SAPIENZA
UNIVERSITÀ DI ROMA

Progettazione e Sviluppo di un simulatore di parcheggio basato sulle API GeneroCity

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Dipartimento di Informatica

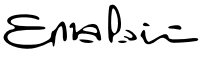
Corso di laurea in Informatica

Lorenzo Passaretti

Matricola 1910723

Relatore

Emanuele Panizzi



A.A. 2022-2023

Contents

Chapter 1

Introduzione

1.1 L'applicazione Generocity

GeneroCity [1] è un'applicazione di smart parking per Android e iOS sviluppata dal Gamification Lab del Dipartimento di Informatica dell'Università degli Studi di Roma "La Sapienza".
Lo scopo dell'applicazione è quello di facilitare lo scambio dei parcheggi all'interno di un'area urbana puntando sulla generosità degli utenti.
L'applicazione consente inoltre di gestire le informazioni delle proprie automobili e di condividerle con i propri familiari.

Figure 1.1: Logo di Generocity

1.2 Funzionamento

Il funzionamento di questa applicazione si basa su due ruoli fondamentali: l'utente giver e l'utente taker. Quando un utente vuole lasciare un parcheggio (giver) può segnalarlo sul suo smartphone indicando un orario approssimativo di quando andrà via. Un utente che invece sta cercando un parcheggio (taker), può indicare un'area in cui ha intenzione di parcheggiare ed un orario.
Quando il server trova un utente giver ed uno taker nella stessa area, con un orario simile e con una macchina di simili dimensioni li abbinerà, creando così un match. Una volta giunto il momento segnato dal match, il giver lascerà il suo parcheggio mentre il taker lo occuperà, completando così il match.
Questo processo di scambio è incentivato da un sistema a punti, il quale premierà i giver che completeranno il loro match.

1.3 Architettura e Backend di Generocity

L'architettura di Generocity si basa sul modello client-server. In questa relazione verrà trattato solamente il lato backend, essendo stato quello il tema del mio tirocinio.
Il lato backend è composto da due interfacce che permettono l'interazione fra l'utente e il database: **Router** e **AppDatabase**.
Il Router processa le informazioni ricavate da una richiesta HTTP ricevuta, tramite l'**handler** il quale riceve la richiesta ed effettua la chiamata API corretta, la quale processa e passa le informazioni ad AppDatabase che si occuperà di eseguire tutte le query necessarie al database per soddisfarla.
Il backend di GeneroCity è scritto nel linguaggio di programmazione "Go" [2] e utilizza un database MariaDB per memorizzare tutti i dati necessari.

1.4 Perché usare Generocity?

Tramite l'utilizzo di Generocity, un automobilista riuscirà a trovare parcheggio con molta più facilità rispetto ad uno che non la utilizza. Questo beneficio comporta un minor tempo di utilizzo dell'automobile, con una conseguente riduzione del traffico cittadino e quindi dell'inquinamento automobilistico.

1.5 Analisi dei competitor

Le altre applicazioni attualmente sul mercato si limitano ad offrire una prenotazione di parcheggi gestiti ed a pagamento. Queste applicazioni sono sicuramente utili ma non risolvono il problema del cercare un parcheggio libero senza dover pagare. Per questo l'app Generocity, una volta diventata abbastanza popolare, risulterà come un applicazione fondamentale per qualunque automobilista cittadino.

1.6 Il tirocinio

Il sito del Gamification Lab [3] descrive il percorso di tirocinio come:

> Ogni progetto è un gruppo di lavoro: quando si entra nel progetto da tirocinante/tesista si lavora in gruppo con altri studenti. In un primo periodo si inizierà a prendere confidenza con gli strumenti di lavoro e con il progetto, poi successivamente si passerà a piccoli lavori (risolvere piccoli bug, altri cambiamenti) ed infine, una volta preso "confidenza" con il progetto e con gli strumenti di lavoro, si sceglie un argomento specifico che costituisce il proprio lavoro di tirocinio/tesi.[4]

Seguendo questa premessa, nei mesi passati al laboratorio mi sono stati assegnati vari task, partendo dai più piccoli bug per prendere dimestichezza col codice, fino ad arrivare al task principale, con riunioni settimanali per aggiornarci sull'avanzamento dei task ed assegnarne di nuovi.
Per poter condividere il codice e gestire l'assegnazione delle task, è stato utilizzata la piattaforma GitLab [5], la quale offre una interfaccia web per la gestione della repository Git [6] del progetto.

Chapter 2

Primi Task

Come accennato nel capitolo 1.6, durante il tirocinio mi sono stati assegnati diversi task, sempre più importanti. Tra questi task possiamo individuare i seguenti:

- Risolvere problemi riguardanti API precedentemente sviluppate;
- Abilitare i linters disattivati;
- Design e sviluppo di nuove funzionalità riguardante il salvataggio dei file ottenuti tramite bluetooth;
- Progettazione di una simulazione tra bot per verificare il corretto funzionamento dell'applicazione.

In questo capitolo verranno analizzati i primi tre tipi di task.

2.1 Minor Bug

La prima tipologia corrisponde ad una serie di piccoli task svolti nel primo mese di tirocinio per prendere dimestichezza con il progetto e gli strumenti usati.
Durante questo primo periodo ho apportato piccole modifiche al progetto per risolvere alcuni task segnalati da altri tirocinanti come "minor bug".

2.1.1 ReadAll in updateMeNickname is deprecated

Questo primo task richiedeva la rimozione dell'ormai deprecato pacchetto "io/ioutil".
La deprecazione di questo pacchetto è avvenuta con l'aggiornamento 1.16 [7] di Go poiché reputato come una "raccolta di cose mal definita e difficile da capire". Per questo motivo ho sostituito tutti gli utilizzi di **ioutil.ReadAll(r.Body)** con la funzione **io.ReadAll(r.Body)** del pacchetto "io".

2.1.2 Remove github.com/pkg/errors dependency

Similmente al task precedente, anche il pacchetto "github.com/pkg/errors" è risultato obsoleto dopo l'aggiornamento 1.13 [8] di Go il quale ha introdotto una nuova gestione degli errori direttamente nella libreria standard di Go. A causa di ciò è risultato necessario rimpiazzare tutti i suoi utilizzi con le funzioni della libreria standard.
Nel dettaglio ho rimpiazzato tutti gli **errors.New()** ed i **errors.Wrap()** con **fmt.Errorf**, rimuovendo così "github.com/pkg/errors" completamente.

2.2 Enable all linters in '.golang-ci.yml' and fix all issues

Il progetto di Generocity utilizza diversi linters [9] per analizzare il codice e segnalare errori, bug, errori stilistici e costrutti sospetti ma al mio arrivo alcuni di questi linters erano disattivati perché aggiunti dopo l'avvio del progetto. Il mio compito è stato quindi di attivare tutti i linters in quel momento disattivati e risolvere tutti i problemi che avrebbero inevitabilmente segnalato nel codice.
Ecco una lista di tutti i linters attivati e di cosa si occupano:

- **errorlint**: questo linter trova codice che causerà problemi con il wrapping degli errori. Ad esempio l'operazione "err == ErrFoo" segnalerà un errore perché comparare wrapped errors con il "==" potrebbe non funzionare. Bisogna quindi sostituire tutte queste comparazioni presenti nel codice con la funzione **errors.Is(err, ErrFoo)**;

- **forcetypeassert**: questo linter trova asserzioni di tipo di cui non vengono gestiti gli errori;

- **goconst**: questo linter trova tutte le stringhe che vengono utilizzate molteplici volte, le quali potrebbero quindi essere rese delle costanti;

- **gocritic**: questo linter verifica la presenza di bug, prestazioni e problemi di stile;

- **nestif**: questo linter trova le istruzioni if annidate che rendono il codice difficile da leggere calcolandone la complessità e segnalando quelle che ne hanno una troppo elevata;

- **prealloc**: questo linter trova le dichiarazioni di slice che potrebbero essere potenzialmente preallocate per evitare una riallocazione successiva alla loro creazione;

- **revive**: questo linter fornisce un framework per lo sviluppo di regole personalizzate da applicare al codice. Nel nostro caso revive ha segnalato una grande varietà di

errori come il mancato utilizzo del CamelCase nei nomi delle variabili oppure un utilizzo non necessario delle else;

- **unconvert**: questo linter trova conversioni di tipo non necessarie;
- **wastedassign**: questo linter trova istruzioni di assegnazione sprecate.

2.3 Allow to modify "modes" field in bluetooth file

L'applicazione, durante un viaggio, raccoglie alcuni dati tramite bluetooth per capire che mezzo di trasporto l'utente stia utilizzando e li immagazzina in un object storage chiamato MinIO.
Col tempo però è risultato necessario un modo per poter modificare i dati immagazzinati nel caso in cui l'applicazione abbia sbagliato nel riconoscere il mezzo utilizzato.

2.3.1 Strutture Dati

Le scansioni effettuate durante un viaggio vengono rappresentate in questa struttura dove:

- **Trip**: l'id del trip da cui sono stati raccolti i dati;
- **Mode**: il mezzo di trasporto su cui si è svolto il viaggio;
- **Scans**: le scansioni effettuate.

```
type BluetoothConnectionsStruct struct {
  Trip  *int         `json:"trip,omitempty"`
  Mode  *string      `json:"mode"`
  Scans []ScanStruct `json:"scans"`
}
```

2.3.2 Modifiche al salvataggio dei file

I dati relativi ad un singolo viaggio venivano salvati in un file JSON dal nome *bluetoothconnections/**userID_time**.json* dove **userID** era l'id dell'utente che ha effettuato il viaggio e **time** era il timestamp di ricezione dei dati da parte del server. Salvare i file con questo nome rendeva però impossibile ricercare il file corretto che l'utente ha intenzione di modificare dato che l'applicazione non conosce il tempo preciso in cui il file è stato memorizzato. Ho quindi modificato il nome con cui i file venivano salvati, passando dal vecchio **userID_time.json** a **tripID.json**. In questo modo l'applicazione, per richiedere la modifica di un file, dovrà specificare solo l'id del trip dov'è stato rilevato

l'errore.
Oltre ai nomi è risultato anche che al momento del salvataggio dei file non venivano effettuati controlli se il file inserito fosse effettivamente valido. Per effettuare questi controlli ho quindi aggiunto due piccole funzioni a quella di salvataggio dei file.
La funzione **CheckTripID** controlla che nel database esista realmente un trip con id uguale a quello del trip di cui si stanno salvando le connessioni bluetooth.

```
func (db *appdbimpl) CheckTripID(tripID int) (bool, error) {
  var ret int
  err := db.c.Get(&ret, "SELECT COUNT(*) FROM trips WHERE `id` = ?", tripID)
  if err != nil {
    return false, fmt.Errorf("select1 statement: %w", err)
  } else if ret > 0 {
    return true, nil
  }
  return false, nil
}
```

La funzione **CheckTripUser** controlla che l'utente che sta salvando il file è lo stesso che ha effettuato il trip.

```
func (db *appdbimpl) CheckTripUser(userID uuid.UUID, tripID int) (bool, error) {
  var parkingUser string
  err := db.c.Get(&parkingUser, "SELECT parkedby FROM parks WHERE `parkid` = ?", tripID)
  if err != nil {
    return false, fmt.Errorf("select1 statement: %w", err)
  }
  if userID.String() != parkingUser {
    return false, nil
  }
  return true, nil
}
```

2.3.3 Bluetooth-mode-update API

Ora che esiste un modo per riconoscere i file memorizzati ho potuto realizzare un nuovo endpoint API */bluetooth/{tripid}/* dove **tripid** indica l'identificatore del trip da modificare.
L'API usa i seguenti parametri:

- **tripid**: identificativo unico del trip da modificare;
- **mode**: stringa enum che corrisponde al corretto mezzo di trasporto con il quale si è svolto il trip ("walk", "metro", "car","tram", "bus").

