SAPIENZA
UNIVERSITÀ DI ROMA

Progettazione e sviluppo di API per il calcolo di statistiche per l'applicazione GeneroCity

Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea in Informatica

Candidato

Lorenzo Camilli

Matricola 1845956

Relatore

Emanuele Panizzi

Anno Accademico 2020/2021

Progettazione e sviluppo di API per il calcolo di statistiche per l'applicazione GeneroCity
Tesi di Laurea. Sapienza – Università di Roma

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

Email dell'autore: camilli.1845956@studenti.uniroma1.it

Sommario

Il seguente elaborato presenta il mio percorso di tirocinio svolto presso il Dipartimento di Informatica dell'università di Roma La Sapienza.

Durante questo percorso ho fatto parte di un gruppo di lavoro che ha progettato una applicazione di *smart parking* denominata **GeneroCity**, sotto la direzione del prof. Emanuele Panizzi. Per smart-parking si intende l'utilizzo di tecnologie al fine di individuare facilmente e in tempo reale i parcheggi disponibili in una determinata zona. Mi sono occupato principalmente dello sviluppo di API (*Application Programming Interface*), ovvero del software che si occupa di fornire funzionalità ad altri componenti, software o hardware. In particolare, in **GeneroCity** ho lavorato su API necessarie al calcolo di alcune statistiche di utilizzo dei parcheggi. Nell'elaborato verranno inizialmente esposte le conoscenze preliminari, e successivamente si affronterà la fase di implementazione e di sviluppo.

Indice

Introduzione

Secondo il recente studio di Nomisma, "Attori e modelli per una mobilità sostenibile", presentato il 26 maggio 2021, in Italia le automobili ci sono 663 ogni 1000 abitanti, la larga maggioranza della popolazione (64%) utilizza quotidianamente l'auto per i propri spostamenti. La congestione dei grandi centri urbani fa perdere nel traffico l'equivalente di oltre 10 giornate lavorative, che diventano 21,5 a Roma e 18,6 a Milano, con conseguenze negative su di noi e sull'ambiente, in termini di stress, qualità dell'aria, spazio pubblico e inquinamento. Parte di questo tempo è speso nella ricerca di un parcheggio, tempo che può arrivare anche a raggiungere svariate decine di minuti. Il problema non è solo italiano: secondo [1] gli americani spendono circa 17 ore all'anno nella ricerca del parcheggio, negli UK questo dato arriva fino a 44 ore. Il riuscire a mitigare questo problema con servizi che puntino all'ottimizzazione, tramite strumenti tecnologici, dell'uso dello spazio pubblico e in particolare dei parcheggi, si inserisce dunque tra i possibili obbiettivi delle cosidette "città intelligenti". L'applicazione **GeneroCity**, attualmente in fase di studio e sperimentazione da parte di un gruppo di ricercatori e studenti del Dipartimento di Informatica della Sapienza di Roma, interviene nella fase del parcheggio con lo scopo di aiutare l'automobilista a ridurne i tempi.

L'idea di base dell'applicazione **GeneroCity** è quella di effettuare lo scambio di parcheggio tra utenti. Un utente, detto *giver*, dichiara al sistema che sta lasciando un parcheggio in una data posizione, un altro utente (il *taker*) tramite una segnalazione usando una funzione apposita o mediante attivazione automatica rilevata dall'applicazione (in modo da ridurre l'uso del telefono alla guida), potrà usufruire del parcheggio lasciato dal giver. Con questo sistema chi cerca parcheggio non dovrà sprecare tempo e benzina girando a vuoto, ma saprà subito dove andare. L'applicazione non richiede l'introduzione di nuovo hardware, ma sfrutta i sensori già disponibili nei telefoni, come il GPS e l'accelerometro, che col passare delle generazioni diventano sempre più precisi.

Il mio lavoro si è concentrato sullo sviluppo di funzionalità (sezione 2.1) che implementino la sezione delle statistiche inerenti il tempo e la distanza di ricerca dello stallo. Queste funzionalità vengono integrate nell'applicazione, in modo da restituire un feedback immediato all'utente sui vantaggi dell'uso dell'app.

Una volta stabilito l'obbiettivo da realizzare con le API (costruzione delle statistiche di tempi e distanze di ricerca dei parcheggi) ho analizzato le informazioni a mia disposizione fornite del sistema per capire quali tra i dati mi sarebbero stati utili e che quindi avrei dovuto estrapolare. Successivamente sono passato alla fase di progettazione, che include anche la definizione di possibili condizioni di errore di raccolta dati e conseguente notifica all'utente. Dopo questa prima fase di studio, ho affrontato lo sviluppo vero e proprio delle API, terminato il quale sono passato ad una fase di test.

La relazione è composta come segue.

Nel Capitolo 1 vengono affrontati gli strumenti che sono stati usati, ai fini dello sviluppo del progetto.

Nel Capitolo 2 è esposta una parte teorica in cui sono presentate cosa sono le API e come funzionano. Un paragrafo è dedicato al protocollo HTTP, parte fondamentale di una API. Viene quindi esposta la struttura dell'applicazione GeneroCity e le sue funzionalità principali. La parte finale affronta nel dettaglio il lavoro da me svolto, in particolare la fase di progettazione delle API.

Infine, il Capitolo 3 tratta lo sviluppo delle API dell' applicazione **GeneroCity**, i calcoli che esse effettuano sui dati del sistema, la loro interazione con il database e come esse reagiscono alle varie condizioni.

Capitolo 1

Architettura e tecnologie

Durante il mio percorso di tirocinio ho avuto l'occasione di sperimentare tecnologie e protocolli per lo sviluppo del lato back-end del progetto, in particolare il linguaggio di programmazione **Go**, le piattaforme **GitLab** e **Docker** ed il software **Postman**. Di seguito espongo le motivazioni che hanno portato alla scelta di questi strumenti rispetto ad altri simili, il loro funzionamento e in che modo sono stati utilizzati.

1.1 Go

Sicuramente parte fondamentale quando si decide di sviluppare un'applicazione, o più in generale un software, è la scelta del linguaggio di programmazione. In questa fase è importante avere ben chiari quali siano i requisiti che il software deve rispettare. In particolare nel caso dell'applicazione GeneroCity si richiedono principalmente reattività, facilità di utilizzo, supporto e stabilità. Pertanto la scelta è ricaduta sul linguaggio open-source con licenza BSD[1], **Go** (conosciuto anche con il nome di *Golang*). Ideato e creato nel 2009 da Google [2], per ovviare alle criticità degli altri linguaggi ma mantenedone le caratteristiche principali. Go garantisce una compilazione efficiente (anche su hardware modesti), facilità di programmazione e velocità di esecuzione, il tutto con una sintassi orientata a quella della famiglia dei linguaggi C, ma con un usabilità e leggibilità molto maggiore. Si contraddistingue specialmente per la sua semplicità e la sua multifunzionalità, i file sorgente si possono organizzare in modo modulare tramite directory che vengono indicate con il nome di *packages*. Le applicazioni che vengono sviluppate in Go hanno una velocità di esecuzione generalmente migliore rispetto a quelle sviluppate con altri linguaggi compilati. Uno dei fattori di maggiore importanza che hanno influito nella scelta di questo linguaggio per il back-end di GeneroCity è l'ampio supporto che esso fornisce. Infatti permette di sviluppare le API dell'applicazione sia per i dispositivi Android che per quelli iOS con lo stesso codice, agevolando la manutenzione e lo sviluppo del software e fornendo strumenti utili a tale scopo, come per esempio `godoc` per l'integrazione della documentazione, `go test` per lo sviluppo di test e `go get` che fornisce la possibilità di recuperare package da remoto. Inoltre Go è provvisto di un ampia gamma di librerie per le più disparate necessità e che permettono di risolvere molti dei problemi che si possono affrontare, facilitando il lavoro del programmatore. Il linguaggio Go ha funzionalità integrate, oltre al supporto di

[1] La licenza BSD è un tipo di licenza open-source, più restrittiva della GPL, ovvero quella che lascia la massima libertà, che permette a chiunque di modificare il codice, con l'unico dovere di citare l'autore. Quindi chiunque può sviluppare in forma chiusa con una licenza proprietaria un programma BSD, anche impedendo ai propri acquirenti di modificarlo e ridistribuirlo a loro volta.

librerie, per la scrittura di programmi simultanei e la concorrenza. La concorrenza si riferisce non solo al parallelismo della CPU, ma anche all'asincronia: consentire l'esecuzione di operazioni lente come un database o la lettura di rete mentre il programma svolge altre attività. Sebbene Go sia ancora un linguaggio giovane, per molti sarà uno standard in futuro; basti pensare che è usato da Dropbox e da aziende come Netflix, Twitch (la piattaforma per live streaming di proprietà di Amazon), dal servizio Uber e anche da Docker (sezione 1.3).

