SAPIENZA
UNIVERSITÀ DI ROMA

# Sviluppo e ingegnerizzazione delle API sull'applicazione GeneroCity


Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea in Informatica

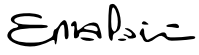
Candidato

Alessio Lucciola

Matricola 1823638

Relatore

Emanuele Panizzi

Anno Accademico 2020/2021

**Sviluppo e ingegnerizzazione delle API sull'applicazione GeneroCity**
Tesi di Laurea. Sapienza – Università di Roma



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

Email dell'autore: lucciola.1823638@studenti.uniroma1.it

# Sommario

Il seguente elaborato presenta il mio percorso di tirocinio svolto presso il *Gamification Lab*, un laboratorio del Dipartimento di Informatica dell'università di Roma La Sapienza che si occupa di progettare, sviluppare e sperimentare applicazioni digitali, in particolare per dispositivi mobili.
Durante questo percorso ho partecipato alla progettazione e lo sviluppo di una applicazione di *smart parking* chiamata **GeneroCity**, sotto la direzione del prof. Emanuele Panizzi. Per *smart parking* (dall'inglese "parcheggio intelligente") si intende l'insieme di strategie atte a velocizzare e facilitare la ricerca di un parcheggio per il proprio mezzo di trasporto. Questi obiettivi si possono perseguire mediante l'utilizzo della tecnologia che può aiutare a ottimizzare i tempi, gli spazi e le risorse che normalmente si consumano nella ricerca di un parcheggio [1].
In questi mesi mi sono principalmente occupato della creazione o la modifica di API con lo scopo di introdurre nuove funzionalità, e di gestire e ristrutturare alcune risorse lato *backend* come il database e il server di archiviazione cloud MinIO [18].
In primo luogo verranno esposte le tecnologie utilizzate e alcune conoscenze preliminari necessarie alla comprensione degli argomenti che seguono. In secondo luogo verrà illustrato in dettaglio il lavoro svolto durante questo percorso, a partire dalla progettazione delle nuove funzionalità, fino ad arrivare alla loro implementazione nel progetto.

# Indice

# Introduzione

I dati raccolti dall'Istituto nazionale di statistica (ISTAT) ci forniscono una fotografia del parco circolante del nostro paese. Nel 2021 risulta che il parco veicoli in Italia era composto da poco meno di 53 milioni di veicoli dei quali, circa 40 milioni, erano autovetture. In Europa, l'Italia si trova al primo posto per numero di auto in rapporto alla popolazione, si stima infatti che nel nostro paese circolino mediamente 655 auto ogni 1000 abitanti e oltre il 64% della popolazione le utilizza abitualmente, anche per piccoli spostamenti [3]. Inoltre, l'attuale pandemia di Covid-19 non ha fatto altro che inasprire ulteriormente questo dato infatti, a causa del maggior rischio di contagio legato alla permanenza sui mezzi pubblici, i cittadini italiani hanno ridotto il tasso di utilizzo del trasporto pubblico, preferendo mezzi individuali [4]. Il continuo e costante aumento di traffico all'interno delle città italiane ha portato a un incremento dei fenomeni di congestione stradale. Con l'aumento delle auto e del traffico si verifica sempre di più il fenomeno della "corsa al parcheggio" che ha degli effetti negativi sia su di noi che sull'ambiente, si pensi allo stress e all'inquinamento. Almeno il 30% del traffico urbano è formato da automobilisti in cerca di parcheggio ed è un dato piuttosto alto se si pensa che la ricerca di un posto costituisce solo l'ultima tratta di un viaggio. Si stima che gli automobilisti impieghino ben 2549 ore di vita alla ricerca di un parcheggio, tempo che viene tolto a momenti quali lavoro, famiglia o svago [5]. Bisogna inoltre considerare i costi che vengono sostenuti dal cittadino e che riguardano ad esempio il costo del carburante che viene consumato alla ricerca di un posto auto. All'estero la situazione non migliora: basti pensare a Londra che ha introdotto dei dazi (*congestion change*) per incoraggiare i cittadini a non utilizzare dei mezzi individuali nel centro città e, nonostante la formula abbia parzialmente funzionato, la media per trovare parcheggio rimane di 67 ore annue [6], oppure a New York in cui in media si arrivano a toccare le 117 ore annue [7].

Tasso di motorizzazione (numero di autovetture ogni 1000 abitanti) nei cinque maggiori Paesi europei nel 2021

| Paese | Auto circolanti ogni 1000 abitanti |
|---|---|
| Italia | 655 |
| Germania | 575 |
| Francia | 570 |
| Spagna | 533 |
| Regno Unito | 528 |

**Figura 0.1.** Fonte: ACEA (Driving Mobility for Europe) [2]

Una soluzione per mitigare questi problemi è quella di ricorrere a soluzioni tecnologiche che permettano di rendere i movimenti all'interno dei centri urbani più efficienti e meno inquinanti, incluso il momento del parcheggio. Sono varie le applicazioni che negli ultimi anni stanno cercando di migliorare l'esperienza generale degli automobilisti: si pensi a Google Maps [8], un servizio di visualizzazione di carte geografiche che consente agli utenti di venire a conoscenza di informazioni relative al proprio tragitto coma la presenza di congestioni, lavori stradali e condizioni meteo

sul percorso; oppure alle applicazioni di *car sharing* come Enjoy [9] che permettono di noleggiare auto di proprietà di terzi per un breve periodo di tempo dopo il quale il mezzo va riportato in uno dei parcheggi riservati sparsi in giro per la città.

## Il progetto GeneroCity

L'applicazione **GeneroCity**, attualmente in sviluppo da parte di alcuni studenti e ricercatori del Dipartimento di Informatica della Sapienza Università di Roma, si inserisce in questo panorama di proposte per migliorare l'esperienza alla guida, concentrandosi sul problema legato alla ricerca di un parcheggio, in particolare, l'applicazione ha come obiettivo quello di aiutare gli automobilisti a ridurre i tempi di ricerca di un parcheggio. L'utilizzo di GeneroCity richiede un semplice smartphone dotato di rete satellitare (GPS) e accelerometro, senza alcun hardware aggiuntivo.

**Figura 0.2.** Logo GeneroCity

L'idea di GeneroCity è quella di permettere lo scambio di parcheggi tra utenti dell'applicazione: un utente chiamato **giver** segnala la disponibilità a lasciare il suo posto auto in una certa posizione e in un certo orario del giorno. Un altro utente chiamato **taker** può segnalare al giver la volontà di prendere il suo posto effettuando un **match** e quindi prenotando il suo posto auto. Il taker arriva quindi dal giver e prende il parcheggio di quest'ultimo, evitando di sprecare tempo e carburante. Per ridurre l'uso del telefono alla guida, le operazioni di "lascia" e "prendi" posto si possono svolgere anche in maniera automatizzata grazie a un modello di *machine learning* presente nell'applicazione che riesce a capire quand'è che l'utente parcheggia e riprende l'auto. Sono state pensate anche delle attività di *gamification*: per ottenere un posto auto gli utenti devo spendere delle monete virtuali, chiamate *GCoins* che si possono ottenere solamente lasciando il proprio posto. Questo permette di aumentare il coinvolgimento e l'interattività tra utenti.
Una volta terminata la fase di sviluppo, l'applicazione verrà rilasciata sui sistemi Android e IOS e sarà scaricabile dai rispettivi *AppStore*.

Il lavoro svolto durante questi mesi di tirocinio si può dividere in due parti che hanno riguardato lo sviluppo e l'ingegnerizzazione delle API di GeneroCity.
In primo luogo mi sono occupato dell'introduzione di nuove funzionalità relative ai match quali l'associazione tra match e parcheggi e la creazione di informazioni derivabili da tale associazione, come la possibilità di capire se un parcheggio è stato preso o lasciato grazie a un match.
In secondo luogo ho svolto ulteriori lavori che hanno riguardato la modifica della logica di varie API esistenti, soprattutto per quanto riguarda la ricezione dei dati in input e come questi vengono salvati in memoria. In alcuni casi, queste modifiche hanno condotto a una profonda ristrutturazione del database con la migrazione di alcuni dati sul server di archiviazione cloud MinIO [18].

La relazione è strutturata come segue:

- Nel Capitolo 1 vengono esposti gli strumenti che sono stati utilizzati ai fini dello sviluppo delle funzionalità sopra elencate.
- Nel Capitolo 2 è presente una breve spiegazione teorica necessaria alla comprensione dei capitoli che seguono. Viene introdotto il funzionamento di un'API e

come queste ricevono una richiesta e inviano una risposta al client sfruttando il protocollo HTTP. Inoltre, è presente un riassunto della struttura lato server di GeneroCity sulla quale ho avuto modo di lavorare.

- Il Capitolo 3 tratta la progettazione e lo sviluppo delle nuove funzionalità precedentemente introdotte e relative ai match.
- Nel Capitolo 4 viene esposto il consistente lavoro di sistemazione del database culminato con la migrazione di alcuni dati al server MinIO mediante vari *script* creati appositamente.
- Nel Capitolo 5 vengono brevemente introdotti altri lavori svolti durante il tirocinio come la modifica al sistema di notifiche sulla posizione del taker.


**Figura 0.3.** Homepage di GeneroCity IOS allo sviluppo attuale

# Capitolo 1

# Tecnologie utilizzate

Durante questo percorso ho avuto modo di approcciarmi ad alcune note tecnologie per lo sviluppo di applicazioni lato backend, partendo da linguaggi di programmazione fino ad arrivare a software per il *testing* delle API. Qui di seguito si espone una breve descrizione delle varie tecnologie utilizzate e i motivi che hanno portato alla loro scelta:

- **Go**: Si tratta di un linguaggio di programmazione *open source*[1] sviluppato da Google. È un linguaggio compilabile la cui sintassi è molto simile a quella del linguaggio C con notevoli influenze provenienti da altri linguaggi come Pascal, Modula e Oberon. Uno dei principali motivi che hanno portato a utilizzare Go in GeneroCity, è l'elevata velocità di conversione che permette di compilare ed eseguire programmi in tempi brevi e generalmente inferiori rispetto a linguaggi della medesima tipologia. Questo lo rende il candidato perfetto in ambienti server dove performance e stabilità giocano un ruolo importante. Inoltre Go si comporta molto bene nella virtualizzazione con i container, un espediente utilizzato nel progetto per testare il funzionamento dell'applicazione lato server. Ulteriori aspetti positivi di Golang sono la funzionalità di *garbage collection*[2], la concorrenza che permette un'esecuzione più rapida di un programma e la presenza di notevoli librerie esterne per la gestione del protocollo HTTP e il funzionamento delle API [10].

- **Docker**: Si tratta di una piattaforma che permette di creare e testare applicazioni impacchettandole in unità standardizzate chiamate *container* che combinano il codice sorgente delle applicazioni con le librerie e le dipendenze del sistema operativo nel quale vengono eseguite. Docker offre la possibilità di creare, implementare, aggiornare e arrestare i container utilizzando dei comandi gestiti da un'unica API. Il funzionamento dei container si basa sulla virtualizzazione e sull'isolamento dei processi integrato nel kernel Linux che consente a più processi di condividere risorse di una singola istanza di un sistema operativo nello stesso modo delle *virtual machine* [12]. Nel progetto, Docker viene utilizzato per costruire un ambiente di sviluppo, isolato dal sistema sul quale viene eseguito e che permette di effettuare dei test. Nel caso in cui si verifichino dei problemi, garantisce la possibilità di effettuare modifiche in maniera rapida ed efficiente prima del rilascio di nuove funzionalità.

[1] Con Open Source si intende un software senza vincoli di copyright e quindi pubblicamente accessibile.

[2] La garbage collection è una modalità di gestione della memoria che permette di deallocare frammenti di memoria non più utilizzati dalle applicazioni tramite un processo che le libera autonomamente.

- **Postman**: Si tratta di un'applicazione che permette di costruire, testare e documentare le API. Postman fornisce un'interfaccia grafica in cui è possibile simulare una chiamata a un'API senza dover mettere mano al codice. Il funzionamento è abbastanza semplice: basta inserire il tipo e l'URL della chiamata HTTP che indentifica l'API e popolare l'*header* o il *body* con eventuali parametri aggiuntivi. Cliccando poi sull'apposito bottone la chiamata viene eseguita e viene mostrato l'output (tipicamente un array di oggetti in formato JSON) accompagnato da un codice di stato HTTP. L'utilizzo di Postman si è reso necessario per testare il comportamento di nuove API e risolvere eventuali criticità riscontrate [11].

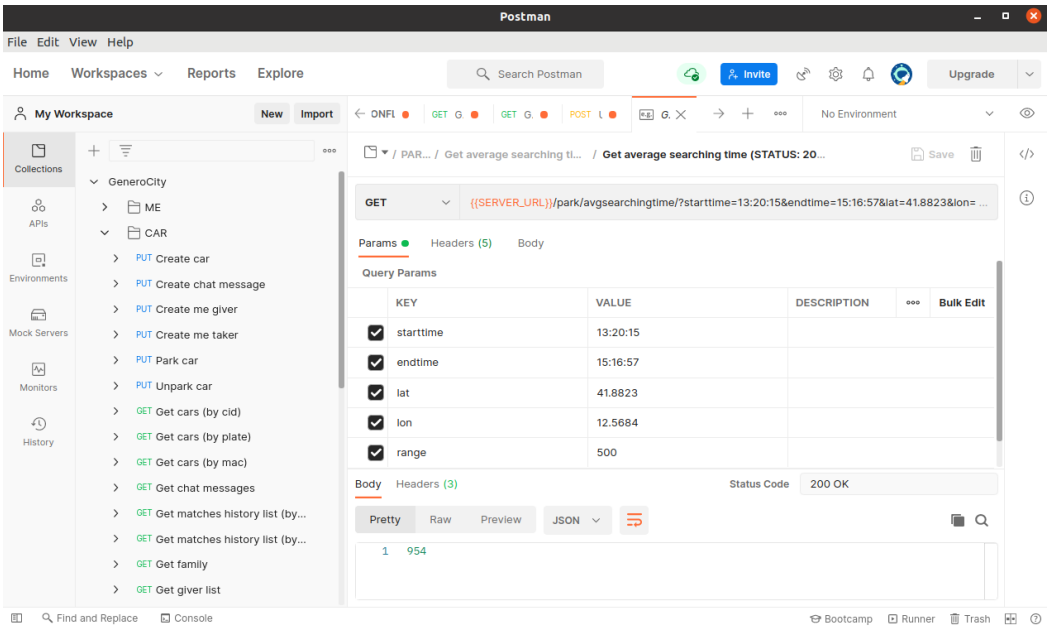



**Figura 1.1.** Postman: esempio di chiamata all'API getAvgSearchingTime che restituisce il tempo medio per trovare parcheggio dagli utenti di Generocity in una determinata zona e fascia oraria.

- **Adminer**: Adminer è uno strumento di gestione di database relazionali che offre un'interfaccia grafica. Questo software si è reso necessario per testare il funzionamento di eventuali interrogazioni al database effettuate dalle API.