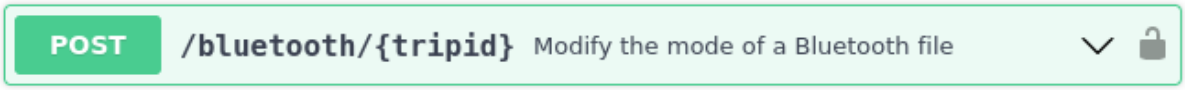


Figure 2.1: Path bluetooth update API

```yaml
/bluetooth/{tripid}:
  parameters:
    - $ref: "#/components/parameters/XId"
    - $ref: "#/components/parameters/XAppBuild"
    - $ref: "#/components/parameters/XAppVersion"
    - $ref: "#/components/parameters/XAppLanguage"
    - $ref: "#/components/parameters/XAppPlatform"
  post:
    tags: [ "Bluetooth" ]
    operationId: updateBluetoothMode
    summary: Modify the mode of a Bluetooth file
    description: "User sends the request using his ID, the trip's id and the new mode "
    security:
      - openid: [ ]
      - openid-dev: [ ]
    parameters:
      - name: tripid
        schema:
          type: integer
        in: path
        required: true
        allowEmptyValue: false
        example: 10
        description: "Name of the bluetooth file to modify. It corresponds to the tripId"
    requestBody:
      required: true
      content:
        application/plain:
            schema:
              type: string
              example: car
              enum: [ "walk", "metro","car","tram","bus" ]
              description: The new mode
    responses:
      "200":
        description: Success
      "400": { $ref: "#/components/responses/BadRequest" }
      "401": { $ref: "#/components/responses/UserNotAuthorized" }
      "404": { $ref: "#/components/responses/NotFound" }
      "500": { $ref: "#/components/responses/InternalServerError" }
```

Figure 2.2: Bluetooth update API documentazione

2.3.4 Implementazione API

Ho implementato l'API nella funzione **updateBluetoothMode**, la quale, dopo aver raccolto i dati, effettua alcuni controlli per verificare la validità della richiesta.

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

  newMode, err := io.ReadAll(r.Body)
  if err != nil {
    var errorMessage types.ErrorMessage
    errorMessage.ErrorMessage = "error reading request body"
    ctx.Logger.WithError(err).Error(errorMessage)
    w.Header().Set("content-type", "application/json")
    w.WriteHeader(http.StatusBadRequest)
    _ = json.NewEncoder(w).Encode(errorMessage)
    return
  }

  correctUser, err := rt.db.CheckTripUser(ctx.UserID, filename)
  if err != nil {
    var errorMessage types.ErrorMessage
    errorMessage.ErrorMessage = "Can't check if the user made that trip"
    ctx.Logger.WithError(err).Error(errorMessage)
    w.Header().Set("content-type", "application/json")
    w.WriteHeader(http.StatusInternalServerError)
    _ = json.NewEncoder(w).Encode(errorMessage)
    return
  } else if !correctUser {
    var errorMessage types.ErrorMessage
    errorMessage.ErrorMessage = "This trip was not made by that user"
    ctx.Logger.WithError(err).Error(errorMessage)
    w.Header().Set("content-type", "application/json")
    w.WriteHeader(http.StatusNotFound)
    _ = json.NewEncoder(w).Encode(errorMessage)
    return
  }

  message, err := rt.fs.UpdateBluetoothMode(context.TODO(), strconv.Itoa(filename),
  string(newMode))
  if err != nil {
    var errorMessage types.ErrorMessage
```

```
    if message == types.ErrorFileNotFound {
      errorMessage.ErrorMessage = "file not found"
      w.WriteHeader(http.StatusNotFound)
    } else {
      errorMessage.ErrorMessage = "can't update bluetooth connections file"
      w.WriteHeader(http.StatusInternalServerError)
    }
    ctx.Logger.WithError(err).Error(errorMessage)
    w.Header().Set("content-type", "application/json")
    _ = json.NewEncoder(w).Encode(errorMessage)
    return
  }
  w.WriteHeader(http.StatusOK)
}
```

Con la funzione **CheckTripUser** 2.3.2, viene controllato se l'utente che sta facendo la richiesta di modificare il file *filename* è lo stesso utente che ha effettuato quel viaggio ed ha quindi il permesso di modificarlo.
Una volta verificato che il viaggio esiste e che l'utente ha il permesso di effettuare la modifica, viene usata la funzione **fs.UpdateBluetoothMode**, la quale verifica che il file esista, legge il file, modifica la *Mode* salvata con quella indicata dall'utente e salva la modifica.

```
func (fs _fsimpl) UpdateBluetoothMode(ctx context.Context, filename string,
newMode string) (string, error) {
  var message string
  destinationFileName := path.Join(fs.cfg.Directory, "bluetoothconnections",
  fmt.Sprintf("%s.json", filename))

  _, err := os.Stat(destinationFileName)
  if os.IsNotExist(err) {
    return types.ErrorFileNotFound, err
  }

  byteValue, err := os.ReadFile(destinationFileName)
  if err != nil {
    return message, fmt.Errorf("opening json file: %w", err)
  }

  var bluetoothStruct types.BluetoothConnectionsStruct
  err = json.Unmarshal(byteValue, &bluetoothStruct)
  if err != nil {
    return message, fmt.Errorf("unmarshal json file: %w", err)
  }
  *bluetoothStruct.Mode = newMode

  fp, err := os.Create(destinationFileName)
  if err != nil {
```

```
    return message, fmt.Errorf("truncating destination file: %w", err)
  }
  defer func() { _ = fp.Close() }()

  err = json.NewEncoder(fp).Encode(bluetoothStruct)
  if err != nil {
    return message, fmt.Errorf("converting connections to JSON: %w", err)
  }
  return message, nil
}
```

Chapter 3

Simulatore di parcheggio

Prima del mio arrivo nel Gamification Lab, il mio collega Marco Wang per la sua tesi triennale [10], aveva creato una prima versione di un bot che simulava il comportamento di un taker, con l'obbiettivo di verificare l'efficacia e la validità del sistema di match.
Questa prima versione di simulazione era però migliorabile aggiungendo un secondo tipo di bot per simulare il comportamento di un giver ed una interfaccia per mostrare il match in tempo reale.
In questo capitolo, si espone la nuova versione del simulatore di parcheggio tra bot con tutte le aggiunte esposte sopra, nonché il tema principale del mio tirocinio.

3.1 Funzionamento

Come spiegato nella sezione 1.2, la simulazione sfrutta il meccanismo di match, il quale prevede due ruoli:

- **Taker**: l'utente che vuole occupare un parcheggio;
- **Giver**: l'utente che sta lasciando un parcheggio.

Vogliamo quindi realizzare una simulazione dove i bot che simulano il comportamento di un giver aspettano parcheggiati che avvenga un match con un bot che simula il comportamento di un taker.
Per fare in modo che tra due bot inizi un match, un bot parcheggiato dovrà inviare al server una richiesta contenente la sua posizione e l'orario di partenza, diventando così un giver, mentre un secondo bot che ha il compito di parcheggiare invia al server una richiesta con la sua posizione, il raggio di ricerca e l'orario desiderato per parcheggiare, diventando così un taker.
Il server è continuamente alla ricerca di possibili match, cercando di abbinare ogni taker con un giver. Per abbinare due bot ed iniziare così un match, il server controlla se la posizione di un giver rientra nell'area di ricerca di un taker e se i loro orari desiderati di "sparcheggio" e parcheggio coincidano. Se le due condizioni precedenti sono soddisfatte,

viene inizializzato un match tra i due bot.
Per avere il maggior numero di match possibili, il server divide le richieste in slot temporali da 15 minuti. Quindi, quando un taker o un giver invia il suo orario desiderato, esso verrà inserito nello slot di 15 minuti corretto. Ad esempio, se un giver inviasse come orario di partenza 11:43, il server lo inserirà nello slot temporale delle 11:45 mentre uno che imposta come orario di partenza 11:48, verrebbe inserito nello slot delle 12:00.

3.2 I Bot

Abbiamo quindi bisogno di due tipi di bot che simuleranno il comportamento dei due tipi di utenti, inviando richieste al server per iniziare e completare un match.
Il funzionamento di un bot è diviso in due parti :

- Configurazione
- Simulazione

Mentre la prima fase di configurazione avviene in maniera iterativa per tutti i bot necessari, la seconda fase di simulazione avviene in modo asincrono. Questo significa che tutti i bot eseguono la simulazione in parallelo sfruttando le **goroutine** [11], le quali rappresentano i thread in Go.

3.2.1 Configurazione

Per far partire il programma dei taker bot bisogna usare il seguente comando:

```
go run ./cmd/taker-bot --config-path ./demo/takerbot.yml
```

Similmente, per far partire i giver bot, si utilizza quest'altro comando:

```
go run ./cmd/giver-bot --config-path ./demo/giverbot.yml
```

Una volta in esecuzione inizia per entrambi la fase di inizializzazione, nella quale viene caricato il file di configurazione passato nei parametri dei comandi sopracitati. Questi file contengono tutte le informazioni necessarie riguardo la creazione dei bot e per assicurare la corretta esecuzione del programma. Queste informazioni sono divise in 6 categorie:

- **Bot**: contiene il numero di bot ed il nome che avranno;
- **Car**: contiene tutte le informazioni necessarie alla creazione e personalizzazione delle macchine che i bot useranno;
- **Header**: contiene gli header delle richieste HTTP che verranno utilizzate;
- **Location**: contiene le coordinate da cui un nuovo bot inizia ad operare ed il loro raggio d'azione;

- **Time**: contiene informazioni temporali come l'intervallo di tempo con cui opera il server e, solo per il taker, il tempo massimo e minimo che dovrà attendere un bot per percorrere un tratto di strada (vedi 3.2.4.4);
- **Host**: contiene l'url al quale fare le richieste e i file in cui salvare i bot creati.