1.2 GitLab

Un altro strumento che si è rivelato essenziale al progetto è **GitLab**, una piattaforma web per la gestione di repository Git. Il suo scopo principale è quello di affiancare i programmatori durante tutto il ciclo di vita del codice, garantendo massima affidabilità nello sviluppo. GitLab si basa sulla tecnologia Git che è un sistema di controllo di versione. Un sistema di controllo di versione (VCS, acronimo di *Version Control Systems*) è quel software che permette di gestire le versioni, modifiche e il rilascio di codice, allo scopo sia di identificare cronologicamente le modifiche, sia di annullare quelle che ne hanno determinato un malfunzionamento. Tra i concetti fondamentali di Git sono: gli **snapshot**, i **commit**, i **repository**, i **branch** e gli **issues** [3].

Snapshot: gli snapshot vengono usati da Git per mantenere traccia della cronologia del codice, memorizzandone lo stato, le versioni precedenti e tutte le modifiche apportate ai file. Fornendo così la possibilità di modificare file in sicurezza o di ripristinare dei cambiamenti.

Commit: il commit è l'azione tramite il quale si genera uno snapshot, essi rappresentano il modo in cui si salvano le modifiche fatte al codice sul repository.

Repository: il repository è una contenitore virtuale di tutti i file. Le principali funzionalità connesse dai repository sono: la clonazione di un repository Git esistente dal server remoto, la possibilità di scaricare le modifiche di un repository e l'invio dei cambiamenti apportati.

Branch: il branch è una ramo nella struttura ad albero del repository. I commit collegati risiedono in uno o più branch, all'interno di un repository. I branch permettono di avere più versioni dello stesso codice che si distaccano dal ramo principale chiamato *main*, che una volta terminate possono essere aggiunte ad esso (Figura 1.1).

Issues: sono strumenti molto utili quando ad un progetto lavorano più persone, poiché consentono di assegnare o ricordare dei compiti da svolgere oppure per segnalare eventuali problemi riscontrati nel codice.

Merge: permette di unire il codice aggiornato, solo dopo aver verificato che non ci siano conflitti, al ramo principale. Consente inoltre di visualizzare le modifiche fatte dall'ultimo aggiornamento del codice.

In GeneroCity, GitLab è stato fondamentale per la gestione del codice all'interno del progetto, infatti è stato utilizzato per avere codice sempre aggiornato (da eventuali cambiamenti apportati da altri membri del gruppo), mantenere una storia delle modifiche fatte dai vari componenti sapendo da chi sono state effettuate e in quale momento o per segnalare eventuali bug riscontrati.

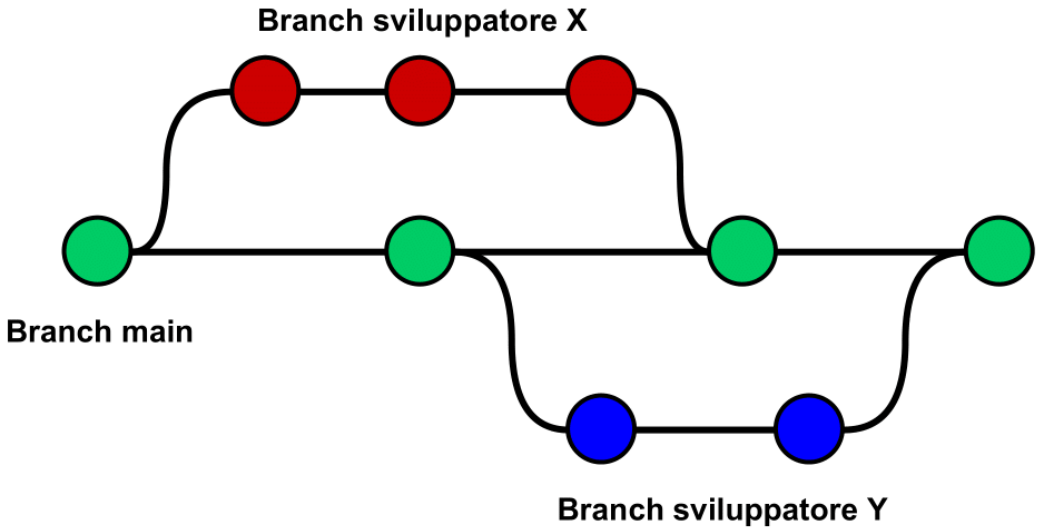

Figura 1.1. Rappresentazione grafica del funzioonamento dei branch in Git. Si possono notare i due rami di sviluppo di due programmatori diversi che verranno uniti nel ramo principale.

1.3 Docker

Docker è una piattaforma che gestisce e automatizza la distribuzione di applicazioni tramite l'uso di *container*, in cui vengono inseriti il codice sorgente, le librerie e tutte le dipendenze necessarie per eseguire il codice, permettendo di rendere eseguibile lo stesso codice su varie macchine in modo affidabile e riproducibile, tramite isolamento dall'architettura su cui viene eseguito. L'isolamento avviene tramite l'uso dei container, che limitano l'accesso dell'applicazione a tutte le risorse del sistema operativo, lasciando l'accesso solo alle funzionalità necessarie all'esecuzione [4]. La comodità di utilizzare i container è data dalla possibilità di costruire facilmente e velocemente un ambiente di sviluppo o di test isolato e indipendente dal sistema su cui viene eseguito, garantendo in caso di problemi di effettuare modifiche sulle funzionalità del software.

La creazione e l'inizializzazione di un'applicazione in un container Docker avviene tramite file chiamati "immagini". La stessa immagine può dar vita a più container. Ogni immagine è definita e creata da un **Dockerfile**, un file di configurazione contenente tutti i comandi necessari per assemblare l'immagine. Una immagine può essere creata partendo da zero, oppure a partire da una già esistente, per esempio scaricandola da Docker Hub[2], sulla quale aggiungere i livelli necessari. Un file Docker è costituito da una serie di comandi formati nel modo seguente: `ISTRUZIONI argomenti`, in cui ogni riga rappresenta un livello. Le istruzioni fornite da Docker, simili a quelle di alcuni linguaggi di programmazione, permettono di svolgere alcune importanti operazioni con lo scopo di creare l'immagine. Tra queste troviamo l'istruzione `FROM`, che indica l'immagine di base da cui partire per la creazione di una nuova. Questa istruzione deve essere la prima istruzione nel file Docker. La possibilità di definire variabili è data dall'istruzione `ARG`; di eseguire comandi durante la fase di compilazione con `RUN`; di copiare file e directory all'interno dell'immagine

[2] `https://hub.docker.com/`

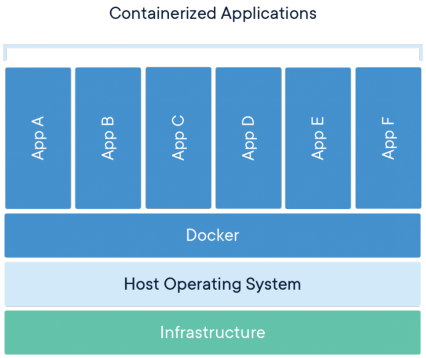

Figura 1.2. Astrazione del funzionamento dei container Docker [3].

con l'istruzione `ADD`. Si ha inoltre la possibilità di montare directory sul container tramite `VOLUME`; quando si vuole specificare in fase di esecuzione la porta su cui il container è in ascolto si utilizza `EXPOSE`. Esistono poi molte altre istruzioni che non verranno affrontate in questo contesto.

```
1  FROM golang:1.16-alpine AS build
2  #Installa le dipendenze necessarie
3  #Esegue comandi per aggiornare le dipendenze
4  RUN apt-get update && apt-get install -y
5  RUN apk add -no-cache git
6  RUN go get github.com/golang/dep/cmd/dep
7  ...
8  #Copia il progetto e lo compila
9  COPY . /go/src/project/
10 RUN go build -o /bin/project
11 ...
```

Listing 1.1. Esempio di Dockerfile [4].

