SAPIENZA
UNIVERSITÀ DI ROMA

Sviluppo di test automatizzati per applicazione di smart parking

Facoltà di ingegneria dell'informazione, informatica e statistica
Informatica

Riccardo Mancini
Matricola 1905638

Relatore
Prof. Emanuele Panizzi

Anno Accademico 2021/2022

Tesi non ancora discussa

Sviluppo di test automatizzati per applicazione di smart parking
Sapienza Università di Roma

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

Email dell'autore: mancini.1905638@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 il periodo del mio tirocinio ho partecipato allo sviluppo del lato back-end dell'applicazione di smart parking GeneroCity. In particolare mi sono concentrato sulla manutenibilitá del codice giá esistente, sviluppando test automatizzati e scrivendone la documentazione. Dopo una prima introduzione e una spiegazione della struttura del back-end di GeneroCity, parlerò del lavoro svolto suddiviso per argomenti.

Indice

Capitolo 1

Introduzione

1.1 GeneroCity

GeneroCity è un'applicazione di smart parking che cerca di rendere piú facile trovare parcheggio ai suoi utenti. L'applicazione riesce ad accorgersi quando un utente sta lasciando il proprio parcheggio, sia automaticamente, sia tramite segnalazione manuale (l'utente prende il ruolo di *giver*). Allo stesso modo può capire se l'utente sta cercando parcheggio (e quindi sta assumendo il ruolo di *taker*).

Se il sistema trova un giver e un taker nella stessa area, esegue un *match* e li mette in contatto, permettendo al taker di sapere preventivamente dove andare, risparmiando tempo e carburante.

Inoltre l'applicazione è dotata di un sistema interno in base al quale per assumere il ruolo di taker si devono spendere delle monete virtuali, che è possibile guadagnare solo portando a buon fine i match come giver. Questo fa sí che tutti siano incentivati a fornire indicazioni sui posti che si stanno liberando, oltre che a riceverle.

1.2 Situazione iniziale

All'inizio del mio tirocinio il lato server dell'applicazione era giá in una fase avanzata dello sviluppo, con la maggior parte delle chiamate API giá implementate.

Era bene quindi cominciare a concentrarsi sulla manutenibilitá del codice, in modo da facilitarne il proseguimento dello sviluppo, apportando alcune modifiche e miglioramenti:

- Semplificare parti che erano state modificate piú volte e che perciò erano diventate in molti casi piú complicate di quanto servisse

- Scrivere dei commenti dettagliati per permettere in futuro, ad altri sviluppatori, di capire rapidamente il significato e lo scopo del codice e quindi di apportare modifiche piú velocemente e facilmente, se necessario
- Creare dei test automatizzati per scovare errori difficili da individuare attraverso un uso abituale dell'applicazione, e per evitare di introdurne inconsapevolmente di nuovi durante un'eventuale modifica del codice

Inoltre, come esercizio per familiarizzare con il codice, mi è stato chiesto di implementare una coppia di richieste API per salvare ed ottenere le statistiche sull'uso medio del carburante della propria macchina.

Capitolo 2

Struttura back-end di GeneroCity

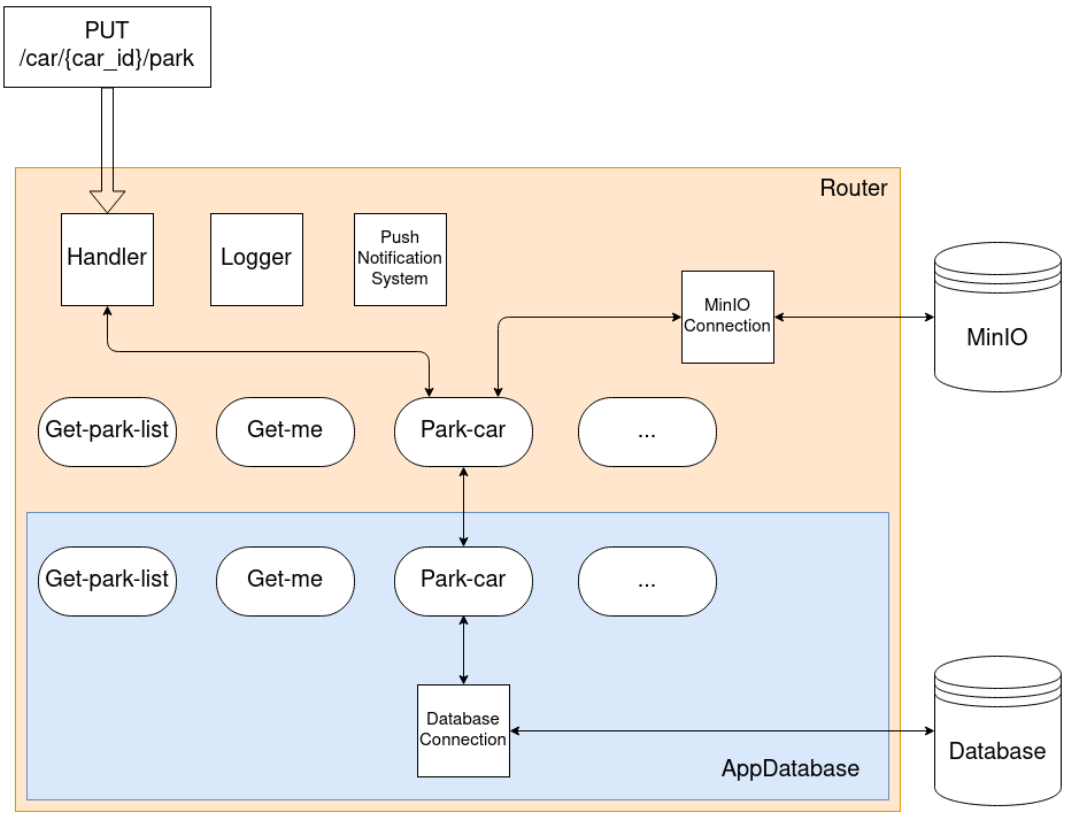

Figura 2.1. Diagramma della struttura del back-end di GeneroCity

Il codice del lato server di GeneroCity è costituito da due parti principali, rappresentate da due interfacce:

- Router
- AppDatabase

Il Router contiene al suo interno un Logger, che come suggerisce il nome permette di salvare i log di ogni evento ed errore, un Push Notification System, che permette di inviare notifiche push agli smartphone che hanno l'applicazione di GeneroCity installata, un oggetto che rappresenta una connessione ad un server MinIO, e un Handler.

