INGEGNERIA DEL SOFTWARE ORIENTATA AI MICRO SERVIZI

Report progetto
a.a. 2020-2021

Parisotto Stefano, 0000931989
Salvatore Maria Francesca, 0000938648

Tecnologie utilizzate per l'implementazione

ACMESky

Camunda
La versione usata è Spring Boot. Il progetto è stato creato usando il link messo a disposizione sul sito di camunda: https://start.camunda.com. Le librerie utilizzate sono le seguenti:

- Camunda 7.15.0
- Spring Boot Version: 2.4.3
- Java: 11

PostgreSQL
È un DBMS completo e ad oggetti rilasciato con licenza libera. L'impostazione della configurazione del server permette la connessione inserendo "postgres" sia come nome utente e sia come password.

pgAdmin
Abbiamo optato per l'utilizzo di questa applicazione basata su un'interfaccia grafica semplice e comoda che consente di amministrare il database di PostgreSQL.

Compagnie aeree

PostgreSQL
È un DBMS completo e ad oggetti rilasciato con licenza libera. L'impostazione della configurazione del server permette la connessione inserendo "postgres" sia come nome utente e sia come password.

GPS

Bing Maps REST Services
Servizi messi a disposizione dalla Microsoft per ottenere posizione e/o informazioni riguardate luoghi e percorsi.

In Generale

Docker
Abbiamo scaricato Docker e installato Postgres tramite un comando predefinito nel terminale ottenendo l'apposito container. Quest'ultimo racchiude il database al cui interno è possibile visualizzare la console del server che si avvia o si ferma a seconda del tasto premuto (play/stop).

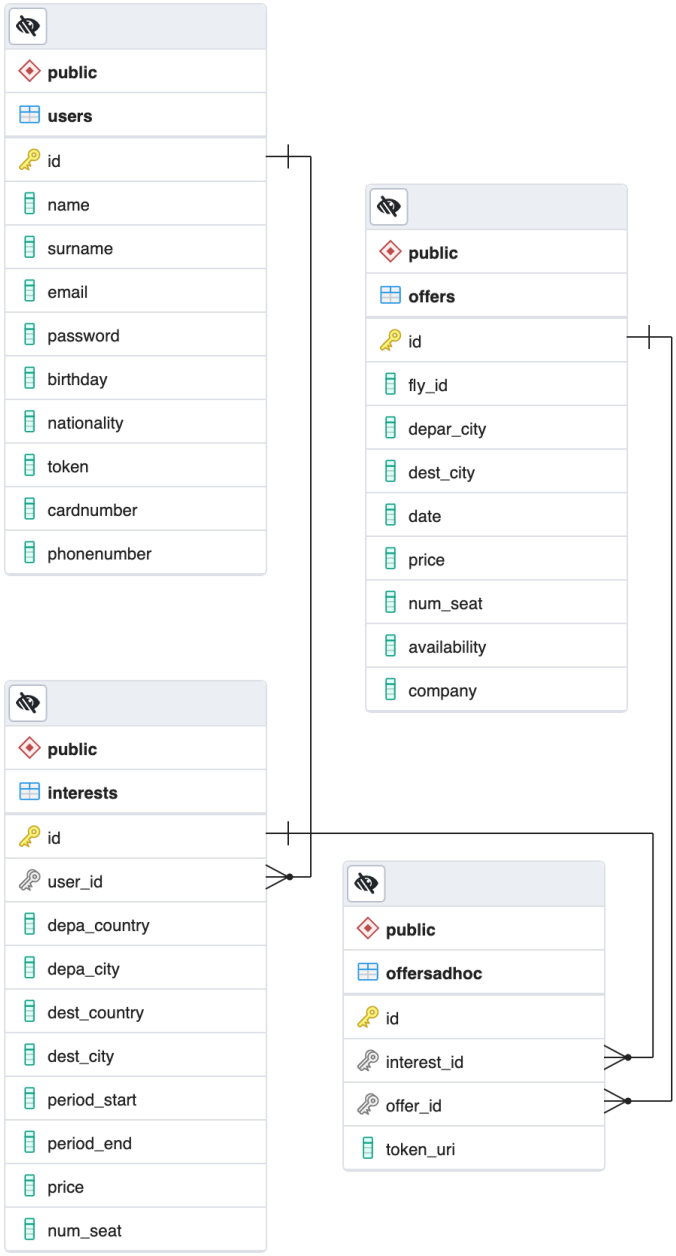
Postman
Una piattaforma facile e veloce per testare i vari servizi sia SOAP che REST.