Grazie a Docker è stato possibile garantire ad ogni sviluppatore di GeneroCity di avere un ambiente di sviluppo con tutto il necessario sul proprio pc in modo semplice.

1.4 Postman

Postman è un software che ha permesso di testare le API in modo facile ed intuitivo. Affronteremo nel dettaglio cosa sono le API e il loro funzionamento nel prossimo capitolo, per ora è sufficiente dire che un API è software che mediante chiamate HTTP (URL) effettua richieste ad un server, ricevendo indietro una risposta. Le richieste possono essere effettuate sia verso un server locale che verso un server online, impostando tutti i dati di una tipica chiamata HTTP, l'*header*, eventualmente

[3] Immagine presa da: `https://www.docker.com/resources/what-container`
[4] Fonte: `https://docs.docker.com/develop/develop-images/dockerfile_best-practices`

il *body*. Una volta scelto il metodo della richiesta (per esempio GET, POST...), si inserisce nel apposito campo del programma l'URL che identifica l'API. Dopo essersi eventualmente autenticati, Postman controlla se i parametri richiesti dalla chiamata (qualora presenti) sono corretti, quindi permette di eseguire l'API, e poi restituisce un array di oggetti JSON contenente l'output oppure, in caso contrario, un errore esplicativo del problema [5]. Dunque Postman permette di effettuare test senza avere l'applicazione, perché permette di valutare in base all'input se viene riportato l'output corretto ed eventualmente di apportare correzioni.

Capitolo 2

Background

2.1 Che cos'è e come funziona una API

Prima di parlare del progetto in sé, è opportuno spiegare nel dettaglio cosa sono le API, poiché queste rappresentano un aspetto fondamentale del mio tirocinio.

Il termine **API** (acronimo di *Application Programming Interface*) indica quel genere di software che offre servizi ad altro software o hardware, tramite un insieme di protocolli e routines volte allo svolgimento di un compito, permettendo una facile comunicazione tra dispositivi diversi. Lo scopo delle API è quello di nascondere i dettagli interni del funzionamento di un sistema, esponendo solo le parti necessarie [6]. Le API semplificano il design e l'architettura dei software, aggiungono flessibilità e sicurezza, e permettono scalabilità e manutenzione nel tempo. In una API agiscono due attori:

- **Client:** colui che effettua la chiamata per ottenere i dati. Non è necessario che conosca il funzionamento del programma sottostante il server, basta solo che sia a conoscenza delle regole necessarie per implementare la chiamata.
- **Server:** l'elemento che su richiesta fornisce i dati corrispondenti alla chiamata ricevuta gestendo e implementando le regole di funzionamento dell'API.

Le API consentono l'accesso alle risorse mantenendo **sicurezza** e **controllo**, in quanto è il programmatore a decidere come e a chi concedere l'accesso. Alcuni esempi di API sono le librerie usate nei linguaggi di programmazione, oppure quelle che consentono ad uno sviluppatore software di accedere a determinate parti di un sistema operativo senza necessariamente conoscerne il funzionamento a basso livello. Le API più utilizzate sono quelle che sfruttano come canale di comunicazione Internet. Queste ultime utilizzano in genere il protocollo HTTP (acronimo di *HyperText Transfer Protocol*, presentato nella prossima sezione) per mettere in comunicazione i vari elementi. I messaggi di risposta di queste API sono in genere sotto forma di file JSON [5], in modo da consentire ad altre app una gestione dei dati più facile. Le principali politiche di rilascio di un'API sono le private, o le pubbliche. Con le private l'API è solo per uso ristretto, interno dell'azienda. Con un rilascio pubblico, le aziende consentono invece agli sviluppatori di terze parti riconosciuti di accedere alle API. Se l'API è disponibile per l'utilizzo da parte del pubblico spesso l'accesso può essere ristretto utilizzando "token API" o convalide del cliente.

[5] Questo genere di file è lo standard di fatto quando si vuole scambiare dati tra client e server. Sono formati da array di valori sotto forma di `chiave:valore`

2.1.1 Messaggi HTTP

HTTP è un protocollo a livello applicativo di tipo client-server, usato principalmente in ambito web. Le comunicazioni tra client e server vengono effettuate tramite messaggi HTTP. I messaggi di richiesta sono costituiti da una **riga di richiesta** (conosciuta anche con il nome di *request-line*) composta da un metodo, una URL e versione del protocollo, una intestazione detta **header**, che contiene metadati inerenti l'host e il client; e da un corpo, **body**, che contiene le informazioni che si vogliono trasmettere [7]. Ogni richiesta ha nella sua request-line un determinato metodo HTTP. I metodi comunicano al server cosa fare con i dati identificati dalla URL. Quelli maggiormente usati nello sviluppo di API sono GET, POST, PUT e DELETE.

GET Ordina al server di trasmettere al client le informazioni identificate nella URL, le informazioni lato server non vengono modificate in questo tipo di richiesta.

PUT Usata quando si vuole creare o aggiornare la risorsa identificata dalla URL. Per esempio, la chiamata `PUT /users/user1` crea un utente di nome user1. Nella richiesta non c'è alcuna indicazione che indichi al server come vadano creati i dati ma solo che deve farlo: queste informazioni risiedono nel corpo della richiesta.

DELETE Opera inversamente al PUT, viene usato quando si vuole cancellare la risorsa identificata dalla URL. Per esempio, `DELETE /users/user1` elimina tutte le informazioni associate alla risorsa identificata da `/users/user1`.

POST Utile quando si vuole inviare grandi quantità di dati, come un file o un'immagine al server.

I messaggi di risposta, sono invece formati da una **riga di stato**, un **header**, che indica il tipo e la versione del server ed il tipo di contenuto restituito, e da un **body** contenente il contenuto restituito. Generalmente nelle API il contenuto viene riportato sotto forma di JSON o XML. Le risposte vengono inoltre affiancate da **codici di stato** che indicano l'esito della richiesta. Il server dovrebbe restituire il codice di riposta più appropriato, in modo da far capire al client il problema che si è presentato ed eventualmente rimediare agli errori. La Tabella 2.1 riporta i codici maggiormente usati.

Tabella 2.1. Codici di stato HTTP.

Codice	Significato
200	OK. Richiesta valida e andata a buon fine.
201	Creato. Richiesta valida, la risorsa è stata creata.
400	Non valida. La richiesta non è stata formata in modo corretto.
401	Non autorizzato. Si richiede l'autenticazione prima di accedere alla risorsa.
403	Proibito. Utente autenticato ma non possiede il permesso necessario.
404	Non trovato. La risorsa richiesta non può essere raggiunta.
500	Errore interno al server. Processo fallito lato server, in circostanze impreviste.

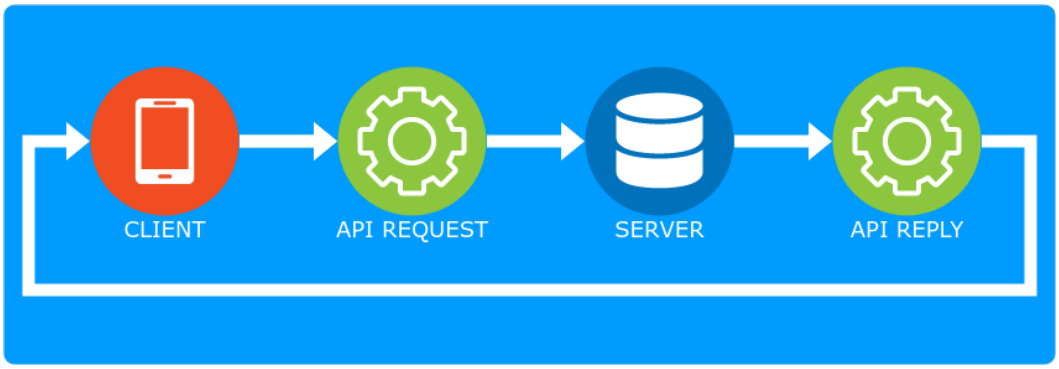

Figura 2.1. Schema di funzionamento di una API. Il client fa una richiesta tramite API, l'API interagisce con il DB e restituisce una risposta.