# Capitolo 2

# Background

Prima di introdurre il lavoro che ho avuto modo di svolgere in questi ultimi mesi, è opportuno sintetizzare l'architettura e il funzionamento generale della struttura dell'applicazione GeneroCity, in particolare il lato *backend*.

## 2.1 Conoscenze preliminari

### 2.1.1 Struttura generale dell'applicazione

Così come buona parte delle applicazioni presenti sul mercato, GeneroCity è composta da due parti:

- Frontend: si tratta della parte visibile all'utente. Egli può interagire con il programma tramite un'apposita interfaccia utente che permette di acquisire dei dati in ingresso, che vengono opportunamente elaborati e resi disponibili alla parte backend.
- Backend: si tratta della parte che permette l'effettivo funzionamento delle interazioni tra utente e applicativo. Il backend gestisce la logica del programma infatti ottiene i dati provenienti dal frontend, li elabora e restituisce una risposta, comunicando direttamente con risorse esterne quali i database.

Durante questo percorso mi sono occupato della gestione delle risorse lato backend, in particolar modo della creazione di nuove API.

### 2.1.2 Definizione e funzionamento di un'API

Una parte consistente del tirocinio ha riguardato la creazione e la modifica di API. Un'API (acronimo di *Application Programming Interface*) rappresenta un insieme di procedure, definizioni e protocolli che permette di offrire servizi ad altro software o hardware. Le API permettono ai servizi presenti all'interno dell'applicazione di comunicare con altri servizi senza necessariamente sapere come questi siano implementati e mostrando solo le parti necessarie. Uno dei principali vantaggi delle API è la possibilità di ottenere un'astrazione a più alto livello infatti evitano ai programmatori di dover riscrivere tutte le funzioni necessarie al programma ogni volta che si aggiunge una nuova funzionalità, garantendo quindi riuso del codice. Le API aggiungono anche un ulteriore strato di sicurezza grazie a un sistema di controllo degli accessi che consente di autorizzare l'accesso ai vari servizi solo alle risorse e agli utenti autorizzati. I dati del client non sono mai direttamente esposti al server così come i dati del server non sono mai esposti al client, infatti la comunicazione tra i due

avviene attraverso piccoli pacchetti di dati (tipicamente scambiati tramite protocollo HTTP) che contengono solo ciò che è necessario a espletare un determinato compito [13].
Esistono due principali tipologie di API [14]:

- SOAP (*Simple Object Access Protocol*): si tratta di un protocollo in cui il formato del messaggio scambiato tra due entità viaggia mediante una richiesta HTTP o SMTP e utilizza il formato XML[3]. Questo protocollo facilita la comunicazione tra programmi che impiegano piattaforme e tecnologie diverse.

- REST (*Representational State Transfer*): si tratta di uno stile architetturale basato sul sistema client-server in cui le richieste vengono gestite tramite il protocollo HTTP e il formato dei messaggi è tipicamente il JSON[4]. Nelle *RestAPI* il client invia una chiamata a un URL specificando un metodo HTTP e passando eventuali dati, tipicamente in formato JSON, nel *request body*. Nel progetto viene utilizzata la tecnologia RestAPI per via della maggiore flessibilità e facilità di configurazione.

Per garantire la sicurezza delle API si possono utilizzare vari schemi di sicurezza: in GeneroCity viene utilizzato OIDC (acronimo di *OpenID Connect*), un protocollo di autenticazione basato sull'emissione di token che permettono al client di accedere alle funzionalità del server che l'ha emesso. I token permettono di ottenere informazioni di base su un utente finale tramite API sfruttando il formato dati JSON.

### 2.1.3 Architettura client-server

Come anticipato nella precedente sezione, nel progetto si utilizzano le API della tipologia REST che si basano sull'architettura client-server. Si tratta di una struttura formata da due attori [15]:

- Client: Si tratta di una componente software ospitata su una macchina (chiamata *host*) che accede a servizi e risorse presenti su un server. Il client effettua una richiesta (utilizzando un protocollo che dipende dall'applicativo), la invia al server e attende una risposta. Non è necessario che il client sappia come il server elabora i dati, bensì si deve solamente occupare di strutturare la richiesta nel formato atteso dal server;

- Server: Così come il client, anche il server è una componente software ospitata su una macchina. Si tratta di un programma in grado di rispondere alle richieste del client. Il server riceve una richiesta, la elabora e infine invia una risposta al client. È il server che regola il funzionamento delle API, nonché la logica del programma.

Il protocollo che regola la trasmissioni di informazioni tra le due entità è il protocollo HTTP (acronimo di *Hypertext Transfer Protocol*). Le REST API utilizzano una soluzione basata su HTTP per il controllo dei servizi web. I messaggi di richiesta sono

[3]Il linguaggio XML (acronimo di *eXtensible Markup Language*) è basato su un meccanismo sintattico che consente di gestire il significato degli elementi contenuti all'interno di un documento come una pagina web. La sintassi è molto simile a quella del linguaggio HTML e consiste di una serie di tag annidati che si possono definire all'interno delle parentesi angolari <> (ad esempio: <tag></tag>)

[4]Il formato JSON (JavaScript Object Notation) viene utilizzato nello scambio di dati in architetture client/server. La sintassi è piuttosto semplice e consiste di una serie di oggetti "chiave: valore".

composti da una *request line* contenente un metodo (ad esempio "GET", "POST" e "DELETE"), un URL che rappresenta la risorsa lato server da invocare e la versione del protocollo. Seguono una sequenza di *header* contenenti metadati sul client e sul server e un eventuale *body* che rappresenta il corpo del messaggio. I messaggi di risposta hanno una struttura piuttosto simile a quelli di richiesta. La differenza risiede nella presenza di una *status line* iniziale contentente uno *status code*, e una stringa che riassume il suo significato [16]. Nelle API il contenuto della risposta viene tipicamente riportato nel body in formato JSON. I principali status-code sono i seguenti:

**Tabella 2.1.** Principali codici di stato HTTP

| Codice | Significato |
|---|---|
| 200 OK | Richiesta valida e andata a buon fine |
| 201 Created | Richiesta valida e risorsa creata |
| 400 Bad Request | Richiesta non formattata correttamente |
| 401 Unauthorized | Autenticazione fallita o non fornita |
| 403 Forbidden | Non si possiede il permesso necessario |
| 404 Not Found | Risorsa non trovata o non disponibile |
| 500 Internal Server Error | Errore generico avvenuto lato server |

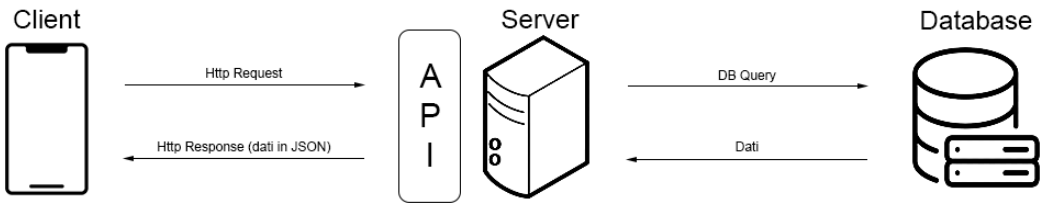



**Figura 2.1.** Funzionamento generale di un'API: il client effettua una richiesta al server tramite API, il server la elabora e genera una risposta (eventualmente interagendo con il database), e la invia al client.

## 2.2 Struttura Backend di GeneroCity

Nel progetto GeneroCity, la parte di backend è costituita da una serie di file e cartelle che gestiscono il funzionamento delle API. Ci sono svariati file di configurazione come il *Dockerfile* e il *Makefile* che contengono una serie di comandi per l'inizializzazione del container su Docker, il *PostmanCollection* che contiene la documentazione relativa alla richieste HTTP da importare su Postman e ulteriori file utili all'esecuzione del progetto. Le cartelle che si occupano del funzionamento dell'applicazione, nonché quelle su cui ho avuto modo di lavorare durante il tirocinio, sono le seguenti:

- **cmd**/: All'interno di questa cartella solo presenti vari *tool* eseguibili tramite comando. Ciascun tool è generalmente costituito da due file, il *load_ configuration* che contiene la configurazione necessaria all'esecuzione del medesimo e il *main* che è un file eseguibile che carica la configurazione precedentemente definita, inizializza le variabili globali ed esegue un determinato compito. Tra questi

tool, va sicuramente menzionato lo *webapi*, nonché l'eseguibile principale che costruisce l'applicazione effettiva, assembla e inizializza tutte le componenti essenziali (database, logger [5], gestore delle notifiche push [6], etc..) ed effettua i dovuti controlli sul loro stato.

- **db**/: Vi sono vari file che permettono di gestire il database come lo *schema* che contiene una copia della sua attuale struttura ed eventuali file di migrazione dei dati [7].

- **service**/: Sono presenti tutti i file che implementano la logica delle API, le strutture dati che queste dovranno ricevere in input e passare in output nella relativa risposta al client e file che permettono l'interazione con il database. In generale ogni API è composta da due parti: vi è un file che gestisce i dati in input, controlla la loro validità e crea la risposta in output, e un ulteriore file che si occupa di realizzare una o più chiamate al database mediante l'utilizzo di comandi *SQL*. Nel momento in cui si avvia l'applicazione, viene invocato un oggetto *router* che funge da ricevitore dell'applicazione. Questo oggetto analizza le richieste in entrata, recupera l'url presente nell'*header* e lo mappa con una delle funzioni che implementano l'API.

- **doc**/: Questa cartella contiene la documentazione delle API nel formato OAS (acronimo di *OpenAPI Specification*) [17], uno standard che permette di comprenderne il funzionamento in maniera semplice e minimale senza dover accedere al codice sorgente.

Per quanto riguarda il **database**, è opportuno specificare la presenza di due tabelle sulle quali ho avuto modo di lavorare, e che hanno subito alcune modifiche che verranno esposte nei capitoli che seguono:

- La tabella *parks* contiene tutte le colonne che rappresentano un parcheggio e contengono informazioni quali latitudine, longitudine, orario e id di utente e macchina che lo hanno eseguito. Ciascun parcheggio si identifica con un id (*parkid*) che gli viene assegnato in maniera incrementale nel momento dell'inserimento nel database.

- La tabella *matches_history* contiene informazioni sui match passati. Le colonne mantengono informazioni sull'id di utente e macchina di taker e giver autori dello scambio, lo stato del match (ad esempio *deleted*, *unsuccess-taker*, *unsuccess-giver*, *success*), orario stimato di arrivo del taker al parcheggio del giver e alcune info su quest'ultimo come l'id e la posizione geografica. Ciascun match si identifica con un id (*historyid*) che gli viene assegnato in maniera incrementale nel momento dell'inserimento nel database.

[5] Logger: Processo che si occupa di analizzare lo stato dell'applicazione, registrando tutte le operazioni svolte e segnalando eventuali errori.

[6] Gestore notifiche push: Processo che si occupa di gestire le notifiche sui dispositivi mobili: ottiene i dati dal server, li invia ai dispositivi e riceve una risposta.

[7] Migration: Si tratta di file che permettono di effettuare modifiche al database come l'aggiunta o la rimozione di una colonna o di un'intera tabella. Sono costituiti da due parti, *migrate-up* (contiene una o più query da applicare al db) e *migrate-down* (contiene una o più query che permettono di ripristinare le modifiche precedentemente apportate).

# Capitolo 3

# Progettazione, sviluppo e documentazione di API

Dopo aver sintetizzato la struttura del progetto e il funzionamento generale dell'applicazione lato server, si introduce il lavoro svolto in questi ultimi mesi di tirocinio. In questa sezione verrà esposta la progettazione e lo sviluppo di API incentrate sui match.

## 3.1 Associazione match e parcheggi

Uno dei primi problemi che ho avuto modo di affrontare è stato quello di associare i parcheggi ai match effettuati da un utente. Inizialmente, nel momento in cui l'utente effettuava un parcheggio, non c'era la possibilità di capire se questo era stato preso grazie a un match in quanto non c'era alcuna associazione tra i due. Nel momento in cui il taker avviava un match con un giver e arrivava da quest'ultimo, tale match andava a buon fine ma non si poteva capire se il taker avesse realmente preso il posto del giver. Bisognava quindi trovare un metodo per associare un parcheggio a tutti i match effettuati dal taker per ottenerlo.
Una prima idea è stata quella di lavorare in maniera retroattiva: nel momento in cui l'utente effettuava un parcheggio, si andavano a prendere tutti match in cui l'utente era il taker e l'orario stimato di arrivo (*schedule*) e il luogo si avvicinavano a data e luogo del parcheggio considerato. Tutti i match trovati si andavano poi a salvare in una nuova tabella contenente l'id del match e l'id del parcheggio. Questa idea presentava notevoli criticità tra le quali si evidenzia la difficoltà nel confrontare l'orario stimato di arrivo del taker e l'orario in cui il parcheggio era stato effettuato. Essendo lo *schedule* del match solamente una stima, bisognava scegliere un range di minuti nel quale selezionare i match da associare a un parcheggio e, sebbene i primi test davano un esito positivo, questa soluzione avrebbe presentato alcune criticità in uno scenario reale.
Dopo aver svolto una riunione con alcuni colleghi, è stata abbandonata l'idea precedentemente esposta in favore della soluzione attualmente adottata: per capire quali sono i match da associare a un determinato parcheggio, si vanno semplicemente a prendere tutti i match che sono stati effettuati dal taker a partire dall'istante del precedente parcheggio. In particolare, si considerano varie possibilità:

- Se l'utente non ha mai effettuato alcun parcheggio: al primo parcheggio si vanno ad associare tutti i match fatti fino a quel momento;

- Se l'utente ha già effettuato dei parcheggi: all'ultimo parcheggio effettuato si vanno ad associare tutti i match fatti a partire dalla data del precedente parcheggio;

Per introdurre questa nuova funzionalità, non è stata creata alcuna nuova API, bensì è stata modificata quella utilizzata per registrare un nuovo parcheggio (chiamata *carPark*). Nel frammento di codice che segue, si mostra la query in linguaggio SQL che permette di recuperare la data del precedente parcheggio effettuato dallo stesso utente con la stessa auto del parcheggio considerato. Se l'utente non ha mai effettuato parcheggi con tale auto allora la variabile *matchSearchingStartTime* viene inizializzata con il suo valore di default che corrisponde all'*istante zero* [8]:

```
1 var matchSearchingStartTime time.Time
2 err = db.c.Get(&matchSearchingStartTime, "SELECT parktime FROM parks WHERE
  ↪  parkstatus = 'park' AND parktime <= ? AND cid = ? AND usedby = ? ORDER
  ↪  BY parkid DESC LIMIT 1;", carPark.ParkTime.String(),
  ↪  carPark.Cid.String(), carPark.Id.String())
3 if err != nil && err != sql.ErrNoRows {
4         return 0, errors.Wrap(err, "select2 statement")
5 }
```