Inoltre il Router implementa un metodo per ogni chiamata API esistente, e contiene un'interfaccia AppDatabase. Questa a sua volta contiene un oggetto che rappresenta la connessione con il database di GeneroCity, e implementa un metodo per ogni tipo di interazione con il database, che nella maggior parte dei casi coincide con una chiamata API.

L'Handler riceve le richieste http e, in base alla chiamata API che sta venendo effettuata, esegue il metodo del Router corrispondente. Per esempio se riceve una richiesta "PUT /car/car_id/park", che serve a registrare un parcheggio sul server, chiama la funzione "Park-car" del Router, che si occupa di effettuare tale operazione.

I metodi del Router a loro volta chiamano i metodi dell'AppDatabase che si occupano di effettuare le operazioni necessarie nel database. Contiunando l'esempio del parcheggio: la funzione "Park-car" del Router chiama il metodo "Is-owner" dell'AppDatabase che interroga il database per assicurarsi che l'utente che ha effettuato la richiesta sia il proprietario della macchina, e cioè sia abilitato ad eseguire questa operazione. Successivamente chiama anche il metodo "Park-car" dell'AppDatabase, che esegue le query necessarie a salvare le informazioni sul parcheggio nel database.

Inoltre alcune chiamate API salvano dei dati in un'istanza di MinIO, perciò i metodi del Router relativi interagiscono con l'oggetto che rappresenta la connessione al server MinIO.

MinIO

MinIO è un High Performance Object Storage distribuito sotto la GNU Affero General Public License, compatibile con Amazon S3. Può manipolare dati non strutturati come foto, video e log files[1]. È costituito da tre componenti: server, client e client SDK.

Il server è progettato per essere minimale e scalabile, ed è abbastanza leggero da poter essere raggruppato insieme allo stack applicativo. Può inoltre essere installato sia su macchine fisiche che virtuali, o lanciato come container docker.

Il client è compatibile con tutti i servizi di cloud storage compatibili con Amazon S3, e il client SDK offre una API, implementata in diversi linguaggi, per potersi infacciare con tali servizi.

Capitolo 3

Semplificazione del codice

3.1 Park/Unpark

3.1.1 Funzionamento

Una delle parti piú importanti del server di GeneroCity è quella che si occupa di salvare nel database le informazioni di ogni evento di parcheggio (*park*) e dell'associato evento di lasciare il parcheggio (*unpark*).

Quando un utente parcheggia la propria macchina e l'applicazione se ne accorge (sia automaticamente, sia tramite segnalazione manuale dell'utente), questa invia una richiesta al server di GeneroCity. La richiesta contiene tutte le informazioni del parcheggio: il codice identificativo del guidatore e della macchina, la data e ora del parcheggio, la posizione, il tipo di parcheggio ed altre informazioni usate per calcolare varie statistiche, come il tempo impiegato a trovare parcheggio. Inoltre la richiesta può includere i dati relativi al viaggio appena concluso, e alle manovre fatte durante l'operazione di parcheggio.

Il server si occupa quindi di salvare le informazioni del parcheggio nel database, e i dati sul viaggio e sulle manovre in un'istanza di MinIO. Successivamente salva nelle tabelle apposite le statistiche aggiornate relative alla macchina e all'utente che hanno eseguito il parcheggio, come il numero totale di parcheggi effettuati e il tempo medio passato a cercare un parcheggio.

Allo stesso modo, quando l'utente esegue un *unpark*, l'applicazione invia una richiesta al server, che aggiunge le relative informazioni nel database.

3.1.2 Modifiche apportate

Al momento dell'inizio del mio tirocinio i *park* e *unpark* venivano registrati nella stessa tabella del database, in modo da facilitare le operazioni che richiedevano i dati di entrambi.

Ogni evento *unpark* prevedeva un record distinto da quello del *park* ad esso associato, con l'aggiunta però di una colonna apposita che permettesse di riconoscere quel record come evento *unpark*. In questo modo però risultava che molte informazioni nel record dell'*unpark* venivano duplicate, come ad esempio l'id della macchina, il tipo del parcheggio e la posizione, o addirittura non erano applicabili, come l'informazione sul tempo impiegato per trovare parcheggio.

Questo si e' rivelato ovviamente poco efficiente sia in termini di occupazione dello spazio nel database che di efficienza: infatti anche query molto semplici, come il richiedere tutti i parcheggi di una macchina, dovevano lavorare sul doppio dei record.

parkid	cid	parkstatus	usedby	parktime	parklat	parklon	parkaccuracy	curb	parktype	searchingtimesec
29382	f80c054a-6356-41c1-9226-6bfebad576d4	unpark	17f4be1a-6f29-4aba-88a4-a399530c0501	2022-07-22 17:56:19	41.9024	12.4466	0			0
29272	f80c054a-6356-41c1-9226-6bfebad576d4	park	17f4be1a-6f29-4aba-88a4-a399530c0501	2022-07-18 17:28:43	41.9024	12.4466	35	Viale Vaticano 46	parallel	120

Figura 3.1. Organizzazione dei record prima delle modifiche apportate

La soluzione quindi è stata quella unire nello stesso record le informazioni relative agli eventi *park* e *unpark* associati, aggiungendo le colonne "unparktime" e "unparkedby": in questo modo, per sapere se una macchina è ancora parcheggiata o ha lasciato libero il posto, è sufficiente controllare se il valore della colonna "unparktime" è NULL oppure no. Questo ha permesso di rimuovere la colonna "parkstatus", necessaria a distinguere il ruolo di ogni record (*park* o *unpark*).

parkid	cid	parkedby	parktime	parklat	parklon	parkaccuracy	curb	parktype	searchingtimesec	unparkedby	unparktime
29272	f80c054a-6356-41c1-9226-6bfebad576d4	17f4be1a-6f29-4aba-88a4-a399530c0501	2022-07-18 17:28:43	41.9024	12.4466	35	Viale Vaticano 46	parallel	120	17f4be1a-6f29-4aba-88a4-a399530c0501	2022-07-22 17:56:19

Figura 3.2. Organizzazione dei record dopo le modifiche apportate

Ovviamente è stato necessario aggiornare tutte le query che andavano ad interrogare la tabella dei parcheggi per recepire i cambiamenti effettuati.