```
# This is the configuration file for taker bot
log:
  level: debug
#Bot's info(name of the bot, and number of the bots that we want to create)
bot:
  name: "bottas"
  quantity: 2
#Car's info
car:
  mac: "ab:ac:dd:aa:aa:a"
  plate: "bbcasi"
  nickname: "botCardos"
  make: "fiat"
  model: "punto"
  color: "black"
  size: 1
#Header of the http request
header:
  contenttype: "application/json;"
  xappbuild: "1"
  xappversion: "1.0.0"
  xapplang: "it"
  xappplatform: "android"
#Where a new bot starts and the range of the area.
#It also indicates the center of the area where the simulation will take place
location:
  lat: "41.897501"
  lon: "12.515542"
  deltalat: "0.03"
  deltalon: "0.03"
  #deltalat and deltalon represents the radius of the area in which the taker will
  #search for a giver. 0.03 means (more or less) 3 km.
time:
  sleep: 2000 #number of milliseconds of each request
  mintime : 200 #min time the bot must wait to traver one segment of the road
  maxtime : 200 #max time the bot must wait to traver one segment of the road
  schedule: 15 #MUST be the same as the match schedule
host:
  url: "http://localhost:3000" #url of the host to send API requests to
  csvfile: "./cmd/taker-bot/config/bots.csv" #where to save the bot
```

3.2.2 Il file CSV

Per non dover creare nuovi bot ad ogni simulazione, quelli creati in una simulazione verranno riciclati per svolgere anche le successive. Per fare ciò, sfrutteremo due file CSV, uno per ogni tipo di bot, all'interno dei quali salveremo ogni bot creato.
Nel file CSV dei giver bot, verranno memorizzati l'id del bot e quello della sua macchina mentre per i taker bot, oltre ai due id precedenti, anche la latitudine e la longitudine di partenza.
Per dare anche un senso di continuità e realismo alle simulazioni, i bot delle due categorie verranno continuamente scambiati (salvando le loro informazioni nel file dell'altra categoria). Questo significa che, ad esempio, una volta che un giver avrà lasciato il suo parcheggio, le sue informazioni verranno salvate nel file CSV dei taker bot, diventando così un taker che partirà dalle coordinate da cui ha appena "sparcheggiato". Al contempo, una volta che un taker bot avrà occupato il parcheggio lasciato da un giver, salverà le sue informazioni nel file CSV dei giver bot, diventando a sua volta un giver.

	C1	C2
1	2118c3bf-f8d1-49bb-9f63-09739442ee82	72522b46-c93c-4450-b5b2-35dcb92a5482
2	84924452-cf61-4182-b3a1-3a0f912b98b5	4d0ba700-c8ba-4e83-b006-f0d83a6772d1
3	6590510c-bc66-4f98-a921-02c788010fba	b60ca985-553c-4395-b30f-484b6610ba9e
4	bdd8fb98-6ce9-409b-96fb-e32f6dc64ce9	9fd5994a-363d-441c-ae75-d6f5223d6963

Figure 3.1: File CSV dei giver

	C1	C2	C3	C4
1	0bf75950-e458-417e-a2c7-4c86ec685274	119a7b7d-3523-4c05-b377-e4e22b4a74b1	41.915154	12.507668
2	0b4d225a-af07-4de4-9c51-5b25a54f42ea	101066d0-353d-4aba-a315-23110ccafd3d	41.882557	12.524346
3	f64515ed-ae46-4a1f-b585-5d3f15e2f801	c457b57d-8477-46bb-8e72-93c407a76b61	41.912075	12.514915
4	963c4fe0-25ca-463d-baad-7abb9c8c2b28	ab2d4477-d95d-45bb-95ef-142fa7354bec	41.885190	12.514169

Figure 3.2: File CSV dei taker

3.2.3 Giver Bot

Il giver bot è il bot che simula il comportamento di un giver. Questo programma utilizza, per rappresentare un singolo bot, una struttura che raccoglie tutte le informazioni utili per ogni bot. Tra queste informazioni troviamo il suo nome, il suo id, l'id della sua macchina, lo schedule con il quale ha cercato un match, il timeslot in cui ha trovato un match e le coordinate alle quali ha parcheggiato.

```
type Bot struct {
  BotName       string
  UserID        uuid.UUID
  Cid           uuid.UUID
  GiverSchedule time.Time
```

```
  MatchSchedule string
  ParkLat       float64
  ParkLon       float64
}
```

3.2.3.1 Creazione

Una volta caricato il file di configurazione viene inizializzato il logger e viene letto il file CSV in cui vengono memorizzati tutti i giver bot creati fin'ora.
Se all'interno del file CSV non dovessero esserci abbastanza bot per raggiungere il numero richiesto nel file di configurazione, vengono creati nuovi bot tramite chiamate API, salvandoli nel file CSV per riutilizzarli in seguito.

```
// Step 3: Create and Load Bots info and writes the csv file
  var csv [][]string //  total bot
  csv = config.ReadCSV(logger, cfg.Host.CsvFile)

  // Create some bots if there aren't enough in the csv
  if len(csv) < cfg.Bot.Quantity {
    for i := len(csv); i < cfg.Bot.Quantity; i++ {
      // The newly created bot
      var bot types.Bot
      // The record to add in the csv file
      var record []string
      bot.BotName = fmt.Sprintf("%s%d", cfg.Bot.Name, len(csv)+1)
      // Create the bot
      uid, reqErr := requests.CreateUser(bot.BotName, logger, cfg)
      bot.UserID, _ = uuid.FromString(uid)
      if reqErr != nil {
        logger.Error(bot.BotName, " not created")
        continue
      } else {
        logger.Print(bot.BotName, " correctly created")
      }

      record = append(record, bot.UserID.String())

      // Create the car
      cid, reqErr := requests.CreateCar(bot.UserID, len(csv)+1, logger, cfg)
      bot.Cid, _ = uuid.FromString(cid)
      if reqErr != nil {
        logger.Error(bot.BotName, "'s car not created")
        continue
      }
      logger.Print(bot.BotName, "'s car correctly created")

      record = append(record, bot.Cid.String())
```

```
      csv = append(csv, record)
    }
    // Rewrite the csv file with the new bots added
    config.WriteCSV(logger, csv, cfg.Host.CsvFile)
  }
```

Per permettere un'esecuzione infinita della simulazione, da questo punto in poi tutto il codice sarà in un for infinito, all'interno del quale verranno caricati sempre nuovi bot che svolgeranno indipendentemente la loro simulazione.

3.2.3.2 Caricamento

Una volta creati i bot mancanti al raggiungimento della quantità richiesta, si passa alla fase di caricamento. In questa fase vengono caricati dal file CSV tutti i bot richiesti, ottenendo le informazioni di ognuno tramite chiamate API.

```
for {
    var botList []types.Bot //  available bot in runtime
    csv = config.ReadCSV(logger, cfg.Host.CsvFile)

    //  Load all the data relative to the bot from the csv file
    //  it stops when we reach the desired number of bots
    for i, record := range csv {
      if len(botList) >= cfg.Bot.Quantity {
        break
      }

      var bot types.Bot
      uid, _ := uuid.FromString(record[0])
      name, err := requests.GetUserNickname(uid, logger, cfg)
      if err != nil {
        logger.Warn("bot not retrieved")
      } else {
        bot.BotName = name
        logger.Print(bot.BotName, "'s info correctly retrieved")
        bot.UserID = uid
        bot.Cid, _ = uuid.FromString(record[1])

        // In this case the already created car is not valid,
        // so we must create a new one
        if !requests.ValidCar(uid, bot.Cid, logger, cfg) {
          cid, reqErr := requests.CreateCar(bot.UserID, i, logger, cfg)
          bot.Cid, _ = uuid.FromString(cid)
          if reqErr != nil {
            logger.Error(bot.BotName, "'s car not created")
            continue
          } else {
```

```
            csv[i][1] = bot.Cid.String()
            botList = append(botList, bot)
            logger.Print(bot.BotName, "'s car correctly created")
          }
        } else {
          botList = append(botList, bot)
          logger.Print(bot.BotName, "'s car correctly retrieved")
        }
      }

      // Generate random coordinates to park the car at a random parking spot
      err = requests.RandomCoordinates(&bot, logger, cfg)
      if err != nil {
        logger.Error(bot.BotName, " failed to generate coordinates for the park")
        continue
      }
      // The newly created car must be parked since the bot is a giver
      park := requests.ParkCarInfo{
        Lat:          bot.ParkLat,
        Lon:          bot.ParkLon,
        ParkTime:     time.Now().Format(time.RFC3339),
        ParkAccuracy: 0.25,
      }

      err = requests.ParkCar(bot.UserID, bot.Cid, park, logger, cfg)
      if err != nil {
        logger.Error(bot.BotName, " has failed to park")
        continue
      }
      logger.Print(bot.BotName, "'s car correctly parked")
    }
```

Essendo un giver bot, è necessario che il bot sia anche parcheggiato prima di iniziare la simulazione. Per questo motivo vengono generate delle coordinate casuali all'interno dell'area specificata nel file di configurazione. Non vogliamo però che il bot parcheggi in un edificio ma solo in un vero parcheggio ai lati di una strada in cui è possibile parcheggiare. Per trovare un parcheggio che risponda ai nostri requisiti, vengono eseguite delle richieste API ad OpenStreetMap [12], il quale ci restituirà le coordinate di tutte le strade sulle quali è possibile parcheggiare una macchina vicine alle coordinate casuali che abbiamo appena generato. Se OpenStreetMap non dovesse aver trovato nessuna strada valida vicino alle nostre coordinate, vengono generate delle nuove coordinate e fatta una nuova richiesta fino al trovare un parcheggio valido.

