e univoco il funzionamento del backend e, in particolare, dei vari endpoint. Ciò permette a chiunque necessiti di informazioni relative all'API di consultare agevolmente questo documento.
OpenAPI può utilizzare il formato JSON (JavaScript Object Notation) oppure, più comunemente, il formato YAML (YAML Ain't Markup Language).

Docker e Docker Compose Docker è una piattaforma per la creazione, distribuzione ed esecuzione di software all'interno di container, mentre Docker Compose è uno strumento per definire e gestire applicazioni multi-container. Insieme, questi strumenti migliorano l'efficienza, la portabilità e la coerenza nello sviluppo e nel deployment di applicazioni moderne.

Uno dei principali vantaggi dell'utilizzo della containerizzazione tramite Docker è la scalabilità e la solidità dei container, i quali, una volta configurati, possono essere eseguiti ovunque in modo sicuro e veloce.

2.5 Il template per la struttura del backend: Fantastic Coffee

Andando ad analizzare nello specifico la struttura del Backend di GeneroCity.

2.5.1 L'architettura del backend

- La directory `cmd/` contiene tutti i pacchetti eseguibili. Ogni directory viene compilata in una singola immagine docker nella pipeline CI/CD. È possibile

eseguire un singolo eseguibile usando `go run ./cmd/package/` nella riga di comando.

  - `webapi` è lo scheletro per l'eseguibile dell'API Web
  - ulteriori pacchetti possono essere creati per i *cron jobs*[1], l'interfaccia a riga di comando per l'amministrazione, ecc.

- La directory `db/` contiene:
  - un `schema.sql` che viene aggiornato utilizzando lo strumento `dbmate`, usato solo per il tracciamento delle modifiche dello schema
  - una directory `migrations` che contiene tutte le migrazioni del database
- La directory `service/` contiene tutto il codice per il servizio corrente, in pacchetti indipendenti (così possono essere riutilizzati o testati).
  - `api` contiene la vera e propria implementazione dell'API
  - `database` contiene il software dedicato all'interazione diretta con il database.
  - `filestorage` contiene la logica per il salvataggio di determinati dati sotto forma di file
  - `types` contiene la definizione di tutte le costanti e strutture dati comuni all'interno del progetto.
  - `utils` contiene funzioni che possono essere riutilizzate dalle varie componenti del progetto in modo tale da ridurre la ridondanza
- La directory `demo/` contiene tutti i file per gli ambienti di sviluppo (come i file docker-compose, dati fittizi del database, configurazioni di test di integrazione, ecc.)
- La directory `doc/` contiene la documentazione delle API

[1] Un cron job è un'attività pianificata per essere eseguita automaticamente su un sistema operativo Unix-like ad un orario o con una frequenza specifica.

Capitolo 3

Testing di un prodotto software

Nello sviluppo di un'applicazione, più nello specifico del modulo backend sono di fondamentale importanza gli *UnitTest*. L'*UnitTesting* è la pratica secondo il quale vengono testate le singole unità del codice. In questo caso specifico per ogni endpoint presente nel backend deve esserci il test associato.

I principali motivi per cui è importante implementare i test all'interno di un progetto sono:

Rilevazione degli errori Come si può banalmente intuire la funzione primaria dei test è la rilevazione degli errori all'interno delle funzioni che compongono il progetto. I test sono molto utili soprattutto per scovare dei *bug*[1] che si generano con degli eventi che sono considerati *edge case*[2].

Facilitazione del Refactoring Durante il processo di ristrutturazione di una determinata porzione di codice, è fondamentale evitare di alterare drasticamente il comportamento del codice in questione. In questo contesto, gli UnitTest rivestono un ruolo cruciale. Poiché sono stati già scritti per la funzione che si sta modificando, dopo aver eseguito il refactoring, l'esecuzione dei test consente di rilevare immediatamente eventuali cambiamenti di comportamento indesiderati dovuti alle modifiche effettuate.

Risparmio di tempo Nonostante sia richiesto del tempo per la corretta scrittura di un buon numero di test, nel lungo termine si avrà un risparmio di tempo e una facilitazione nella correzione di eventuali bug.

[1] Un bug in un programma è un errore nel codice che causa un malfunzionamento o un comportamento indesiderato del codice.

[2] Un edge case (o caso limite) in informatica è una situazione particolare che si verifica con condizioni non standard e inusuali.

3.1 Eliminazione della ridondanza

Grazie alla scrittura dei test è stata trovata una grande ridondanza di codice per l'invio di messaggi di errore da parte del backend. Nello specifico veniva ripetuta numerose volte questa porzione di codice:

```
if err != nil {
    var errorMessagetypes.ErrorMessage
    errorMessage.ErrorMessage = "lat param not well formed"
    ctx.Logger.WithError(err).Error(errorMessage)
    w.Header().Set("content-type", "application/json")
    w.WriteHeader(http.StatusBadRequest)j
    _ = json.NewEncoder(w).Encode(errorMessage)
}
```

Grazie all'introduzione della seguente funzione è stato possibile eliminare migliaia di righe di codice.

```
func HandleHTTPError(
    w http.ResponseWriter,
    ctx reqcontext.RequestContext,
    err error,
    message string,
    status int
) {
        var errorMessage types.ErrorMessage
        errorMessage.ErrorMessage = message
        if err != nil {
                ctx.Logger.WithError(err).Error(errorMessage)
        } else {
                ctx.Logger.Error(errorMessage)
        }
        w.Header().Set("content-type", "application/json")
        w.WriteHeader(status)
        _ = json.NewEncoder(w).Encode(errorMessage)
}
```

Questa funzione riceve in ingresso, oltre ai componenti per la riposta HTTP, due parametri che identificano il codice di errore generato e il messaggio di errore che si vuole ritornare.

3.1.1 Importanza del riuso del codice

Uno dei concetti fondamentali della programmazione è il riuso del codice, non solo per ridurre la dimensione complessiva del progetto in termini di righe di codice, ma soprattutto per migliorarne la manutenibilità.

Ad esempio, creare una funzione unica per la gestione degli errori consente di avere una struttura standard condivisa da tutti i componenti in modo tale da

mantenere uno standard coerente all'interno di tutto il progetto. In caso di errore, questa funzione può essere modificata in un solo punto, e la correzione avrà effetto uniformemente in ogni sua istanza. In questo modo è necessario testare la funzione e si ha la garanzia che essa funzioni correttamente in tutto il progetto.