3.1.3 Script di migrazione dati

Avendo modificato la struttura del database, è stato necessario creare uno script che migrasse le vecchie informazioni nel nuovo formato. Questo semplice script raccoglie per ogni macchina tutti i suoi *park* e *unpark*, e per ogni riga di *unpark*, dopo aver spostato le sue informazioni nella riga del *park* corrispondente, la elimina.

```
var carList []uuid.UUID
err = db.Select(&carList, query: "SELECT DISTINCT cid FROM parks")
if err != nil { errors.Wrap(err, "Can't get car ids from db") }

tx, err := db.Beginx()
if err != nil { errors.Wrap(err, "can't begin a new transaction") }
for _, car := range carList {
    var parkList []CarUnpark
    err = db.Select(&parkList, query: `
                    SELECT parkedby as id, parkid, parkstatus, parktime
                    FROM parks
                    WHERE cid = ?
                    ORDER BY parkid`,
        car)
    if err != nil {...}

    var firstPark = parkList[0]
    if firstPark.ParkStatus == "unpark" {
        _, err = tx.Exec( query: "DELETE FROM parks WHERE parkid = ?", firstPark.ParkId)
        if err != nil {...}
    }

    for index := 1; index < len(parkList); index++ {
        park := parkList[index]
        if park.ParkStatus == "unpark" {
            if parkList[index-1].ParkStatus == "unpark" {
                _ = tx.Rollback()
                return errors.Wrap(err, message: "Unparks should always follow parks")
            }

            _, err = tx.Exec( query: `
                UPDATE parks
                SET unparkedby = ?, parkstatus = 'unpark', unparktime = ?
                WHERE parkid = ?`,
                park.Id, park.ParkTime, parkList[index-1].ParkId)
            if err != nil {...}
            _, err = tx.Exec( query: "DELETE FROM parks WHERE parkid = ?", park.ParkId)
            if err != nil {...}
        }
    }
}
```

Figura 3.3. Script di migrazione dati

Questo script per poter funzionare ha bisogno della colonna "parkstatus" nella tabella dei parcheggi, che specifica se un record è un park o un unpark. Perciò è stato necessario aggiungere al database le due colonne per le informazioni sull'unpark,

poi eseguire lo script, e solo successivamente si è potuto modificare nuovamente il database per rimuovere la colonna "parkstatus".

3.2 Unificazione delle structs

Per manipolare i dati ottenuti dalle interrogazioni del database, vengono usate delle struct, che si possono definire come collezioni di variabili.

Poiché il codice è stato scritto a piú riprese da diverse persone, il numero di struct usate per rappresentare vari tipi di dati era diventato molto alto, nonostante molte di queste rappresentassero spesso la stessa cosa. Mi è stato quindi chiesto di sostituire tutte le strutture duplicate con un'unica struct, leggermente piú generalizzata.

Erano presenti due gruppi di struct da unificare:

- Quelle che si occupavano delle informazioni su un determinato parcheggio
- Quelle che si occupavano delle informazioni su una macchina e sul suo stato

Spesso le informazioni contenute nelle varie struct appartenenti allo stesso gruppo risultavano replicate, con pochissime differenze; piú precisamente nel codice era giá presente per ognuno dei due gruppi una struct "principale" che conteneva molti campi, e diverse struct piú piccole che erano composte da un sottoinsieme di questi. In rari casi le struct piú piccole avevano anche uno o due campi non presenti nella struct principale. Ho quindi eliminato le piú piccole sostituendole con quelle principali, alle quali ho aggiunto i pochi campi in piú.

Capitolo 4

Documentazione in linea

Nonostante fosse giá presente una documentazione fatta con ApiDoc, questa andava solo a spiegare a grandi linee il compito di ogni chiamata API, elencando gli argomenti da inserire nella richiesta e quelli che si ricevono nella risposta.

Ciò è sufficiente solo per chi si occupa dello sviluppo nel lato front-end, ma completamente inadeguato per chi invece lavora sul codice del server, perché non fornisce indicazioni esaustive su come le API lavorano al loro interno.

Per questo motivo ho ritenuto opportuno scrivere una documentazione in linea (commenti) che permettesse, a chi in futuro dovesse apportare delle modifiche o integrazioni al codice, di capire in modo veloce come farlo.

4.1 Importanza dei commenti

I commenti sono uno degli strumenti che contribuiscono maggiormente alla mantenibilitá del codice. A volte può risultare molto difficile capire cosa sta cercando di fare un frammento di codice, o perché non siamo stati noi a scriverlo, o perché è passato molto tempo da quando l'abbiamo scritto.

È di fondamentale importanza quindi scrivere dei commenti esaustivi, che rendano il codice piú chiaro e leggibile, spieghino i motivi che hanno portato a prendere delle decisioni non ovvie, avvertano di possibili side effects, e permettano di comprendere parti di codice che hanno una logica complicata.

In assenza di tale documentazione si rischia di non essere in grado di decifrare il codice in tempi rapidi o di fraintenderne gli obiettivi e addirittura di prendere decisioni che erano state evitate in partenza per motivi validi ma non immediatamente evidenti.

4.2 Tipi di commenti

I commenti si possono distinguere in due macro categorie: quelli che si occupano di spiegare cosa sta facendo un frammento di codice, e quelli che spiegano il perché un frammento di codice sta facendo qualcosa.