```
// RandomCoordinates Generate Random coordinates in an area to park
func RandomCoordinates(bot *types.Bot, logger logrus.FieldLogger,
cfg config.BotConfiguration) error {
  /*
```

```
    .
    .
    .
    Section to generate a set or random coordinates inside the
    requested area
    .
    .
    .
    */
    // In this next section we make the request to Open Street Map to find
    // the nearest road to the generated coordinates where the car can park

    // Construct the custom client.
    client := overpass.NewWithSettings("https://overpass-api.de/api/interpreter",
cfg.Bot.Quantity, http.DefaultClient)

    // Make the query.
    query := fmt.Sprint("[out:json];(way(around:50," + fmt.Sprintf("%f", cx+x) + ","
+ fmt.Sprintf("%f", cy+y) + ")
[highway~'^(primary|secondary|tertiary|residential)$'][name];);out geom;")
    result, err := client.Query(query)
    if err != nil {
      logger.Error(bot.BotName, " has failed to send API request to open street map")
      return err
    }

  // If the request didn't find any available roads we'll go back to generate
    // a new set of  random coordinates
    if len(result.Ways) > 0 {
      // The request could return 0 or many roads, but we just need one.
      for _, c := range result.Ways {
        // For each road we get many coordinates set but we only need one
        // (in our case, we use the center point of the road)
        i := len(c.Geometry) / 2
        bot.ParkLat = c.Geometry[i].Lat
        bot.ParkLon = c.Geometry[i].Lon
        return nil
      }
    }
    time.Sleep(time.Duration(2) * time.Second)
  }
}
```

Ora che i giver bot sono stati creati e parcheggiati è finalmente possibile iniziare la simulazione nella quale ogni bot agirà indipendentemente dagli altri giver.
In questo punto creiamo anche un canale per catturare i segnali SIGKILL e SIGINTERRUPT così che la simulazione non possa essere interrotta durante la sua esecuzione ma solo una volta che tutti i bot hanno terminato il loro match.

```
  // Step 4: Simulation
  // each bot is now independent of the others to complete the task
  // Catch SIGTERM or SIGINTERRUPT. By doing this we can't interrupt a not ended match
  cancelChan := make(chan os.Signal, 1)
  signal.Notify(cancelChan, syscall.SIGTERM, syscall.SIGINT)

  // Start the simulation for each bot
  for i := 0; i < len(botList); i++ {
    wg.Add(1)
    go simulate(botList[i], logger, cfg)
  }

  wg.Wait()
  // Check if any SIGTERM or SIGINTERRUPT signals were caught
  if len(cancelChan) >= cap(cancelChan) {
    sig := <-cancelChan
    logger.Println("Caught signal ", sig)
    break
  }
}
```

3.2.3.3 Simulazione del giver

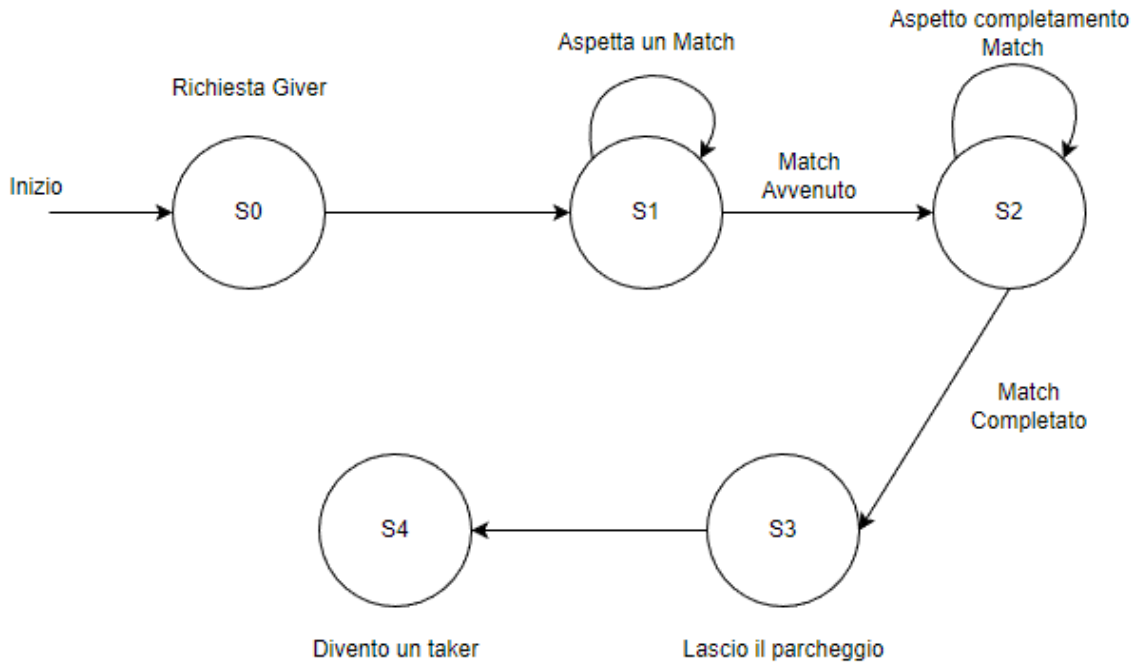


Figure 3.3: Automa della simulazione di un giver bot

Come possiamo vedere nella figura 3.3, la simulazione di un giver è composta da 5 stati diversi:

- **S0**: in questo primo stato, il bot invia al server una richiesta per segnalarsi come giver. Con questa richiesta, il bot invia anche il suo range di ricerca e l'orario in cui ha intenzione di lasciare il parcheggio;

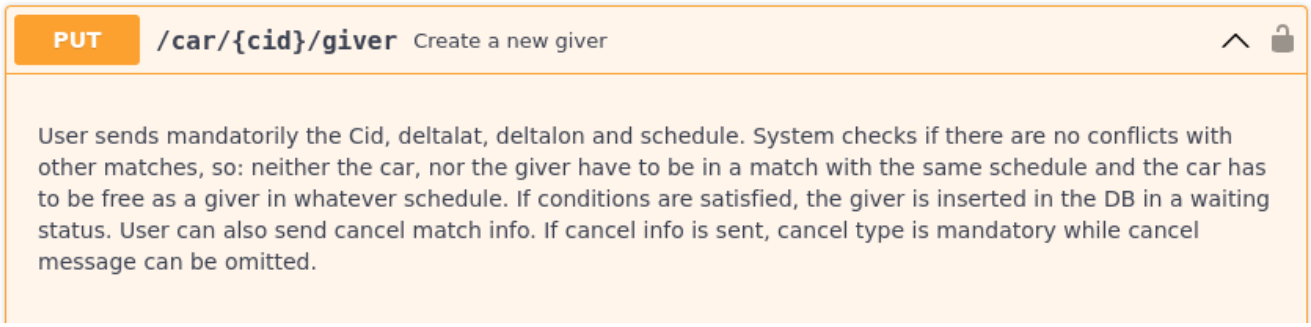


Figure 3.4: Richiesta API per segnalarsi giver

```
// For now, we're using this time for the schedule,
// but we could also change it for a custom one
bot.GiverSchedule = time.Now().Add(time.Minute)
// Send a giver request
err := requests.PutGiver(bot.UserID, bot.Cid, bot.GiverSchedule, logger, cfg)
if err != nil {
  logger.Error(bot.BotName, " has failed to send giver request")
  Wait(cfg.Time.Sleep)
  wg.Done()
  return
}
logger.Print(bot.BotName, " has correctly send the giver request")
```

- **S1**: in questo secondo stato, il giver attende che il server stabilisca un match tra di lui ed un taker facendo polling all'API di match, la quale restituirà informazioni riguardanti il taker che memorizzeremo per utilizzarle nell'ultima fase. Inoltre, viene anche creato un secondo canale per catturare i segnali SIGKILL e SIGINTERRUPT cosicché la simulazione di questo bot sia interrompibile se non dovesse riuscire a trovare un match;

```
// Catch SIGTERM or SIGINTERRUPT. We're using a second one to
// stop the simulation IF a match has not been found yet
  stopChan := make(chan os.Signal, 1)
  signal.Notify(stopChan, syscall.SIGTERM, syscall.SIGINT)

  // We'll need to memorize some taker information
  var takerInfo *requests.TakerInfo
  logger.Print(bot.BotName, " is waiting for matches...")

  for { // Waiting for a match
    Wait(cfg.Time.Sleep)
```

```
    // Check if any SIGTERM or SIGINTERRUPT signals were caught
    if len(stopChan) >= cap(stopChan) {
      sig := <-stopChan
      logger.Println("Caught signal ", sig)
      wg.Done()
      return
    }
    // Check for matches
    takerInfo,err = requests.GetMatchTakerInfo(bot.UserID, bot.Cid, logger, cfg)
    if err != nil {
      logger.Error(bot.BotName, " has failed to get match taker info")
      wg.Done()
      return
    }

    // If the giver request get matched, retrieve taker info
// and exit the waiting loop
    if !takerInfo.MatchSchedule.IsZero() {
      bot.MatchSchedule = takerInfo.MatchSchedule.Format(time.RFC3339)
      logger.Println(bot.BotName, " has found a match")
      break
    }
  }
// Save the Uid and Cid of the giver
  var oldBot []string
  oldBot = append(oldBot, bot.UserID.String())
  oldBot = append(oldBot, bot.Cid.String())

  // Save the Uid and Cid of the taker
  var newBot []string
  newBot = append(newBot, takerInfo.TakerID.String())
  newBot = append(newBot, takerInfo.TakerCid.String())

  park := requests.ParkCarInfo{
    Lat:          bot.ParkLat,
    Lon:          bot.ParkLon,
    ParkTime:     time.Now().Format(time.RFC3339),
    ParkAccuracy: 0.25,
  }
```

- **S2**: ora che un match tra questo bot ed un altro è iniziato, il giver bot deve attendere il completamento del match da parte del taker. Il match terminerà quando il taker avrà raggiunto la posizione del giver, segnalandolo al server. Per verificare se il match è stato concluso viene fatto il polling della stessa chiamata API usata per attendere l'avvio del match;

```
// Waiting for the match to end
```

```
for {
    Wait(cfg.Time.Sleep)
    takerInfo,err = requests.GetMatchTakerInfo(bot.UserID, bot.Cid, logger, cfg)
    if err != nil {
       logger.Error(bot.BotName, " has failed to get old match info")
       wg.Done()
       return
    }
    // Check if the old match ended
    if takerInfo.MatchSchedule.IsZero() {
       logger.Println(bot.BotName, " previous match ended")
       break
    }
}
```

- **S3**: ora che il match è stato completato il giver deve lasciare il posto in cui ha parcheggiato al taker, il quale lo occuperà subito dopo;

```
// Unpark the car in order to be able to park for the next simulation
  err = requests.UnparkCar(bot.UserID, bot.Cid, park, logger, cfg)
  if err != nil {
    logger.Error(bot.BotName, " has failed to unpark")
    wg.Done()
    return
  }
  logger.Println(bot.BotName, " has unparked correctly")
```

- **S4**: in quest'ultimo stato i due bot che hanno appena concluso il match, hanno terminato il loro lavoro e sono quindi pronti a scambiarsi. Per fare questo scambio il giver salverà nel suo file CSV l'id del taker e della sua macchina memorizzati alla fine dello stato S1, al posto del suo record. In questo modo il giver non sarà più un giver mentre il taker lo sarà diventato. Allo stesso modo, il taker farà lo stesso scambio nel suo file CSV.