2.2 Struttura back-end dell'app GeneroCity

In questo paragrafo vedremo come le tecniche generali precedentemente presentate sono state usate nell'applicazione GeneroCity. Ricordiamo che un'applicazione è formata da una parte di **font-end** e da una parte di **back-end**, in comunicazione: il front-end è la parte di interfaccia grafica con cui interagisce l'utente, è la parte visibile, serve ad acquisire o presentare dati all'utente. La parte di **back-end**, non visibile all'utente, riceve le chiamate dal front-end, e si occupa di implementare tutto il funzionamento e la logica dell'applicazione e di elaborare i dati ricevuti. Il mio lavoro di tirocinio si concentra sulla parte back-end, spiegata brevemente nel seguito.

Il back-end è costituito da file e cartelle. Ci sono i file di configurazione, il Dockerfile (sezione 1.3), il file con la collezione delle API da importare in Postman (sezione 1.4), quelli utili all'esecuzione del progetto. Le cartelle principali che si occupano di far funzionare l'applicazione sono: la cartella `cmd/` che ha lo scopo di costruire e gestire tutte le strutture necessarie al funzionamento dell'applicazione, essa contiene anche l'eseguibile principale, la cartella `database/` predisposta alla creazione e modifica del database e la cartella `service/` che contiene tutti i file per il funzionamento delle API. Grazie alla modularità garantita dai packages Go, ogni directory contiene tutti i file predisposti allo svolgimento di un singolo compito.

2.2.1 Inizializzazione dell'app

La cartella `cmd/` si occupa di gestire tutti gli strumenti per la fase di inizializzazione e di "costruzione" del progetto partendo dall'eseguibile principale (il `main`). Un file di questo package contiene la configurazione dell'applicazione, ovvero la struttura che compone l'applicazione con tutte le dipendenze necessarie.
Il `main` carica la configurazione ed inizializza le variabili globali, poi costruisce l'applicazione effettiva, assemblando e inizializzando tutte le componenti e effettuando i dovuti controlli sulla loro corretta costruzione e sul loro stato. Tra le componenti utili all'applicazione ci sono: la cache, utile a velocizzare la comunicazione, il server web, per la comunicazione tra i componenti, il servizio di autenticazione, il log degli errori (in cui vengono salvati eventuali errori di esecuzione), il gestore delle notifiche, il database ed altri.

2.2.2 Database

Il package `database/` mantiene tutti i file utili per la creazione e l'aggiornamento del database dell'applicazione. In particolare è presente un file con la copia attuale dello schema del database (ovvero struttura delle varie tabelle ed i collegamenti tra esse), e i file predisposti alle migrazioni. I file di migrazione dello schema sono quei file che, mediante l'ausilio di alcuni strumenti, permettono di aggiornare o ripristinare in automatico versioni e modifiche fatte ad un database (es. aggiunta o rimozione di tabelle e colonne) già in utilizzo, permettendo di adattare i dati al cambiamento dei requisiti in maniera sicura. Nel nostro caso, tra le tabelle più importanti ci sono quella predisposta a mantenere informazioni sugli utenti, quella che mantiene le informazioni sulle auto, quella per i parcheggi (coordinate, l'orario, utente e auto che ha effettuato il parcheggio...) e quelle con la storia dei match tra gli utenti, ovvero i dati che associano i vari utenti (con le rispettive automobili) che hanno scambiato un parcheggio.

2.2.3 Gestione delle API

Il package `service/` rappresenta il punto di ingresso dell'applicazione. Esso implementa all'interno della funzione `New()` un router (costruito utilizzando una libreria di Go, che implementa una comunicazione HTTP tra un client ed un server) interno, ovvero quella componente che farà da "ricevitore" dell'applicazione confrontando le richieste in entrata e chiamando un gestore per gli URL, presenti nell'**handler**. L'handler è il file che si occupa di mappare tutte le funzioni che implementano le API con l'URL per le chiamate, dunque si occupa di una delle operazioni più importanti per GeneroCity. Nel package `service/` sono presenti inoltre tutti i file che implementano la logica delle API, le strutture che dovranno essere restituite come output ed infine i file che permettono l'interazione delle API con il database. Infatti ogni API di GeneroCity è composta di due parti: il file principale che svolge la funzione di ricevere i dati in input, fare i controlli necessari e dare il corretto output, con annesso codice di stato, ed una parte che si occupa di interagire con il database mediante comandi SQL, chiamato dal file principale. Queste due parti sono tenute separate per fornire maggiore flessibilità ai cambiamenti in futuro. Tra le API che hanno un ruolo fondamentale ci sono:

createMe Si occupa di creare un utente nel sistema, inizializzare tutti i dati ad esso associati ed assegnargli un identificativo univoco (quest'ultimo utile per effettuare query sul database), prende dal body il nickname da assegnare all'utente.

createCar Prende in input dal body vari dati, come targa, nome, dimensioni, modello e colore dell'auto, e crea un "oggetto" *auto* associato ad uno o più utenti (per esempio ad utenti nella stessa famiglia).

parkCar Una volta passato nel path dell'API l'identificativo dell'auto e nel body le coordinate geografiche del parcheggio, il momento del parcheggio ed il viaggio effettuato (un array JSON che mantiene informazioni su ogni coordinata del viaggio effettuato per arrivare a quel parcheggio), se i controlli su tali dati sono andati a buon fine, aggiorna lo stato dell'auto in "parcheggiata" e aggiunge una nuova riga nella tabella dei parcheggi.

unparkCar Opera in maniera opposta alla precedente e aggiorna lo stato dell'automobile da "parcheggiata" a "non parcheggiata".

createMeGiver Serve per effettuare lo scambio di parcheggio, segnalando al sistema (e agli altri utenti) il luogo e il momento in cui l'utente lascia il parcheggio.

createMeTaker Complementare alla precedente, indica al sistema (e agli altri utenti) che l'utente cerca parcheggio in una determinata area, indicata dalle coordinate geografiche inserite nel body della richiesta.

createChatMessage Permette di mandare un messaggio ad un altro utente, il testo viene passato nel body.

getParkList Selezionando un intervallo di tempo e ritorna un array con tutti i parcheggi effettuati in quell'intervallo, con informazioni come inizio e fine del parcheggio, via e auto con cui si è fatto il parcheggio.

getFamily Restituisce l'elenco di tutti gli utenti che hanno accesso all'automobile selezionata.

Sono poi presenti altre API che permettono all'utente di visualizzare, modificare o eliminare dati.

2.3 Lavoro svolto

Il percorso che ho fatto in questi mesi mi ha permesso di avvicinarmi, per quanto possibile in sede di un tirocinio universitario, con quella che in ingegneria del software viene definita **metodologia agile**, ovvero una serie di tecniche che puntano allo sviluppo di software che sia focalizzato sull'obiettivo di consegna, in tempi brevi e frequenti (*early delivery*), e sia funzionante e di qualità. Infatti, settimanalmente venivano svolte riunioni con il professore ed il team volte a fare il punto della situazione e a progettare il lavoro per la settimana successiva, promuovendo quindi lo sviluppo iterativo ed incrementale del progetto e l'organizzazione autonoma del lavoro. Ho imparato a seguire una pianificazione a breve termine che gestisce l'andamento del progetto, tenendo anche conto di eventuali cambiamenti in corso d'opera.

Dopo un periodo di apprendimento e di studio del progetto, che era già in fase di sviluppo quando ho cominciato il tirocinio, nel quale ho dovuto capire il funzionamento e le tecnologie utilizzate, mi è stato assegnato il primo dei miei due compiti: l'aggiornamento del codice, e lo sviluppo delle API.