3.2 Implementazione dei test mancanti

Una delle prime *issue*[3] che sono state risolte era quella relativa alla mancanza di test per determinati endpoint dell'applicazione.

3.2.1 Libreria per gli UnitTest: GO-SQLMock

Per effettuare gli UnitTest dell'API di GeneroCity è utilizzata una libreria esterna chiamata **go-sqlmock** (4).

Sqlmock è una libreria *mock*[4] che implementa sql/driver. Ha un unico scopo: simulare qualsiasi comportamento di un driver SQL nei test, senza la necessità di una connessione a un database reale.

Questa libreria è ora completa e stabile ed offre il supporto a:

- Concorrenza e connessioni multiple.
- Mocking delle funzionalità relative al Context di go1.8 e dei parametri sql nominati.
- Non richiede alcuna modifica al tuo codice sorgente.
- Il driver permette di simulare il comportamento di qualsiasi metodo del driver SQL.
- Non ha dipendenze di terze parti.

[3] Il sistema di issue è un meccanismo che permette agli sviluppatori che partecipano ad un progetto di sollevare un problema all'interno della repository, discuterne e andare poi a risolvere la problematica.

[4] Il termine "mock" nel contesto del software development si riferisce a un oggetto simulato che imita il comportamento di un componente reale, ma che è controllato in modo predefinito.

Esempio

```
func TestShouldUpdateStats(t *testing.T) {
        db, mock, err := sqlmock.New()
        if err != nil {
                t.Fatalf("an error '%s' was not expected when
                 ↪  opening a stub database connection", err)
        }
        defer db.Close()

        mock.ExpectBegin()
        mock.ExpectExec("UPDATE
         ↪  products").WillReturnResult(sqlmock.NewResult(1, 1))
        mock.ExpectExec("INSERT INTO product_viewers").WithArgs(2,
         ↪  3).WillReturnResult(sqlmock.NewResult(1, 1))
        mock.ExpectCommit()

        if err = recordStats(db, 2, 3); err != nil {
                t.Errorf("error was not expected while updating
                 ↪  stats: %s", err)
        }

        if err := mock.ExpectationsWereMet(); err != nil {
                t.Errorf("there were unfulfilled expectations: %s",
                 ↪  err)
        }
}
```

3.2.2 Creazione di MockFS, il sistema per simulare il FileSystem

Durante la creazione dei test mancanti è sorta la necessità di implementare un sistema per simulare il comportamento del *FileSystem*, in quanto alcuni degli endpoint necessitano di salvare dei dati sotto forma di file JSON.

Per realizzare queste funzioni abbiamo preso ispirazione proprio dalla libreria **go-sqlmock** citata in precedenza, andando a realizzare le seguenti funzioni.

PatchBleVehicleMode	Simula il comportamento di aggiornamento del tipo di veicolo in un trip.
ListBleTrips	Simula la restituzione della lista di tutti i trip.
GetBleTrip	Simula la restituzione di uno specifico trip.
SaveBleTrip	Simula il salvataggio di un trip.
ExtractTripJSON	Simula l'estrazione dei dati dei trip sotto forma di JSON.
ExtractDriverBehavior	Simula l'estrazione del comportamento del conducente basato sulla lista di manovre delle auto fornita.
GetCarManeuversRAM	Simula la restituzione delle manovre dell'auto in RAM.
GetCarManeuver	Simula la restituzione di una lettura/chiusura per una manovra dell'auto basata sull'ID della manovra dell'auto fornito.
SaveTripPoints	Simula il salvataggio dei punti del viaggio con l'ID e il tipo di punti del viaggio forniti.
GetModel	Simula la restituzione del modello in byte per la versione e il nome del modello specificati.
SaveModel	Simula il salvataggio del modello con il modello e il nome del modello forniti.
GetLastVersionModel	Simula la restituzione dell'ultima versione del modello per il nome del modello fornito.
SaveBluetoothConnections	Simula il salvataggio delle connessioni Bluetooth con la struttura di connessioni Bluetooth fornita.
UpdateBluetoothMode	Simula l'aggiornamento della modalità Bluetooth con il filename e la nuova modalità specificati.
ListBluetoothConnections	Simula la restituzione di una lista di connessioni Bluetooth.
SaveCarManeuvers	Simula il salvataggio delle manovre dell'auto con l'ID del parco e le manovre fornite.
SaveParkSamples	Simula il salvataggio dei campioni di parcheggio con l'ID del parco e i campioni di parcheggio forniti.
ListCarManeuvers	Simula la restituzione di una lista di manovre dell'auto.

Tabella 3.1 Metodi per il MockFS

Per mantenere una coerenza con la libreria sopra citata, è stato realizzato il seguente sistema per passare alla funzione gli argomenti, i valori di ritorno o l'eventuale errore da ritornare.

```
// MockOperation is a struct that represents a mock operation
type MockOperation struct {
        operation      string
        expectedInput  []interface{}
        expectedOutput []interface{}
        expectedError  error
}

// WithArgs sets the input arguments that the mock operation expects
func (o *MockOperation) WithArgs(args ...interface{}) *MockOperation {
        o.expectedInput = args
        return o
}

// WillReturnValues sets the output values for the mock operation
func (o *MockOperation) WillReturnValues(output ...interface{}) {
        o.expectedOutput = output
}

// WillReturnError sets the error for the mock operation
// should be nil if no error is expected
func (o *MockOperation) WillReturnError(err error) {
        o.expectedError = err
}
```

Sono stati inoltre aggiunti dei metodi standard per la gestione degli errori in modo da facilitare il debugging[5] durante la fase di test.

```
// mockFSError is a custom error type used to signal errors in the mockFS struct
type mockFSError struct {
        errorType         string
        expectedOperation string
        givenOperation    string
        expectedArguments int
        givenArguments    int
        expectedArgument  interface{}
        givenArgument     interface{}
}