SoapUI
È un'applicazione di servizi Web open source che ci ha permesso di eseguire vari test sulle Api SOAP del progetto.

Logica ed implementazione

ACMESky

Schema relazionale del DB



Segue una breve descrizione delle tabelle:

- *Users* contiene le informazioni riguardante tutti gli iscritti al servizio;
- *Interests* raccoglie gli interessi di tutti gli utenti. Ogni utente può avere più interessi, ma ogni interesse fa riferimento ad un singolo utente. I campi *depa_city* e *dest_city* sono opzionali poiché è possibile che all'utente interessino dei voli nazionali e non si troverebbe costretto ad aggiungere un interesse per ogni aeroporto dello stato;

- *Offers* contiene tutte le offerte trovate/ricevute dalle compagnie aeree. Il campo *fly_id* è univoco per ogni compagnia aerea, ma non lo è con le altre compagnie;
- *Offersadhoc* serve per collegare ogni offerta con l'interesse a cui essa è collegata. Inoltre, contiene un campo (*token_uri*) che viene utilizzato da Acmesky per salvare il link dell'offerta.

Percorsi logici

1) **Registrazione**: l'utente tramite l'apposito form html manda i dati al server di ACMESky che crea il profilo e restituisce una conferma oppure una pagina di errore.
2) **Login**: l'utente tramite apposita pagina messa a disposizione da ACMESky manda e-mail e password al server, che le controlla e, se corrette, l'utente può eseguire le operazioni di modifica delle informazioni, interessi ed eliminazione dell'account.
3) **Aggiunta di una nuova offerta LM**: ACMESky può ricevere in qualsiasi momento un'offerta da qualsiasi compagnia aerea che invierà agli utenti interessati effettuando un *match* tra offerta e tutti gli interessi del database.
4) **Controllo giornaliero delle nuove offerte**: ACMESky esegue giornalmente un match tra gli interessi degli utenti e le offerte arrivate da ogni compagnia aerea associata, per poi inviare i relativi riscontri positivi agli utenti tramite Prontogram.
5) **Controllo di una offerta da parte di un utente**: tramite il servizio Prontogram, l'utente potrà controllare le offerte inviate da ACMESky e procedere all'acquisto qualora fossero ancora valide.
6) **Acquisto di una offerta**: l'utente che decide di acquistare un'offerta valida dovrà compilare i campi richiesti (es. numero biglietti e nominativi) e subito dopo verrà rindirizzato alla pagina della banca per completare l'acquisto. Infine, otterrà la conferma che tutto è andato a buon fine con la ricezione dei biglietti (ed eventuale servizio noleggio prenotato) o con un messaggio di errore.

Problema del controller che espone le rotte

Di default, l'engine di Camunda mette a disposizione delle rotte Rest con le quali è possibile gestire l'intera istanza. In particolare, è disponibile una rotta (*IP:PORT/engine-rest/message)* tramite la quale è possibile mandare messaggi a Camunda. Dal momento che è stato richiesto un portale Web tramite un browser, un utente non riuscirebbe facilmente a mandare messaggi a Camunda in questo modo. Perciò si è realizzato, usando la libreria contenuta nella versione Spring di Camunda (javax.ws), un controller in grado di esporre delle rotte agli utenti che, tramite esse, possono fare richieste al controller e questo avrà il compito di mandare i messaggi alle varie istanze di ACMESky.
Il controller, quando ottiene i dati dal client, manda un messaggio a Camunda con i dati ricevuti dall'utente e rimane in attesa che finisca l'esecuzione. Una volta che l'esecuzione sarà completata, il controller prende le variabili che sono nell'istanza e poi li passa al client tramite una pagina html.
Il comportamento generale è in seguito descritto brevemente:

1. il client manda una richiesta ad ACMESky;
2. il controller la riceve, fa partire/sveglia un'istanza di ACMESky che elabora il tutto;
3. quando il processo finisce, il controller ottiene le variabili di output che vengono ritornate dall'esecuzione.
4. Il controller riempie una pagina html con le variabili ottenute e la restituisce all'utente come risposta alla richiesta inizialmente ricevuta

In poche parole: il client non sveglia direttamente i *messageTask*, ma chiama il controller che li sveglia ed inoltre Camunda non contatta direttamente il client, ma passa anche lui per il controller.
Questa soluzione porta un "problema": usa un modello richiesta e risposta e quindi il ACMESky non può contattare l'utente se questo non gli scrive. Inoltre, l'utente dovrà avere sempre una pagina html di risposta ad ogni sua richiesta.