2.3.1 Aggiornamento struttura del codice

Poiché l'applicazione era già stata avviata tempo prima, sorgeva la necessità di un aggiornamento e di alcune modifiche che garantissero una scalabilità maggiore del progetto in futuro. In particolare mi sono occupato di:

- **Modifica di librerie:** ho dovuto modificare quei metodi che usano versioni vecchie di librerie, cercando di adattarli quando possibile alla nuova versione, altrimenti usando una libreria alternativa che potesse svolgere lo stesso compito.
- **Riprogettazione file di configurazione:** era necessario riposizionare il codice presente nel main che rappresentava i vari elementi utili al progetto (database, server...) in file di configurazione appositi, riprogettando le chiamate e gli usi di essi, per favorire la suddivisione del codice in base alle funzioni svolte.

- **Adattamento delle API:** ho dovuto riallineare le API pre-esistenti alle modifiche della nuova versione.
- **Riscrittura routine autenticazione:** a seguito di alcuni cambiamenti è risultato necessario riscrivere la routine che si occupa del token di autenticazione per il software di testing delle API.
- **Individuazione di bug:** segnalare e possibilmente risolvere alcuni piccoli bug emersi durante il lavoro.

Il compito che mi è stato affidato si è rivelato inizialmente ostico, ha dunque richiesto del tempo per capire come operare. Inoltre mi sono trovato davanti ad problemi inaspettati, che mi hanno richiesto di capire da dove fossero generati e come risolverli. Grazie anche alla collaborazione con altri colleghi del team, il lavoro preparatorio mi ha permesso di prendere confidenza con il codice, con le modalità e i tempi di gestione di un progetto del genere.

2.3.2 Progettazione delle API

L'altro compito che mi è stato assegnato è quello di progettare e sviluppare delle specifiche API per GeneroCity. L'intento era quello di fornire dati che evidenziassero all'utente i vantaggi derivanti dall'uso di GeneroCity, ovvero quello ottimizzare i tempi e minimizzare il percorso per il parcheggio.

La prima cosa che ho dovuto fare insieme ad un collega che si occupa della parte di front-end è ragionare mediante **need-finding**[6] e sondaggi, su quali dati potessero essere utili allo sviluppo. L'idea di base che è emersa, considerando anche lo stato attuale dell'applicazione, è quella di evidenziare la distanza percorsa ed il tempo impiegato dall'utente ad effettuare parcheggi.

Inizialmente l'idea era quella di rappresentare dati solamente inerenti all'utente, ma durante le riunioni settimanali raffinando i requisiti è emerso che potesse essere utile avere dati anche riguardo alle auto, in quanto queste possono essere usate da vari membri della famiglia che usano l'applicazione. I dati di interesse inerenti alle statistiche di ricerca dell'utente, che verranno quindi inseriti nel database dell'applicazione sono:

- **Tempo totale:** ovvero il tempo totale in secondi impiegato dall'utente per effettuare i parcheggi.
- **Distanza totale:** la distanza totale in chilometri impiegata dall'utente per cercare parcheggio.
- **Media tempo di ricerca:** media totale in secondi del tempo impiegato a cercare parcheggio, indica all'utente quanto tempo impiega mediamente (e quindi indirettamente quanto ne risparmia) nella ricerca di parcheggio.
- **Media distanza di ricerca:** media totale in chilometri della distanza percorsa dall'utente a cercare parcheggio.
- **Numero di match:** ovvero il numero di parcheggi che l'utente ha trovato usando la funzionalità di scambio parcheggio dell'applicazione.

[6] Il need-finding è un insieme di tecniche prevalentemente basate su domande e questionari, da somministrare ai possibili utenti finali di un'applicazione, che aiutano nella fase di progettazione delle funzionalità, poiché permettono di immaginare cosa realmente interessa all'utente, indirizzando così lo sviluppo nella corretta direzione, risparmiando risorse ed evitando lo sviluppo di funzionalità non usate.

- **Numero parcheggi senza match:** numero di parcheggi che l'utente ha trovato senza usare la funzione di scambio.
- **Numero di parcheggi:** numero totale dei parcheggi effettuati.

I dati di interesse inerenti alle auto sono:

- **Tempo totale:** il tempo totale, in secondi, di ricerca dei parcheggi svolti con una stessa auto.
- **Media tempo:** tempo medio, in secondi, impiegato a ricercare parcheggio.
- **Distanza totale:** chilometri percorsi dall'auto a cercare parcheggio.
- **Distanza media:** media dei chilometri percorsi da un'auto per cercare parcheggio.
- **Numero di match:** ovvero il numero di parcheggi fatti con la stessa auto usando la funzionalità di scambio parcheggio dell'applicazione.
- **Numero parcheggi senza match:** numero di parcheggi fatti con la stessa auto senza usare la funzione di scambio.
- **Numero parcheggi:** il numero totale di parcheggi fatti da qualsiasi utente con la stessa auto.

Con questi dati chi si occupa di progettare la parte dell'interfaccia dell'applicazione ha la possibilità di presentare all'utente quanto tempo impiega nella fase di parcheggio, permettendo ad esso di prendere coscienza di tali informazioni ed invogliandolo sempre di più all'utilizzo dell'applicazione come strumento di supporto nell'impiego di tempo in maniera efficiente.

L'idea per arrivare a ottenere questi dati è la seguente: dopo che si è calcolato i dati del tempo di ricerca e distanza di ricerca, associati ad ogni parcheggio, si effettuano i calcoli e si salvano nel database tramite un API che effettua la `PUT`, in modo di evitare di ripetere il calcolo ad ogni chiamata delle API che effettuano la `GET` aumentando l'efficienza.

Lo schema di progettazione del funzionamento dell'API per le statistiche sull'utente è mostrata nella Figura 2.2. Come prima cosa si controlla che l'id dell'utente sia corretto, sebbene è raro che si verifichi un tale errore. Se il controllo non va a buon fine si ritorna il codice di stato 400, con spiegazione dell'errore associato. Altrimenti si può proseguire con la funzione che interagisce con il database, se le query vengono effettuate correttamente si ritornano i dati statistici. Quando l'interazione col database non va a buon fine le cause possono essere due: l'id dell'utente non è presente nella tabella, allora ritorna il codice 404 che indica che la risorsa non è stata trovata; oppure può accadere che il database non sia raggiungibile e quindi ritorna codice di stato 500.

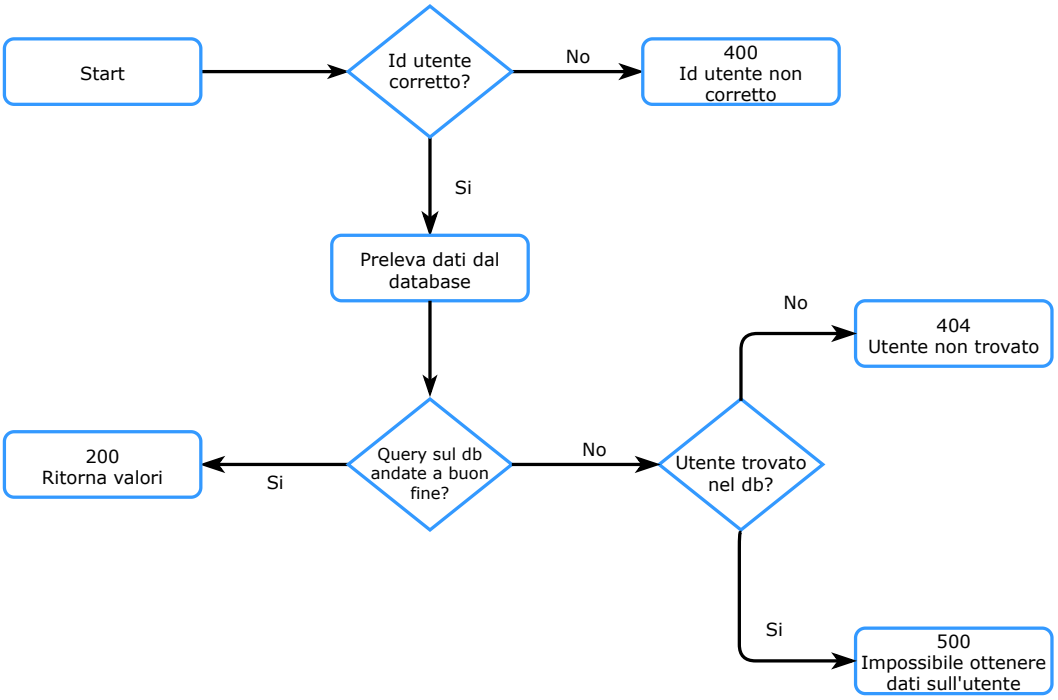

Figura 2.2. Funzionamento API per ottenimento delle statistiche sull'utente.