```

[5] Il debugging è il processo di individuazione e risoluzione degli errori o difetti all'interno di un programma software.

```
12 // Error returns the error message
13 func (e *mockFSError) Error() string {
14         message := e.errorType + ": "
15         switch e.errorType {
16         case types.ErrorMockFSWrongOperation:
17                 message += fmt.Sprintf("%s was executed, %s were
                    ↪  expected.", e.givenOperation,
                    ↪  e.expectedOperation)
18         case types.ErrorMockFSWrongArgumentCount:
19                 message += fmt.Sprintf("function executed with %d
                    ↪  arguments, %d were expected.", e.givenArguments,
                    ↪  e.expectedArguments)
20         case types.ErrorMockFSWrongArgumentType:
21                 message += fmt.Sprintf("expected %T, got %T",
                    ↪  e.givenArgument, e.expectedArgument)
22         case types.ErrorMockFSWrongOutputType:
23                 message += fmt.Sprintf("expected %T, got %T",
                    ↪  e.givenArgument, e.expectedArgument)
24         case types.ErrorMockFSWrongArgumentValue:
25                 message += fmt.Sprintf("function executed with
                    ↪  argument %v, %v was expected.", e.givenArgument,
                    ↪  e.expectedArgument)
26         }
27         return message
28 }
29
30 // wrongOperationError returns a new mockFSError with the given
   ↪  operation and the expected operation
31 func wrongOperationError(given string, expected string) error {
32         return &mockFSError{
33                 errorType:         types.ErrorMockFSWrongOperation,
34                 givenOperation:    given,
35                 expectedOperation: expected,
36         }
37 }
```

Analizzando nello specifico le varie porzioni di codice sopra esposte:

MockOperation Questa *struct* è la rappresentazione di un'operazione che viene effettuata durante l'esecuzione di un test. I campi che la compongono rappresentano rispettivamente:

- **operation** - il nome dell'operazione da eseguire
- **expectedInput** - la lista di parametri in input che la funzione si aspetta
- **expectedOutput** - la lista di valori di output della funzione che ci si aspetta vengano prodotti

- **expectedError** - l'errore che eventualmente si vuole ritornare se si vuole testare un malfunzionamento del codice.

WithArgs Questo metodo accetta in input una lista di argomenti in input che vengono inseriti nel rispettivo campo della *MockOperation*. Questo metodo ritorna poi l'operazione in modo tale da potergli concatenare uno dei due metodi per l'output, *WillReturnValues* oppure *WillReturnError*.

WillReturnValues Questo metodo *terminale*[6] salva nel campo predisposto l'output che ci si aspetta dall'esecuzione del codice.

WillReturnError Questo metodo *terminale* salva nel campo predisposto l'errore che ci si aspetta dall'esecuziione del codice.

mockFSError Questa *struct* è stata realizzata per una migliore classificazione dei possibili errori e poterne gestire al meglio la rappresentazione. Sono quindi state realizzate varie funzioni che creano un nuovo *mockFSError* basato sui parametri passati in input.

wrongOperationError Questa funzione presa come esempio crea un nuovo errore di tipo *ErrorMockFSWrongOperation* con l'operazione e i valori di output passati come paramento.

Error Per concludere, questo metodo genera un stringa di errore che viene riportata all'interno dei test per facilitare lo sviluppo, basata sul tipo dell'errore.

3.2.3 Uso del MockFS nei test

Avendo creato il sistema di MockFS è stato possibile inserire all'interno dei test dei relativi endpoint che, durante l'elaborazione della richiesta necessitano l'accesso al filesystem per scrivere o leggere dei dati.

Per poter utilizzare il sistema creato, è necessario creare un istanza di Mockfs all'interno del test sulla quale verranno applicati i metodi sopra riportati.

```
1 mockfs, ok := router.fs.(*tests.Mockfs)
2 if !ok {
3     t.Fatal("error casting")
4 }
```

[6] In questo caso per metodo terminale si intende un metodo che non restituisce l'istanza su cui è stato applicato, in modo tale da non poter applicare altri metodi su di esso.

Ottenuta l'istanza si procede nel seguente modo:

```
mockfs.ExpectExec("ListBleTrips").WithArgs(context.TODO(),
↪  test.tripIDs).WillReturnValues(
    map[int]*types.Trip{
        1: &tripSample[0],
    },
)
```

In questo specifico caso il sistema si aspetterà di eseguire il metodo `ListBleTrips` ricevendo come argomenti il contesto della richiesta, che in questo caso è simulato tramite `context.TODO()` in quanto durante l'esecuzione degli UnitTest non vengono effettuate delle vere richieste al Server.

Infine ci si aspetta che dall'esecuzione di questo metodo venga ritornato un oggetto con chiave 1 e come valore un `tripSample`, la rappresentazione in JSON di un *trip* effettuato.

Trip Sample

```
var tripSample = types.Trip{
{
    AppInfo: types.AppInfo{
        Version:      "1.0.0",
        Platform:     "android",
        BuildVersion: 1,
    },
    Vehicle: "car",
    TripPoints: []types.TripPoint{
        {
        Lat:       12.9455173,
        Lon:       41.5392761,
        Speed:     0,
        Timestamp: createTimestampForTrip(),
        Devices: []types.BleDevices{
            {
                DeviceUUID:
                ↪  uuid.FromStringOrNil("41ffef41-ab44-3d74-96e5-855b7336480f"),
                RSSI:       -104,
            },
        },
    },
    },
    Comment:  "",
    AvgSpeed: 0,
    }
}
```

Capitolo 4

Implementazione di un meccanismo di raccolta dati.

In questo capitolo verrà trattato il compito principale svolto durante il mio tirocinio, ossia la progettazione e l'implementazione di un sistema per la raccolta e la gestione dei dati provenienti dai sensori integrati nei dispositivi Android e iOS, sviluppati da altri membri del team per il software frontend.

4.1 Uso dei dati raccolti

Il processo di raccolta dei dati è finalizzato principalmente all'acquisizione di un ampio spettro di informazioni riguardanti le azioni, le abitudini e i comportamenti degli utenti, ottenute tramite l'impiego di vari sensori. Queste informazioni sono fondamentali per la generazione di statistiche che permettono una comprensione approfondita dei diversi comportamenti degli utenti e per il monitoraggio di potenziali anomalie. Tuttavia, l'obiettivo primario è la costruzione di un dataset[1] fondamentale per l'addestramento di un modello di machine learning.

Il modello, una volta addestrato e integrato nell'applicazione finale, ha l'obiettivo principale di offrire un'esperienza d'uso altamente precisa, analizzando in modo efficiente i dati raccolti dai sensori del dispositivo su cui l'applicazione è eseguita. Inoltre, l'algoritmo è progettato per fornire un'esperienza personalizzata per ciascun utente, suggerendo opzioni specifiche basate sull'analisi dei dati relativi al comportamento individuale. Questo approccio permette all'applicazione di adattarsi dinamicamente alle esigenze e preferenze degli utenti, migliorando in maniera significativa la loro esperienza complessiva.