Elenco delle classi

I service Task sono stati implementati in Java. Tutte le tabelle nel database hanno il corrispettivo in classi java (User, Offer, Interest, OffersAdHoc). Seguito, una lista dei servizi scritti in java.

CheckCredentials

Prende in input l'e-mail e la password e restituisce in output ID e checkLogin. Quest'ultima è una variabile booleana che serve a controllare successivamente, se le credenziali utente sono corrette. Inoltre, se è *true*, allora la variabile ID contiene l'ID dell'utente; in caso contrario non verrà settato nessun ID.

GetUserInformation

Prende in input l'ID utente restituendo tutte le informazioni che gli riguardano.

UpdateInterest

Revisiona gli interessi dell'utente tramite tre operazioni: eliminare, aggiornare o inserire un nuovo interesse. Nel farlo, prende in input un JSON che trova salvato nelle istanze di Camunda chiamato "interests". Il Json contenente un array, in cui ogni elemento è formato da una coppia di interesse (dati relativi) e operazione, presenta il formato seguente:

```
{
        "interests": [{
                "operation": "DELETE/INSERT/UPDATE",
                "interest": {
                        ...
                        vari campi
                        ...
                },
                {
                "operation": ...
                ...
                }
        }]
}
```

GetInterests

A differenza di GetUserInterest che si riferisce agli interessi di un utente specifico basandosi sull'ID, questa classe ritorna tutti gli interessi presenti nel database così da poter successivamente controllare le offerte delle compagnie aeree.

CheckInterests

Prende in input una lista di interessi e proposte di volo e controlla presenza dei match. In caso affermativo, restituisce delle possibili offerte. Per questa ragione esistono due classi:

1. "Proposal" identica a "Offer", ma con due campi in meno ed è concettualmente diversa: una proposta è un volo con i relativi dati e che diventa un'offerta quando combacia con almeno un interesse;
2. "PossibleOffers" è una classe di supporto: al suo interno contiene una mappa che ha come chiavi delle proposte ognuna di esse associate a una lista degli ID dei vari utenti che presentano un match con un interesse, di conseguenza trasformandole in offerte. In più, ha delle istruzioni per poterle convertire in Json.

CreateOffer

Data una proposta, crea la relativa offerta e la aggiunge nel database.
È possibile che l'offerta sia già stata precedentemente inserita all'interno del database. In questo caso, si limiterà a recuperare l'identificativo che ha questa offerta all'interno del database.

CreateTokenUri
Crea il link dell'offerta per gli utenti interessati.

SaveOfferAdHoc
Prende in input l'identificativo di una offerta, quello dell'interesse a cui essa è collegata ed il link dell'offerta (creato precedentemente) e salva sul database una tupla contenente questi dati nella tabella OfferAdHoc.

CheckOffer
Prende l'ID del volo e quello della compagnia (che dovrebbe essere univoco) e controlla se quel volo è ancora disponibile.

UpdateStateOffer
Come si può intuire dal nome, questa classe si limita ad aggiornare lo stato dell'offerta all'interno del database.

CheckTokenURI
In input prende il tokenUri e se il link è valido, allora restituisce i dati utili per iniziare l'acquisto.
La validità del link non è solamente basata sul fatto che sia stato ritrovato nel database, ma è anche basato sul controllo temporale. I link che fanno riferimento a voli passati non saranno considerati più validi.

ChangeInfo
Aggiorna le informazioni di un utente basandosi suoi dati passati in input.

CreateUser
Aggiunge un utente al database, se la mail passata in input non è già stata utilizzata per la creazione di un altro account e se i dati in input sono validi. In output restituisce la variabile checkID che serve per controllare se *CreateUser* è riuscita a salvare l'utente nel database o ha avuto un errore.