**Listing 3.1.** API carPark: recupero dell'istante del precedente parcheggio

Per salvare l'id del parcheggio associato ai match, è stata aggiunta una nuova colonna nella tabella *matches_history* (che contiene le info su tutti i match passati) chiamata *parkid*. Tale modifica è stata fatta attraverso una *migration*.
Si vanno quindi a prendere gli id di tutti i match effettuati dall'utente e l'auto considerata (a partire dall'istante precedentemente ottenuto) e si aggiorna il campo *matches_history.parkid* nelle colonne relative ai match selezionati. Per evitare eventuali conflitti, è stato aggiunto un controllo che abilita la modifica dell'id del parcheggio associato al match, solamente se questo non è mai stato definito (e quindi ha valore *null*):

```
1 ALTER TABLE matches_history
2 ADD COLUMN IF NOT EXISTS parkid int(11) DEFAULT NULL,
3 ADD FOREIGN KEY IF NOT EXISTS `matches_history_ibfk_1` (`parkid`)
  ↪  REFERENCES `parks` (`parkid`) ON UPDATE CASCADE;
```

**Listing 3.2.** Codice SQL per l'aggiunta dalla colonna *parkid* nella tabella *matches_history*.

[8] Istante zero: "0000-00-00 00:00:00"

```
var parkMatches []int
err = tx.Select(&parkMatches, "SELECT historyid FROM matches_history WHERE takercid = ? AND takerid = ? AND starttime > ? AND starttime < ?", carPark.Cid.String(), carPark.Id.String(), matchSearchingStartTime, carPark.ParkTime)
if err != nil && err != sql.ErrNoRows {
    _ = tx.Rollback()
    return 0, errors.Wrap(err, "select3 statement")
}

if len(parkMatches) != 0 {
    for _, matchid := range parkMatches {
            var checkMatchExistance int
            err = tx.Get(&checkMatchExistance, "SELECT COUNT(1) FROM matches_history WHERE historyid = ? AND parkid IS NOT NULL", matchid)
            if err != nil {
                    _ = tx.Rollback()
                    return 0, errors.Wrap(err, "get2 statement")
            }
            if checkMatchExistance == 0 {
                    _, err = tx.Exec("UPDATE matches_history SET parkid = ? WHERE historyid = ?", parkId, matchid)
                    if err != nil {
                            _ = tx.Rollback()
                            return 0, errors.Wrap(err, "insert4 statement")
                    }
            }
    }
}
```

**Listing 3.3.** API carPark: aggiornamento dei match con l'id del parcheggio associato.

Quindi, ogni volta che il taker effettua un parcheggio si vanno a recuperare tutti i match che sono stati effettuati a partire dal precedente parcheggio e a questi viene associato l'id del parcheggio appena effettuato.

### 3.1.1 Lista dei match associati a un parcheggio

Una volta risolto il problema precedentemente esposto, è sorta la necessità di creare una nuova API per ottenere la lista dei match associati a un certo parcheggio. Il funzionamento dell'API è piuttosto semplice e si serve della colonna *matches_history.parkid* precedentemente aggiunta. Segue un grafico che illustra il funzionamento generale dell'API:

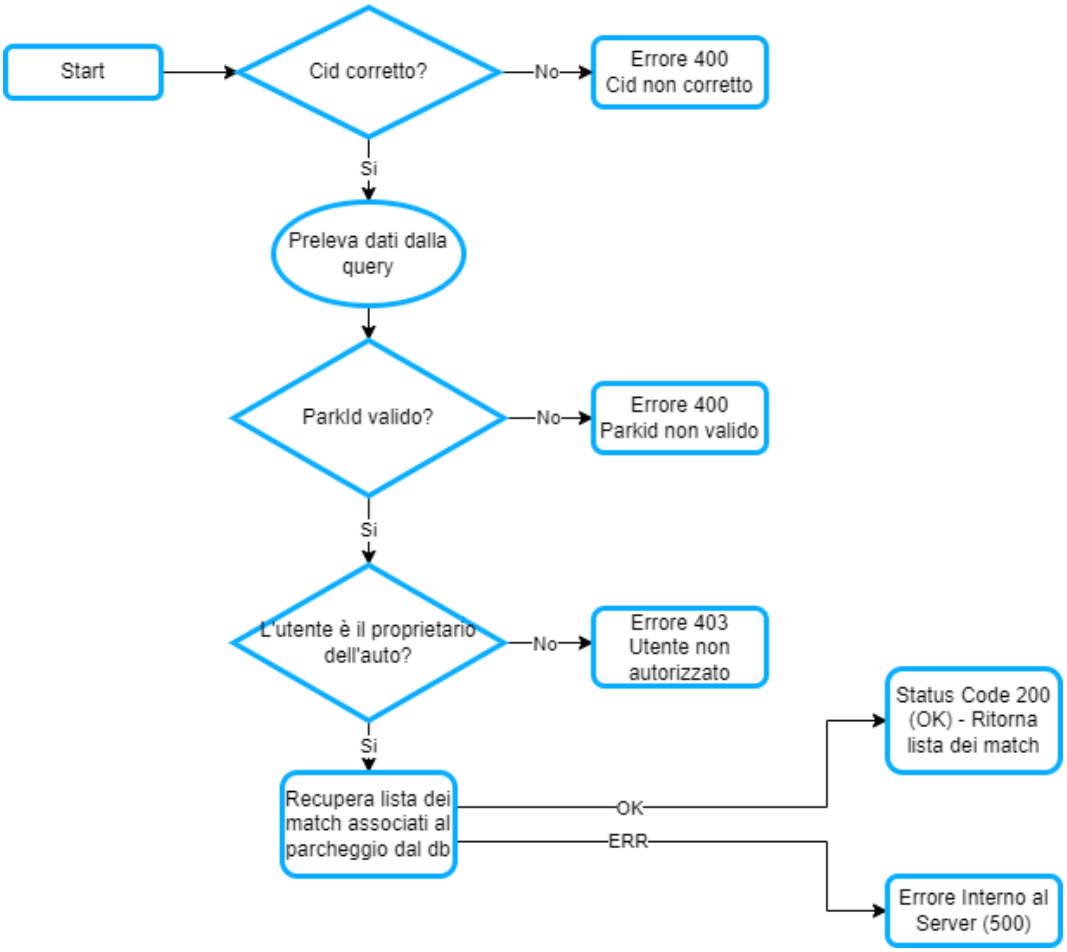



**Figura 3.1.** Progettazione API *getParkMatches*

La seguente API prende in input l'id di un auto (*cid*) e l'id del parcheggio (*parkid*) del quale recuperare i match associati. Controlla che il *parkid* sia valido e che l'utente che ha effettuato la richiesta sia il proprietario dell'auto. Se tutti i controlli vanno a buon fine allora si recupera la lista dei match dal database. L'API ottiene due liste:

- takerMatches: si tratta di tutti i match effettuati dall'utente come taker e che quindi hanno eventualmente permesso di ottenere il parcheggio considerato. Sono tutti quei match che presentano nel campo *parkid*, l'id del parcheggio passato in input;
- giverMatches: si tratta di tutti i match effettuati dall'utente come giver e che quindi hanno eventualmente permesso di lasciare il parcheggio considerato a un altro utente. Sono tutti quei match che presentano nel campo *giverparkid*, l'id del parcheggio passato in input.

L'oggetto ritornato all'utente sarà quindi composto da due chiavi *takerMatches* e *giverMatches* i cui valori sono array di oggetti JSON con la seguente struttura:

```
type Match struct {
        TakerId        *uuid.UUID       `json:"takerid" db:"takerid"`
        TakerCid       *uuid.UUID       `json:"takercid" db:"takercid"`
        TakerLat       *float64         `json:"takerlat" db:"takerlat"`
        TakerLon       *float64         `json:"takerlon" db:"takerlon"`
        TakerDeltaLat *float64          `json:"takerdeltalat"
        ↪  db:"takerdeltalat"`
        TakerDeltaLon *float64          `json:"takerdeltalon"
        ↪  db:"takerdeltalon"`
        TakerSize      *float32         `json:"takersize" db:"takersize"`
        TakerNickname *string           `json:"takernickname"
        ↪  db:"takernickname"`
        GiverId        *uuid.UUID       `json:"giverid" db:"giverid"`
        GiverCid       *uuid.UUID       `json:"givercid" db:"givercid"`
        GiverLat       *float64         `json:"giverlat" db:"giverlat"`
        GiverLon       *float64         `json:"giverlon" db:"giverlon"`
        GiverSize      *float32         `json:"giversize" db:"giversize"`
        GiverNickname *string           `json:"givernickname"
        ↪  db:"givernickname"`
        GiverParkId    int              `json:"giverparkid"
        ↪  db:"giverparkid"`
        ParkType       *string          `json:"parktype" db:"parktype"`
        Schedule       *time.Time       `json:"schedule" db:"schedule"`
        Status         string           `json:"status" db:"status"`
        Created        *time.Time       `json:"created" db:"created"`
        Updated        *time.Time       `json:"updated" db:"updated"`
        HistoryId      int              `json:"historyid" db:"historyid"`
        TimeDistance   time.Duration    `json:"timedistance" db:"-"`
        ParkId         int              `json:"parkid" db:"parkid"`
        CancelInfo     CancelJsonStruct `json:"cancelinfo" db:"cancelinfo"`
}
```

**Listing 3.4.** Struttura dei file JSON che rappresentano un match.

Si mostra ora il codice dell'API appena illustrata:

```
func (rt *_router) getParkMatches(w http.ResponseWriter, r *http.Request,
↪  ps httprouter.Params, ctx reqcontext.RequestContext) {
        cidUuid, err := uuid.FromString(ps.ByName("cid"))
        if err != nil {
                var errorMessage types.ErrorMessage
                errorMessage.ErrorMessage = "cid not well formed"
                ctx.Logger.WithError(err).Error(errorMessage)
                w.Header().Set("content-type", "application/json")
                w.WriteHeader(400)
                _ = json.NewEncoder(w).Encode(errorMessage)
                return
        }

        b, err := rt.db.IsOwner(types.Driver{Id: ctx.UserId, Cid: cidUuid})
        if err != nil {
                var errorMessage types.ErrorMessage
                errorMessage.ErrorMessage = "can't check owner"
                ctx.Logger.WithError(err).Error(errorMessage)
                w.Header().Set("content-type", "application/json")
```

```
19                  w.WriteHeader(500)
20                  _ = json.NewEncoder(w).Encode(errorMessage)
21                  return
22          } else if !b {
23                  var errorMessage types.ErrorMessage
24                  errorMessage.ErrorMessage = "user not authorized"
25                  ctx.Logger.WithError(err).Error(errorMessage)
26                  w.Header().Set("content-type", "application/json")
27                  w.WriteHeader(403)
28                  _ = json.NewEncoder(w).Encode(errorMessage)
29                  return
30          }
31
32          params := r.URL.Query()
33          var parkId int
34          if params.Get("parkid") == "" {
35                  var errorMessage types.ErrorMessage
36                  errorMessage.ErrorMessage = "parkid param is required"
37                  ctx.Logger.Error(errorMessage)
38                  w.Header().Set("content-type", "application/json")
39                  w.WriteHeader(400)
40                  _ = json.NewEncoder(w).Encode(errorMessage)
41                  return
42          } else {
43                  parkId, err = strconv.Atoi(params.Get("parkid"))
44                  if err != nil {
45                          var errorMessage types.ErrorMessage
46                          errorMessage.ErrorMessage = "parkid param not well
                            ↪  formed"
47                          ctx.Logger.WithError(err).Error(errorMessage)
48                          w.Header().Set("content-type", "application/json")
49                          w.WriteHeader(400)
50                          _ = json.NewEncoder(w).Encode(errorMessage)
51                          return
52                  }
53          }
54
55          takerMatches, giverMatches, err := rt.db.GetParkMatches(parkId)
56          if err != nil {
57                  var errorMessage types.ErrorMessage
58                  errorMessage.ErrorMessage = "can't get matches associated
                    ↪  with this park"
59                  ctx.Logger.WithError(err).Error(errorMessage)
60                  w.Header().Set("content-type", "application/json")
61                  w.WriteHeader(500)
62                  _ = json.NewEncoder(w).Encode(errorMessage)
63          } else {
64                  type parkMatchesStruct struct {
65                          TakerMatch []types.Match `json:"takerMatches"`
66                          GiverMatch []types.Match `json:"giverMatches"`
67                  }
68                  parkMatchesRet := parkMatchesStruct{TakerMatch:
                    ↪  takerMatches, GiverMatch: giverMatches}
69
70                  w.Header().Set("content-type", "application/json")
71                  _ = json.NewEncoder(w).Encode(parkMatchesRet)
72          }
73  }
```

**Listing 3.5.** Sviluppo API *getParkMatches*: fase di controllo dei dati in input.

```
...
var takerMatch []types.Match
err := db.c.Select(&takerMatch, "SELECT matches_history.historyid,
↪  matches_history.takerid, matches_history.takercid, matches_history.takerlat,
↪  matches_history.takerlon, matches_history.giverid, matches_history.givercid,
↪  matches_history.giverlat, matches_history.giverlon, matches_history.schedule,
↪  matches_history.status, matches_history.starttime as created,
↪  matches_history.endtime as updated FROM matches_history WHERE
↪  matches_history.parkid = ? AND matches_history.status != 'waiting'", parkId)
if err != nil && err != sql.ErrNoRows {
        return nil, nil, errors.Wrap(err, "select1 statement")
}

for i, tm := range takerMatch {
        var currentMatch types.Match
        var giverParkId sql.NullInt64
        err = db.c.Get(&giverParkId, "SELECT giverparkid FROM matches_history WHERE
        ↪  historyid = ?", tm.HistoryId)
        if err != nil && err != sql.ErrNoRows {
                return nil, nil, errors.Wrap(err, "get1 statement")
        }

        if giverParkId.Valid {
                err = db.c.Get(&currentMatch, "SELECT parks.parktype, y.size as
                ↪  giversize, w.nickname as givernickname FROM matches_history,
                ↪  parks, cars as y, users as w WHERE parks.parkid =
                ↪  matches_history.giverparkid AND y.cid =
                ↪  matches_history.givercid AND w.id = matches_history.giverid AND
                ↪  matches_history.historyid = ?", tm.HistoryId)
                if err != nil && err != sql.ErrNoRows {
                        return nil, nil, errors.Wrap(err, "get2 statement")
                }
                takerMatch[i].GiverParkId = int(giverParkId.Int64)
                takerMatch[i].ParkType = currentMatch.ParkType
                takerMatch[i].GiverSize = currentMatch.GiverSize
                takerMatch[i].GiverNickname = currentMatch.GiverNickname
        }

        if tm.TakerCid != nil {
                err = db.c.Get(&currentMatch, "SELECT x.size as takersize,
                ↪  z.nickname as takernickname FROM matches_history, cars as x,
                ↪  users as z WHERE x.cid = matches_history.takercid AND z.id =
                ↪  matches_history.takerid AND matches_history.historyid = ?",
                ↪  tm.HistoryId)
                if err != nil && err != sql.ErrNoRows {
                        return nil, nil, errors.Wrap(err, "get3 statement")
                }
                takerMatch[i].TakerSize = currentMatch.TakerSize
                takerMatch[i].TakerNickname = currentMatch.TakerNickname
        }
}
...
```

**Listing 3.6.** Sviluppo API *getParkMatches*: recupero della lista dei match associati a un parcheggio come taker. Si esclude il codice del recupero dei match effettuati come giver in quanto molto simile a quello esposto.