```
  csv := config.ReadCSV(logger, cfg.Host.CsvFile)
  config.SubstituteCSV(logger, csv, oldBot, newBot, cfg.Host.CsvFile)

  logger.Println(bot.BotName, " became a taker")
  logger.Println("Waiting for every match to end")
  wg.Done()
```

Terminate tutte le 5 fasi ed effettuato lo scambio tra i due bot, il giver bot che ha appena concluso segnala di aver finito con il metodo **wg.Done**. Il programma attende quindi che tutti i bot terminino la loro esecuzione con **wg.Wait** 3.2.3.2, controlla eventuali segnali SIGKILL e SIGINTERRUPT e ricomincia il caricamento di una nuova serie di giver bot 3.2.3.2.

3.2.4 Taker Bot

Come per il giver bot, il taker bot è il bot che simula il comportamento di un taker. Anche i taker utilizzano una struttura simile ai giver per rappresentare un singolo bot. Tra le informazioni memorizzate in questa struttura troviamo il suo nome, il suo id, l'id della sua macchina, lo schedule con il quale ha cercato un match, il timeslot in cui ha trovato un match, le coordinate del giver con cui ha iniziato un match, le sue coordinate di partenza, l'id del trip che rappresenta il match e l'id della simulazione.

```
type Bot struct {
  BotName       string
  UserID        uuid.UUID
  Cid           uuid.UUID
  GiverSchedule time.Time
  MatchSchedule string
  GiverLat      float64
  GiverLon      float64
  StartingLat   float64
  StartingLon   float64
  CurrentTripID int
  SimulationID  int
}
```

3.2.4.1 Creazione e Inizio Simulazione

La creazione dei bot necessari è identica a quella del giver bot 3.2.3.1.
Una volta che siamo sicuri dell'esistenza di almeno un bot, se ne è appena stato creato uno o era già presente nel file CSV, possiamo considerata iniziata una simulazione, salvandola quindi nel database.

```
// Now that we have at least one bot ready start the simulation
  simulationID, err := requests.StartSimulation(csv[0][0], logger, cfg)
  if err != nil {
    logger.Error("failed to start simulation")
    return err
  }
```

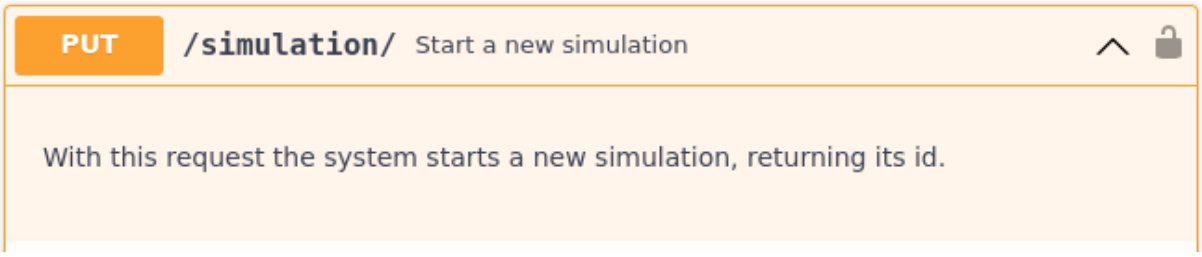


Figure 3.5: API per iniziare la simulazione

Con abbastanza bot nel file CSV e la simulazione ufficialmente iniziata possiamo iniziare, come per il giver bot, un ciclo infinito per permettere un'esecuzione continua della simulazione fino all'interruzione da parte dell'utente.

3.2.4.2 Caricamento

Il caricamento dei bot necessari dal file CSV è simile a quello del giver bot con l'eccezione che questa volta verranno memorizzate, per ogni bot, anche l'id della simulazione ottenuto quando la simulazione è iniziata, e le coordinate di partenza del bot, le quali erano state salvate nel file CSV quando erano un giver oppure nel processo di creazione che, in questo caso, saranno delle coordinate di default inserite nel file di configurazione. Inoltre, al contrario dei giver bot, i taker non verranno parcheggiati una volta caricati.

3.2.4.3 Ricerca Giver

A questo punto abbiamo caricato tutti i bot necessari in una lista. Dobbiamo quindi cercare dei giver con cui effettuare dei match ed assegnare ad ogni giver trovato, uno dei taker appena caricati.
Con la richiesta **requests.GetGiverList()**, ci verrà restituita una lista di giver bot disponibili per fare un match ed il loro schedule. Filtriamo ora tutti i giver trovati per cercare quelli compatibili con i nostri taker bot. Quelli compatibili per un match verranno salvati in una lista apposita ed assegnati, uno ad uno, ai taker bot necessari per soddisfarli tutti. Se dovessero essere presenti più taker bot che giver bot, quelli in eccesso verranno utilizzati in un prossimo ciclo della simulazione.

```
// Step 4: Search for giver requests and assign schedule
    // search for giver request until there are one or more valid giver requests
    var giverList []requests.GiverList

    logger.Print("searching giver..")
    for {
      // If we get a SIGKILL signal it's necessary to stop the simulation
      if len(cancelChan) >= cap(cancelChan) {
        sig := <-cancelChan
        logger.Println("Caught signal ", sig, ". Interrupting Simulation")
        // Stop the simulation by putting an EndTime in the db
        err = requests.StopSimulation(botList[0].UserID, simulationID, logger, cfg)
        if err != nil {
          logger.Error("failed to stop the simulation")
        }
        return nil
      }
      // Waiting the giver request
      Wait(cfg.Time.Sleep)
      giverList, err = requests.GetGiverList(botList[0].UserID, botList[0].Cid, logger, cfg)
```

```
    if err != nil {
      logger.Error("failed to get giver list")
    }

    if len(giverList) > 0 {
      var validTime = time.Now()

      // Take the valid schedule for bot to complete the match
      _, min, sec := validTime.Clock()
      if min%cfg.Time.Schedule != 0 || sec != 0 {
        validTime = validTime.Add(time.Duration(-(min%cfg.Time.Schedule)*60-sec) *
                  time.Second).Add(time.Duration(cfg.Time.Schedule) * time.Minute)
      }

      // Filter the request with a valid schedule
      n := 0
      for _, giver := range giverList {
        if giver.Schedule.Format(time.RFC3339) == validTime.Format(time.RFC3339) {
          giverList[n] = giver
          n++
        }
      }

      giverList = giverList[:n]

      if n > 0 {
        break
      }
    }
  }
  min := min(len(giverList), len(botList))
  logger.Print("found ", len(giverList), " givers, ", min, " requests are going to be fulfilled")

  // Assign giver schedule to the bot
  for i := 0; i < min; i++ {
    botList[i].GiverSchedule = *giverList[i].Schedule
  }
```

Ora che ogni taker ha il suo giver con cui poter effettuare un match è finalmente possibile iniziare la simulazione nella quale ogni bot agirà indipendentemente dagli altri. A differenza dei giver bot, i taker bot faranno in seguito utilizzo di un' API esterna la quale ha un limite di due richieste contemporaneamente. Per questo motivo viene creato un semaforo condiviso tra tutti i bot in modo tale che soltanto due bot alla volta potranno fare quella richiesta.

```
  // Step 5: Freedom: the bot is ready to simulate
  // each bot is now independent of the others to complete the task
```

```
    // The api we use to get the road only accepts 2 request at the time,
    // so we need a semaphore to limit them
    sem := make(chan bool, 2)

    // Start the simulation for each bot
    for i := 0; i < min; i++ {
      wg.Add(1)
      go simulate(botList[i], logger, cfg, sem)
    }
    wg.Wait()