DeleteUser
Preso in input l'ID di un utente, se valido, lo rimuove dal database.

GetUserInterests
Preso in input l'ID di un utente, se valido, restituisce tutti i suoi interessi (anche quelli passati).

InitPayment
Genera un identificativo numerico casuale per un pagamento e calcola il prezzo.

MergeInterests
Ha il compito di unire insieme gli interessi (che possono essere unititi senza perdere offerte) al fine di ridurre il numero di interessi che si mandato alle compagnie aeree.

RemovePayment
Prende in input un identificativo di un pagamento e lo rimuove dai pagamenti ancora da completare.

MessageSender
Serve per mandare un messaggio dall'istanza del cliente a quella di ACMEsky. Prende in input il nome del messaggio, l'identificativo dell'istanza e le variabili da inserire dentro al messaggio.

SendData
Passa i valori necessari per creare la pagina per il pagamento dall'istanza di ACMEsky a quella del client.

SendTickets
Passa i dati dei biglietti ed eventuale prenotazione del noleggio al client.

Classi "Send[...]"
Servono per chiamare i servizi esterni esplicitati dai vari nomi che seguono.

Gestione offerte
"Controllo offerta periodico" in realtà è un timer che scatta ogni volta che bisogna ricevere nuove offerte. Per esigenze implementative è stato sostituito da un *Message Start Event* in modo che fosse più semplice testare tutto il meccanismo ogni qualvolta fosse necessario.
Successivamente i voli di interesse vengono raccolti nel database e mandati ad ogni compagnia aerea associata ad ACMESky. In questa transazione, il *Service Task "unione degli interessi"* fa in modo che:

- interessi comuni di due o più persone vengano inviati una sola volta alle varie compagnie aeree per evitare che poi si abbiano da gestire più offerte che sono identiche tra loro.

Es. Mario e Luigi sono entrambi interessati alla tratta Bologna-Roma. Il loro interesse è lo stesso per cui verrà inviato alle compagnie aeree una sola volta;

- il sistema non invii più volte la stessa offerta a un profilo che è interessato.

Es. Se l'offerta sulla tratta precedente dura un mese, Mario e Luigi la riceveranno una volta soltanto e non una volta al giorno fino alla fine della durata dell'offerta;

La classe *"MergeInterests"* esplicita il meccanismo con cui gli interessi simili per tratte e tempo sono inglobati e inviate. Dietro tutti i casi gestiti e commentati al suo interno, la logica è quella di fondere, quando possibile, gli interessi più specifici con quelli più generali.

Es. dati due interessi: il primo (ITA, Bologna) -> (FRA, Parigi) ed il secondo (ITA, Bologna) -> (FRA, _); il primo interesse sarà inserito all'interno del secondo poiché quest'ultimo richiede già tutti i voli che portano da Bologna ad un aeroporto qualsiasi della Francia e, essendo il primo interesse un caso specifico del secondo, non si andranno a perdere offerte.
In questo caso però, il primo interesse non verrà cancellato: semplicemente non verrà richiesto esplicitamente alle compagnie aeree, ma verrà comunque tenuto in conto quando si farà matching in seguito (qualora esistesse l'offerta).

All'interno dell'esempio è stato usato il termine "inserito" invece di "ignorato", questo perché è possibile che le date di inizio e fine e/o il prezzo massimo non coincidono tra i due interessi. Per risolvere questo, l'algoritmo andrà a portare le date ed il prezzo dell'interesse finale in modo che riesca ad includere entrambi gli interessi.

Banca

Descrizione

È un semplicissimo servizio bancario che permette di eseguire pagamenti utilizzando carte di credito. Si è voluto non implementare nessun servizio di registrazione/login per non rendere troppo complessa la sua progettazione. Espone tre funzionalità:

- **requestPayment**: passando i dati della carta, un quantitativo di soldi da pagare ed un identificativo, la banca congela momentaneamente i soldi su quel conto ed avvisa ACMESky che è stato eseguito un pagamento;
- **compensation**: se l'identificativo del pagamento è valido, allora rende di nuovo disponibili i soldi che erano stati congelati nel conto;
- **concludePayment**: se l'identificativo del pagamento è valido, allora preleva i soldi dal conto.