Nella prima categoria sono presenti i seguenti tipi di commenti:

- Commenti di funzioni
- Commenti teorici
- Commenti guida

Commenti di funzioni Spiegano il funzionamento di intere funzioni, e servono quindi ad evitare allo sviluppatore di dover leggerne tutto il corpo prima di poter capire a cosa servono.

```
// ParkCar inserts the given park information in the database,
// and it returns the id of the newly created park.
// If the car was not unparked before this request, ParkCar
// returns a types.DUPLICATE_PARK message instead.
// After inserting park information, it creates a new trip in the
// database, and it updates all matches without parkid
// made by the car and driver since the car's last park
func (db *appdbimpl) ParkCar(carPark types.ParkInfo) (int64, int, error)
    {...}
```

Commenti teorici Spiegano concetti che potrebbero essere al di fuori delle conoscenze dello sviluppatore, ma che sono fondamentali per capire cosa sta facendo un frammento di codice. Per esempio possono spiegare la teoria che sta dietro alcune funzioni matematiche particolarmente complesse.

Commenti guida Sono commenti che potrebbero sembrare inutili, visto che non servono a spiegare qualcosa che non è chiaro o i motivi delle scelte fatte. Si limitano a descrivere cosa fa un frammento di codice, in modo da rendere piú facile e veloce allo sviluppatore leggerlo. Dividono quindi il codice in blocchi piú leggibili, e lo rendono piú ordinato, dandogli un ritmo definito. Non devono però trasformarsi in commenti banali, che descrivono cose talmente ovvie da non avere bisogno di essere commentate.

Esempio di commento banale:

```
count += 1; // Increase counter by one
```

Esempio di commento guida:

```
// Insert park information in database, send error response if
// car was not unparked before request
// or if given park time is older than last unpark
parkId, message, err := rt.db.ParkCar(carPark)
if err != nil {...
   errorMessage.ErrorMessage = "can't insert carpark"
   ...
} else if message == types.DUPLICATE_PARK {...
   errorMessage.ErrorMessage = "can't park without having unparked first"
   ...
} else if message == types.PARK_TIME_INVALID {...
   errorMessage.ErrorMessage = "given park time older than last
       park/unpark"
   ...
}
```

Alla seconda categoria appartengono:

- Commenti di progettazione
- Commenti di motivazione

Commenti di progettazione Aiutano a far capire come il codice è stato progettato, descrivendo le tecniche usate e come e perché vengono usati certi algoritmi. Sono una versione piú ad alto livello dei Function comments, e sono di solito usati per descrivere grandi parti di codice.

Commenti di motivazione Si occupano di spiegare perché il codice sta facendo qualcosa. Aiutano chi sta leggendo a capire come mai il codice sta eseguendo azioni che potrebbero sembrare inutili o addirittura sbagliate, se non se ne conoscono le ragioni.

```
// If last park doesn't have an unpark time, it means the
// car is not unparked, so it cannot be parked again.
else if parkInfo.UnparkTime == nil {
_ = tx.Rollback()
return 0, types.DUPLICATE_PARK, nil
```

4.3 Commenti usati

Ho iniziato scrivendo i commenti di funzione per tutte le funzioni piú importanti, in modo da poter spiegare ogni funzione a cosa serve, e come è implementata a grandi linee. Successivamente ho cominciato ad aggiungere i commenti guida, rendendo molto piú leggibile tutta la codebase. Infatti, a causa delle molte linee di codice usate per fare error checking, spesso il codice rischiava di diventare confusionario. Infine ho aggiunto alcuni commenti di motivazione per spiegare come mai erano state fatte alcune scelte.

Capitolo 5

Chiamate API implementate

Durante il periodo del mio tirocinio mi è stato chiesto di implementare due chiamate API per ottenere e aggiornare le statistiche sul consumo di carburante medio di una macchina. Le chiamate sono le seguenti:

- GetAvgRefuel
- UpdateAvgRefuel

GetAvgRefuel Restituisce il numero di rifornimenti effettuati, la data dell'ultimo rifornimento, e la distanza media percorsa e il tempo medio trascorso tra due rifornimenti.

UpdateAvgRefuel È progettata per essere effettuata ad ogni rifornimento, in modo da poter aggiornare le statistiche con i dati raccolti dall'ultimo rifornimento.

L'equazione utilizzata per calcolare le nuove medie della distanza percorsa e del tempo trascorso è la seguente:

$$m_n = \frac{(n-1)m_{n-1} + a_n}{n}$$

dove:

- m_n: nuova media da calcolare
- m_{n-1}: media precedente
- a_n: distanza percorsa (o tempo passato) dall'ultimo rifornimento
- n: numero totale di rifornimenti effettuati

UpdateAvgRefuel riceve nel body della richiesta un oggetto json contenente la data del rifornimento corrente e la distanza percorsa dall'ultimo rifornimento. Dalla

data ricevuta e quella presente nel database relativa al rifornimento precedente può calcolare il tempo passato da quest'ultimo. Con queste informazioni, le vecchie medie e il numero di rifornimenti fatti può calcolare le nuove medie, che andrá a salvare nel database insieme alla data ricevuta nella richiesta.

Capitolo 6

Test automatizzati

L'argomento principale del mio tirocinio è stato lo sviluppo di test automatizzati per l'api di GeneroCity. Questi test vengono eseguiti nella pipeline ogni volta che viene fatto il push di un nuovo commit sulla repository di SapienzaApps, in questo modo si possono trovare subito possibili errori aggiunti durante una modifica al codice. Lo sviluppo di questi test ha anche permesso di portare alla luce bug giá esistenti che erano sfuggiti al testing manuale che era stato fatto fino ad allora.

6.1 Perché usare test automatizzati

Il software testing è il processo di eseguire un prodotto software con il fine di determinare se il sistema si comporta come dovrebbe e di identificare possibili malfunzionamenti. Ciò può essere attuato sia manualmente, sia tramite lo sviluppo di strumenti di test automatizzati.