    // Check if any SIGTERM or SIGINTERRUPT signals were caught
    if len(cancelChan) >= cap(cancelChan) {
      sig := <-cancelChan
      logger.Println("Caught signal ", sig, ". Interrupting Simulation")
      // Stop the simulation by putting an EndTime in the db
      err = requests.StopSimulation(botList[0].UserID, simulationID, logger, cfg)
      if err != nil {
        logger.Error("failed to stop the simulation")
      }
      break
    }
  }
```

Similmente al giver bot, anche il taker controllera a questo punto se sono state intercettati segnali SIGKILL o SIGINTERRUPT ma questa volta, quando il segnale sarà intercettato, la simulazione verrà interrotta inserendo nella tupla del database una data di fine simulazione.

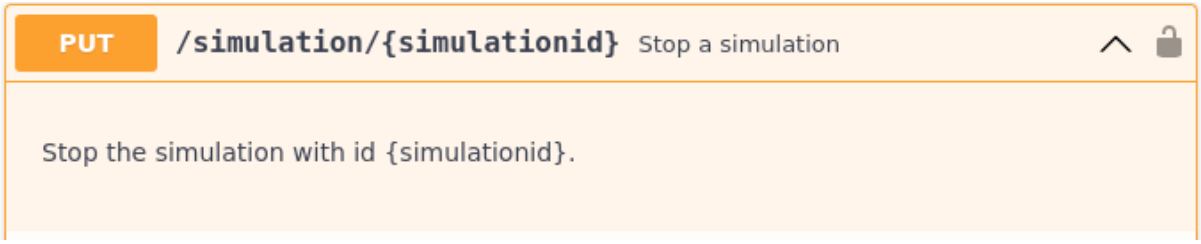


Figure 3.6: API per interrompere la simulazione

id	timeStart	timeEnd
78	2023-06-08 17:52:53	2023-06-08 17:53:52
79	2023-06-08 17:54:37	2023-06-08 17:55:38
80	2023-06-08 17:57:25	2023-06-08 17:59:21
81	2023-06-08 18:08:59	<null>

Figure 3.7: Simulazioni salvate nel database

3.2.4.4 Simulazione del taker

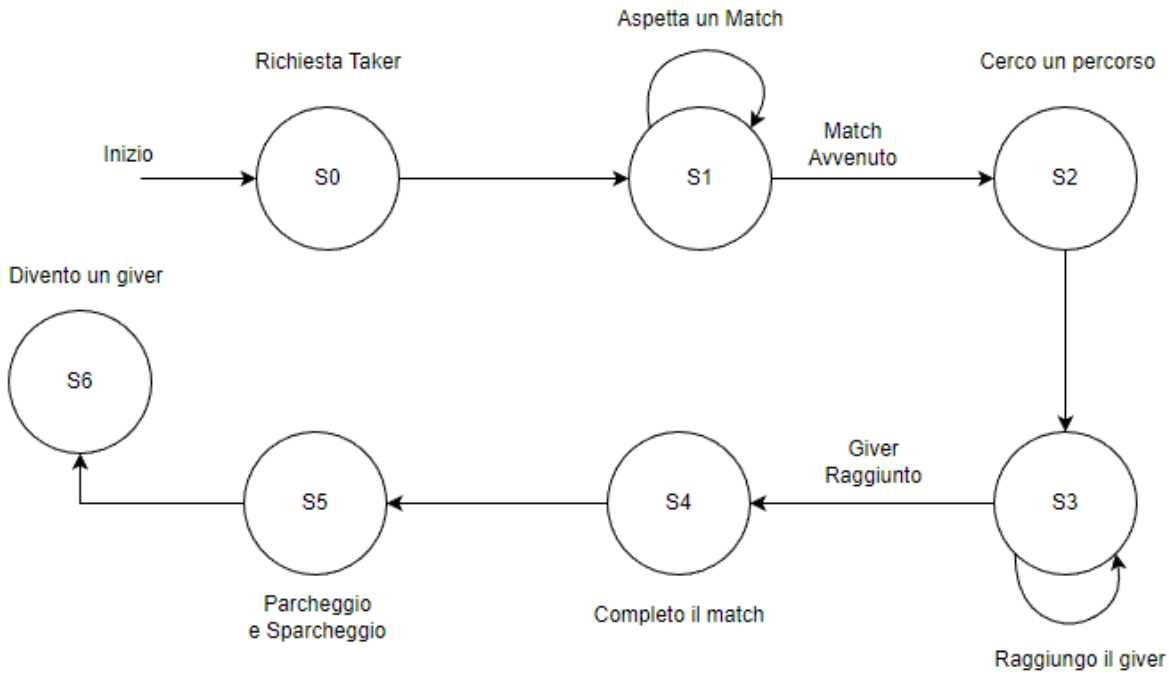


Figure 3.8: Automa della simulazione di un taker bot

Come possiamo vedere nella figura 3.8, la simulazione di un taker è composta da 7 stati diversi:

- **S0**: in questo primo stato, il bot invia al server una richiesta per segnalarsi come taker. Con questa richiesta, il bot invia anche le sue coordinate, il suo range di ricerca e l'orario;

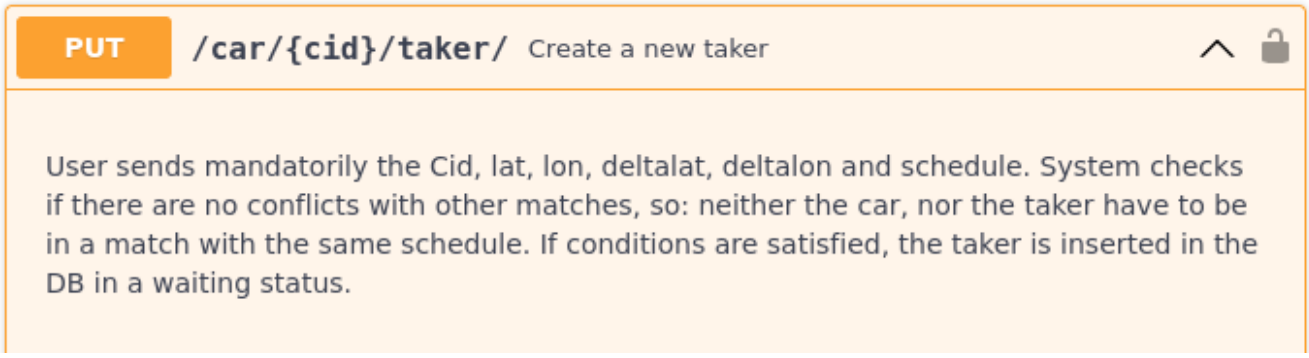


Figure 3.9: Richiesta API per segnalarsi taker

```
// Send a taker request
  err := requests.PutTaker(bot.UserID, bot.Cid, bot.GiverSchedule, logger, cfg)
  if err != nil {
    logger.Error(bot.BotName, " has failed to send taker request")
```

```
    wg.Done()
    return
  }
  logger.Print(bot.BotName, " has correctly send the taker request")
```

- **S1**: nel secondo stato, il taker attende che il server stabilisca un match tra di lui ed un giver facendo polling all'API di match, la quale restituirà informazioni riguardanti il giver che memorizzeremo per utilizzarle nell'ultima fase se c'è stato un match;

```
var giverInfo *requests.GiverInfo
  logger.Print(bot.BotName, " is waiting for matches...")

  for { // Waiting for the match to start
    Wait(cfg.Time.Sleep)
    giverInfo,err = requests.GetMatchGiverInfo(bot.UserID, bot.Cid, logger, cfg)
    if err != nil {
      logger.Error(bot.BotName, " has failed to get match giver info")
      wg.Done()
      return
    }
    // If the taker request get matched, retrieve giver info and exit the
    // waiting loop
    if giverInfo.GiverLat != 0 && giverInfo.GiverLon != 0 {
      bot.CurrentTripID = giverInfo.GiverParkID
      bot.GiverLat = giverInfo.GiverLat
      bot.GiverLon = giverInfo.GiverLon
      bot.MatchSchedule = giverInfo.MatchSchedule.Format(time.RFC3339)
      break
    }
  }
```

- **S2**: ora che il match è iniziato, il taker deve trovare una strada da percorrere per raggiungere il giver. Viene quindi usata la funzione **requets.FindPath()**, la quale effettua delle richieste API a Project OSMR [13] che restituirà un grande set di coordinate divise in segmenti, rappresentanti la strada più veloce che collega il taker con il giver.
  Come accennato in precedenza, Project OSMR permette solo due richieste alla volta. Sfruttiamo quindi il semaforo creato per dare il lock alla funzione a solamente due bot per volta.
  Per approfondire questa funzione vedasi la sezione 3.2.4.5;

```
// Now we save the simulationID in the trip tied to this match
// (we got it in the GetMatchGiverInfo)
  err = requests.UpdateTripSimulation(bot.UserID, bot.SimulationID,
    bot.CurrentTripID, logger, cfg)
```

```
if err != nil {
  logger.Error(bot.BotName," failed to add simulationId to the trip")
}

// Start notifying the user with his position
logger.Print(bot.BotName, " is updating taker position...")
// As stated before the api only allows 2 requests at the time
sem <- true
// Find a route to follow to get to the giver
path, pathLength, err := requests.FindPath(bot.GiverLat, bot.GiverLon,
   bot.StartingLat, bot.StartingLon)
if err != nil {
  logger.Error(bot.BotName, " has failed to find a path")
  wg.Done()
  return
}
Wait(1000)
func() { <-sem }()
```

- **S3** : in questo stato, il taker ha ottenuto un percorso valido per raggiungere il giver e non gli resta che seguirlo. Per dare un senso di realtà, il taker percorre ogni segmento dell'intero percorso in un tempo casuale, scelto tra il tempo massimo e minimo inseriti nel file di configurazione, moltiplicato per la lunghezza del segmento che si sta percorrendo. Questo implica che il taker impiegherà più tempo a percorrere un pezzo di strada lungo rispetto ad uno più corto.
  Inoltre, salviamo nel database ogni set di coordinate raggiunte dal taker. Salvare queste coordinate sarà necessario per visualizzare il percorso di ogni match 3.3;

```
// Follow said route
  for i, point := range path {
    // Save in the db the coordinates we're reaching
    err = requests.SaveCoordinates(bot.UserID, logger, cfg,   bot.CurrentTripID,
        point.Y(), point.X())
    if err != nil {
      logger.Error(bot.BotName, " has failed to update taker position")
    }

    takerPos := requests.TakerPosition{
      ParkLat:  point.Y(),
      ParkLon:  point.X(),
      Schedule: bot.MatchSchedule,
      Eta:      path.Length() - i,
    }

    // Calculate the time the taker will need to reach those coordinates
    randomNum := cfg.Time.MinTime
    if (cfg.Time.MaxTime - cfg.Time.MinTime) > 0 {
```

```
      bg := big.NewInt(cfg.Time.MaxTime - cfg.Time.MinTime)
      n, err := rand.Int(rand.Reader, bg)
      if err != nil {
        logger.Error(bot.BotName," has failed to generate a random wait number")
      }
      randomNum = n.Int64() + cfg.Time.MinTime
    }
    // The time needed is defined by a random time with a max and a min
    // written in the config file * the length of this road segment
    Wait(int(randomNum) * pathLength[i])
    // Update position
    err = requests.UpdateTakerPosition(bot.UserID,bot.Cid,logger,cfg, takerPos)
    if err != nil {
      logger.Error(bot.BotName, " has failed to update taker position")
    }
  }
```

- **S4**: una volta terminato il percorso da percorrere, il taker avrà raggiunto la posizione del giver e quindi il parcheggio che deve occupare. Deve quindi segnalare al server la sua intenzione di completare il match che a sua volta informerà anche il giver di poter lasciare il parcheggio;

```
// Complete the match
err = requests.CompleteMatch(bot.UserID, bot.Cid, bot.MatchSchedule, logger, cfg)
if err != nil {
  logger.Error(bot.BotName, " has failed to complete the match")
  wg.Done()
  return
}
logger.Println(bot.BotName, " has completed the match")
```

- **S5**: ora che il match è stato segnato come completato il giver starà lasciando il parcheggio. Il taker deve quindi occupare il parcheggio appena lasciato per poi lasciarlo subito dopo in modo tale da essere pronto per la sua prossima simulazione;

```
park := requests.ParkCarInfo{
    Lat:          bot.GiverLat,
    Lon:          bot.GiverLon,
    ParkTime:     time.Now().Format(time.RFC3339),
    ParkAccuracy: 0.25,
  }

// Park the car in order to assign a parkid to the match
err = requests.ParkCar(bot.UserID, bot.Cid, park, logger, cfg)
if err != nil {
  logger.Error(bot.BotName, " has failed to park")
  wg.Done()
```

```
  return
}
logger.Println(bot.BotName, " has parked correctly")

// Unpark the car in order to be able to park next time as a giver
err = requests.UnparkCar(bot.UserID, bot.Cid, park, logger, cfg)
if err != nil {
  logger.Error(bot.BotName, " has failed to unpark")
  wg.Done()
  return
}
logger.Println(bot.BotName, " has unparked correctly")
```

- **S6**: in quest'ultimo stato i due bot che hanno appena concluso il match sono pronti a scambiarsi. Per fare questo scambio il taker salverà nel suo file CSV l'id, l'id della macchina e la posizione del giver, al posto del suo record. In questo modo il taker non sarà più un taker mentre il giver lo sarà diventato. Allo stesso modo, il giver farà lo stesso scambio nel suo file CSV.