Tecnologie utilizzate: Jolie, SOAP

Implementazione

Non è stata implementato nessun database, ma si sono inseriti all'interno del codice i dati di alcune carte valide ed il credito di queste. Nel caso in cui le richieste diano errore, allora verranno rimessi i soldi nella carta ed annullato il pagamento.
Funzionalità esposte:

- **requestPayment**: (paymentId: int, cost: double, card: string, secretNumber: int) -> (error: bool, sid: string). Presi in input i dati per il pagamento, viene controllato se sono dei dati validi e, se lo sono, viene rimosso momentaneamente un quantitativo di soldi pari al valore di *cost* dal conto e viene mandato un messaggio SOAP ad ACMESky per avvisarla che è stato effettuato il pagamento;
- **compensation**: (sid: string) se il sid del processo è valido, allora vengono rimessi i soldi momentaneamente rimossi dal conto;
- **concludePayment**: (sid: string) se il sid del processo è valido, allora vengono i soldi precedentemente rimossi vengono aggiunti al conto di ACMESky.

Compagnie aeree

Descrizione

Il servizio permette di comprare biglietti per voli aerei da una destinazione ad un'altra in una data fissata. Inoltre, permette di cercare voli tramite una funzionalità di ricerca classica. Il servizio espone queste funzionalità:

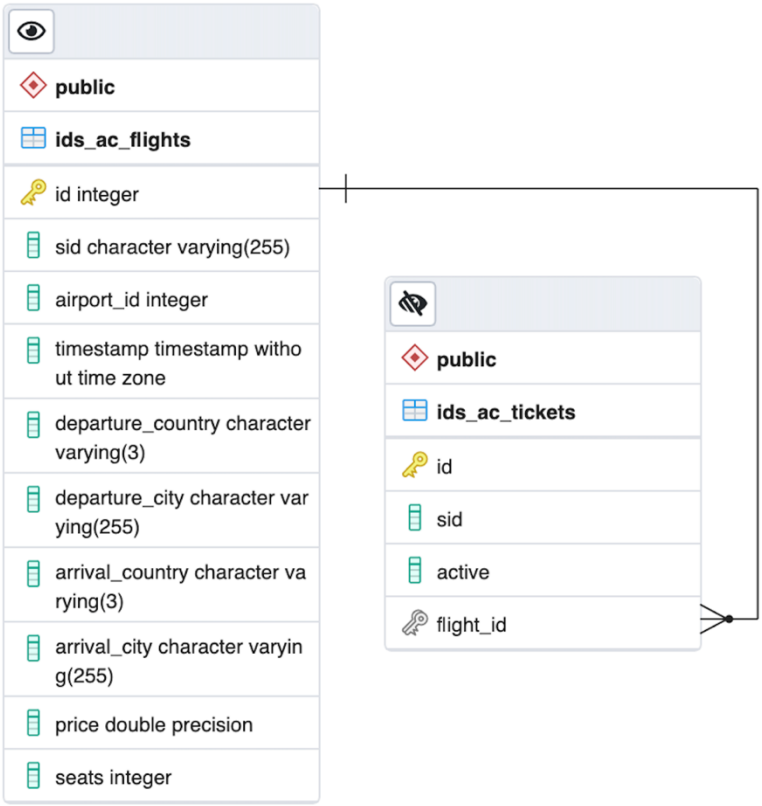
- **searchOffers**: dato in ingresso una lista di interessi, restituisce i dati dei voli aerei che combaciano con i singoli interessi;
- **availableSeats**: dato in ingresso un ID di un volo, restituisce quanti posti disponibili sono rimasti per quel volo;
- **purchaseOffer**: crea i biglietti per il volo di interesse che dovranno essere confermati o annullati successivamente;
- **confirmPurchase**: conclude l'acquisto iniziato precedentemente con la funzionalità *purchaseOffer* validando i biglietti precedentemente creati;
- **cancelPurchase**: annulla l'acquisto iniziato precedentemente con la funzionalità *purchaseOffer* invalidando i biglietti precedentemente creati.