4.2 Struttura dei dati raccolti

Una delle principali problematiche nella gestione dei dati consiste nel garantire un'uniformità che consenta di creare una struttura solida e coerente. Questa uniformità

[1] Un **dataset** è una raccolta organizzata di dati, generalmente strutturati in modo tale da permettere analisi, manipolazioni e interpretazioni accurate.

è essenziale per assicurare che i dati possano essere immagazzinati in modo efficiente, riducendo la possibilità di ridondanze e inconsistenze. Inoltre, una struttura uniforme facilita notevolmente l'interazione con i dati stessi, permettendo operazioni di analisi, manipolazione e aggiornamento in modo rapido ed efficace. Raggiungere questo equilibrio tra efficienza nell'immagazzinamento e semplicità nelle operazioni è cruciale per il buon funzionamento dell'intero sistema, poiché influisce direttamente sulla capacità di ricavare informazioni rilevanti dai dati raccolti.

4.2.1 Progettazione della struttura del database

Durante la fase di progettazione di questa nuova funzionalità all'interno del software è sorto il problema della variabilità di forma dei dati inviati dai vari sensori.

Struttura personalizzata per ogni sensore

In prima battuta si pensava di realizzare una struttura personalizzata e dedicata ad ogni sensore implementato, ma in questo modo si sarebbe ottenuta una struttura complessiva poco scalabile e sarebbe stato costoso e complesso operare con i dati di più sensori contemporaneamente.

Struttura unica e flessibile

Per affrontare le problematiche sopra esposte, si è scelto di progettare una struttura flessibile che consenta a ogni sensore di inviare i dati nel formato più adatto alle proprie specifiche esigenze. Tuttavia, per garantire un'efficace archiviazione delle informazioni, sono state previste alcune componenti obbligatorie che tutti i sensori devono rispettare. Questo approccio consente di mantenere l'efficienza nella raccolta e gestione dei dati, pur offrendo la necessaria adattabilità alle diverse caratteristiche dei sensori, assicurando così un equilibrio tra flessibilità operativa e coerenza strutturale.

Durante una fase preliminare di analisi dei dati e della loro possibile forma sono state individuate delle componenti fondamentali in comune tra tutti i vari sensori implementati fino a questo momento. Le componenti individuate sono:

userID

Questo data rappresenta l'utente che ha inviato al server un determinato dato, utile per tenere traccia di chi fornisce determinati dati ed eventualmente trovare dei pattern relativi all'utilizzatore.

key

Questo non è strettamente un dato rilevato dal sensore, ma è un campo dedicato ad una migliore indicizzazione. Può essere utilizzato ad esempio per specificare la versione del sensore che ha raccolto quei dati.

datetime

Di fondamentale importanza il *datetime* o anche detto *timestamp*[2] che in questo caso rappresenta il momento esatto in cui il dato che viene comunicato al server è stato raccolto dal sensore.

confidence

Questo dato, raccolto sotto forma di valore numerico decimale, rappresenta il grado di confidenza con la quale il sensore ha determinato l'azione che l'utente stava compiendo in quel determinato momento. Questo valore va da 0, che rappresenta il valore minimo di confidenza, a 1 che rappresenta il valore massimo di confidenza.

action

Questo dato indica l'azione identificata dal sensore del dispositivo come eseguita dall'utente. Ad esempio, un'azione rilevata dal sensore potrebbe essere *walking*, qualora l'utente stesse camminando in quel momento.

sensor

Un'altra informazione fondamentale da tracciare è il sensore che ha effettuato la rilevazione del dato specifico.

Per evitare incoerenze nelle informazioni dovute all'utilizzo di nomi diversi con lo stesso significato semantico, è stato deciso di non lasciare ai dispositivi la libertà di scegliere arbitrariamente il nome del sensore. Ad esempio, la presenza di dati nel database registrati come rilevati dal sensore "*wifi*" e altri dal sensore "*wi-fi*" potrebbe generare problemi di inconsistenza. Per affrontare questa problematica, è stata progettata una seconda tabella, denominata *sensors*, nella quale saranno salvati tutti i nomi dei sensori registrati e autorizzati all'invio dei dati. Questo approccio riduce al minimo il rischio di incoerenze, creando una dipendenza tra le due tabelle. Il nome del sensore all'interno della tabella *sensors* è stato definito come univoco per evitare inutili duplicati.

data

Questo campo è dedicato ai veri e propri dati che vengono rilevati e inviati dal sensore al server. Questo campo deve necessariamente essere inviato al fine di poter lavorare correttamente i dati una volta immagazzinati ma, non potendone conoscere a priori la struttura si è optato per rendere questo campo di tipo JSON.[3] Grazie a questa funzionalità dei *MariaDB* è possibile quindi definire un campo di una tabella

[2]Il termine timestamp si riferisce a un indicatore temporale che rappresenta il momento esatto in cui un evento si verifica o un'operazione viene eseguita. In informatica, un timestamp è generalmente espresso in termini di data e ora, e viene utilizzato per registrare il tempo preciso di operazioni, transazioni, modifiche ai dati, o per sincronizzare attività in sistemi distribuiti.

[3]JSON (JavaScript Object Notation) è un formato di scambio dati leggero e facilmente leggibile, utilizzato per rappresentare e trasmettere strutture di dati tra un server e un client o tra diverse applicazioni.

come stringa di formato JSON, il che consente di validarne in maniera automatica la struttura senza però conoscerne la forma a priori,

Formattazione corretta di una stringa JSON Nel caso in cui un dispositivo invii una stringa JSON con la seguente formattazione:

```
{
    "name": "Federico Raponi",
    "gender": "Male",
    "age": 23,
    "phoneNumber": [
        { "type": "personal", "number": "3274015148" }
    ]
}
```

il sistema riconoscerebbe la corretta formattazione senza però creare problemi sull'effettivo contenuto di questa stringa.

Formattazione sbagliata di una stringa JSON Nel caso in cui un dispositivo invii una stringa JSON con la seguente formattazione:

```
{
    "name": "Federico Raponi",
    "gender": "Male",
    "age: 23
    "phoneNumber": [
        { "type": "personal", "number": "3274015148" }

}
```

non vedrebbe accettato dal sistema in quanto non rispetterebbe i vincoli definiti dal tipo del campo della tabella.