```
  // Save the Uid and Cid of the taker
  var oldBot []string
  oldBot = append(oldBot, bot.UserID.String())
  oldBot = append(oldBot, bot.Cid.String())

  // Save the Uid, the Cid and the park coordinates of the giver
  var newBot []string
  newBot = append(newBot, giverInfo.GiverID.String())
  newBot = append(newBot, giverInfo.GiverCid.String())
  newBot = append(newBot, fmt.Sprintf("%f", bot.GiverLat))
  newBot = append(newBot, fmt.Sprintf("%f", bot.GiverLon))

  // Now we replace the taker with the giver in the taker's csv file.
  // By doing this a giver becomes a taker and will
  // start from where it left as a giver.
  // (The old giver does the same thing for the taker,
  // allowing it to become a giver)
  csv := config.ReadCSV(logger, cfg.Host.CsvFile)
  config.SubstituteCSV(logger, csv, oldBot, newBot, cfg.Host.CsvFile)

  logger.Println(bot.BotName, " became a giver")
  logger.Println("Waiting for every match to end")
  wg.Done()
}
```

Terminate tutte le 7 fasi ed effettuato lo scambio tra i due bot, il bot che ha appena concluso segnala di aver finito con il metodo **wg.Done**. Il programma attende quindi che tutti i bot terminino la loro esecuzione con **wg.Wait** 3.2.4.3, controlla eventuali segnali

SIGKILL e SIGINTERRUPT e ricomincia il caricamento di una nuova serie di taker bot 3.2.4.2.

3.2.4.5 Funzione Find Path

In questa sezione vorrei approfondire la funzione FindPath.

```
func FindPath(giverLat,giverLon,takerLat,takerLon float64)(geo.PointSet,[]int,error){}
```

Come spiegato prima, questa funzione esegue delle richieste API a Project OSMR [13] per ottenere un percorso da seguire in macchina che possa condurre il taker fino al giver. Per semplificare potremmo dividere la funzione in due parti:

- **La richiesta**
- **Analisi del risultato**

Nella prima parte viene eseguita la richiesta ad OSMR sfruttando il client offerto dal pacchetto **github.com/gojuno/go.osrm** [14].

```
// We use an external api to get the road to travel to get to the giver
  client := osrm.NewFromURL("https://router.project-osrm.org")

  ctx, cancelFn := context.WithTimeout(context.Background(), time.Second)
  defer cancelFn()

  resp, err := client.Route(ctx, osrm.RouteRequest{
    Profile: "car",
    Coordinates: osrm.NewGeometryFromPointSet(geo.PointSet{
      {takerLon, takerLat},
      {giverLon, giverLat},
    }),
    Steps:       osrm.StepsTrue,
    Annotations: osrm.AnnotationsFalse,
    Overview:    osrm.OverviewFalse,
    Geometries:  osrm.GeometriesPolyline6,
  })
  if err != nil {
    log.Printf("route failed: %v", err)
    return nil, nil, err
  }
```

Nella seconda parte analizziamo invece la risposta ricevuta dalla precedente chiamata.

```
var path geo.PointSet
  var pathLength []int

  for _, route := range resp.Routes {
```

```
    for _, leg := range route.Legs {
      // leg.Steps =The entire trip
      for i, step := range leg.Steps {
        // step = one segment of the entire road
        for _, p := range step.Geometry.PointSet {
          // We memorize the length of this segment as
          // the number of coordinate sets in it
          // We limit it at 15 to avoid a long waiting
          if step.Geometry.PointSet.Length() >= 15 {
            pathLength = append(pathLength, 15)
          } else {
            pathLength = append(pathLength, step.Geometry.PointSet.Length())
          }
          // Every coordinate set of this segment
          point := p
          path.Push(&point)
          // We only use the first coordinate set of each segment because otherwise we
          // would have way too many coordinates to follow and memorize for each trip.
          break
        }
        // Sometimes the last segment is composed of more than a single coordinates
        // set. This check is needed to get the real last coordinate
        if i == len(leg.Steps)-1 {
          lastPoint := step.Geometry.PointSet[len(step.Geometry.PointSet)-1]
          if lastPoint != *path.Last() {
            path.Push(&lastPoint)
          }
        }
      }
    }
  }
  return path, pathLength, nil
}
```

Questa chiamata API ci restituisce un enorme quantità di coordinate, abbastanza da rappresentare alla perfezione ogni piccolo segmento di strada dell'intero percorso. Nella nostra simulazione non abbiamo però bisogno di rappresentare alla perfezione l'intero percorso considerando che il taker notifica la sua posizione ad ogni cambio di strada e che memorizzando nel database ogni set di coordinate raggiunte dal taker rischieremmo di avere un numero enorme di tuple per ogni match. Prendiamo quindi solamente la prima coordinata di ogni strada che compone il percorso in modo tale da avere una rappresentazione realistica del percorso, senza però occupare troppo spazio nel database. Memorizziamo inoltre la lunghezza (numero di coordinate) di ogni strada del percorso per usarla in seguito, limitandola però a 15 per non avere attese troppo lunghe.
La funzione restituisce quindi il percorso che il taker dovrà seguire e la lunghezza di ogni segmento che lo compone in due array.

3.2.5 Modifiche database

Per poter memorizzare le simulazioni e distinguere i trip fatti da un bot da quelli fatti da un utente normale è risultato necessario fare delle modifiche al database.
Per gestire al meglio le simulazioni, ho quindi aggiunto al database due nuove tabelle ed apportato modifiche ad alcune già esistenti.
Ho innanzitutto aggiunto la nuova tabella **simulations**, la quale rappresenta tutte le simulazioni. In questa tabella possiamo capire se una simulazione è terminata oppure è in corso lasciando l'attributo **timeEnd** "null". Inoltre, per indicare che un trip appartiene ad una simulazione, ho anche aggiunto l'attributo di chiave esterna **simulationId** nella tabella **trips**.
Oltre le simulazioni, era anche necessario memorizzare tutte le coordinate di ogni trip fatto da un bot. Per questo motivo ho creato la tabella **simulated_trip_points** la quale contiene appunto tutte le coordinate legate ad un trip.

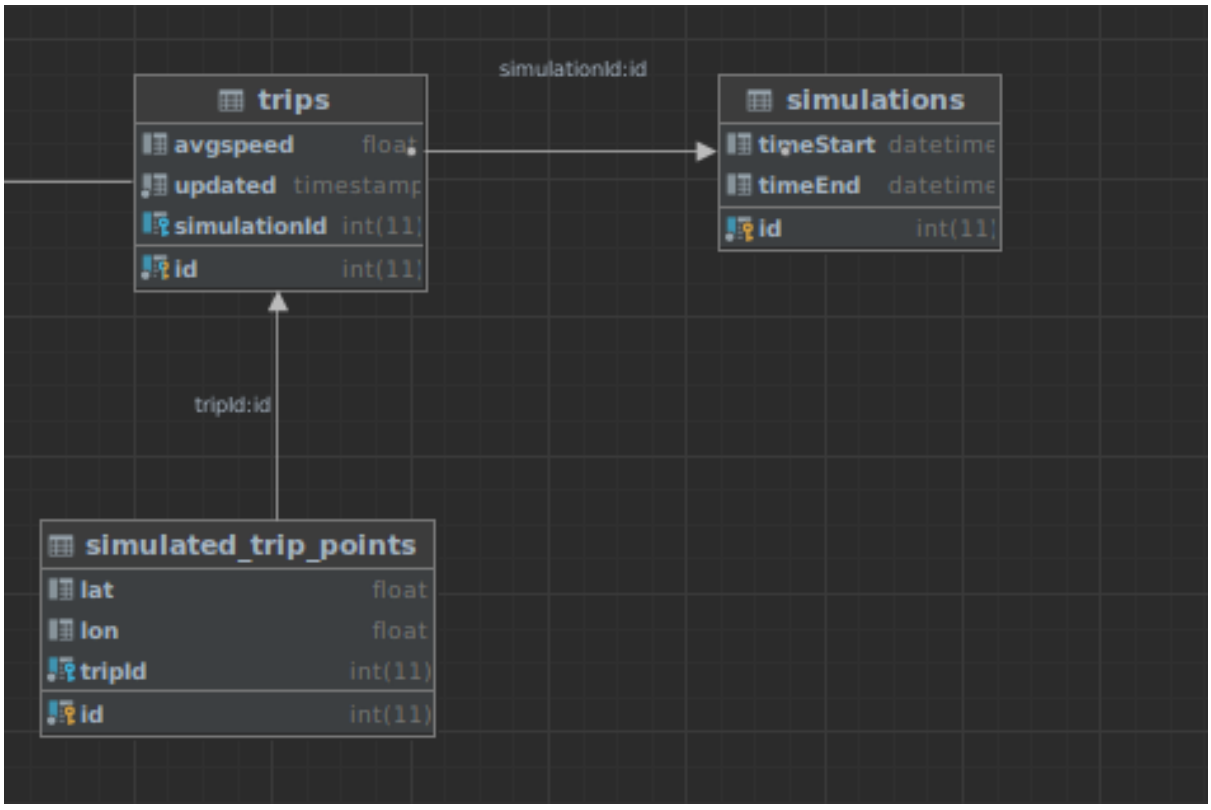


Figure 3.10: Diagramma delle tabelle inerenti alle simulazione

3.2.6 Logger

Nella simulazione viene fatto uso di un logger, il quale registra ogni azione commessa dai due programmi. Tramite l'uso del logger è possibile analizzare l'intera simulazione, mostrandoci ogni errore che può capitare durante l'esecuzione.

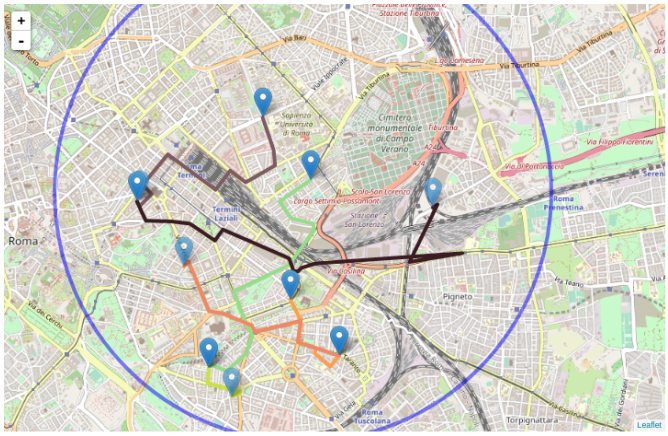
```
wasa@gamlab-vm:~/GolandProjects/generocity-api-server$ go run ./cmd/taker-bot --c
onfig-path ./demo/takerbot.yml
INFO[0000] bottas4's info correctly retrieved
INFO[0000] bottas4's car correctly retrieved
INFO[0000] searching giver..
INFO[0006] found 1 givers, 1 requests are going to be fulfilled
INFO[0006] bottas4 has correctly send the taker request
INFO[0006] bottas4 is waiting for matches...
INFO[0008] bottas4 is updating taker position...
INFO[0047] bottas4  has completed the match
INFO[0047] bottas4  has parked correctly
INFO[0047] bottas4  has unparked correctly
INFO[0047] correctly written the csv file
INFO[0047] bottas4  became a giver
INFO[0047] Waiting for every match to end
INFO[0047] +-+-+-+-+ Restarting +-+-+-+-+
```