### 3.1.2 Recupero informazioni sui match associati a un parcheggio

Una volta creata l'infrastruttura per l'associazione dei match ai vari parcheggi effettuati da un taker, si è aperta la possibilità di ottenere numerose nuove informazioni relative alla funzionalità di scambio del posto. Avendo la possibilità di ottenere facilmente la lista di tutti i match associati a un parcheggio, è stato possibile offrire all'utente le seguenti informazioni:

- **totalMatchNumber**: numero totale dei match effettuati dall'utente per ottenere il parcheggio;
- **successMatchNumber**: numero di match "success" (ossia i match andati a buon fine) effettuati dall'utente per ottenere il parcheggio;
- **unsuccessTakerMatchNumber**: numero di match "unsuccess-taker" (ossia i match annullati dal taker) effettuati dall'utente per ottenere il parcheggio;
- **unsuccessGiverMatchNumber**: numero di match "unsuccess-giver" (ossia i match annullati dal giver) effettuati dall'utente per ottenere il parcheggio;
- **expiredMatchNumber**: numero di match "expired" (ossia i match scaduti) effettuati dall'utente per ottenere il parcheggio;
- **deletedMatchNumber**: numero di match "deleted" (ossia i tentativi di match annullati prima di trovare un giver) effettuati dall'utente per ottenere il parcheggio;
- **takerMatchId**: id del match *"success"* in cui l'utente era il taker e che lo ha portato a ottenere il parcheggio da un altro utente;
- **giverMatchId**: id del match *"success"* in cui l'utente era il giver e che lo ha portato a lasciare il parcheggio a un altro utente.

I primi sei flag rappresentano delle semplici informazioni sul numero dei match effettuati dall'utente nell'intervallo di tempo tra il parcheggio considerato e il precedente. Queste informazioni possono essere utili per costruire delle statistiche sull'utilizzo della funzionalità di scambio posto.
Gli ultimi due flag, risolvono uno dei problemi elencati nelle sezioni precedenti ossia quello di capire se l'utente ha ottenuto il parcheggio oppure lo ha lasciato grazie a un match andato a buon fine.
Il flag *takerMatchId* ritorna un intero che rappresenta l'id di un match (oppure *null*). Si va a controllare se nella lista dei match associati al parcheggio, l'ultimo match (in ordine cronologico) è del tipo "*success*", quindi:

1. Nel caso di più match "*success*" considero solo l'ultimo;
2. Se non ho alcun match "*success*" allora significa che il parcheggio non è stato ottenuto per via di un match;
3. Se ho almeno un match "*success*" ma dopo di questo ho altri match (di altri tipi) allora significa che l'utente alla fine non ha preso il posto del giver ed è andato via.

Il flag *giverMatchId* ritorna un intero che rappresenta l'id di un match (oppure *null*). Si va a prendere la lista dei match con campo *giverparkid* (contenente l'informazione sull'id del parcheggio del giver) uguale all'id del parcheggio considerato e si prende

l'ultimo match. Si controlla quindi che questo match sia di tipo "*success*": se non lo è allora si ritorna *null* (significa che l'utente non ha lasciato il posto a nessuno), altrimenti significa che c'è stato un incontro tra il giver e il taker. Si vede se il taker si è veramente parcheggiato dal giver (si controlla che il *parkid* del parcheggio preso dal taker (nonché il parcheggio lasciato dal giver) faccia riferimento allo stesso match "*success*" del giver) o alla fine non ha preso quel posto (quindi il taker ha effettuato match successivi riguardanti quel parcheggio) e in tal caso si ritorna *null*.
Per ritornare queste informazioni, è stata modificata l'api *getParkList* che recupera la lista dei parcheggi effettuati da un utente. A questa lista sono state aggiunte tutte le nuove informazioni sopra elencate. Qui di seguito tutte le modifiche apportate al codice:

```
var taker sql.NullInt64
err = db.c.Get(&taker, "SELECT historyid FROM (SELECT historyid, status FROM
↪  matches_history WHERE parkid = ? ORDER BY historyid DESC LIMIT 1) AS lastMatch
↪  WHERE lastMatch.status = 'success'", ret[count].ParkId)
if err != nil {
        if err != sql.ErrNoRows {
                return nil, errors.Wrap(err, "get2 statement")
        }
}
type LastSuccessfulGiverMatch struct {
        ParkId    sql.NullInt64 `db:"parkid"`
        HistoryId string        `db:"historyid"`
        TakerId   uuid.UUID     `db:"takerid"`
        TakerCid  uuid.UUID     `db:"takercid"`
}
var giver sql.NullInt64
var lastSuccessfulGiverMatch LastSuccessfulGiverMatch
err = db.c.Get(&lastSuccessfulGiverMatch, "SELECT lastGiverSuccessMatch.historyid
↪  AS historyid, lastGiverSuccessMatch.takerid AS takerid,
↪  lastGiverSuccessMatch.takercid AS takercid, lastGiverSuccessMatch.parkid AS
↪  parkid FROM (SELECT historyid, status, takercid, takerid, parkid FROM
↪  matches_history WHERE giverparkid = ? AND status = 'success' ORDER BY historyid
↪  DESC LIMIT 1) AS lastGiverSuccessMatch WHERE lastGiverSuccessMatch.historyid =
↪  (SELECT historyid FROM matches_history WHERE giverparkid = ? ORDER BY historyid
↪  DESC LIMIT 1)", ret[count].ParkId, ret[count].ParkId)
if err != nil {
        if err != sql.ErrNoRows {
                return nil, errors.Wrap(err, "get3 statement")
        }
}
if lastSuccessfulGiverMatch.ParkId.Valid {
        err = db.c.Get(&giver, "SELECT * FROM (SELECT historyid FROM
        ↪  matches_history WHERE parkid = ? AND takerid = ? AND takercid = ? ORDER
        ↪  BY historyid DESC LIMIT 1) AS lastTakerMatch WHERE
        ↪  lastTakerMatch.historyid = ?", lastSuccessfulGiverMatch.ParkId.Int64,
        ↪  lastSuccessfulGiverMatch.TakerId, lastSuccessfulGiverMatch.TakerCid,
        ↪  lastSuccessfulGiverMatch.HistoryId)
        if err != nil {
                if err != sql.ErrNoRows {
                        return nil, errors.Wrap(err, "get4 statement")
                }
        }
}
var totalmatchnumber int
err = db.c.Get(&totalmatchnumber, "SELECT COUNT(*) FROM matches_history WHERE
↪  parkid = ? AND status != \"waiting\"", ret[count].ParkId)
if err != nil {
        return nil, errors.Wrap(err, "get5 statement")
}
```

```
35 var successmatchnumber int
36 err = db.c.Get(&successmatchnumber, "SELECT COUNT(*) FROM matches_history WHERE
   ↪  parkid = ? AND status = \"success\"", ret[count].ParkId)
37 if err != nil {
38         return nil, errors.Wrap(err, "get6 statement")
39 }
40 var unsuccesstakermatchnumber int
41 err = db.c.Get(&unsuccesstakermatchnumber, "SELECT COUNT(*) FROM matches_history
   ↪  WHERE parkid = ? AND status = \"unsuccess-taker\"", ret[count].ParkId)
42 if err != nil {
43         return nil, errors.Wrap(err, "get7 statement")
44 }
45 var unsuccessgivermatchnumber int
46 err = db.c.Get(&unsuccessgivermatchnumber, "SELECT COUNT(*) FROM matches_history
   ↪  WHERE parkid = ? AND status = \"unsuccess-giver\"", ret[count].ParkId)
47 if err != nil {
48         return nil, errors.Wrap(err, "get8 statement")
49 }
50 var expiredmatchnumber int
51 err = db.c.Get(&expiredmatchnumber, "SELECT COUNT(*) FROM matches_history WHERE
   ↪  parkid = ? AND status = \"expired\"", ret[count].ParkId)
52 if err != nil {
53         return nil, errors.Wrap(err, "get9 statement")
54 }
55 var deletedmatchnumber int
56 err = db.c.Get(&deletedmatchnumber, "SELECT COUNT(*) FROM matches_history WHERE
   ↪  parkid = ? AND status = \"deleted\"", ret[count].ParkId)
57 if err != nil {
58         return nil, errors.Wrap(err, "get10 statement")
59 }
```

**Listing 3.7.** Modifica API *getParkListByParkId*: recupero delle informazioni sul numero dei match (in base alla loro tipologia) e sui flag *takerMatchId* e *giverMatchId* che permettono di capire se il parcheggio è stato preso o lasciato grazie a un match.

## 3.2 Tempo e distanza media per la ricerca di un parcheggio

La seguente sezione vuole esporre un'ulteriore funzionalità introdotta in GeneroCity ossia la possibilità di conoscere il tempo e la distanza media per trovare parcheggio in una determinata area geografica e fascia oraria. Per ottenere queste informazioni è stato necessario creare rispettivamente due nuove API, *getAvgSearchingTime* e *getAvgSearchingDistance*.

### 3.2.1 Tempo medio

Per quanto riguarda il **tempo**, l'obiettivo è quello di conoscere quanto si impiega a trovare parcheggio in una determinata zona e in una determinata fascia oraria, in base ai dati di tutti gli utenti. Il funzionamento dell'API appositamente creata si basa sull'utilizzo di un dato *searching time* associato a un parcheggio e che rappresenta il tempo impiegato dall'utente nel cercarlo. Per calcolare questo dato vengono utilizzati i *tripjson*, degli array di oggetti *trippoint* che rappresentano la posizione dell'utente durante il suo viaggio in macchina e che vengono raccolti in intervalli di qualche secondo. Ogni parcheggio ha un *tripjson* associato dal quale è possibile calcolare il tempo di ricerca. Una volta che questo viene calcolato, viene salvato nel campo

*searchingtime* della tabella *parks* e viene utilizzato dall'API *getAvgSearchingTime*. La logica dell'API è la seguente:

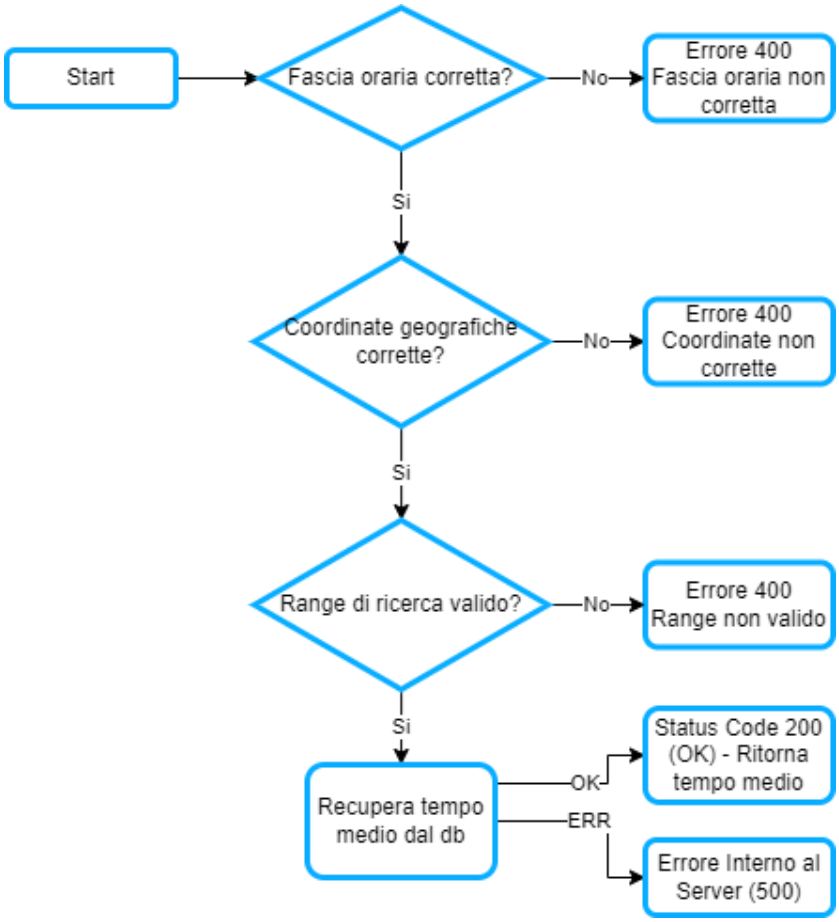



**Figura 3.2.** Progettazione API *getAvgSearchingTime*

L'API prende in input latitudine e longitudine, un tempo di inizio e fine che rappresentano una fascia oraria e un range nonché un valore che permette di identificare il luogo geografico (in metri) nel quale selezionare i parcheggi di interesse. Il range è facoltativo e, se non specificato, viene impostato di default a 500 metri. Prima di tutto si verifica la validità dei dati in input e quindi che tempo di inizio (*starttime*) e tempo di fine (*endtime*) rappresentino un orario nel formato "hh:ss:mm" (ore, minuti, secondi), che la latitude (*lat*) e la longitudine (*lon*) rappresentino delle coordinate valide e che il range, se specificato, sia un numero positivo. Se i controlli vanno a buon fine, l'API utilizza una specifica funzione (*squaring*) per calcolare un *geoSquare* utilizzando latitudine, longitudine e range, ossia un rettangolo che rappresenta un'area geografica nella quale selezionare i parcheggi dei quali calcolare il tempo medio di ricerca. Tale funzione ritorna quattro valori che identificano i confini del rettangolo.