4.3 Creazione delle nuove tabelle nel database

Per implementare le due nuove tabelle necessarie all'interno della struttura del database, sono stati creati dei file di migrazione che seguono il principio di *idempotenza* precedentemente discusso.

Questi file di migrazione definiscono le operazioni necessarie per la creazione e la configurazione delle tabelle, inclusa la definizione delle colonne, dei tipi di dati e dei vincoli. Inoltre, includono meccanismi per gestire eventuali modifiche future, come l'aggiunta di colonne o l'aggiornamento di tipi di dati, senza compromettere l'integrità del database esistente.

4.3.1 Creazione della tabella Sensors

Questa tabella è estremamente semplice, infatti è composta da un solo campo di tipo *VARCHAR* identificato come *name* a cui è stato attribuito il ruolo di *PRIMARY KEY*[4] della tabella.

```
-- migrate:up
CREATE TABLE IF NOT EXISTS sensors
(
        name VARCHAR(255) NOT NULL PRIMARY KEY
);

-- migrate:down
DROP TABLE IF EXISTS sensors;
```

4.3.2 Creazione della tabella Sensor Data

Per il vero e propri immagazzinamento dei dati, è stata realizzata la tabella *sensor_data* tramite la seguente migrazione

```
-- migrate:up
CREATE TABLE IF NOT EXISTS `sensor_data`
(
  `userid`     VARCHAR(36)  NOT NULL REFERENCES `users` (`id`) ON
  ↪  DELETE CASCADE,
  `key`        VARCHAR(255) NOT NULL,
  `data`       JSON         NOT NULL,
  `datetime`   DATETIME     NOT NULL,
  `sensor`     VARCHAR(255) NOT NULL,
  `action`     VARCHAR(255) NOT NULL,
  `confidence` DOUBLE       NOT NULL,
  PRIMARY KEY (`sensor`, `datetime`, `key`, `userid`)
);
-- migrate:down
DROP TABLE IF EXISTS `sensor_data`
```

Con la creazione di queste due nuove tabelle la struttura del database è pronta per immagazzinare tutti i dati raccolti dai sensori.

4.4 Struttura dei nuovi endpoint

Dopo aver progettato e implementato la nuova struttura del database, sono stati introdotti nuovi endpoint per consentire ai vari client di interagire con i dati all'interno delle nuove tabelle. Questi endpoint sono stati sviluppati per garantire un'interazione efficiente e sicura con le risorse del database, permettendo operazioni

[4]Una *PRIMARY KEY* è un vincolo in un database relazionale che identifica univocamente ogni record (o riga) all'interno di una tabella.

di lettura, scrittura, aggiornamento e cancellazione sui dati in modo sicuro e scalabile. La progettazione degli endpoint ha seguito le rispettato le linee guida per la creazione di una REST API, assicurando che ogni operazione fosse intuitiva e rispondesse adeguatamente alle richieste dei client.

Per facilitare un'interazione ottimale con queste nuove strutture e per garantire una solida integrazione con il team di sviluppo del frontend, è stata data priorità alla creazione di una documentazione esaustiva e ben strutturata. Tale documentazione è fondamentale per assicurare che gli sviluppatori possano comprendere chiaramente le funzionalità degli endpoint e utilizzarli correttamente nei loro processi di sviluppo.

4.4.1 Documentazione dei nuovi endpoint

La documentazione dei nuovi endpoint è stata integrata, estendendo il file già esistente all'interno del progetto, denominato `api.yaml`. Questo file è stato aggiornato per includere tutte le informazioni necessarie alla comprensione e all'utilizzo efficiente degli endpoint, come:

- **Descrizione dettagliata di ogni endpoint**: Per ogni endpoint sono stati specificati i vari metodi (*GET, PUT, POST, DELETE*) supportati.
- **Codici di stato HTTP**: Elenco dei possibili codici di stato restituiti da ciascun endpoint, con spiegazioni dettagliate su cosa ciascun codice rappresenti.
- **Parametri opzionali e non**: sono stati specificati inoltre tutti i parametri necessari all'interazione con questi endpoint, come il token di autenticazione[5] o altri *path parameter*[6] da inserire nel path della richiesta.

Questa documentazione non solo fornisce una guida completa per l'uso degli endpoint, ma funge anche da riferimento centrale per tutti gli sviluppatori coinvolti nel progetto, garantendo che tutti operino sulla base delle stesse informazioni e seguano linee guida coerenti.

4.4.2 I nuovi endpoint

Per l'interazione sono da parte dell'utente sono stati realizzati due endpoint di creazione e modifica, rispettivamente utilizzando il metodo `PUT` e `PATCH`, un endpoint per la cancellazione dei dati inseriti tramite metodo `DELETE` e infine uno per richiedere i dati dal server tramite richieste di tipo `GET`.

[5] Un **token di autenticazione** è una stringa univoca utilizzata per verificare l'identità di un utente o di un sistema durante l'accesso a un servizio o un'applicazione. Quando un utente effettua l'accesso, il sistema genera un token e lo invia all'utente, che lo include in tutte le successive richieste al server. Questo token consente al server di riconoscere l'utente senza dover richiedere nuovamente le credenziali per ogni richiesta. In questo modo, si migliora sia la sicurezza sia l'efficienza della comunicazione tra client e server.

[6] I **path parameter** (parametri di percorso) sono segmenti variabili all'interno di un URL che vengono utilizzati per identificare risorse specifiche su un server. Vengono definiti all'interno del percorso (path) di un endpoint API e consentono di passare informazioni direttamente nell'URL.

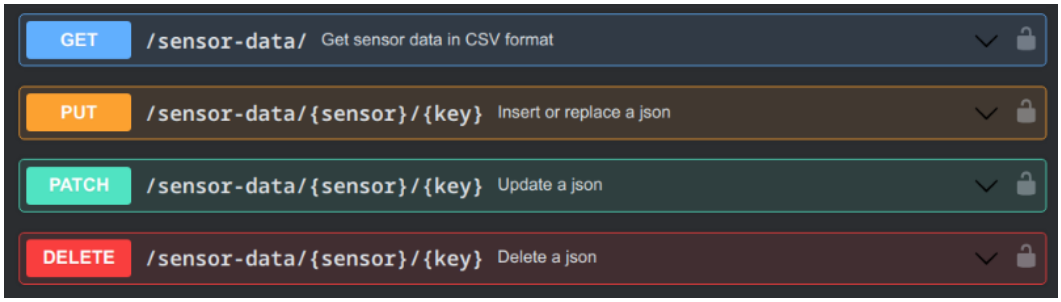

Figura 4.1 Nuovi endpoint per l'interazione con la tabella *sensor_data*

Inserimento dei dati tramite il metodo PUT

Per l'inserimento dei dati all'interno del database è stato scelto di utilizzare il metodo `PUT` in quanto l'inserimento di un dato, per valori uguali nei campi che compongono la chiave, deve andare a creare una nuova risorsa nel caso non ci sia un riscontro, deve invece sovrascriverla nel caso sia già presente. Per questo motivo è stato scelto di utilizzare il metodo `PUT`, in quanto idempotente, rispetto al metodo `POST`.