I test automatizzati offrono una serie di vantaggi rispetto a quelli manuali:

- Sono riutilizzabili, sia perché danno la possibilitá di creare molti test con lo stesso codice a cui si possono apportare piccole modifiche, sia perché permettono di ripetere gli stessi test per un numero arbitrario di volte, avendo la sicurezza di eseguirli sempre nelle stesse condizioni
- Assicurano di venire svolti correttamente rimuovento la possibilitá di errore umano durante il loro svolgimento
- Aiutano a testare tutti gli scenari possibili, compresi i casi limite
- Permettono di testare facilmente scenari particolarmente complessi
- Sono eseguibili piú velocemente e piú frequentemente di quelli manuali

6.2 Tipi di test

I test automatizzati si possono generalmente raggruppare in quattro categorie:

- Unit test
- Integration test
- System test
- Acceptance test

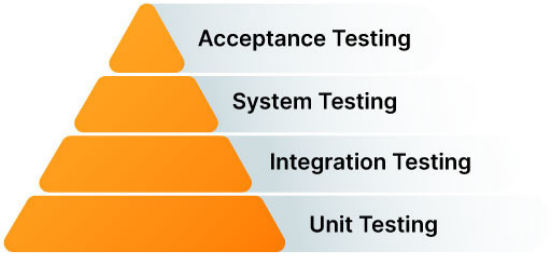

Figura 6.1. Piramide dei test

Unit test Sono quelli che agiscono al piú basso livello, verificano che le singole unitá di codice, come funzioni e classi, funzionino correttamente. Sono utili a scovare errori nelle fasi iniziali dello sviluppo, permettendo di correggerli facilmente e riducendo il loro impatto. Dividendo il codice in piccole unitá, lo rendono inoltre piú facile da testare. Spesso, per isolare le varie unitá, richiedono l'utilizzo di test doubles come mock, stub, etc. Per velocizzarne lo sviluppo si usano di solito test parametrizzati, che permettono con lo stesso codice di testare tutte le combinazioni di input e output attesi.

In alcune metodologie di sviluppo software, come il test driven design, sono scritti prima ancora di implementare le unitá di cui si occupano, in modo da poter definire chiaramente come queste devono comportarsi.

Integration test Operano su un intero componente software costituito da diverse unitá che sono giá state testate singolarmente, combinandole e testandole insieme. Permettono quindi di verificare il corretto funzionamento di tale componente e controllare che le varie unitá interagiscano tra loro correttamente.

System test Agisce sull'intero sistema software con tutti i suoi componenti completamente integrati. Viene eseguito in condizioni simili a quelle dell'ambiente in cui il software è progettato per essere usato. Controlla che il sistema nella sua interezza funzioni correttamente, e che il suo design e comportamento siano quelli definiti nelle specifiche dei requisiti.

Acceptance test È l'ultimo stadio del percorso di testing prima di rilasciare il prodotto al pubblico. Tramite l'acceptance testing ci si assicura che il software soddisfi tutti i requisiti prestabiliti.

6.3 Test doubles

Lo scopo dei test automatizzati è di verificare parti di codice, e per farlo correttamente è bene eliminare tutte le componenti esterne al codice stesso (nel caso di GeneroCity il database e il server MinIO). Usare dei sostituti (*test doubles*) permette di avere diversi vantaggi:

- Rimuove il bisogno di avere in esecuzione, durante i test, dei server che possono renderne molto piú dispendioso lo svolgimento. Soprattutto quando questi devono essere eseguiti ogni volta che viene fatto un push sulla repository remota.
- Rimuove il rischio di fallimento delle componenti esterne. Queste non rientrano nell'ambito di ciò che deve essere testato, cioè il codice
- Offre una migliore gestione di questi componenti, per esempio permette di decidere cosa il database restituirá a seconda della query ricevuta

Esistono diversi tipi di test doubles, ognuno con il suo campo d'applicazione. Sono riportati di seguito.

Dummy: Sono la tipologia piú semplice, non hanno alcuna funzionalitá e servono solo ad essere passati come parametri che non verranno mai utilizzati

Stubs: Contengono dei dati predefiniti dal test, e restituiscono sempre quelli. Sono utili quando non si ha bisogno di verificare le interazioni con l'oggetto che stanno sostituendo.

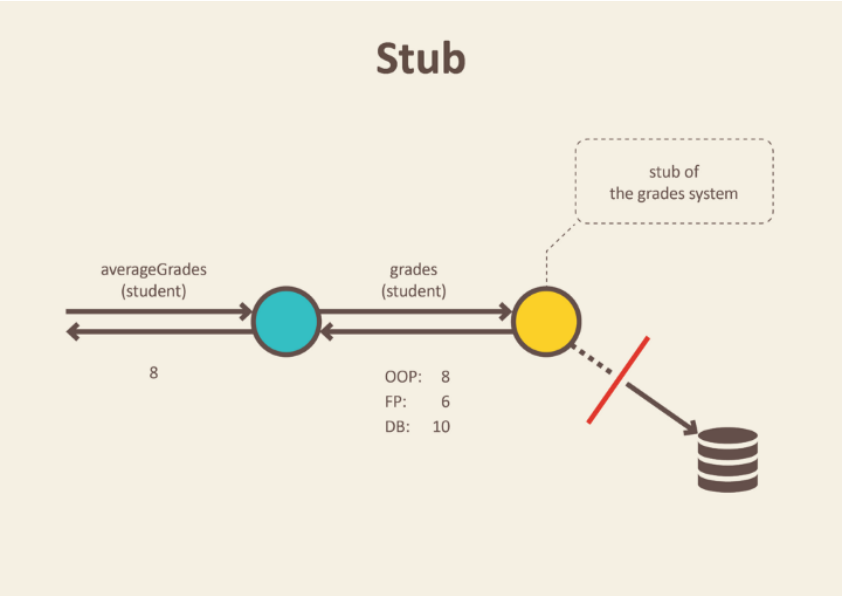

Figura 6.2. Esempio di funzionamento di uno Stub: restituisce un risultato prestabilito senza interrogare il database

Spies: Sono come gli stubs, ma permettono anche di salvare le informazioni ricevute dalla funzione chiamante. Possono esssere usati per esempio per simulare un servizio di email che mantiene il conto di quanti messaggi ha ricevuto.