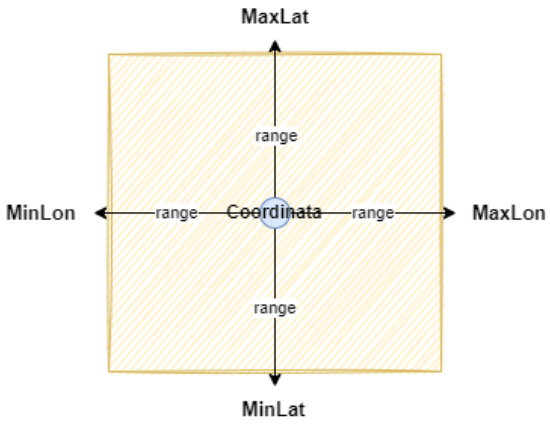



**Figura 3.3.** Immagine che rappresenta il calcolo del *geoSquare*

```
minLat, maxLat, minLon, maxLon := locationutils.Squaring(lat, lon,
↪  searchRange/1000)
```

**Listing 3.8.** Funzione *squaring*

Viene quindi invocata la query SQL che effettua la media del tempo di ricerca dei parcheggi effettuati nella fascia oraria e nell'area geografica precedentemente calcolata. Sono presenti due query per gestire due casi differenti:

- Se *starttime*<*endtime* allora si utilizza la seguente fascia oraria: [starttime, endtime];
- Se *starttime*>*endtime* allora si utilizza la seguente fascia oraria: [starttime, 23:59:59]U[00:00:00, endtime];

Il secondo caso si rende necessario per fare in modo che sia possibile calcolare il tempo medio in una fascia oraria dove l'endtime si trova nel giorno successivo rispetto allo starttime (si pensi al caso in cui il tempo di inizio sia "23:00:00" e il tempo di fine sia "02:00:00"). Qui di seguito il codice che implementa l'API:

```
func (rt *_router) getAvgSearchingTime(w http.ResponseWriter, r *http.Request, ps
↪  httprouter.Params, ctx reqcontext.RequestContext) {
        params := r.URL.Query()
        var parkRequest = types.ParkRequest{}
        location, err := time.LoadLocation("Europe/Rome")
        if err != nil {
                var errorMessage types.ErrorMessage
                errorMessage.ErrorMessage = "can't load location"
                ctx.Logger.Error(errorMessage)
                w.Header().Set("content-type", "application/json")
                w.WriteHeader(400)
                _ = json.NewEncoder(w).Encode(errorMessage)
                return
```

```
13          }
14          if params.Get("starttime") == "" || params.Get("endtime") == "" {
15                  var errorMessage types.ErrorMessage
16                  errorMessage.ErrorMessage = "starttime and endtime are required"
17                  ctx.Logger.Error(errorMessage)
18                  w.Header().Set("content-type", "application/json")
19                  w.WriteHeader(400)
20                  _ = json.NewEncoder(w).Encode(errorMessage)
21                  return
22          } else {
23                  startTime, err := time.ParseInLocation(types.DEFAULT_TIME_LAYOUT,
                    ↪  params.Get("starttime"), location)
24                  if err != nil {
25                          var errorMessage types.ErrorMessage
26                          errorMessage.ErrorMessage = "starttime param not well
                            ↪  formed"
27                          ctx.Logger.WithError(err).Error(errorMessage)
28                          w.Header().Set("content-type", "application/json")
29                          w.WriteHeader(400)
30                          _ = json.NewEncoder(w).Encode(errorMessage)
31                          return
32                  }
33                  endTime, err := time.ParseInLocation(types.DEFAULT_TIME_LAYOUT,
                    ↪  params.Get("endtime"), location)
34                  if err != nil {
35                          var errorMessage types.ErrorMessage
36                          errorMessage.ErrorMessage = "endtime param not well formed"
37                          ctx.Logger.WithError(err).Error(errorMessage)
38                          w.Header().Set("content-type", "application/json")
39                          w.WriteHeader(400)
40                          _ = json.NewEncoder(w).Encode(errorMessage)
41                          return
42                  }
43                  parkRequest.StartTime = &startTime
44                  parkRequest.EndTime = &endTime
45          }
46          if params.Get("lat") == "" || params.Get("lon") == "" {
47                  var errorMessage types.ErrorMessage
48                  errorMessage.ErrorMessage = "latitude and longitude are required"
49                  ctx.Logger.Error(errorMessage)
50                  w.Header().Set("content-type", "application/json")
51                  w.WriteHeader(400)
52                  _ = json.NewEncoder(w).Encode(errorMessage)
53                  return
54          } else {
55                  lat, err := locationutils.ParseLatitude(params.Get("lat"))
56                  if err != nil {
57                          var errorMessage types.ErrorMessage
58                          errorMessage.ErrorMessage = "lat param not well formed"
59                          ctx.Logger.WithError(err).Error(errorMessage)
60                          w.Header().Set("content-type", "application/json")
61                          w.WriteHeader(400)
62                          _ = json.NewEncoder(w).Encode(errorMessage)
63                          return
64                  }
65                  lon, err := locationutils.ParseLongitude(params.Get("lon"))
66                  if err != nil {
67                          var errorMessage types.ErrorMessage
68                          errorMessage.ErrorMessage = "lon param  not well formed"
69                          ctx.Logger.WithError(err).Error(errorMessage)
70                          w.Header().Set("content-type", "application/json")
71                          w.WriteHeader(400)
72                          _ = json.NewEncoder(w).Encode(errorMessage)
73                          return
```

```
74                  }
75                  parkRequest.Latitude = (*float64)(&lat)
76                  parkRequest.Longitude = (*float64)(&lon)
77                  searchRange := 500.0 //default
78                  if params.Get("range") != "" {
79                          inputSearchRange, err :=
                            ↪  strconv.ParseFloat(params.Get("range"), 64)
80                          if err != nil {
81                                  var errorMessage types.ErrorMessage
82                                  errorMessage.ErrorMessage = "range param not well
                                    ↪  formed"
83                                  ctx.Logger.WithError(err).Error(errorMessage)
84                                  w.Header().Set("content-type", "application/json")
85                                  w.WriteHeader(400)
86                                  _ = json.NewEncoder(w).Encode(errorMessage)
87                                  return
88                          }
89                          if inputSearchRange <= 0 {
90                                  var errorMessage types.ErrorMessage
91                                  errorMessage.ErrorMessage = "range must be greater
                                    ↪  than 0, if specified"
92                                  ctx.Logger.WithError(err).Error(errorMessage)
93                                  w.Header().Set("content-type", "application/json")
94                                  w.WriteHeader(400)
95                                  _ = json.NewEncoder(w).Encode(errorMessage)
96                                  return
97                          }
98                          searchRange = inputSearchRange
99                  }
100                 minLat, maxLat, minLon, maxLon := locationutils.Squaring(lat, lon,
                    ↪  searchRange/1000)
101                 parkRequest.GeoSquare = types.GeoSquare{MinLat: minLat, MaxLat:
                    ↪  maxLat, MinLon: minLon, MaxLon: maxLon}
102         }
103         avgSearchingTime, err := rt.db.GetAvgSearchingTime(parkRequest)
104         if err != nil {
105                 var errorMessage types.ErrorMessage
106                 errorMessage.ErrorMessage = "can't get average parking time"
107                 ctx.Logger.WithError(err).Error(errorMessage)
108                 w.Header().Set("content-type", "application/json")
109                 w.WriteHeader(500)
110                 _ = json.NewEncoder(w).Encode(errorMessage)
111         } else {
112                 w.Header().Set("content-type", "application/json")
113                 _ = json.NewEncoder(w).Encode(avgSearchingTime)
114         }
115 }
```

**Listing 3.9.** Sviluppo API *getAvgSearchingTime*: fase di controllo dei dati in input

```
1 func (db *appdbimpl) GetAvgSearchingTime(parkRequest types.ParkRequest) (float64,
  ↪  error) {
2         var avgSearchingTime float64
3         var ret sql.NullFloat64
4
5         if parkRequest.StartTime.Before(*parkRequest.EndTime) {
```

```
6                 err := db.c.Get(&ret, "SELECT AVG(searchingtimesec) FROM parks
                  ↪  WHERE searchingtimesec > 0 AND parkstatus = \"park\" AND
                  ↪  (CAST(parktime AS TIME) BETWEEN ? AND ?) AND parklat >= ? AND
                  ↪  parklat <= ? AND parklon >= ? AND parklon <= ?",
                  ↪  parkRequest.StartTime.Format(types.DEFAULT_TIME_LAYOUT),
                  ↪  parkRequest.EndTime.Format(types.DEFAULT_TIME_LAYOUT),
                  ↪  parkRequest.GeoSquare.MinLat, parkRequest.GeoSquare.MaxLat,
                  ↪  parkRequest.GeoSquare.MinLon, parkRequest.GeoSquare.MaxLon)
7                 if err != nil {
8                         return 0, errors.Wrap(err, "get1 statement")
9                 }
10        } else {
11                err := db.c.Get(&ret, "SELECT AVG(searchingtimesec) FROM parks
                  ↪  WHERE searchingtimesec > 0 AND parkstatus = \"park\" AND
                  ↪  ((CAST(parktime AS TIME) BETWEEN ? AND \"23:59:59\") OR
                  ↪  (CAST(parktime AS TIME) BETWEEN \"00:00:00\" AND ?)) AND
                  ↪  parklat >= ? AND parklat <= ? AND parklon >= ? AND parklon <=
                  ↪  ?", parkRequest.StartTime.Format(types.DEFAULT_TIME_LAYOUT),
                  ↪  parkRequest.EndTime.Format(types.DEFAULT_TIME_LAYOUT),
                  ↪  parkRequest.GeoSquare.MinLat, parkRequest.GeoSquare.MaxLat,
                  ↪  parkRequest.GeoSquare.MinLon, parkRequest.GeoSquare.MaxLon)
12                if err != nil {
13                        return 0, errors.Wrap(err, "get2 statement")
14                }
15        }
16        if ret.Valid {
17                avgSearchingTime = ret.Float64
18        }
19        return avgSearchingTime, nil
20 }
```

**Listing 3.10.** Sviluppo API *getAvgSearchingTime*: recupero del tempo medio dal database.

### 3.2.2 Distanza media

L'idea e il funzionamento dell'API *getAvgSearchingDistance* sono esattamente identici all'API relativa al tempo medio appena esposta quindi non si andranno a spiegare nuovamente tutti i dettagli implementativi. L'obiettivo è quello di conoscere, in media, quanta distanza viene percorsa per trovare parcheggio in una determinata zona e in una determinata fascia oraria, in base ai dati di tutti gli utenti. L'unica differenza tra le due API è che, anziché utilizzare il campo *searchingtime*, viene utilizzato il *searchingdistance* che rappresenta la distanza percorsa per trovare un parcheggio e viene calcolato utilizzando i *trippoint* come per il tempo di ricerca.

## 3.3 Aggiunta di motivazioni sulla cancellazione di un match

Rimanendo sempre sui match, un'ulteriore aggiunta al progetto è stata la possibilità di aggiungere una motivazione dietro alla cancellazione di un match, sia da parte del giver, sia da parte del taker. Tali motivazioni vanno inviate nel momento in cui l'utente cancella volontariamente un match, per cui si è reso necessario effettuare delle modifiche alle API *removeMeTaker* e *removeMeGiver* che consentono di annullare il match in base al fatto che l'utente sia, rispettivamente, il taker o il giver. Le modifiche apportate alle due API sono equivalenti per cui ci si limiterà ad esporre quelle effettuate all'API *removeMeTaker*.
Il primo compito è stato quello di capire quali informazioni si vogliono ottenere

dall'utente che cancella un match. Dopo aver ragionato con il team di lavoro è stato deciso di raccogliere i seguenti dati:

- **CancelType**: si tratta di un intero che rappresenta la tipologia di cancellazione nonchè un motivo standard per cui l'utente ha deciso di cancellare il match (ad esempio il valore "2" corrisponde a "Utente parcheggiato in doppia fila");
- **CancelMessage**: si tratta di una stringa che rappresenta un messaggio facoltativo che l'utente può inviare insieme alla tipologia di cancellazione (ad esempio "Non ho potuto parcheggiare in quanto la macchina ostruiva il traffico").

Una volta definita la struttura di un **CancelInfo**, è stato necessario capire dove salvare queste informazioni. Dopo varie considerazioni, è stata aggiunta una nuova colonna *cancelinfo* nella tabella *matches_history* in maniera da poter associare ai vari match passati le eventuali informazioni sulla loro cancellazione. All'interno di questo campo viene salvato un oggetto json contenente i due campi *canceltype* e *cancelinfo*. Qui di seguito la migration utilizzata per l'aggiunta della nuova colonna:

```
1 ALTER TABLE matches_history ADD COLUMN IF NOT EXISTS cancelinfo longtext
  ↪  DEFAULT '';
```

**Listing 3.11.** Aggiunta nuova colonna "*cancelinfo*" nella tabella "*matches_history*".

È stato poi necessario modificare l'API per salvare le info sulla cancellazione. Inizialmente, l'API *removeMeTaker* prendeva un solo parametro *schedule* nel request body (una stringa in formato *"datetime"* che viene utilizzata per annullare il match all'orario specificato). Il campo schedule è stato mantenuto, ed è inoltre stato aggiunto un oggetto facoltativo *cancelinfo* (in formato json) che contiene i due campi sopra citati. La nuova struttura del body è la seguente:

```
1 {
2     "schedule": "2022-03-02T12:09:59Z",
3     "cancelinfo":
4         {
5             "canceltype": 2,
6             "cancelmessage": "Non ho potuto parcheggiare in quanto la
              ↪  macchina ostruiva il traffico"
7         }
8 }
```

**Listing 3.12.** Esempio di request body dell'API *removeMeTaker*.

È stata quindi modificata l'API per aggiungere un controllo sulla validità dei dati inseriti, in particolare, se viene inviato l'oggetto *cancelinfo* ci si assicura che il campo *canceltype* venga specificato in quanto obbligatorio (si ricorda che *cancelmessage* è facoltativo). Infine è stata aggiunta la query per il caricamento di tali informazioni sul database. Qui di seguito il codice delle modifiche apportate all'API *removeMeTaker*:

```
type CancelJsonStruct struct {
        CancelType    *int   `json:"canceltype" db:"canceltype"`
        CancelMessage string `json:"cancelmessage" db:"cancelmessage"`
}
...
type removeMeTakerBody struct {
        Schedule   *time.Time             `json:"schedule"`
        CancelInfo types.CancelJsonStruct `json:"cancelinfo"`
}
var takerBody removeMeTakerBody
err = json.NewDecoder(r.Body).Decode(&takerBody)
if err != nil {
        var errorMessage types.ErrorMessage
        errorMessage.ErrorMessage = "json not well formed"
        ctx.Logger.WithError(err).Error(errorMessage)
        w.Header().Set("content-type", "application/json")
        w.WriteHeader(400)
        _ = json.NewEncoder(w).Encode(errorMessage)
        return
} else if takerBody.Schedule == nil {
        var errorMessage types.ErrorMessage
        errorMessage.ErrorMessage = "empty schedule not allowed"
        ctx.Logger.WithError(err).Error(errorMessage)
        w.Header().Set("content-type", "application/json")
        w.WriteHeader(400)
        _ = json.NewEncoder(w).Encode(errorMessage)
        return
} else if takerBody.CancelInfo.CancelMessage != "" &&
↪  takerBody.CancelInfo.CancelType == nil {
        var errorMessage types.ErrorMessage
        errorMessage.ErrorMessage = "canceltype value is required if cancel message
        ↪  is specified"
        ctx.Logger.WithError(err).Error(errorMessage)
        w.Header().Set("content-type", "application/json")
        w.WriteHeader(400)
        _ = json.NewEncoder(w).Encode(errorMessage)
        return
}
...
if takerBody.CancelInfo.CancelType != nil {
        cancelInfo, err := json.Marshal(takerBody.CancelInfo)
        if err != nil {
                var errorMessage types.ErrorMessage
                errorMessage.ErrorMessage = "can't marshall cancel info file"
                ctx.Logger.WithError(err).Error(errorMessage)
                w.Header().Set("content-type", "application/json")
                w.WriteHeader(500)
                _ = json.NewEncoder(w).Encode(errorMessage)
                return
        }
        matchCancelInfo := types.MatchCancelInfo{HistoryId: *retMatchHistoryId,
        ↪  CancelInfo: string(cancelInfo)}
        _, err = rt.db.UpdateMatchCancelInfo(matchCancelInfo)
        if err != nil {
                var errorMessage types.ErrorMessage
                errorMessage.ErrorMessage = "can't update match cancel info"
                ctx.Logger.WithError(err).Error(errorMessage)
                w.Header().Set("content-type", "application/json")
                w.WriteHeader(500)
                _ = json.NewEncoder(w).Encode(errorMessage)
                return
        }
}
```

**Listing 3.13.** Modifica API *removeMeTaker*: fase di controllo dei dati nel body e creazione del json da salvare sul server.