C'è un ulteriore servizio esposto, che non è pubblico, ma riguarda i dipendenti della compagnia aerea. Esso serve per aggiungere una offerta LM ed avvisare ACMESky di quest'ultima.

Tecnologie utilizzate: Jolie, SOAP, PostgresSQL

Implementazione

Tutte le offerte ed i voli di interesse sono mantenuti all'interno di un database relazionale con il seguente schema:



- All'interno di *ids_ac_flights* vengono tenuti tutti i voli messi a disposizioni dalla compagnia aerea.
- All'interno di *ids_ac_tickets* vengono inseriti tutti i biglietti creati dalla compagnia aerea

NOTA: Il campo *airport_id* contiene l'identificativo della compagnia aerea che propone il volo. Questo campo è stato aggiunto solamente per evitare di realizzare un database diverso per ogni compagnia aerea. In questo modo, ne avremo uno in grado di raccogliete tutti i voli delle compagnie. Questo è stato fatto in modo da utilizzare il minor numero di risorse del computer su cui sono stati eseguiti i test.

Funzionalità esposte:

- **searchOffers**: (preferenceID: int, fromTimestamp: string, toTimestamp: string, departureCountry: string, departureCity: string, arrivalCountry: string, arrivalCity: string, maxPrice: double)* → (preferenceID: int, airportID: int, offerID: int, timestamp: string, departureCountry: string, departureCity: string, arrivalCity: string, arrivalCountry: string, price: double, seats: int)*. Riceve in input una lista di interessi e, per ognuno di questi, risponde con tutti voli che soddisfano i requisiti di arrivo e partenza in quel range di orario e di prezzo. Nel caso in cui si riceva lo stesso interesse replicato, verranno mandati in risposta le stesse offerte replicate e quindi sta all'altra parte stare attenta a quello che chiede e/o gestire la risposta;
- **availableSeats**: (offerID: int) -> (availableSeats: int, seats: int). Presa l'ID dell'offerta, cerca all'interno del database quanti biglietti sono attivi, in quel momento, per quella offerta e restituisce tale cifra ed il numero di biglietti totali per quel volo;
- **purchaseOffer**: (offerID: int, seats: int) -> (sid: string, ticket*: int). Se l'ID del volo è valido e ci sono ancora un numero di posti disponibili maggiori o uguali a quelli richiesti con il campo *seats*, allora vengono generati *n* biglietti, dove *n* è il valore chiesto con *seats*. In risposta verranno mandati gli identificativi dei biglietti e l'identificativo per poter completare il pagamento o annullarlo;
- **confirmPurchase**: (sid: string). Ricevuto il messaggio, si procede a confermare i biglietti creati con quell'identificativo;

- **cancelPurchase**: (sid: string). Ricevuto il messaggio, si procede a rendere nulli i biglietti creati con quell'identificativo.
- **addLMOffer**: (depaCountry: string, depaCity: string, destCountry: string, destCity: string, price: double, num_seat: int, timestamp: string, company: string). Aggiunge una nuova tupla all'interno del database nella tabella *ids_ac_flights* settando *airport_id* con il codice della compagnia aerea a cui viene mandato questo messaggio. Questa funzionalità non è stata inserita nel wsdl della compagnia aerea perché non si vuole rendere accessibile a tutti questa funzionalità.

Compagnia di noleggio

Descrizione

Il servizio di compagnia di noleggio permette di prenotare un tassista per potersi far portare dove si desidera in una determinata data. Espone due servizi:

- **Reserve**: prenota un servizio di trasporto prendendo come parametri un nominativo ed il luogo ed ora di dove si presenterà il tassista. Risponderà con l'identificativo della prenotazione;
- **CancelReserve**: disdice una prenotazione precedentemente fatta utilizzando l'identificativo della prenotazione.

Tecnologie utilizzate: Jolie, SOAP.

Implementazione

Non fa utilizzo di database, ma salva i dati all'interno di una struttura dati chiave-valore dove la chiave è l'ID della prenotazione (generato casualmente dal software della compagnia) e il valore è un tipo composto dai dati inerenti alla prenotazione: il nominativo, la data ed il luogo dove essere prelevati.