Fakes: Hanno un'implementazione funzionante, ma semplificata rispetto a quella degli oggetti che sostituiscono. Sono utili nel caso in cui ció che si vuole simulare deve avere un comportamento stateful, cosa che gli stubs non hanno. Per esempio possono sostituire il comportamento di un database.

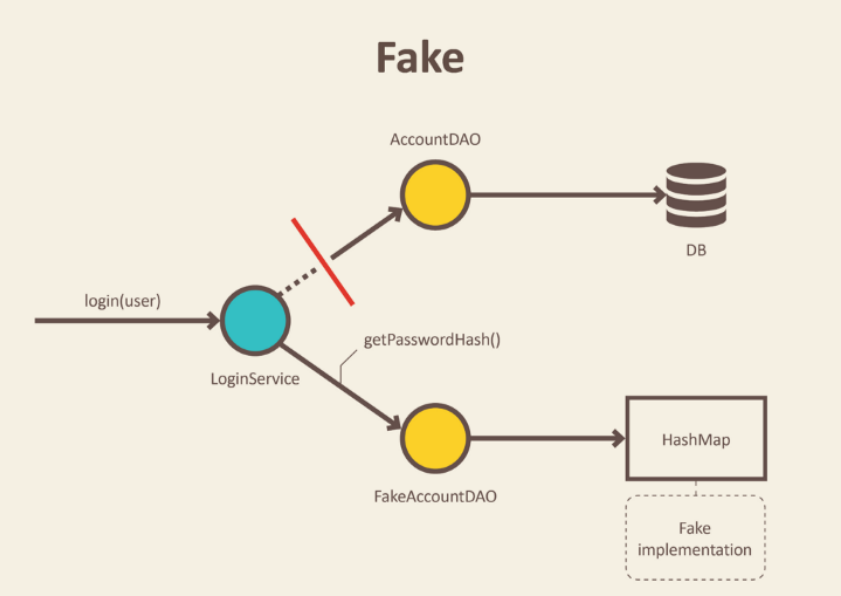

Figura 6.3. Esempio di funzionamento di un Fake: la sua implementazione usa una semplice hash map al posto del database

Mocks: Sono pre-programmati con delle aspettative su come verranno usati e quali valori riceveranno. Nel caso le loro aspettative non vengano soddisfatte, lanciano un errore. La caratteristica che distingue i mock dagli altri test double è che i test che li usano rientrano nella categoria del white-box testing, in cui i test conoscono la struttura del codice da testare (visto che è l'unico modo per impostare le aspettative). Questo solitamente non è auspicabile visto che i test si dovrebbero occupare di verificare la correttezza del codice a prescindere da come è implementato.

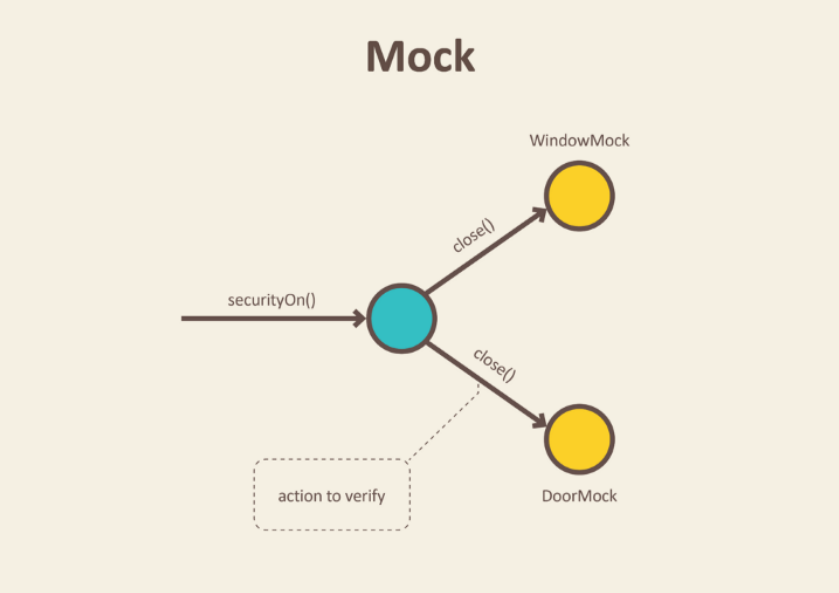

Figura 6.4. Esempio di funzionamento di un Mock: sostituisce il vero oggetto, e controlla se l'azione ricevuta è corretta

6.4 Approccio utilizzato

Lo scopo principale del mio tirocinio era quello di scrivere dei test per la parte back-end di GeneroCity. In particolare dovevo sviluppare test automatizzati per le componenti che costituiscono la struttura portante del sistema, ovvero le interfacce Router e AppDatabase, che come detto in precedenza lavorano insieme per processare le richieste http, interrogare il database, e preparare le risposte dell'API.

6.4.1 Idea iniziale

Inizialmente avevo preso la decisione di fare degli unit test per ogni funzione di AppDatabase e per ogni funzione di Router.

In quanto unit test non potevo far usare al Router l'AppDatabase implementato da GeneroCity, quindi ho creato uno stub apposito, in modo da poter testare tutti i comportamenti delle funzioni del Router a seconda di cosa le funzioni di AppDatabase chiamate da loro avrebbero restituito.

Per i motivi elencati nel paragrafo sui Test Doubles non potevo usare un server di MinIO, sono quindi andato alla ricerca di una libreria che fornisse un mock apposito. Non avendola trovata, ho deciso di scriverne uno io.

Per i test delle funzioni dell'AppDatabase invece, al posto dell'oggetto DB della libreria standard di Go, ho usato un mock fornito dalla libreria go-sqlmock.