Il corpo del body viene decodificato nella struct *removeMeTakerBody* e, se la decodifica va buon fine, si effettua il controllo sulla validità dei dati inseriti. Nel caso in cui il *canceltype* venga specificato (e quindi vengono inviate le motivazioni sulla cancellazione del match), allora viene creato un oggetto json da salvare sul database. Qui di seguito la query SQL che svolge il compito di salvare il dato:

```
func (db *appdbimpl) UpdateMatchCancelInfo(matchCancelInfo types.MatchCancelInfo)
↪  (int, error) {
        var message int

        tx, err := db.c.Beginx()
        if err != nil {
                return message, errors.Wrap(err, "begin1 transaction")
        }

        _, err = tx.Exec("UPDATE matches_history SET cancelinfo = ? WHERE historyid
        ↪  = ?", matchCancelInfo.CancelInfo, matchCancelInfo.HistoryId)
        if err != nil {
                _ = tx.Rollback()
                return message, errors.Wrap(err, "update1 statement")
        }

        return message, tx.Commit()
}
```

**Listing 3.14.** Query SQL *updateMatchCancelInfo*: caricamento sul database delle informazioni sulla cancellazione del match.

Infine è stato necessario modificare le API *getMatchesHistoryList* (che ritorna di tutti i match effettuati da un utente) e *getParkMatches* (che ritorna la lista dei match associati a un parcheggio), per ottenere anche i dati relativi alla cancellazione del match. Dopo aver aggiunto il campo *cancelinfo* nel json ritornato all'utente, è stato aggiunto un pezzo di codice per il recupero delle relative informazioni dal database:

```
...
for i, match := range matchesHistory {
        var cancelInfo string
        err = db.c.Get(&cancelInfo, "SELECT cancelinfo FROM matches_history WHERE
        ↪  historyid = ?", match.HistoryId)
        if err != nil && err != sql.ErrNoRows {
                return nil, errors.Wrap(err, "select3 statement")
        }
        if cancelInfo != "" {
                var cancelJson types.CancelJsonStruct
                err = json.Unmarshal([]byte(cancelInfo), &cancelJson)
                if err != nil {
                        return nil, errors.Wrap(err, "can't retrieve match cancel
                        ↪  info")
                } else {
                        matchesHistory[i].CancelInfo.CancelType =
                        ↪  cancelJson.CancelType
```

```
15                          matchesHistory[i].CancelInfo.CancelMessage =
                            ↪  cancelJson.CancelMessage
16                  }
17          }
18  }
```

**Listing 3.15.** Query SQL *getMatchesHistoryList*: recupero delle informazioni sulla cancellazione del match dal database. Una volta recuperata la stringa dal db, si trasforma in un oggetto json che viene associato al rispettivo match nella lista da inviare all'utente.

## 3.4 Aggiornamento della documentazione

Durante la creazione delle nuove API precedentemente esposte, è stato necessario aggiornare la documentazione. In precedenza, la documentazione si trovava all'interno di un file presente sulla piattaforma *GitLab* da aggiornare manualmente. Questa soluzione si è dimostrata obsoleta a causa della difficoltà nel tenere continuamente traccia dei cambiamenti apportati alle API, per cui si è scelto di adottare una nuova strategia. Tutta la documentazione è stata riscritta in formato OpenAPI [17] (già introdotto nelle precedenti sezioni), in un file presente all'interno della cartella *"doc/"*. Ogni volta che si effettua una modifica a un'API, la si può documentare immediatamente in tale file e alla successiva *build* dell'applicazione, questa viene automaticamente aggiornata e resa disponibile a tutti gli sviluppatori di GeneroCity.

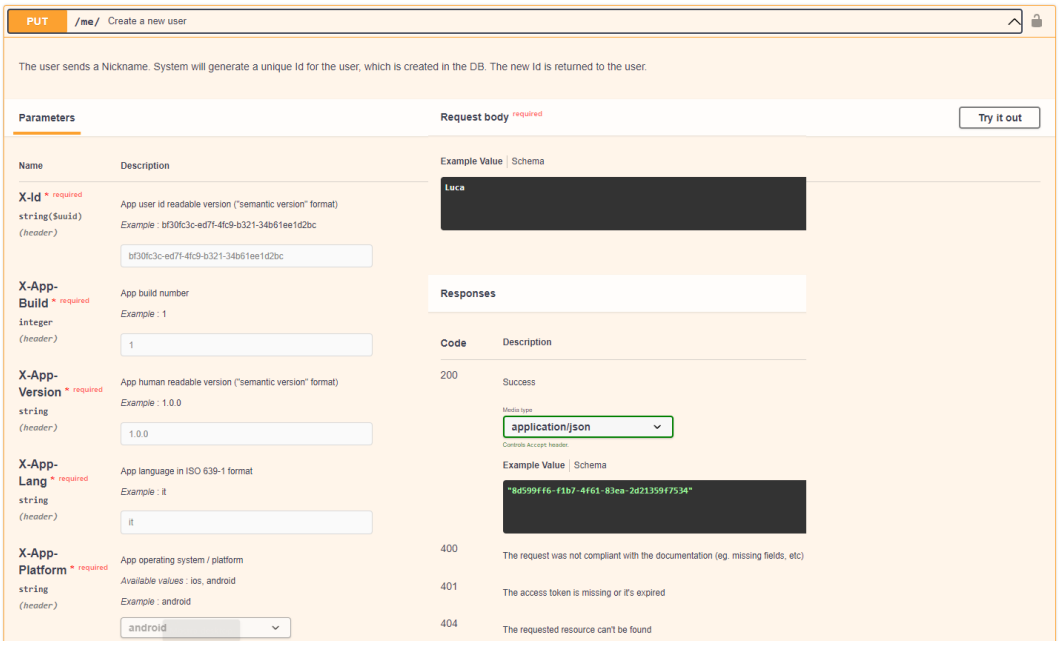



**Figura 3.4.** Esempio di documentazione di un API di GeneroCity in formato OpenAPI. In alto figura l'url che permette di invocare la funzione associata. In basso a sinistra i vari parametri da inserire nell'header. In basso a destra il request body e i possibili codici di stato nella risposta.

# Capitolo 4

# Trasferimento dei file json dal database al server Minio

Un ulteriore compito che ho dovuto affrontare è stata la sistemazione del database con il trasferimento di alcuni dati su un server esterno. In GeneroCity, buona parte dei dati raccolti vengono inseriti nel database. Nel momento in cui si effettuava un parcheggio, oltre a dati inerenti al parcheggio stesso (ad esempio latitudine, longitudine, orario, etc..) venivano salvati anche dei dati utilizzati da altri tirocinanti di GeneroCity per allenare dei modelli di *machine learning*. All'interno della tabella *parks* vi erano due colonne che presentavano dei problemi:

- **parksamples**: colonna che conteneva dei campionamenti del sensore di movimento raccolti durante il parcheggio dell'auto. Il problema con questi dati era che occupavano molto spazio (fino a decine di megabytes per ogni parcheggio effettuato) a tal punto che circa il 90% dello spazio utilizzato era destinato ai parksamples;
- **driverbehaviour**: colonna che conteneva delle informazioni legate a manovre effettuate con l'auto. Il problema con questi dati era che venivano raccolti nel momento in cui si effettuava un parcheggio, nonostante la loro indipendenza da quest'ultimo.

È quindi sorta la necessità di trovare una soluzione ai problemi sopra elencati che è culminata con il trasferimento dei suddetti dati sul server **MinIO**, un server di archiviazione cloud scritto in Go [18]. Nelle sezioni che seguono verranno spiegate le soluzioni adottate nel dettaglio.

## 4.1 Migrazione dei ParkSamples

Per quanto riguarda i **parksamples**, la soluzione è stata quella di trasferire tutti i dati che li riguardavano presenti nel database, sul *cloud server MinIO* [18]. Prima di effettuare il trasferimento dei dati, sono state rimosse dal codice tutte le parti in cui questi dati venivano caricati sul database. È quindi stata modificata l'API *carPark* per permettere, durante il parcheggio dell'auto, di caricare il *parksample* nel *body* della richiesta, sul server MinIO.

```
...
if len(carPark.ParkSamples) != 0 {
                parkSamples, err := json.Marshal(carPark.ParkSamples)
                if err != nil {
                        var errorMessage types.ErrorMessage
                        errorMessage.ErrorMessage = "can't create parksamples file"
                        ctx.Logger.WithError(err).Error(errorMessage)
                        w.Header().Set("content-type", "application/json")
                        w.WriteHeader(500)
                        _ = json.NewEncoder(w).Encode(errorMessage)
                        return
                }
                _, err = rt.minio.PutObject(context.TODO(), rt.minioBucket,
                ↪  fmt.Sprintf("parksamples/\%d.json", parkId),
                ↪  bytes.NewReader(parkSamples), int64(len(parkSamples)),
                ↪  minio.PutObjectOptions{})
                if err != nil {
                        var errorMessage types.ErrorMessage
                        errorMessage.ErrorMessage = "can't upload park samples
                        ↪  file"
                        ctx.Logger.WithError(err).Error(errorMessage)
                        w.Header().Set("content-type", "application/json")
                        w.WriteHeader(500)
                        _ = json.NewEncoder(w).Encode(errorMessage)
                        return
                }
        }
...
```

**Listing 4.1.** Codice che permette il caricamento dei parksamples sul server MinIO

Il seguente frammento di codice, controlla il contenuto del parametro "parksamples" passato in input e se valido ne ricava un file json tramite l'operazione di marshal. Viene quindi caricato un nuovo oggetto sul server con il nome del parcheggio associato al dato il cui contenuto è il json precedentemente creato.

Il compito successivo è stato quello di creare un eseguibile per trasferire il valore dei parksamples contenuti nel database, al server MinIO.

```
type datasetConfiguration struct {
        Config struct {
                Path string `conf:"default:/conf/config.yml"`
        }
        Log      apilogger.LogSettings
        Datasets struct {
                Host   string `conf:""`
                Access string `conf:""`
                Secret string `conf:""`
                Secure bool   `conf:"default:true"`
                Bucket string `conf:""`
        }
        Database struct {
                Driver string `conf:"default:mysql"`
                DSN    string `conf:"default:-"`
        }
}

func loadConfiguration() (datasetConfiguration, error) {
```

```
20          // Create configuration defaults
21          var cfg datasetConfiguration
22
23          // Try to load configuration from environment variables and command line
            ↪  switches
24          if err := conf.Parse(os.Args[1:], "CFG", &cfg); err != nil {
25                  if errors.Is(err, conf.ErrHelpWanted) {
26                          usage, err := conf.Usage("CFG", &cfg)
27                          if err != nil {
28                                  return cfg, fmt.Errorf("generating config usage:
                                    ↪  %w", err)
29                          }
30                          fmt.Println(usage) //nolint:forbidigo
31                          return cfg, conf.ErrHelpWanted
32                  }
33                  return cfg, fmt.Errorf("parsing config: %w", err)
34          }
35
36          // Override values from YAML if specified and if it exists (useful in
            ↪  k8s/compose)
37          fp, err := os.Open(cfg.Config.Path)
38          if err != nil && !os.IsNotExist(err) {
39                  return cfg, fmt.Errorf("can't read the config file, while it exists:
                    ↪  %w", err)
40          } else if err == nil {
41                  yamlFile, err := ioutil.ReadAll(fp)
42                  if err != nil {
43                          return cfg, fmt.Errorf("can't read config file: %w", err)
44                  }
45                  err = yaml.Unmarshal(yamlFile, &cfg)
46                  if err != nil {
47                          return cfg, fmt.Errorf("can't unmarshal config file: %w",
                            ↪  err)
48                  }
49                  _ = fp.Close()
50          }
51          return cfg, nil
52  }
```

**Listing 4.2.** Codice che carica la configurazione necessaria per eseguire lo script.

```
1   ...
2           // Retrieve data from DB
3           type parkSamples struct {
4                   ParkId      int    `db:"parkid"`
5                   ParkSamples string `db:"parksamples"`
6           }
7
8           var parkSamplesList []parkSamples
9           err = db.Select(&parkSamplesList, "SELECT parkid, parksamples FROM parks
            ↪  WHERE parkstatus != 'unpark'")
10          if err != nil {
11                  panic(errors.Wrap(err, "Can't get data from db"))
12          }
13
14          //Upload objects
15          for _, obj := range parkSamplesList {
16                  ps := []byte(obj.ParkSamples)
17                  _, err = dest.PutObject(context.TODO(), cfg.Minio.Bucket,
                    ↪  fmt.Sprintf("parksamples/%d.json", obj.ParkId),
                    ↪  bytes.NewReader(ps), int64(len(ps)), minio.PutObjectOptions{})
```

```
18                 if err != nil {
19                         logger.WithError(err).Error("error while uploading a file
                           ↪  to server")
20                         return errors.Wrap(err, "can't upload file to server")
21                 }
22         }
23
24         return nil
25 ...
```

**Listing 4.3.** Codice che recupera i dati relativi ai parksamples dal database e li carica sul server MinIO. Per semplicità è stata esclusa la parte di codice che si occupa di inizializzare le componenti necessarie.