La Figura 2.3 rappresenta l'idea dietro l'API per l'ottenimento delle statistche sull'auto. Viene inizialmente controllata la correttezza dell'id dell'auto, se il controllo va a buon fine si verifica che l'utente che ha effettuato la chiamata all'API sia associato all'auto di interesse, se non lo è ritorna il codice 403 (l'utente non è autorizzato all'accesso alla risorsa). Altrimenti si possono effettuare le query sul database, se non raggiungibile ritorna codice 500 se invece non si trova l'id dell'utente allora ritorna 404, negli altri casi in cui le query vanno a buon fine e si restituisce il codice di stato 200.

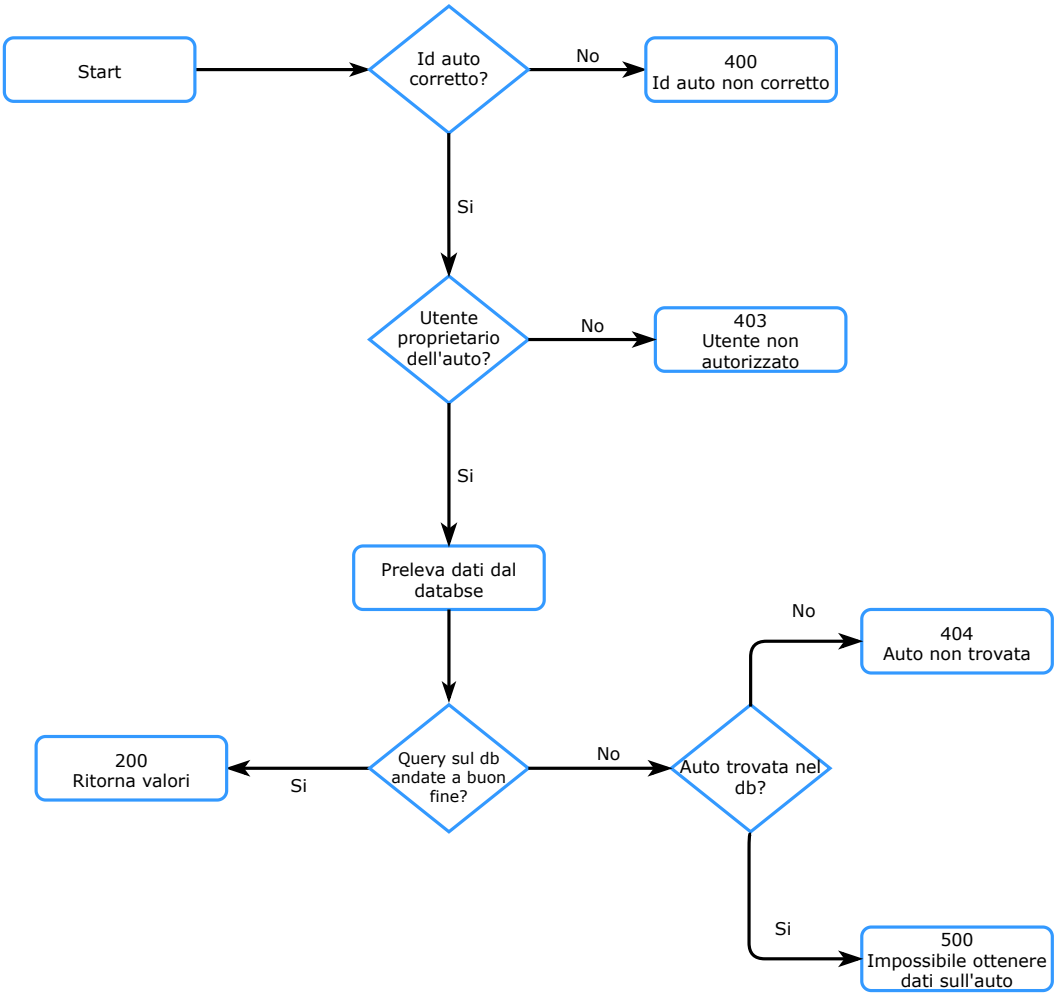

Figura 2.3. Funzionamento API per ottenimento delle statistiche sull'auto.

Capitolo 3

Implementazione API

In questo capito presento la parte di sviluppo in linguaggio Go mediante anche l'ausilio di alcune parti di codice.

3.1 Estrapolazione dei dati tramite l'API car park

Come prima cosa per ogni parcheggio vanno associati i valori inerenti al tempo di ricerca e la distanza percorsa, tramite i quali si estrapoleranno tutte le altre statistiche. Nel sistema per parcheggio non si intende solamente il luogo fisico, ma un oggetto che contiene le informazioni sul viaggio, sullo stato dell'auto e sul momento e luogo effettivo del parcheggio.

Successivamente, prendendo in considerazione l'intero viaggio dell'auto da un parcheggio all'altro, estrapolo solo i dati inerenti la fase di ricerca. Per fare questo ho ampliato una API già esistente, **car-park**. Questa API prende in input dal body anche un campo **tripjson**, ovvero una stringa con struttura di un file JSON, che rappresenta il viaggio dell'auto, sotto forma di coordinate geografiche, da un parcheggio all'altro. Proprio il **tripjson** rappresenta il punto di partenza di tutto lo sviluppo, questo campo è un insieme di posizioni, detti **tripoint**, formati come segue:

```
1  {
2    "speed" : 21,
3    "searching" : 0,
4    "lat" : 41.9018889,
5    "lon" : 12.511498,
6    "date" : "2021-10-28T13:24:58+02:00"
7  }
```

Listing 3.1. Esempio di un dato in tripjson.

Di seguito la spiegazione dei vari valori:

- Il campo **speed** indica la velocità dell'auto in quel momento.
- **Searching** indica se l'auto è in fase di parcheggio, quando vale 0 allora l'auto non sta cercando parcheggio. La fase di ricerca viene attivata o tramite deduzione di un modello di machine learning (in questo caso sarà presente nei

tripoint anche un campo denominato "falsepositive"), o tramite indicazione esplicita dell'utente.

- I due valori **lat** e **lon** indicano le coordinate geografiche in quell'istante.
- Il campo **date** rappresenta il momento della cattura dei dati.

Quindi l'idea è quella di scorrere l'intero JSON del viaggio individuando il primo punto in cui comincia la ricerca di parcheggio (quello che ha valore **searching** maggiore di 0), da questo punto in poi calcolare la distanza temporale e spaziale di un punto dal punto precedente. Dopo aver effettuato le somme delle distanze, queste vengono associate al parcheggio di riferimento. Il tutto è mostrato nella seguente parte di codice:

```
err = json.Unmarshal([]byte(carPark.TripJson), &data)
...
var startSearching bool
var searchingDist float64
var searchingTime int
for i := 0; i < len(data)-1; i++ {
        if data[i].Searching > 0 {
                startSearching = true
        }
        if data[i].FalsePositive {
                startSearching = false
        }
        if startSearching {
          searchingDist += locationutils.HaversineDistance
          ↪  ((data[i+1].Lat),(data[i+1].Lon),
          ↪  (data[i].Lat),(data[i].Lon))
          diff := data[i+1].Date.Sub(data[i].Date)
          searchingTime = searchingTime + int(diff.Seconds())
  }
}
carPark.SearchingDistance = searchingDist
carPark.SearchingTime = searchingTime
...
```

Listing 3.2. Codice di carPark per il calcolo dei tempi e distanze di ricerca.

Nel dettaglio il funzionamento è il seguente. Prima si converte la stringa `tripJson` passata dal body (`carPark.Tripstrut`, rappresenta il valore tripjson preso dla body dell'API) in un oggetto iterabile in Go, chiamato `data`, usando il metodo `Unmarshal` della libreria "encoding/json" [9]. Dopo aver fatto opportuni controlli (per esempio che i vari dati esistano e che siano corretti), si effettua l'algoritmo visto nel codice.