6.4.2 Problemi dell'idea iniziale

Scrivendo i test è cominciato ad apparire evidente che la maggior parte dei test per AppDatabase sarebbero stati inutili, in quanto molte funzioni erano costituite da semplici sequenze di query sql e non avevano alcuna logica da verificare, se non quella racchiusa nelle query stesse, che però non potevano essere testate, se non attraverso un database di test.

Inoltre, limitandosi a fare degli unit test, non si poteva controllare che le funzioni del Router e quelle dell'AppDatabase interagissero correttamente.

Un altro problema era che, a causa dell'implementazione della libreria messa a disposizione dagli sviluppatori di MinIO per interfacciarsi al server tramite Go, non era possibile in nessun modo creare un mock.

6.4.3 Soluzione adottata

Ho quindi deciso di scrivere degli integration test verificassero il funzionamento delle intere chiamate API, e che unissero quindi gli unit test delle funzioni del Router e quelli delle corrispondenti funzioni dell'AppDatabase. In questo modo potevo testare sia l'interazione tra funzioni del Router e dell'AppDatabase, sia la correttezza della loro logica, se necessario.

Per quanto riguarda le parti di codice che interagiscono MinIO invece, mi sono limitato a dare per scontata la loro correttezza. Sono infatti riuscito solo a trovare il modo di creare una sorta di stub del server MinIO che restituisse un codice di successo ad ogni richiesta. In futuro bisognerá testarle tramite un server MinIO creato appositamente.

6.5 Strumenti usati per i test

Per la stesura dei test mi sono servito principalmente di due librerie:

- testing[2]
- go-sqlmock[3]

testing È contenuta nella libreria standard di Go, fornisce il supporto per creare test automatizzati, subtest, benchmark e offre varie funzionalitá basilari.

go-sqlmock Simula il comportamento del database. Ogni test imposta le query che il mock si deve aspettare, e i risultati che deve restituire. Se durante il test il mock riceve query diverse, nell'ordine sbagliato o non le riceve tutte, il test fallisce.

Ciò significa che la correttezza delle query e del loro ordine non viene testata, e viene data per scontata.

6.6 Struttura dei test

Per evitare eccessive ripetizioni ho usato la tecnica del Table Driven Testing, che consiste, per ogni scenario nel test, nell'avere una tabella che in ogni riga contiene sia i valori da dare in input, sia tutti gli output corrispondenti che ci si aspetta di ricevere. Il test itera sulle righe della tabella ed esegue un subtest per ognuna di esse.

Di seguito riporto come esempio il test per la chiamata GetCars, che restituisce all'utente le informazioni della macchina avente l'id, la targa o il codice MAC dati in input.

I test sono cosí strutturati: si preparano tutti i valori che verranno usati da tutti o la maggior parte dei test.

```
func TestGetCars(t *testing.T) {
   // Prepare variables used by all tests
   var (
      uid    = existingUsers[0]
      cid    = existingCars[1]
      car, _ = uuid.FromString(cid)
      size   = float32(4.5)
      fuel   = 10
      photo  = ""
      carInfo = types.CarInfo{
         Plate: "fg56hil",
         MAC:   "5c:08:fc:f9:ad:9b",
         Cid:   car,
         Make: "fiat",
         Model: "punto",
         Color: "green",
         Size: &size,
         Fuel: &fuel,
         Photo: &photo,
      }

      headers := http.Header{
         "Accept":       {"application/json"},
         "X-App-Build":  {"1"},
         "X-App-Version": {"1.0.0"},
         "X-App-Lang":   {"it"},
         "X-App-Platform": {"android"},
         "X-Id":         {uid},
         "Content-Type": {"application/plain"},
      }
   )
```

Successivamente si crea la tabella dei test e un ciclo che itera sulle righe della tabella, eseguendo un subtest con i valori di ogni riga.

```
// Test table
var tests = map[string]struct {
   plate              string
   mac                string
   cid                string
   carExists          bool
   expectedResponseCode int
}{
   "TEST plate":                {"fg56hil", "", "", true, http.StatusOK},
   "TEST plate does not exist": {"fg56hil", "", "", false, http.StatusNotFound},
   "TEST mac":                  {"", "5c:08:fc:f9:ad:9b", "", true, http.StatusOK},
   "TEST mac does not exist":   {"", "5c:08:fc:f9:ad:9b", "", false, http.StatusNotFound},
   "TEST cid":                  {"", "", cid, true, http.StatusOK},
   "TEST cid not well formed":  {"", "", "not a valid cid", true, http.StatusBadRequest},
   "TEST cid does not exist":   {"", "", cid, false, http.StatusNotFound},
   "TEST nothing":              {"", "", "", false, http.StatusBadRequest},
}
for name, test := range tests {
   t.Run(name, func(t *testing.T) {
```

Il subtest, tramite la funzione *getTestRouter* scritta per evitare ripetizioni, crea il Router che si avvarrá dell'AppDatabase contenente il mock del database.

```
        // Get router, handler and mock db
        router, mockStruct, err := getTestRouter()
        if err != nil {
           t.Fatal(err)
        }
        handler := router.Handler()
        mock := mockStruct.mock

        // Close connection at the end of the test
        defer func() {
           _ = mockStruct.mockDB.Close()
           _ = router.Close()
        }()
```