Modifica dei dati tramite il metodo PATCH

Per la modifica dei dati all'interno della tabella è consentito effettuare una richiesta di tipo `PATCH`, questo tipo di richiesta, a differenza delle richieste di tipo `PUT`, consentono di inviare all'interno del corpo della richiesta soltanto i campi che si è intenzionati a modificare, senza dover inviare l'intera risorsa.

Questo è utile sopratutto quando si vuole modificare qualche campo di una risorsa, alleggerendo il peso del corpo della richiesta e dei dati che devono essere elaborati dal server.

Eliminazione dei dati tramite metodo DELETE

Per l'eliminazione di una risorsa dal database è stato predisposto questo endpoint, che per identificare il *record* da eliminare utilizzerà i campi *sensor* e *key* ottenuti come *path parameter*, il campo *datetime* ottenuto tramite *query parameter*[7], infine il campo *userID* si ottiene tramite lo *uid* presente negli *Headers* della richiesta.

```
1  func (rt *_router) sensorDataDelete(w http.ResponseWriter, r
   ↪  *http.Request, ps httprouter.Params, ctx
   ↪  reqcontext.RequestContext) {
2          datetime := r.URL.Query().Get("datetime")
3          _, err := time.Parse(time.RFC3339, datetime)
4          if err != nil && datetime != "" {
5                  utils.HandleHTTPError(w, ctx, err, "datetime must be
                   ↪  in rfc3339 format", http.StatusBadRequest)
6                  return
7          }
8
9          sensor := ps.ByName("sensor")
10         exist, err := rt.db.CheckSensor(sensor)
11         if err != nil {
12                 utils.HandleHTTPError(w, ctx, err, "can't check
                   ↪  sensor", http.StatusInternalServerError)
13                 return
14         }
15         if !exist {
16                 utils.HandleHTTPError(w, ctx, nil, "sensor not
                   ↪  found", http.StatusBadRequest)
17                 return
18         }
19
20         ok, err := rt.db.DeleteSensorData(ctx.UserID,
           ↪  ps.ByName("key"), datetime, sensor)
21         if err != nil {
22                 utils.HandleHTTPError(w, ctx, err, "can't delete app
                   ↪  data", http.StatusInternalServerError)
23                 return
24         }
25         if !ok {
26                 utils.HandleHTTPError(w, ctx, nil, "app data not
                   ↪  found", http.StatusNotFound)
27                 return
28         }
29         w.WriteHeader(http.StatusNoContent)
30 }
```

[7] I query parameter sono una parte dell'URL utilizzata per passare dati a una pagina web. Vengono aggiunti alla fine di un URL dopo il simbolo ? e consistono in coppie chiave-valore separate dal simbolo =. Se ci sono più parametri, vengono separati da &.

Richiesta dei dati tramite metodo GET

Per interrogare il server e richiedere dati tramite questo endpoint, oltre a fornire un *uid* per identificare i dati di un determinato utente, è necessario inviare anche un *Log-Token*. Questo token, riservato esclusivamente agli amministratori, consente di accedere a tutti i dati raccolti, inclusi quelli relativi ad altri utenti.

Come da richiesta, per facilitare poi l'utilizzo dei dati ottenuti tramite questo endpoint, il formato scelto con il quale questi vengono ritornati al client è il CSV.

Il formato CSV (Comma-Separated Values) offre diverse potenzialità, rendendolo uno strumento molto versatile per la gestione dei dati:

- **Semplicità**: Il CSV è un formato di testo semplice, leggibile sia dalle persone che dalle macchine.
- **Compatibilità**: È supportato da un'ampia gamma di software, inclusi fogli di calcolo (come Excel), database, e strumenti di analisi dei dati.
- **Portabilità**: Essendo un formato di testo, i file CSV sono molto leggeri e possono essere facilmente trasferiti.
- **Facilità di elaborazione**: È facile scrivere script o utilizzare strumenti per analizzare, manipolare e filtrare i dati contenuti in un file CSV.

4.5 Ottimizzazione dell'endpoint per la richiesta dati

Una volta realizzate le sopra citate funzioni, si è effettuato un *refactor*[8] dell'endpoint dedicato all'invio dei dati dal server verso i client richiedente. Le modifiche apportate hanno riguardato sia la struttura del sistema sia il processo di elaborazione dei dati da parte del server e la modalità con cui vengono restituiti.

4.5.1 Modifica della struttura del database

La modifica alla struttura del database è stata necessaria per favorire e semplificare l'introduzione del sistema di paginazione. Questo è stato possibile grazie alla semplice introduzione di un indice auto-incrementale con la funzione di chiave primaria all'interno della tabella. La modifica ha ovviamente mantenuto il vincolo di univocità definito in precedenza basato su i seguenti campi:

- sensor
- datetime
- key
- userID

[8] Il refactor (o refactoring) è il processo di miglioramento del codice di un programma senza alterarne il comportamento esterno.

Oltre al vincolo è stato aggiunto anche un indice, chiamato *datetime_index*, basato sul campo *datetime*, che serve a favorire ed ottimizzare le operazione di filtraggio dei dati utilizzando quel campo della tabella.

Come da procedura, per apportare queste modifiche, è stata creata una migrazione che rispettasse le regole di idempotenza.

```
1  -- migrate:up
2  ALTER TABLE sensor_data
3          ADD COLUMN IF NOT EXISTS id INT AUTO_INCREMENT,
4          DROP PRIMARY KEY,
5          ADD PRIMARY KEY (id),
6          ADD INDEX IF NOT EXISTS datetime_index (datetime),
7          ADD CONSTRAINT sensor_data_sensor_datetime_key_userid_unique
           ↪  UNIQUE IF NOT EXISTS (sensor, datetime, `key`, userid);
8
9  -- migrate:down
10 ALTER TABLE sensor_data
11         DROP CONSTRAINT IF EXISTS
           ↪  sensor_data_sensor_datetime_key_userid_unique,
12         DROP PRIMARY KEY,
13         DROP COLUMN IF EXISTS id,
14         ADD PRIMARY KEY (`sensor`, `datetime`, `key`, `userid`),
15         DROP INDEX IF EXISTS datetime_index;
```

4.5.2 Implementazione della paginazione per l'invio dei dati

Una volta modificata la struttura del database è stata modificata la funzione per ottenere i dati in formato CSV.