Si creano le due variabili per la distanza e il tempo, e quella booleana, inizializzata a `false`, che indica se la ricerca è iniziata. Si itera poi sulla struct[7] che rappresenta tutto il trip. Se il punto su cui si sta iterando ha il campo "searching" maggiore di 0, si mette a `true` la variabile `startSearching`. Il secondo controllo serve per evitare i falsi positivi; infatti se nel trip è presente il campo "falsepositive", allora

[7] Una struct è un tipo definito dal programmatore che rappresenta una raccolta di campi, quando si ritiene opportuno cha tali campi vadano raggruppati in un unico elemento anziché essere tenuti separati.

vuol dire che c'è stato un errore nella rilevazione automatica. Quindi, se la variabile `startSearching` è `true`, si calcola la distanza delle coordinate tra gli ultimi due punti, usando il metodo della libreria che implementa la formula di Haversine [8]. Tale formula calcola la distanza in chilometri tra due punti geografici, date la longitudine e la latidudine. Tale distanza viene poi sommata a quella già calcolata in precedenza. Si effettua anche il calcolo per il tempo, in secondi. Infine le ultime due righe del codice associano i valori alla struct `CarPark`. Quest'ultima viene quindi passata alla funzione che interagisce con il databse per effettuare l'aggiornamento dei dati appena calcolati.

Illustriamo ora la parte di codice predisposta al calcolo degli altri dati.

```
1  ...
2  err = tx.Get(&stat, "SELECT SUM(searchingtimesec) AS totaltimesearching,
   ↪  SUM(searchingdistance) AS totaldistance, AVG(searchingtimesec) AS
   ↪  avgtimesearching, AVG(searchingdistance) AS avgdistance FROM parks
   ↪  WHERE usedby = ?  AND parkstatus = 'park' AND searchingtimesec > 0",
   ↪  userId.String())
3  if err != nil {
4          _ = tx.Rollback()
5          return nil, errors.Wrap(err, "select1 statement")
6  }
7
8  err = tx.Get(&stat, "SELECT COUNT(*) AS parknumber FROM parks WHERE
   ↪  parkstatus = 'park' AND usedby = ?", userId.String())
9  if err != nil {
10         _ = tx.Rollback()
11         return nil, errors.Wrap(err, "select2 statement")
12 }
13
14 err = tx.Get(&stat, "SELECT COUNT(*) AS matchnumber FROM matches_history
   ↪  WHERE takerid = ? AND status = 'success' ", userId.String())
15 if err != nil {
16         _ = tx.Rollback()
17         return nil, errors.Wrap(err, "select3 statement")
18 }
19 ...
```

Listing 3.3. Codice per il calcolo dei dati in `CarPark`.

Per ogni tipologia di dati raccolti (sottosezione 2.3.2) si effettua la query per prelevare dalla tabella "parks" il tempo di ricerca e la distanza, e si calcola la somma e la media con i nuovi dati. Successivamente, si calcola il numero di parcheggi effettuati dall'utente, si procede quindi con il calcolo del numero di parcheggi avvenuti con lo scambio tra utenti. Vengono cioè considerati tutti quei parcheggi dove l'utente è stato coinvolto come *taker* e in cui il match è andato a buon fine, sottraendo questo dato al totale dei parcheggi si ottiene il numero di parcheggi senza scambio. Anche questi valori vengono inseriti nella tabella delle statistiche. Tutti questi nuovi dati vengono associati, grazie anche alla flessibilità garantita da Go, ai corretti campi della varibile `stat`, che gestisce i valori da restituire al front-end.

Nella parte di codice che segue si effettua l'aggiornamento dati sull'utente.

```
_, err = tx.Exec("INSERT INTO user_stat SET userid = ? ON DUPLICATE KEY
 ↪  UPDATE parknumber = ?,  totaltimesearching = ?, totaldistance = ?,
 ↪  avgtimesearching = ?, avgdistance = ? , matchnumber = ?, nomatchnumber
 ↪  = ?", userId.String(), stat.ParkNumber, stat.TotalTimeSearching,
 ↪  stat.TotalDistance, stat.AvgTimeSearching, stat.AvgDistance,
 ↪  stat.MatchNumber, stat.NoMatchNumber)
if err != nil {
        _ = tx.Rollback()
        return nil, errors.Wrap(err, "update1 statement")
}
```

Listing 3.4. Aggiornamento dei dati sull'utente.

Da notare che per l'aggiornamento dei dati è stata usata la dicitura `INSERT ... ON DUPLICATE KEY UPDATE`, in questo modo se la tupla che identifica l'utente già esiste nella tabella, si effettua solo l'aggiornamento dei dati, altrimenti viene creata. Questa operazione (come anche le precedenti) viene racchiusa in una transazione[8] in modo tale da mantenere il database in uno stato coerente anche in caso di errori imprevisti oppure in caso di aggiornamento simultaneo della risorsa. Se quindi sorge un errore viene annullata l'intera transazione e riportati i valori allo stato precedente (ovvero si fa un **roolback**).

Per il calcolo delle statistiche sulle auto l'idea è analoga a quella per il calcolo delle statistiche sugli utenti. Di seguito è riportata la parte di codice interessata.

[8] Una transazione fa in modo di rendere un blocco di istruzioni atomico: o avvengono con successo tutte le istruzioni oppure in caso di errore tutto resta invariato. Permettono il corretto ripristino dagli errori e mantengano coerente un database anche in caso di errore di sistema, quando l'esecuzione si interrompe e le operazioni potrebbero rimanere incomplete, mettendo il database in uno stato non previsto. Forniscono inoltre isolamento tra i programmi che accedono contemporaneamente a un database.

```
err = tx.Get(&stat, "SELECT AVG(searchingdistance) AS avgcardistance,
  ↪  SUM(searchingdistance) AS totalcardistance, SUM(searchingtimesec) AS
  ↪  totalcartime, AVG(searchingtimesec) AS avgcartime FROM parks WHERE cid
  ↪  = ? AND parkstatus = 'park' AND searchingtimesec > 0", cid.String())
if err != nil {
        _ = tx.Rollback()
        return nil, errors.Wrap(err, "select1 statement")
}

err = tx.Get(&stat, "SELECT COUNT(*) AS parknumber  FROM parks WHERE cid =
  ↪  ? AND parkstatus = 'park'", cid.String())
if err != nil {
        _ = tx.Rollback()
        return nil, errors.Wrap(err, "select2 statement")
}

err = tx.Get(&stat, "SELECT COUNT(*) AS matchnumber FROM matches_history
  ↪  WHERE takercid = ? AND status = 'success'", cid.String())
if err != nil {
        _ = tx.Rollback()
        return nil, errors.Wrap(err, "select3 statement")
}
```

Listing 3.5. Aggiornamento dei dati sull'utente.

Sebbene inizialmente l'idea fosse quella di delegare il calcolo e gli aggiornamenti dei dati ad API separate, ci si è accorti che la soluzione migliore era quella di sfruttare una stessa API sia per il calcolo sia per l'aggiornamento. Infatti, in questo modo la parte di front-end non deve chiamare l'API che effettua i calcoli ogni volta che si accede alla schermata di presentazione dei dati, anche se questi non sono cambiati, ma è sufficiente solo che chiami le API che effettuano la `GET` delle statistiche. Queste API sono presentate nel prossimo paragrafo.

3.2 Ottenimento statistiche sull'utente e sulle auto

GeneroCity è stata dotata di due API, per il prelievo di statistiche utente e auto rispettivamente. Anche se queste due API sono state implementate in modo separato, il funzionamento è pressoché analogo: lo scopo infatti è quello di prelevare i dati inerenti e all'occorrenza riportali sotto forma di risposta HTTP. In particolare, la parte principale (quella che gestisce input, output e chiamate al database) dell'API `getMeStatistics` che si occupa delle statistiche dell'utente opera nel seguente modo:

```
...
userStat, err := rt.db.GetMeStatistics(ctx.UserId)
if err != nil {
        var errorMessage types.ErrorMessage
        errorMessage.ErrorMessage = "can't get user statistics from DB"
        ctx.Logger.WithError(err).Error(errorMessage)
        w.Header().Set("content-type", "application/json")
        w.WriteHeader(500)
        _ = json.NewEncoder(w).Encode(errorMessage)
} else if userStat == nil {
        var errorMessage types.ErrorMessage
        errorMessage.ErrorMessage = "user not found"
        ctx.Logger.WithError(err).Error(errorMessage)
        w.Header().Set("content-type", "application/json")
        w.WriteHeader(404)
        _ = json.NewEncoder(w).Encode(errorMessage)
} else {
        w.Header().Set("content-type", "application/json")
        _ = json.NewEncoder(w).Encode(userStat)
}
```

Listing 3.6. Gestione output API `getMeStatistics`.