- **Reserve**: (name: string, address: string, timestamp: string) --> (reserved: boolean, id: string). Presi in input i dati per la prenotazione, salva in memoria la prenotazione e genera un ID casuale. In ritorno restituirà un messaggio contenente un valore positivo per *reserved* e l'ID generato per la prenotazione. Nel caso in cui la prenotazione non vada a buon fine il campo *reserved* sarà negativo.
- **CancelReserve**: (id: string) → (canceled: boolean). Prende in input l'ID della prenotazione e, se valido, viene rimosso dalla memoria la prenotazione con tale identificativo e restituisce in output *canceled* impostato con un valore positivo; mentre, nel caso in cui l'ID non sia valido, il valore di *canceled* sarà impostato a negativo.

Prontogram

Descrizione

Servizio di messaggistica dove ogni utente può inviare e leggere messaggi.
Per semplificare l'applicazione si è deciso di non utilizzare una registrazione e nemmeno un meccanismo di login con credenziali. Espone tre funzionalità:

- **Leggere i propri messaggi**: permette di richiedere i messaggi che sono stati ricevuti da un determinato numero;
- **Inviare un messaggio**: permette di inviare un messaggio ad un determinato numero utilizzando però una *reference* valida;
- **Chiedere una reference**: serve per ottenere una *reverence* che ci permetterà di inviare un messaggio.

Tecnologie utilizzate: Javascript, express, nanoid, REST.

Implementazione

Non è stato utilizzato nessun database, ma viene tutto salvato semplicemente su una struttura dati e, di conseguenza, se il servizio viene spento, vengono persi tutti i messaggi ricevuti fino a quel momento.
Servizi esposti:

- **/api/messages/<receiver>/<messageId>** : *receiver* è il numero di cui si vuole ricevere il messaggio e serve per recuperare l'id-esimo messaggio inviato a tale numero. Dopo aver aggiunto il messaggio alla lista dei messaggi per quel numero, risponderà con un json contente tre campi: *statusCode*, *exixts* e *text.* Il primo campo mi conferma che la risposta è stata elaborata correttamente; il secondo campo è una variabile booleana che mi informa se c'è stato un errore durante l'esecuzione della mia richiesta; infine, se non ci sono stati errori, il campo text conterrà il testo del messaggio.
- **/api/messages/:receiver** : nel body della richiesta ci devono essere anche la *reference*, il numero del mittente ed il contenuto del messaggio che si vuole mandare. L'output sarà un json contente uno *statusCode* di errore ed il campo error sestato a true nel caso in cui manchino alcuni dei parametri appena descritti o se la reference non è valida. In caso contrario, ovvero quello in cui non ci sia stato errore, allora l'output sarà sempre un json, ma con i campi *statusCode* impostato a 200 ed *errors* a false e se nella memoria sarà aggiunto il messaggio appena creato;
- **/api/reference** : restituisce un json contente uno *statusCode* ed un campo *reference* contenente la reference da usare per inviare un messaggio.

NOTE: Ai fini di facilitare il debugging è stata implementata una interfaccia estremamente semplice per visualizzare i messaggi di un determinato numero

GPS

Descrizione

Servizio che date due località che posso essere Città e/o vie, restituisce la distanza in linea d'aria in kilometri da queste due. Espone una singola funzionalità che è quella appena descritta.

Tecnologie utilizzate: PHP, REST

Implementazione

GET http://localhost:4010?origin=<YOUR_ORIGIN_POINT>&destination=<YOUR_DESTINATION_POINT>

Questo servizio fa utilizzo delle Bing Maps REST Services (https://docs.microsoft.com/en-us/bingmaps/rest-services/). In particolare, sono state utilizzate due funzionalità: quella per ottenere informazioni riguardo ad un luogo/via e l'altra che date due coordinate, restituisce la loro distanza.
La prima funzionalità citata sopra viene utilizzata due volte per ottenere le coordinate dei due luoghi passati in input (che sono in forma di stringa e non in forma di coordinata). Una volta ricevuta risposta, che non sia un errore, si procede a chiedere la distanza alla seconda funzionalità citata. L'output ottenuto verrà poi ripulito ed inserito nel campo *distance* della risposta.
La risposta del servizio sarà un JSON con due campi: *distance* e *statusCode.*
Il primo campo contiene il valore sopra descritto; il secondo permette di rivelare la presenza di eventuali errori.