Questo eseguibile è composto da due parti: un file che definisce e carica la configurazione necessaria al funzionamento dell'eseguibile e un ulteriore file che inizializza le componenti necessarie (in questo caso database e server MinIO) ed esegue un determinato compito. Nel primo frammento viene definita la configurazione necessaria che viene inizializzata a partire dai valori contenuti in variabili d'ambiente o da comando. Nel secondo frammento di codice vengono recuperate tutte le coppie *(parkid, parksamples)* dal database e successivamente, viene creato un nuovo oggetto *"parkid.json"* (dove parkid è l'id del parcheggio) con all'interno il valore del parksamples. Per eseguire tale script basta lanciarlo tramite comando. Qui di seguito un esempio su come lanciarlo in locale utilizzando gli strumenti messi a disposizione sul container *Docker*:

```
1 go run ./cmd/migrate-parksamples --minio-host 127.0.0.1:8065 --minio-bucket
  ↪  generocity --minio-access minio-access --minio-secret minio-secret
  ↪  --minio-secure false --database-driver "mysql" --database-dsn
  ↪  "root:root@tcp(127.0.0.1:3306)/gc?parseTime=true&loc=Europe\%2FRome"
  ↪  --log-level info
```

**Listing 4.4.** Comando per l'avvio dell'eseguibile "migrate-parksamples".

## 4.2 Migrazione dei DriverBehaviour

Per quanto riguarda i **driverbehaviour**, bisognava separare la logica delle manovre da quella dei parcheggi quindi, prima effettuare il trasferimento dai dati dal database al server, è stato oppurtuno creare una nuova tabella "cars_maneuvers" nel database per migrare tutti i dati in un ambiente separato.

```
CREATE TABLE `cars_maneuvers` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `userid` varchar(36) NOT NULL DEFAULT '',
  `cid` varchar(36) NOT NULL DEFAULT '',
  `datetime` datetime DEFAULT current_timestamp(),
  PRIMARY KEY (`id`),
  KEY `cars_maneuvers_ibfk_1` (`userid`),
  KEY `cars_maneuvers_ibfk_2` (`cid`),
  CONSTRAINT `cars_maneuvers_ibfk_1` FOREIGN KEY (`userid`) REFERENCES
  ↪  `users` (`id`) ON UPDATE CASCADE,
  CONSTRAINT `cars_maneuvers_ibfk_2` FOREIGN KEY (`cid`) REFERENCES `cars`
  ↪  (`cid`) ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

**Listing 4.5.** Nuova tabella con un id incrementale, l'id di utente e auto che hanno eseguito la manovra e istante di caricamento sul database.

È stato quindi creato un eseguibile per trasferire tutti i dati dalla tabella *parks* alla nuova tabella appena illustrata. Per ogni colonna nella tabella *parks* con campo "driverbehaviour" non vuoto e valido (alcuni dati risultavano corrotti e quindi non sono stati copiati nella nuova tabella), è stata aggiunta una colonna all'interno della nuova tabella *cars_ maneuvers*. Come si può facilmente notare dal codice, l'unico campo non presente nella nuova tabella è proprio quello riservato ai driverbehaviour. Questi non vengono caricati nel database ma direttamente sul server MinIO: per ogni colonna valida aggiunta nel database, viene creato un nuovo oggetto con all'interno il json che contiene le informazioni sulle manovre. Per collegare le colonne del database al rispettivo oggetto sul server viene utilizzato il nome dell'oggetto stesso infatti, in fase di creazione, gli viene assegnato come nome l'id della colonna presente sul database.

```
...
type CarManeuver struct {
        UserId          uuid.UUID `json:"userid" db:"userid"`
        Cid             uuid.UUID `json:"cid" db:"cid"`
        ParkTime        time.Time `json:"parktime" db:"parktime"`
        DriverBehaviour string    `json:"driverbehaviour" db:"driverbehaviour"`
}
type DriverBehaviour struct {
        Type     string `json:"type"`
        Sequence string `json:"sequence"`
}

logger.Println("Started : Retrieving old \"driverbehavior\" data from db")
var carManeuverList []CarManeuver
err = db.Select(&carManeuverList, "SELECT usedby as userid, cid, driverbehaviour,
  ↪  parktime FROM parks WHERE parkstatus != 'unpark' AND driverbehaviour != ''")
if err != nil {
        panic(errors.Wrap(err, "Can't get data from db"))
}

tx, err := db.Beginx()
if err != nil {
        return errors.Wrap(err, "can't begin a new transaction")
}
for _, obj := range carManeuverList {
```

```
25          var driverBehaviour DriverBehaviour
26          err = json.Unmarshal([]byte(obj.DriverBehaviour), &driverBehaviour)
27          if err != nil || driverBehaviour.Type == "" || driverBehaviour.Sequence ==
            ↪  "" {
28                  logger.Println("Item discarded: found a non valid driver behavior
                    ↪  field")
29          } else {
30                  logger.Println("Item ok: copying a valid driver behavior to
                    ↪  cars_maneuvers table")
31                  res, err = tx.Exec("INSERT INTO cars_maneuvers SET userid = ?, cid
                    ↪  = ?, datetime = ?", obj.UserId, obj.Cid, obj.ParkTime)
32                  if err != nil {
33                          _ = tx.Rollback()
34                          return errors.Wrap(err, "Error while copying an item")
35                  }
36                  ps := []byte(obj.DriverBehaviour)
37              _, err = dest.PutObject(context.TODO(), cfg.Datasets.Bucket,
                ↪  fmt.Sprintf("carmaneuvers/driverbehaviour/\%d.json",
                ↪  res.LastInsertId()), bytes.NewReader(ps), int64(len(ps)),
                ↪  minio.PutObjectOptions{})
38          }
39  }
40  err = tx.Commit()
41  if err != nil {
42          return errors.Wrap(err, "Error during commit")
43  }
44  ...
```

**Listing 4.6.** Eseguibile che prende tutti i dati necessari ad effettuare la migrazione dei *driverbehaviour* dalla tabella *parks* alla tabella *cars_maneuvers*. Il json che rappresenta la manovra viene direttamente caricato sul server MinIO con nome l'id della colonna che mantiene le informazioni sull'utente e la macchina che l'hanno effettuata.

E' stato poi necessario creare un'API per caricare nuove manovre. La logica dietro l'API è la seguente:

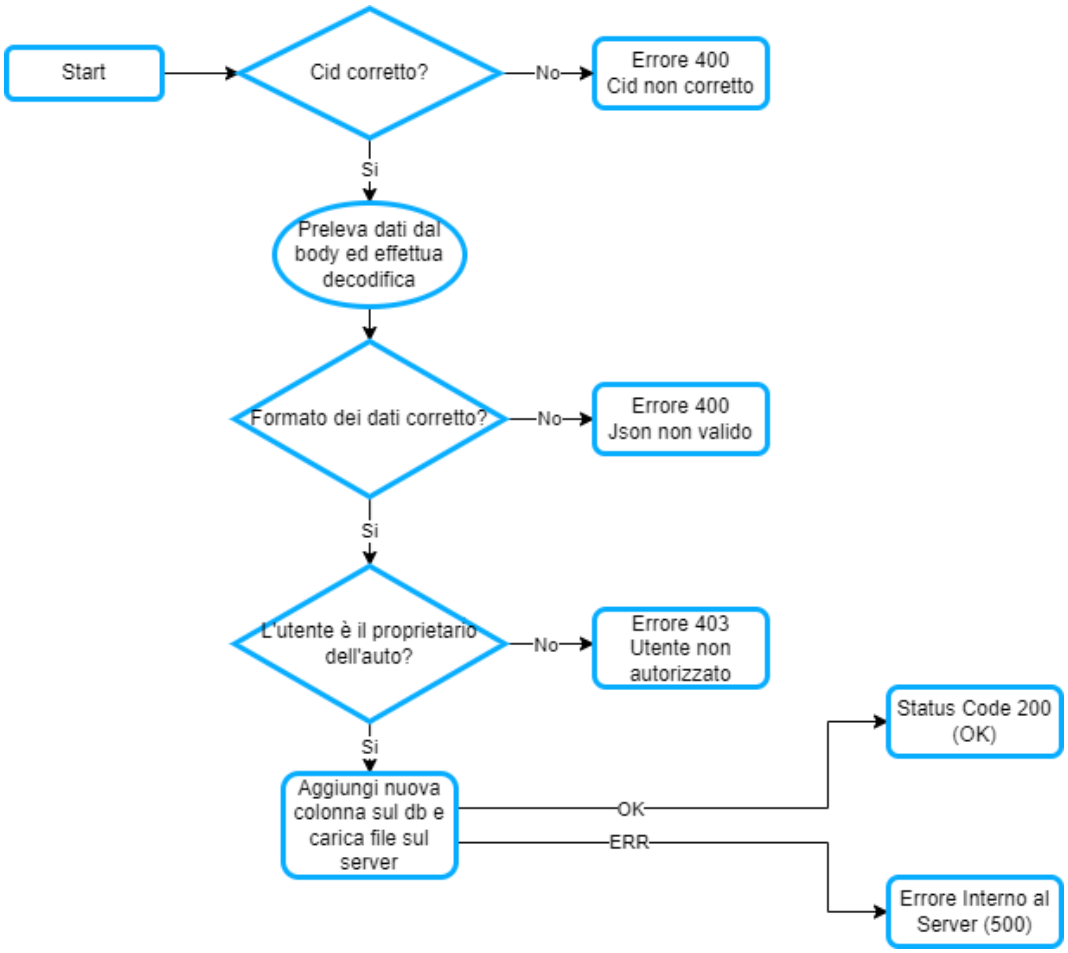



**Figura 4.1.** Progettazione API *createCarManeuver*

La seguente API prende in input l'id dell'utente e l'id della macchina (*cid*) e un *body* contenente un oggetto json che rappresenta una manovra. Viene inizialmente controllata la validità dei dati inviati quindi se il cid è valido e se il json nel body rappresenta realmente una manovra (questo per evitare la collezione di dati non validi). Infine si controlla che l'utente sia il proprietario dell'auto e sia quindi autorizzato ad aggiungere la manovra. Se tutto va a buon fine viene aggiunta una nuova colonna nel database con campi l'id dell'utente, l'id della macchina, l'istante di aggiunta della manovra e un id incrementale associato alla colonna. Viene inoltre aggiunto un oggetto nel server MinIO contenente il json nel body. Qui di seguito il codice dell'API:

```
1 ...
2 func (rt *_router) createCarManeuver(w http.ResponseWriter, r *http.Request, ps
  ↪ httprouter.Params, ctx reqcontext.RequestContext) {
3         cidUuid, err := uuid.FromString(ps.ByName("cid"))
4         if err != nil {
5                 var errorMessage types.ErrorMessage
6                 errorMessage.ErrorMessage = "cid not well formed"
```

```
7                ctx.Logger.WithError(err).Error(errorMessage)
8                w.Header().Set("content-type", "application/json")
9                w.WriteHeader(400)
10               _ = json.NewEncoder(w).Encode(errorMessage)
11               return
12       }
13
14       var carManeuver types.CarManeuver
15       err = json.NewDecoder(r.Body).Decode(&carManeuver)
16       if err != nil {
17               var errorMessage types.ErrorMessage
18               errorMessage.ErrorMessage = "json not well formed"
19               ctx.Logger.WithError(err).Error(errorMessage)
20               w.Header().Set("content-type", "application/json")
21               w.WriteHeader(400)
22               _ = json.NewEncoder(w).Encode(errorMessage)
23               return
24       } else if carManeuver.DriverBehaviour == "" {
25               var errorMessage types.ErrorMessage
26               errorMessage.ErrorMessage = "driver behaviour param is mandatory"
27               ctx.Logger.WithError(err).Error(errorMessage)
28               w.Header().Set("content-type", "application/json")
29               w.WriteHeader(400)
30               _ = json.NewEncoder(w).Encode(errorMessage)
31               return
32       }
33       carManeuver.Cid = cidUuid
34       carManeuver.UserId = ctx.UserId
35
36       var driverBehaviour types.DriverBehaviour
37       err = json.Unmarshal([]byte(carManeuver.DriverBehaviour), &driverBehaviour)
38       if err != nil {
39               var errorMessage types.ErrorMessage
40               errorMessage.ErrorMessage = "can't unmarshall file"
41               ctx.Logger.WithError(err).Error(errorMessage)
42               w.Header().Set("content-type", "application/json")
43               w.WriteHeader(500)
44               _ = json.NewEncoder(w).Encode(errorMessage)
45               return
46       } else if driverBehaviour.Type == "" || driverBehaviour.Sequence == "" {
47               var errorMessage types.ErrorMessage
48               errorMessage.ErrorMessage = "driver behaviour type and sequence are
                 ↪  mandatory"
49               ctx.Logger.WithError(err).Error(errorMessage)
50               w.Header().Set("content-type", "application/json")
51               w.WriteHeader(400)
52               _ = json.NewEncoder(w).Encode(errorMessage)
53               return
54       }
55
56       b, err := rt.db.IsOwner(types.Driver{Id: carManeuver.UserId, Cid:
         ↪  carManeuver.Cid})
57       if err != nil {
58               var errorMessage types.ErrorMessage
59               errorMessage.ErrorMessage = "can't check the ownership"
60               ctx.Logger.WithError(err).Error(errorMessage)
61               w.Header().Set("content-type", "application/json")
62               w.WriteHeader(500)
63               _ = json.NewEncoder(w).Encode(errorMessage)
64               return
65       } else if !b {
66               var errorMessage types.ErrorMessage
67               errorMessage.ErrorMessage = "user not authorized"
68               ctx.Logger.WithError(err).Error(errorMessage)
```

```
69                  w.Header().Set("content-type", "application/json")
70                  w.WriteHeader(403)
71                  _ = json.NewEncoder(w).Encode(errorMessage)
72                  return
73          }
74
75          id, err := rt.db.CreateCarManeuver(carManeuver)
76          if err != nil {
77                  var errorMessage types.ErrorMessage
78                  errorMessage.ErrorMessage = "can't create a new driver behaviour"
79                  ctx.Logger.WithError(err).Error(errorMessage)
80                  w.Header().Set("content-type", "application/json")
81                  w.WriteHeader(500)
82                  _ = json.NewEncoder(w).Encode(errorMessage)
83          } else {
84                  driverBehaviour := []byte(carManeuver.DriverBehaviour)
85                  _, err = rt.minio.PutObject(context.TODO(), rt.minioBucket,
                    ↪  fmt.Sprintf("carmaneuvers/driverbehaviour/%d.json", id),
                    ↪  bytes.NewReader(driverBehaviour), int64(len(driverBehaviour)),
                    ↪  minio.PutObjectOptions{})
86                  if err != nil {
87                          var errorMessage types.ErrorMessage
88                          errorMessage.ErrorMessage = "can't create driver behaviour
                            ↪  file"
89                          ctx.Logger.WithError(err).Error(errorMessage)
90                          w.Header().Set("content-type", "application/json")
91                          w.WriteHeader(500)
92                          _ = json.NewEncoder(w).Encode(errorMessage)
93                          return
94                  }
95                  w.Header().Set("content-type", "application/json")
96                  _ = json.NewEncoder(w).Encode(id)
97          }
98  }
```

**Listing 4.7.** Sviluppo API createCarManeuver: fase di controllo dei dati in input e caricamento oggetto sul server MinIO.