Dopo aver controllato che l'id dell'utente sia della forma corretta, viene chiamata la funzione `GetMeStatistics(ctx.UserId)` che preleva i dati di quell'utente dal database. Il risultato viene associato alla variabile `userStat`, una struct Go, che implementa i dati da riportare in output come visto nella sottosezione 2.3.2. Quindi, in base all'esito delle operazioni sul database, viene restituito un determinato output. In caso non si riesca ad interagire col database o fallisce qualche operazione su di esso, viene restituito il codice di stato 500 con il messaggio di errore esplicativo. Se invece l'interazione con il database va a buon fine ma non si riesce a prelevare i dati (in teoria questa condizione è estremamente rara) viene ritornato 404. Infine se tutto va come deve, si restituisce il codice 200 e la struct `userStat` sotto forma di JSON. Nel codice appena mostrato è visibile l'intera costruzione del messaggio di risposta HTTP, oltre al codice di stato. In particolare, con `content-type` è indicato il tipo di output (usando la dicitura richiesta dal protocollo), e quindi la struct Go che rappresenta l'output viene convertita con il metodo `json.NewEncoder` in un JSON, leggibile dal front-end. Un esempio di corretto output di questa API è il seguente:

```
{
    "userid": "62e1550d-72a7-4256-8b24-677b05cb58f2",
    "totaltimesearching": 1351,
    "totaldistance": 2.78237,
    "avgtimesearching": 75.0556,
    "avgdistance": 0.154576,
    "matchnumber": 12,
    "nomatchnumber": 6,
    "parknumber": 18
}
```

Listing 3.7. Esempio di un output dell'API per le statistiche sull'utente.

Il comportamento dell'API per le statistiche sulle auto è similare. Nell'URL viene passato l'id dell'auto (`/car/:cid/statistics`, dove al posto di `:cid` va inserito l'id dell'auto di cui si vuole ottenere le informazioni), viene fatto un controllo di correttezza dell'id, e poi viene verificato che l'utente che chiama l'API sia effettivamente associato all'auto, secondo quanto riportato nella opportuna tabella del database. Se il controllo è positivo, viene ritornata la struttura con i dati visti nella sottosezione 2.3.2, altrimenti si ritorna un errore con codice 403, che indica che l'utente non è autorizzato ad accedere alla risorsa:

```
...
b, err := rt.db.IsOwner(types.Driver{Id: carStat.UserId, Cid: carStat.Cid})
if err != nil {
    var errorMessage types.ErrorMessage
    errorMessage.ErrorMessage = "can't check the ownership"
    ctx.Logger.WithError(err).Error(errorMessage)
    w.Header().Set("content-type", "application/json")
    w.WriteHeader(500)
    _ = json.NewEncoder(w).Encode(errorMessage)
    return
} else if !b {
    var errorMessage types.ErrorMessage
    errorMessage.ErrorMessage = "user not authorized"
    ctx.Logger.WithError(err).Error(errorMessage)
    w.Header().Set("content-type", "application/json")
    w.WriteHeader(403)
    _ = json.NewEncoder(w).Encode(errorMessage)
    return
}
...
```

Listing 3.8. API `getCarStatistics`

Le API sono poi state testate sia simulandone il funzionamento con Postman (sezione 1.4), sia realmente usando l'applicazione sul campo. Successivamente, ho scritto la documentazione seguendo lo standard OpenAPI, spiegando il funzionamento, le precondizioni, e gli output. La scrittura della documentazione è importante, poiché permette di far capire il funzionamento del codice anche a sviluppatori che

subentreranno in futuro, come è auspicabile in contesti del genere. OpenAPI permette infatti, mediante poche righe di codice basato su parole chiave, di auto-generare specifiche di un programma che siano leggibili sia da programmi che da persone interessate a capirne il funzionamento e anche di testarle.

Conclusioni

Al termine del periodo di tirocinio ho contribuito all'aggiornamento del codice e allo costruzione di statistiche relative alle fase di ricerca del parcheggio, sviluppando due nuove API e integrandone una già esistente. Prima dell'aggiunta di queste nuove funzionalità GeneroCity era già un applicazione funzionante, ma era necessario aggiungere una sezione per le statistiche per evidenziare all'utilizzatore l'impiego di tempo, e quindi indirettamente anche dei costi, di ricerca di parcheggio.
Questa relazione riporta ciò che ho imparato durante tutto il periodo, e come sono arrivato a poter sviluppare un codice che perseguisse lo scopo prefissato. Ho avuto la possibilità di acquisire nuove conoscenze che si sono rivelate utili a portare a termine il lavoro. Mi sono interfacciato con strumenti per lo sviluppo di software in maniera pratica e agile, ho utilizzato il linguaggio di programmazione Go, il cui utilizzo è in ampia crescita, e la piattaforma GitLab. Ho ampliato la mia conoscenza sul protocollo HTTP, su come esso comunica e come è formato, e sui vari codici di stato. Questo stesso protocollo è servito poi per capire il funzionamento dietro alle API. Ho potuto sperimentare l'importanza che le API hanno in ambito informatico, e le ampie possibilità che esse forniscono. Infine ho potuto lavorare con i file JSON e apprezzarne la struttura, capendo il motivo per cui essi hanno un ampio utilizzo nello sviluppo software. Ho imparato inoltre a costruire interazioni con un database. Tutto questo lavorando in contatto con altre persone e seguendo quanto più possibile la metodologia agile, ragionando quali fossero i problemi e come poterli risolvere.
Complessivamente considero questa esperienza formativa non solo sul piano tecnico ma anche su quello personale. Mi ha dato la possibilità di vivere un ambiente simile a quello lavorativo, basato sulla collaborazione e sul conseguimento degli obiettivi prefissati, imparando a gestire il tempo e come affrontare i problemi.

Vorrei ora lasciare alcune possibili idee per sviluppi futuri. Le statistiche che ho sviluppato possono essere ampliare, per esempio aggiungendo informazioni inerenti al carburante consumato, ai costi dello stesso, e all'impatto ambientale relativo alla riduzione di emissioni CO_2, o inserendo quanto tempo si impiega a cercare parcheggio in una determinata zona di interesse. Si potrebbe anche aggiungere una classifica o un badge per segnalare gli utenti virtuosi, in modo da incentivare l'uso dell'applicazione, basata per esempio sul numero di parcheggi lasciati ad altre persone o sui parcheggi trovati con essa. Queste innovazioni potrebbero rendere l'interazione tra l'app e utenti più divertente.

Bibliografia

[1] A. Neha, J Cook, R. Kumar, I. Kuznetsov, Y. Li, H.J. Liang, A. Miller, A. Tomkins, I. Tsogsuren, I. Wang "*Hard to Park? Estimating Parking Difficulty at Scale*", Association for Computing Machinery, 2019, https://doi.org/10.1145/3292500.3330767

[2] Go (programming language), https://en.wikipedia.org/w/index.php?title=Go_(programming_language)&oldid=1055413071

[3] Introduzione a GitLab, https://www.coretech.it/it/service/knowledge_base/Programmazione/GitLab/Introduzione-a-GitLab.php

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

[5] Datrevo, Postman per testare le API, http://www.datrevo.com/postman-per-testare-le-api/

[6] What is an API?, https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces, 2017

[7] Hypertext Transfer Protocol, https://it.wikipedia.org/w/index.php?title=Hypertext_Transfer_Protocol&oldid=12307540,

[8] Haversine formula, https://en.wikipedia.org/wiki/Haversine_formula

[9] Documentazione package GO encoding/json, https://pkg.go.dev/encoding/json#Unmarshal