```
wasa@gamlab-vm:~/GolandProjects/generocity-api-server$ go run ./cmd/giver-bot --
config-path ./demo/giverbot.yml
INFO[0000] giverBot5's info correctly retrieved
INFO[0000] giverBot5's car correctly retrieved
INFO[0004] giverBot5's car correctly parked
INFO[0004] giverBot5 has correctly send the giver request
INFO[0004] giverBot5 is waiting for matches...
INFO[0006] giverBot5  has found a match
INFO[0046] giverBot5  previous match ended
INFO[0046] giverBot5  has unparked correctly
INFO[0046] correctly written the csv file
INFO[0046] giverBot5  became a taker
INFO[0046] Waiting for every match to end
INFO[0046] +-+-+-+-+ Restarting +-+-+-+-+
```

Figure 3.11: Log ottenuti durante una simulazione tra due bot

3.3 Visualizzazione Web

Avendo una simulazione funzionante che salva anche tutte le informazioni necessarie per ricostruirla in seguito, è rimasto solo di visualizzarla su una mappa durante l'esecuzione. Per fare ciò ho realizzato una semplice pagina web in HTML che comunica con il programma tramite delle API appositamente realizzate.



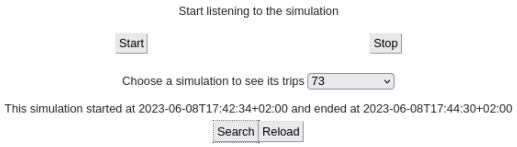


Figure 3.12: Visualizzazione web di una simulazione

Questa pagina sfrutta la mappa offerta dalla libreria JavaScript open-source Leaflet [15]. Grazie a questa libreria è possibile mostrare una mappa con markers per indicare la posizione di tutti i bot ed il percorso che hanno seguito.

3.3.0.1 Simulazione in tempo reale

Una volta premuto il pulsante "Start", la pagina inizia a fare richieste API al server ogni due secondi per ottenere le informazioni della simulazione in corso (se presente).

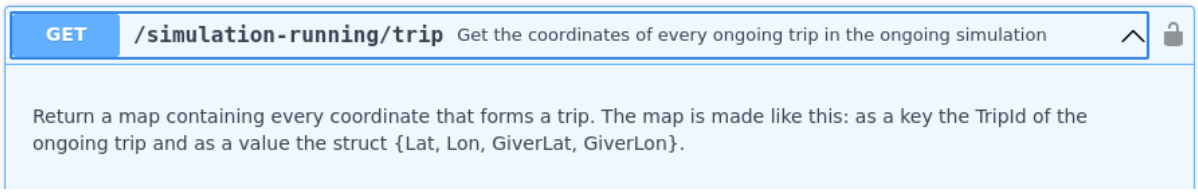


Figure 3.13: API per ottenere la simulazione in corso

Se ci dovessero essere simulazioni in corso, l'API restituirà un JSON contenente tutti i trip in corso. Ogni elemento nel JSON sarà una coppia formata dall'id del trip ed un array di oggetti **MapPoints**. Questo array rappresenta tutte le coordinate di un trip salvate nel database dal taker mentre stava percorrendo la strada per raggiungere il giver (memorizzate in 3.2.4.4). Insieme alle coordinate attuali, vengono passate anche le coordinate del giver da raggiungere così da poterlo mostrare sulla mappa.

```
type MapPoints struct {
  Lat      float64 `json:"lat" db:"lat"`
  Lon      float64 `json:"lon" db:"lon"`
  GiverLat float64 `json:"giverlat" db:"giverlat"`
  GiverLon float64 `json:"giverlon" db:"giverlon"`
}
```

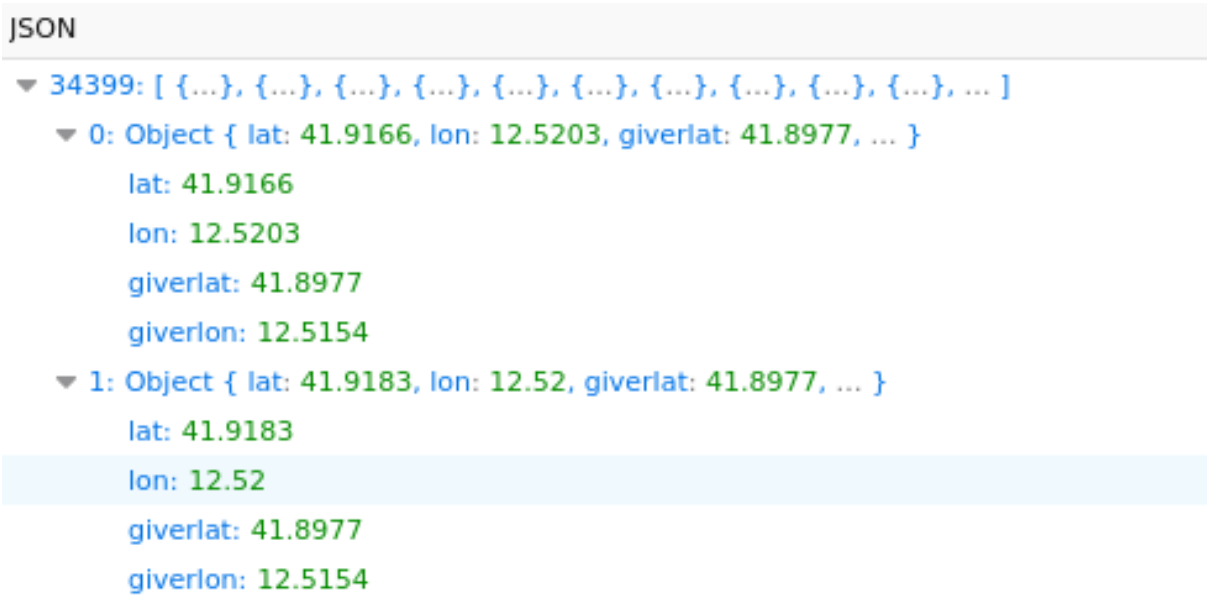


Figure 3.14: Esempio di risposta

Una volta ricevuta una risposta, tutti i trip in corso vengono visualizzati sulla mappa ed aggiornati ogni due secondi finché non verrà premuto il pulsante "Stop".
Possiamo distinguere tre tipi di marker:

- **Partenza Taker** : il marker più scuro;

- **Posizione Taker** : il marker semitrasparente;
- **Posizione Giver** : il marker leggermente più chiaro di quello del taker.

Figure 3.15: Simulazione in corso

3.3.0.2 Vecchie simulazioni

Tramite questa pagina è anche possibile visualizzare simulazioni ormai terminate. Appena la pagina viene caricata, viene fatta una richiesta API al server per ottenere l'id e la data di tutte le simulazioni già terminate. Questo elenco appena ottenuto viene inserito nel menù a tendina, il quale permetterà all'utente di selezionare una qualsiasi simulazione, mostrando anche la data in cui è stata svolta.

Figure 3.16: Menù a tendina

Una volta scelta una simulazione, basterà premere il pulsante “Search” per visualizzare tutti i trips svolti durante quella simulazione. 3.12

Chapter 4

Conclusione

Ritengo che Generocity abbia del vero potenziale come applicazione e che una volta pubblicata diventerà davvero essenziale nella vita di un automobilista di città. Per questo motivo, sono felice di essere stato parte di questo progetto e spero di essere riuscito a migliorarlo con il mio lavoro.
Grazie a questa esperienza ho avuto modo di migliorare le mie competenze nell'uso di tecnologie già studiate ed apprenderne di nuove, oltre all'aver sperimentato il lavoro e la collaborazione in un grande gruppo.
Vorrei quindi ringraziare il laboratorio Gamification Lab e tutti i suoi membri, i quali mi hanno aiutato molto in questa mia esperienza.

4.1 Sviluppi Futuri

In futuro, la simulazione potrebbe essere migliorata considerando anche un maggior numero di tipi bot che utilizzano macchine diverse, in modo da verificare che il matching funzioni bene anche considerando le dimensioni delle auto. Inoltre potrebbe essere utile creare un unico file di configurazione per entrambi i tipi di bot ed un unico comando per far iniziare l'esecuzione di entrambi i comandi.

Bibliography

[1] *Generocity*. 2023. URL: https://www.generocity.it/. (accessed: 12-06-2023).

[2] *Go*. 2023. URL: https://go.dev/. (accessed: 22-06-2023).

[3] *GamificationLab*. 2023. URL: http://gamificationlab.uniroma1.it/. (accessed: 12-06-2023).

[4] *Com'è strutturato il tirocinio / la tesi"*. 2023. URL: http://gamificationlab.uniroma1.it/tirocini_e_tesi/struttura/. (accessed: 12-06-2023).

[5] *GitLab*. 2023. URL: https://about.gitlab.com/. (accessed: 23-06-2023).

[6] *Git*. 2023. URL: https://git-scm.com/. (accessed: 23-06-2023).

[7] *Go 1.16*. 2023. URL: https://go.dev/doc/go1.16. (accessed: 13-06-2023).

[8] *Go 1.13*. 2023. URL: https://go.dev/blog/go1.13-errors. (accessed: 14-06-2023).

[9] *Linters*. 2023. URL: https://golangci-lint.run/usage/linters/. (accessed: 22-06-2023).

[10] Marco Wang. *Progettazione e Sviluppo di bot per l'applicazione Generocity*. 2023.

[11] *Goroutine*. 2023. URL: https://go.dev/tour/concurrency/1. (accessed: 23-06-2023).

[12] *OpenStreetMap API*. 2023. URL: https://wiki.openstreetmap.org/wiki/Overpass_API. (accessed: 18-06-2023).

[13] *Osmr project*. 2023. URL: http://project-osrm.org/docs/v5.5.1/api/#general-options. (accessed: 19-06-2023).

[14] *Gojuno library*. 2023. URL: https://github.com/gojuno/go.osrm. (accessed: 19-06-2023).

[15] *Leafletjs*. 2023. URL: https://leafletjs.com/. (accessed: 20-06-2023).