```
func (db *appdbimpl) CreateCarManeuver(carManeuver types.CarManeuver) (int, error) {
        var message int

        tx, err := db.c.Beginx()
        if err != nil {
                return message, errors.Wrap(err, "begin1 transaction")
        }

        res, err := tx.Exec("INSERT INTO cars_maneuvers SET userid = ?, cid = ?",
        carManeuver.UserId, carManeuver.Cid)
        if err != nil {
                _ = tx.Rollback()
                return message, errors.Wrap(err, "insert1 statement")
        }

        lastId, err := res.LastInsertId()
        if err != nil {
                _ = tx.Rollback()
                return 0, errors.Wrap(err, "get1 last insert id")
        }

        return int(lastId), tx.Commit()
}
```

**Listing 4.8.** Sviluppo API createCarManeuver: aggiunta nuova colonna sul database

Una volta sistemati i *driverbehaviour* nella nuova tabella e aver rimosso eventuali dati corrotti, è stato necessario creare un ulteriore eseguibile per recuperare i file salvati sul server MinIO e creare due file in formato CSV necessari al modello di *machine learning* sopra citato. Un file contiene solamente le informazioni sulle manovre contenute sul server MinIO. L'altro file contiene anche le info presenti sul database quindi id dell'utente, id dell'auto e istante di creazione della manovra.

| id | userid | cid | datetime | driverbehaviour |
|---|---|---|---|---|
| 1 | a77a6c58-125e-49b2-9648-06be95dc49b8 | 4bed6fea-954f-41a7-84ae-4637dc9e14ba | 2022-02-01 17:31:24 +0100 CET | {"type": "HarshRight", "sequence": "[0.0=[3.3921003E-4, -0.0013334 |
| 2 | a77a6c58-125e-49b2-9648-06be95dc49b8 | 4bed6fea-954f-41a7-84ae-4637dc9e14ba | 2022-02-01 17:31:24 +0100 CET | {"type":"SafeAcceleration","sequence":"{0.1=[0.003473118, 0.29808 |
| 3 | a77a6c58-125e-49b2-9648-06be95dc49b8 | 4bed6fea-954f-41a7-84ae-4637dc9e14ba | 2022-02-01 17:31:24 +0100 CET | {"type":"SafeAcceleration","sequence":"{1.91=[5.904995E-4, -2.3102 |
| 4 | a77a6c58-125e-49b2-9648-06be95dc49b8 | 4bed6fea-954f-41a7-84ae-4637dc9e14ba | 2022-02-01 17:31:24 +0100 CET | {"type":"SafeAcceleration","sequence":"{0.1=[0.005340852, -0.01820 |
| 5 | 2f05e2ab-6d64-42e7-b8e2-7f4357133183 | 992c000d-2a88-454e-8c95-66ec89f3c32c | 2021-04-27 18:04:52 +0200 CEST | {"type":"SafeAcceleration","sequence":"{0.0=[0.0, 0.0, 0.0, 0.0, 0.0], |
| 6 | 9d71efe6-011a-429a-8f86-05be94089aa2 | d9a27a2b-aeb6-418b-8de7-a76ad29d829f | 2021-04-28 21:17:07 +0200 CEST | {"type":"SafeAcceleration","sequence":"{0.0=[0.0, 0.0, 0.0, 0.0, 0.047884 |

**Figura 4.2.** Struttura del file CSV con tutte le informazioni sulla manovra.

È stato fatto in modo che lo script venga eseguito ogni notte in modo da aggiornare i file CSV con nuove eventuali manovre aggiunte durante la giornata. Si mostra ora il codice contenuto nel main dell'eseguibile e che crea e carica i due file (si esclude il codice che carica la configurazione):

```
...
// Retrieve data from DB
type carsManeuvers struct {
        Id              int       `db:"id"`
        UserId          uuid.UUID `db:"userid"`
        Cid             uuid.UUID `db:"cid"`
        DateTime        time.Time `db:"datetime"`
        DriverBehaviour string    `db:"driverbehaviour"`
```

```
}

logger.Debug("DB: Retrieving driverbehaviour information from db")
var carsManeuversList []carsManeuvers
err = db.Select(&carsManeuversList, "SELECT * FROM cars_maneuvers")
if err != nil {
        panic(errors.Wrap(err, "Can't get data from db"))
}

logger.Debug("MinIO: Retrieving driverbehaviour files from server")
serverFiles := make(map[int]string)
for obj := range source.ListObjects(ctx, cfg.Minio.Bucket,
↪  minio.ListObjectsOptions{
        Prefix:    "carmaneuvers/driverbehaviour/",
        Recursive: false,
}) {
        fileobj, err := source.GetObject(ctx, cfg.Minio.Bucket, obj.Key,
        ↪  minio.GetObjectOptions{})
        if err != nil {
                return err
        }
        f, err := io.ReadAll(fileobj)
        if err != nil {
                return errors.Wrap(err, "MinIO: Error while reading file")
        }
        s := string(f)
        fileid, _ := strconv.Atoi(strings.Split(strings.Split(string(obj.Key),
        ↪  "/")[2], ".")[0])
        serverFiles[fileid] = s
        logger.Debug("MinIO: Found file with id " + strconv.Itoa(fileid))
}

// Car Maneuvers converter (struct to csv)
logger.Debug("CSV: Creating csv files")
var cmb, drb bytes.Buffer
cm := csv.NewWriter(&cmb)
dr := csv.NewWriter(&drb)
_ = cm.Write([]string{"id", "userid", "cid", "driverbehaviour", "datetime"})
_ = dr.Write([]string{"driverbehaviour"})
for _, row := range carsManeuversList {
        if row.DriverBehavior == "" {
                row.DriverBehavior = serverFiles[row.Id]
        }
        _ = cm.Write([]string{strconv.Itoa(row.Id), row.UserId.String(),
        ↪  row.Cid.String(), row.DriverBehavior, row.DateTime.String()})
        _ = dr.Write([]string{row.DriverBehavior})
}
cm.Flush()

//Upload objects
logger.Debug("Minio: Uploading files to server")
csvcm := cmb.Bytes()
_, err = dest.PutObject(context.TODO(), cfg.Datasets.Bucket,
↪  "carmaneuvers/carsmaneuvers.csv", bytes.NewReader(csvcm), int64(len(csvcm)),
↪  minio.PutObjectOptions{})
if err != nil {
        logger.WithError(err).Error("error while uploading a file to server")
        return errors.Wrap(err, "can't upload file to server")
}

csvdr := drb.Bytes()
_, err = dest.PutObject(context.TODO(), cfg.Datasets.Bucket,
↪  "carmaneuvers/driverbehaviour.csv", bytes.NewReader(csvdr), int64(len(csvdr)),
↪  minio.PutObjectOptions{})
```

```
65 if err != nil {
66         logger.WithError(err).Error("error while uploading a file to server")
67         return errors.Wrap(err, "can't upload file to server")
68 }
69
70 logger.Debug("Log: Operation ended successfully")
71 ...
```

**Listing 4.9.** Eseguibile che ottiene il valore delle colonne nella tabella cars_maneuvers, vi associa i dati relativi alle manovre presenti sul server MinIO e crea i file CSV necessari al modello di machine learning. I suddetti file vengono poi caricati sul server MinIO. Per semplicità, si esclude la parte di codice che carica la configurazione e inizializza le componenti necessarie.

Dopo aver terminato la sistemazione di *parksamples* e *driverbehaviour*, è stato necessario togliere le relative colonne dalla tabella *parks* attraverso una *migration*:

```
1 ALTER TABLE parks DROP COLUMN IF EXISTS driverbehaviour;
2 ALTER TABLE parks DROP COLUMN IF EXISTS parksamples;
```

**Listing 4.10.** Query in linguaggio SQL per l'eliminazione delle colonne *parksamples* e *driverbehaviour* dalla tabella *parks*.

# Capitolo 5

# Ulteriori lavori svolti

In questo capitolo vengono esposti altri lavori che ho avuto modo di svolgere durante il tirocinio. L'obiettivo è stato quello di risolvere dei problemi che impedivano la prosecuzione del lavoro ad altri collaboratori di GeneroCity.

## 5.1 Modifica alla gestione delle notifiche sull'aggiornamento della posizione del taker

Un'ulteriore modifica al progetto ha riguardato la gestione delle notifiche relative alla posizione del taker durante il viaggio verso il giver. La posizione del taker viene aggiornata mediante un'API *updateTakerPosition* che ottiene informazioni quali latitudine, longitudine e istante corrente e le invia al giver che può così visualizzare la posizione del taker sulla mappa. In precedenza, per inviare la posizione del taker al giver con un dispositivo **Android**, veniva utilizzata una notifica inviata ogni tre secondi utilizzando apposite librerie messe a disposizione da *Google* [19]. Questo metodo presentava delle criticità: Google limita deliberatamente il numero delle notifiche inviate entro un certo periodo, se queste riguardano lo stesso "argomento" (in questo caso l'aggiornamento della posizione del taker) che viene specificato in un parametro (*collapseKey*) nel momento dell'invio della notifica. Questo portava a un blocco dell'invio delle notifiche con la conseguente impossibilità di ottenere la posizione del taker.
Una soluzione, è stata quella di sostituire le notifiche con dei *data payload* ossia dei messaggi di dati che contengono solo un campo dati senza l'invio di una notifica. Questo ha permesso al taker di inviare la sua posizione senza limiti sul numero di aggiornamenti. Qui di seguito il codice che ha riguardato la modifica:

```
...
if giverNotificationInfo.Os == "Android" {
customDataMap := map[string]interface{}{
        "taker-id":      takerPosition.DriverId.String(),
        "taker-cid":     takerPosition.Cid.String(),
        "giver-cid":     giverNotificationInfo.GiverCid.String(),
        "lat":           fmt.Sprintf("%f", *takerPosition.ParkLat),
        "lon":           fmt.Sprintf("%f", *takerPosition.ParkLon),
        "eta":           fmt.Sprintf("%f", *takerPosition.Eta),
        "schedule":      takerPosition.Schedule.String(),
        "x-gc-category": types.NOTIFICATION_TAKERPOSITION,
}
var customMessage fcm.Message
customMessage.Data = customDataMap
customMessage.To = giverNotificationInfo.PushToken
AndroidCustomNotification := mobilepush.CustomNotification{FirebaseNotification:
↪  &customMessage}
_, _, err = rt.push.SendCustomPush(AndroidCustomNotification)
if err != nil {
        ctx.Logger.WithError(err).Warning("can't send takerposition notification")
        return
}
```

**Listing 5.1.** Sviluppo API updateTakerPosition: aggiornamento sistema di invio della posizione del taker con la creazione di un *messaggio push*

# Conclusioni

In questa relazione è stato descritto tutto il lavoro svolto durante l'attività di tirocinio. Nel momento in cui ho iniziato a lavorare al progetto, l'applicazione **GeneroCity** era già funzionante e includeva buona parte delle funzionalità previste durante la sua progettazione. Il lavoro di ingegnerizzazione delle API che ho avuto modo di svolgere in questi mesi, è stato essenziale per rafforzare maggiormente quelli che sono i servizi offerti da GeneroCity. In particolare, è stata aggiunta la possibilità di verificare se un parcheggio è stato ottenuto o lasciato grazie a un match effettuato con un altro utente, un'informazione in principio mancante e fondamentale per gli scopi dell'applicazione. Oltre a questo, sono state aggiunte numerose nuove informazioni ricavabili dall'esperienza degli utenti, come il numero di match andati (o meno) a buon fine e il tempo e distanza necessari a ottenere un parcheggio in una determinata fascia oraria e luogo geografico. Questi dati, oltre a rappresentare un inizio per la costruzione di un robusto sistema di statistiche, possono aiutare sviluppatori di GeneroCity e altri ricercatori nel campo dello *smart parking* a ideare tecnologie sempre più efficienti che permettano di migliorare la viabilità all'interno dei centri urbani e facilitare la ricerca di un parcheggio.
Inoltre, ho avuto modo di modificare la logica di alcune API esistenti, portando a una ristrutturazione di alcune risorse lato *backend*. La sistemazione del database e la conseguente migrazione di tutti i file json nel server cloud MinIO, si è rivelata necessaria per permettere agli sviluppatori attuali e futuri di GeneroCity di lavorare in un ambiente pulito, efficace ed efficiente.

Per concludere, vorrei lasciare qualche idea per eventuali sviluppi futuri. Data la possibilità di capire se un parcheggio è stato effettuato grazie a un match, si potrebbero aggiungere delle statistiche sul carburante consumato sfruttando o meno la funzionalità di scambio posto, anche tenendo conto del risparmio economico e dell'impatto ambientale relativo alla riduzione di emissioni di $CO_2$. Si potrebbe anche aggiungere una classifica sugli utenti che hanno effettuato più parcheggi grazie a un match andato a buon fine. Questo permetterebbe di aumentare ancora di più il coinvolgimento degli utenti e incentivare l'uso dell'applicazione.

# Ringraziamenti

Sono tante le persone che ho incontrato durante questo percorso, a partire dai professori fino ad arrivare ai colleghi di corso con cui ho avuto modo di condividere questa fantastica esperienza.

Vorrei ringraziare il prof. Emanuele Panizzi che mi ha seguito costantemente in questo percorso fornendomi tutte le conoscenze necessarie a svolgere il tirocinio nel miglior modo possibile. Lo ringrazio per la sua gentilezza, disponibilità e immensa professionalità con cui svolge il proprio lavoro.

Vorrei ringraziare Alba, Enrico e tutti gli altri collaboratori di GeneroCity che mi hanno permesso di crescere sia dal punto di vista tecnico, sia personale.

Vorrei ringraziare tutti i miei amici che mi hanno supportato e sopportato in questi lunghi anni e hanno condiviso con me tutte le gioie e, talvolta, le frustrazioni che ne sono derivate.

Infine, un ringraziamento speciale alla mia famiglia che mi ha sostenuto nell'intraprendere questa avventura. Grazie al loro sostegno, ho avuto il coraggio di osare e spingermi fino a questo punto. Se sto scrivendo queste parole è anche grazie a loro.

*Labor omnia vincit.*

# Bibliografia


[1] Cosa vuol dire "smart parking" https://www.internet4things.it/smart-city/smart-parking-cose-come-funziona-esempi-e-vantaggi/

[2] Acea: Report vehicles in use in Europe (January 2021) https://www.acea.auto/files/report-vehicles-in-use-europe-january-2021-1.pdf#page=18

[3] Istat: Pubblico registro automobilistico http://dati.istat.it/Index.aspx?DataSetCode=DCIS_VEICOLIPRA

[4] Openpolis: Con la pandemia si è ridotta la frequenza dell'uso di mezzi pubblici https://www.openpolis.it/con-la-pandemia-si-e-ridotta-la-frequenza-nelluso-di-mezzi-pubblici/

[5] SmartParkingSystems: Il parcheggio, da problema a risorsa https://smartparkingsystems.com/il-parcheggio-da-problema-a-risorsa/

[6] Express UK: Parking nightmare, You won't believe the amount of time drivers spend looking for a space https://www.express.co.uk/life-style/cars/827333/Car-park-drivers-looking-for-spaces-cities-research

[7] Usa Today: Parking pain causes financial and-personal strain https://eu.usatoday.com/story/money/2017/07/12/parking-pain-causes-financial-and-personal-strain/467637001/

[8] Google Maps https://it.wikipedia.org/wiki/Google_Maps

[9] Enjoy https://enjoy.eni.com/it/roma/home

[10] Ionos: Golang, il linguaggio di programmazione semplice di Google, https://www.ionos.it/digitalguide/server/know-how/golang/

[11] Postman, https://www.postman.com/

[12] IBM Cloud Learn Hub: Cos'è Docker?, https://www.ibm.com/it-it/cloud/learn/docker

[13] RedHat: Cos'è un API, https://www.redhat.com/it/topics/api/what-are-application-programming-interfaces

[14] RedHat: Rest e Soap https://www.redhat.com/it/topics/integration/whats-the-difference-between-soap-rest

[15] Wikipedia: Sistema Client/Server, https://it.wikipedia.org/wiki/Sistema_client/server

[16] IBM: The HTTP protocol, https://www.ibm.com/docs/en/cics-ts/5.2?topic=concepts-http-protocol

[17] Swagger: OpenAPI Specification, https://swagger.io/specification/

[18] Minio https://min.io/

[19] Firebase https://firebase.google.com/