L'obiettivo di questa ottimizzazione è recuperare un certo numero di record tramite una query al database, spostare il cursore al record successivo e inviare i dati ottenuti al client. Dopo l'invio dei primi dati, la stessa connessione HTTP viene utilizzata per trasmettere i dati successivi, ottenuti tramite una nuova query. Questo processo si ripete finché ci sono record disponibili nel database.

Connessione HTTP persistente

Per far si che questo meccanismo funzioni è stata sfruttata una caratteristica delle connessioni HTTP, infatti, fintanto che la connessione è aperta, è possibile continuare a inviare e ricevere dati:

1. **Streaming di Dati**: In applicazioni che richiedono l'invio continuo di dati, come lo streaming di video o l'invio di grandi quantità di dati, la connessione persistente è fondamentale. Il client e il server possono mantenere aperta la connessione e continuare a scambiarsi dati finché è necessario. In questo specifico caso l'invio dei dati paginati può essere visto come uno stream.

2. **Chiusura della Connessione**: La connessione rimane aperta fino a quando il client o il server decidono di chiuderla. Il server può decidere di chiudere la connessione dopo un certo periodo di inattività (timeout), o il client può chiuderla quando ha finito di ricevere i dati.

Modifiche alla funzione di query

Nella funzione che esegue la query, è stato introdotto un ordinamento basato sull'ID auto-incrementale, un filtro per selezionare solo i record con un ID maggiore del cursore, e l'aggiornamento del cursore all'ultimo ID del record recuperato con quella query.

```
func (db *appdbimpl) GetCsvSensorData(sensor string, key string,
↪  startDate string, endDate string, userid *uuid.UUID, limit int,
↪  cursor int) (*[]types.SensorQueryEntry, int, error) {
        data := make([]types.SensorQueryEntry, 0)
        values := make([]interface{}, 0)

        query := "SELECT * FROM sensor_data WHERE id > ?"
        values = append(values, cursor)

        // Resto del codice invariato

        query += " ORDER BY id LIMIT ?"
        values = append(values, limit)

        err := db.c.Select(&data, query, values...)
        if err != nil {
                return nil, cursor, err
        }

        if len(data) > 0 {
                cursor = data[len(data)-1].ID
        }

        return &data, cursor, nil
}
```

Modifiche alla funzione principale

La funzione principale, responsabile della verifica della validità della richiesta e del controllo della presenza del *Log-Token*, è stata modificata per mantenere aperta la connessione con il client una volta stabilita. Durante ogni iterazione della query, utilizzando la funzione descritta in precedenza, invia i dati paginati al client e chiude la connessione solo quando non ci sono più dati disponibili che soddisfano i parametri e i filtri della richiesta.

```
1  func (rt *_router) sensorDataGetCSV(w http.ResponseWriter, r
   ↪  *http.Request, _ httprouter.Params, ctx
   ↪  reqcontext.RequestContext) {
2          token := r.Header.Get("Log-Token")
3          var uid *uuid.UUID
4          switch token {
5          case "":
6                  uid = &ctx.UserID
7          case rt.logToken:
8                  uid = nil
9          default:
10                 utils.HandleHTTPError(w, ctx, nil, "unauthorized",
                    ↪  http.StatusUnauthorized)
11                 return
12         }
13
14         // Codice per la validazione della richiesta
15
16         sensor := r.URL.Query().Get("sensor")
17         startDate := r.URL.Query().Get("startDate")
18         endDate := r.URL.Query().Get("endDate")
19
20         cursor := 0
21         limit := types.CSVBufferSize
22
23         w.Header().Set("Content-Type", "text/csv")
24         w.Header().Set("Content-Disposition",
           ↪  "attachment;filename=SensorData.csv")
25         _, _ =
           ↪  w.Write([]byte("DATETIME,SENSOR,KEY,USERID,CONFIDENCE,ACTION,DATA\n"))
26
27         flag := true
28         var data *[]types.SensorQueryEntry
29         var err error
30
31         for flag {
32                 data, cursor, err = rt.db.GetCsvSensorData(sensor,
                    ↪  key, startDate, endDate, uid, limit, cursor)
33
34                 if err != nil {
35                         utils.HandleHTTPError(w, ctx, err, "error
                           ↪  getting sensor data",
                           ↪  http.StatusInternalServerError)
36                         return
37                 }
38
39                 var csvData string
```

```
40                  for _, sd := range *data {
41                          csvData = sd.Datetime.Format(time.RFC3339) +
                            ↪  "," + sd.Sensor + "," + sd.Key + "," +
                            ↪  sd.UserID.String() + "," +
                            ↪  fmt.Sprintf("%f", *sd.Confidence) + "," +
                            ↪  *sd.Action + ",\"" + fromJSONToCSV(w, sd,
                            ↪  ctx) + "\"\n"
42                          _, _ = w.Write([]byte(csvData))
43                  }
44
45                  if len(*data) < limit {
46                          flag = false
47                  }
48          }
49  }
```

Da notare come, all'interno di questa funzione venga chiamato il metodo *fromJSONToCSV*, che serve per convertire il campo *data* che viene ritornato dalla query sotto forma di JSON, in una stringa compatibile con il formato CSV.

```
1   func fromJSONToCSV(w http.ResponseWriter, sd types.SensorQueryEntry,
    ↪  ctx reqcontext.RequestContext) string {
2           jsonData, err := json.Marshal(sd.Data)
3           if err != nil {
4                   utils.HandleHTTPError(w, ctx, err, "error
                    ↪  marshalling sensor data",
                    ↪  http.StatusInternalServerError)
5                   return ""
6           }
7           jsonString := strings.ReplaceAll(string(jsonData), "\\",
            ↪  "\"")
8           jsonString = jsonString[1 : len(jsonString)-1]
9           return jsonString
10  }
```

Grazie a queste migliorie apportate al codice, la gestione della sicurezza è migliorata, soprattutto nel caso in cui il volume di dati da inviare diventi molto elevato. Questo approccio evita il sovraccarico del server, consentendo di inviare la stessa quantità di dati in modo frazionato, riducendo così il carico di ogni singola richiesta.