Poi imposta le query che tale mock si deve aspettare e cosa deve restituire. Alcune query per evitare ripetizioni, essendo usate dalla maggior parte dei test, vengono impostate da funzioni apposite. Per esempio la query usata per controllare se un utente esiste viene impostata dalla funzione mockCheckUser.

```
// Prepare expected db queries and their return values
mockCheckUser(mock, uid)

resCar := carInfo
if test.plate != "" {
   resCar.Plate = test.plate
   q := mock.ExpectQuery(regexp.QuoteMeta("SELECT cars.cid, cars.plate,
       cars.make, cars.model, cars.color, cars.size FROM cars, drivers
       WHERE cars.plate = ? AND drivers.cid = cars.cid AND drivers.tba = 0
       LIMIT 1")).
      WithArgs(test.plate)
   if test.carExists {
      q.WillReturnRows(sqlmock.NewRows([]string{"cid", "plate", "make",
          "model", "color", "size"}).
         AddRow(cid, resCar.Plate, resCar.Make, resCar.Model, resCar.Color,
             *resCar.Size))
   } else {
      q.WillReturnError(sql.ErrNoRows)
   }
} else if test.mac != "" {
   resCar.MAC = test.mac
   q := mock.ExpectQuery(regexp.QuoteMeta("SELECT cars.cid, cars.plate,
       cars.make, cars.model, cars.color, cars.size FROM cars, drivers
       WHERE cars.mac = ? AND drivers.cid = cars.cid AND drivers.tba = 0
       LIMIT 1")).
      WithArgs(test.mac)
   if test.carExists {
      q.WillReturnRows(sqlmock.NewRows([]string{"cid", "plate", "make",
          "model", "color", "size"}).
         AddRow(cid, resCar.Plate, resCar.Make, resCar.Model, resCar.Color,
             *resCar.Size))
   } else {
      q.WillReturnError(sql.ErrNoRows)
   }
} else if test.cid != "" {
   tCid, err := uuid.FromString(test.cid)
   if err != nil {
      resCar.Cid = tCid
   }
   q := mock.ExpectQuery(regexp.QuoteMeta("SELECT cars.cid, cars.plate,
       cars.make, cars.model, cars.color, cars.size, cars.photo, cars.fuel
       FROM cars, drivers WHERE cars.cid = ? AND drivers.cid = cars.cid
       AND drivers.tba = 0 LIMIT 1")).
      WithArgs(cid)
   if test.carExists {
      q.WillReturnRows(sqlmock.NewRows([]string{"cid", "plate", "make",
          "model", "color", "size", "photo", "fuel"}).
         AddRow(cid, resCar.Plate, resCar.Make, resCar.Model, resCar.Color,
             *resCar.Size, *resCar.Photo, *resCar.Fuel))
   } else {
      q.WillReturnError(sql.ErrNoRows)
   }
}
```

Successivamente il subtest prepara l'url e il body (se necessario) della richiesta http relativa alla chiamata API da testare, ed esegue la richiesta http passandola direttamente all'handler del router, in modo da evitare di dover avviare un server che la debba ricevere.

```
        // Prepare request
        url := fmt.Sprintf("/car/?cid=%s&plate=%s&mac=%s", test.cid, test.plate,
            test.mac)
        var body io.Reader = nil
        req, _ := http.NewRequest("GET", url, body)
        req.RequestURI = url
        req.Header = headers

        // Make request, get response
        response := httptest.NewRecorder()
        handler.ServeHTTP(response, req)
```

Infine controlla che la risposta contenga gli stessi valori attesi dal subtest. Se i risultati non sono corretti, le funzioni vanno in errore o le query ricevute dal mock database non sono quelle impostate, il subtest fallisce.

```
        // Check response code and body
        if response.Code != test.expectedResponseCode {
           t.Fatalf("Wrong response code: %d", response.Code)
        }
        if test.expectedResponseCode == http.StatusOK {
           var res types.CarInfo
           err = json.Unmarshal(response.Body.Bytes(), &res)
           if err != nil {
              t.Fatal(err)
           }
           switch {
           case res.Plate != resCar.Plate:
              t.Fatalf("Plate %s != %s", res.Plate, resCar.Plate)
           case res.MAC != resCar.MAC:
              t.Fatalf("MAC %s != %s", res.MAC, resCar.MAC)
           case res.Cid.String() != resCar.Cid.String():
              t.Fatalf("Cid %s != %s", res.Cid.String(), resCar.Cid.String())
           case res.Make != resCar.Make:
              t.Fatalf("Make %s != %s", res.Make, resCar.Make)
           case res.Model != resCar.Model:
              t.Fatalf("Model %s != %s", res.Model, resCar.Model)
           case res.Color != resCar.Color:
              t.Fatalf("Color %s != %s", res.Color, resCar.Color)
           case *res.Size != *resCar.Size:
              t.Fatalf("Plate %f != %f", *res.Size, *resCar.Size)
           case *res.Fuel != *resCar.Fuel:
              t.Fatalf("Fuel %d != %d", *res.Fuel, *resCar.Fuel)
           case *res.Photo != *resCar.Photo:
              t.Fatalf("Photo %s != %s", *res.Photo, *resCar.Photo)
           }
        }
     })
  }
}
```

Capitolo 7

Conclusioni

7.1 Errori emersi grazie alla stesura dei test

Essendo il codice per cui andavano scritti i test automatici giá stato collaudato manualmente, non mi aspettavo di trovare grandi errori. Come giá detto questi test serviranno principalmente per assicurarsi che future modifiche non danneggino il codice giá esistente e funzionante.

Nonostante ció, alcuni errori relativi al controllo dei permessi sono venuti alla luce: alcune chiamate API relative alla macchina (come per esempio get-car-statistics, usata per ottenere le statistiche di una macchina) controllano che chi ha fatto la richiesta ne sia il proprietario o il guidatore, a seconda della chiamata. In alcune funzioni si controllava che la richiesta venisse fatta dal guidatore della macchina, mentre si sarebbe dovuto controllare che venisse fatta dal proprietario della macchina; in altre funzioni tale controllo veniva fatto invece erroneamente.

7.2 Possibili futuri miglioramenti

In futuro potrebbe essere utile scrivere dei test che interagiscano con un database di test per poter controllare la correttezza delle query usate. Probabilmente questi non saranno eseguiti ad ogni push dei cambiamenti, ma manualmente, solo quando ce ne sará bisogno, in modo da non appesantire inutilmente il server di SapienzaApps. Lo stesso si può dire per le parti di codice che interagiscono con MinIO.

Bibliografia

[1] MinIO `https://en.wikipedia.org/wiki/MinIO`

[2] Package Go /dev/testing https://pkg.go.dev/testing

[3] Package go-sqlmock https://github.com/DATA-DOG/go-sqlmock