Capitolo 5

Contribuzioni minori al progetto

5.1 L'app per la raccolta dati: GeneroCity BLE

Durante il periodo di tirocinio ho partecipato anche alla raccolta dati relativi al sistema di GeneroCity BLE. Questo progetto ha lo scopo di recuperare dati tramite la tecnologia del Bluetooth Low Energy.

5.1.1 Scopo di GeneroCity BLE

L'obiettivo di questa applicazione separata è quella di iniziare a raccogliere dei dati relativi al numero di dispositivi bluetooth che si trovano nei dintorni dell'utente che sta raccogliendo i dati.

Quest'applicazione infatti si basa sulla creazione di *trip*, dei veri e propri tragitti effettuati dall'utente nei quali, ad ogni intervallo di tempo deciso in fase progettuale, rileva il numero di dispositivi bluetooth nelle vicinanze, associando alla rilevazione eseguita le coordinate.

Una volta terminato un trip si ottiene un resoconto della quantità di dispositivi scansionati, il tutto rappresentato come un'insieme di punti, colorati in modo diverso in base alla quantità di dispositivi bluetooth nei paraggi.

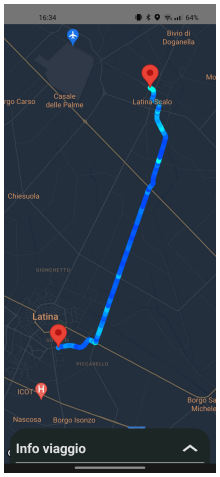

Figura 5.1 Rappresentazione grafica di un trip fornita dall'applicazione

Ad ogni trip, nel caso dell'applicazione per Android, l'utente deve specificare manualmente il mezzo che ha utilizzato per lo spostamento. I mezzi disponibili sono: Automobile, Bus, Tram, Metro e a piedi. Grazie alla raccolta dati effettuata, si possono effettuare delle analisi sulle informazioni raccolte, in modo tale da realizzare ad esempio la seguente *heatmap*[1].

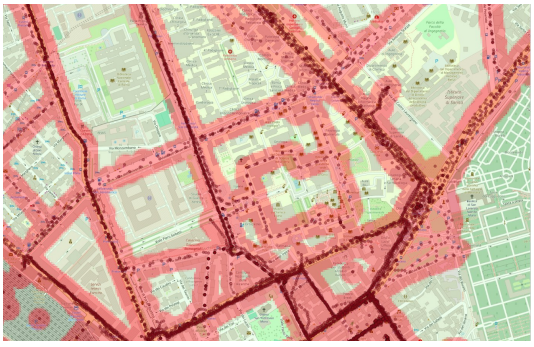

Figura 5.2 Heatmap della quantità di dispositivi bluetooth scansionati

[1]Una heatmap è una rappresentazione grafica dei dati dove i singoli valori contenuti in una matrice sono rappresentati da colori.

5.1.2 Il mio contributo

Il mio contributo al progetto non si è limitato alla semplice raccolta dei dati, cui ho contribuito personalmente eseguendo un centinaio di viaggi. Un aspetto significativo del mio lavoro è stato l'individuazione di un bug critico che impediva il corretto flusso dei dati tra client e server, compromettendo di fatto la raccolta delle informazioni.

Figura 5.3 Bug riscontrato durante la raccolta dati tramite GeneroCity BLE

Il problema risiedeva nella gestione errata da parte dell'applicazione di un determinato *status code* restituito dal server. Questo codice veniva inviato quando il dispositivo non riusciva a trasmettere i dati entro un certo intervallo di tempo, a causa di una connessione assente o estremamente lenta. In tali situazioni, l'applicazione avrebbe dovuto ripetere automaticamente il tentativo di invio una volta ristabilita la connessione. Tuttavia, ciò non avveniva. L'applicazione restava bloccata in uno stato intermedio, incapace sia di inoltrare nuovi dati raccolti, sia di trasmettere quelli già accumulati in coda. La risoluzione di questo bug ha quindi contribuito al miglioramento della solidità dell'applicazione e garantendo un processo di raccolta dati più fluido.

Capitolo 6

Conclusioni

Lo scopo del progetto di tirocinio è stato lo sviluppo e progettazione di una soluzione software creativa per risolvere il problema della crescente congestione del traffico, nonché l'innegabile difficoltà nel trovare posti auto in tutte le città italiane. La soluzione "GeneroCity" ha mostrato il contributo che tecnologie emergenti come BLE, apprendimento automatico tramite machine learning e REST API faranno un giorno al futuro con soluzioni reali ed accessibili per gli utenti della città.

Nella fase di sviluppo, diversi problemi tecnici erano coinvolti come fare la progettazione della struttura del backend, attuare gli endpoint REST e creare un meccanismo per salvare e gestire i dati. Tali dati, che sono recuperati tramite sensori incorporati come GPS e Bluetooth, devono rendere il godimento dell'applicazione implicito, offrendo un'esperienza nella ricerca e condivisione di parcheggi in modo efficiente e con un'alta sicurezza alla guida, poiché i conducenti non saranno distratti dall'utilizzo dell'applicazione.

Il testing dell'implementazione del software con l'aiuto di strumenti come GO-SQLMock e MockFS aveva lo scopo di garantire l'integrità del sistema e di aiutare nella manutenzione del codice. Il caricamento dei dati tramite l'impaginazione è stato un'altra funzione che ha contribuito a far funzionare l'applicazione con un'esperienza senza interruzioni per l'utente.

Questo lavoro quindi punta a migliorare il mantenimento della qualità della vita urbana attraverso un'applicazione pratica per facilitare la ricerca di parcheggi nella vita di tutti i giorni.

Bibliografia

[1] SMART PARKING: Il parcheggio: da problema a risorsa
https://smartparkingsystems.com/il-parcheggio-da-problema-a-risorsa/

[2] EUROSTAT: Passenger cars per 1000 inhabitants reached 560 in 2022
https://ec.europa.eu/eurostat/web/products-eurostat-news/w/ddn-20240117-1

[3] GoLang
https://go.dev/

[4] GO-SQLMock documentation
https://github.com/DATA-DOG/go-sqlmock