Fly on the Cloud

Matteo Fanfarillo
Facoltà di Ingegneria Informatica
Università degli Studi di Roma Tor Vergata
Alatri, Italia
matteo.fanfarillo99@gmail.com

Luca Capotombolo
Facoltà di Ingegneria Informatica
Università degli Studi di Roma Tor Vergata
Monterotondo, Italia
capoluca99@gmail.com

***Abstract*—L'obiettivo di questo documento è quello di descrivere l'architettura dell'applicazione, le scelte progettuali effettuate, l'implementazione realizzata, le limitazioni riscontrate e la piattaforma software usata per lo sviluppo.**

Keywords — Comunicazione, container, database, discovery, logging, microservizio, pattern, testing set, training set, volo.

I. Architettura e scelte progettuali

A. Descrizione dell'applicazione

L'applicazione, dal nome *Fly on the Cloud*, consente di effettuare l'acquisto di biglietti aerei. Esistono due tipologie di utenti: i turisti, che acquistano i biglietti, e le compagnie aeree, che forniscono le informazioni relative ai voli. Entrambe le tipologie di utenti hanno la necessità di registrarsi al sito per poter sfruttare le funzionalità del sistema.

Il turista può acquistare i biglietti aerei per conto di una o più persone. A tal proposito, se vuole, ha la possibilità di selezionare il posto (o i posti) a sedere e i servizi aggiuntivi di cui vuole usufruire (i.e. bagaglio in stiva aggiuntivo medio, bagaglio in stiva aggiuntivo grande, bagaglio speciale, animale domestico in cabina, assicurazione bagagli e trasporto neonato). Prima della conferma di prenotazione, il sistema fornirà qualche suggerimento sull'acquisto dei biglietti: in particolare, indicherà all'utente in quale data potrebbe essere più conveniente effettuare la prenotazione nell'ottica di risparmiare sul prezzo dei biglietti. Tale suggerimento viene generato sulla base dell'andamento al variare del tempo dei prezzi dei biglietti dei voli già effettuati. Per portare a termine la prenotazione, l'utente dovrà effettuare il pagamento.

La compagnia aerea può aggiungere un nuovo volo disponibile all'interno del sistema e può cambiare il prezzo di un volo inserito precedentemente, della selezione dei posti a sedere e dei servizi aggiuntivi.

B. Descrizione dell'architettura

Come richiesto dalle specifiche, per lo sviluppo dell'applicazione è stata usata un'architettura a microservizi, dove ciascun microservizio implementa una specifica funzionalità dell'applicazione. In particolare, sono stati individuati i seguenti microservizi: **Front-end**, iscrizione al sito (**Registration**), prenotazione di un volo (**Booking**), gestione delle informazioni sui voli (**Flights Management**), suggerimento sull'acquisto dei biglietti (**Suggestions**) e pagamento (**Payment**). Più precisamente:

- Il microservizio di front-end, che svolge anche il ruolo di **API gateway**, comunica direttamente con tutti gli altri microservizi in modo tale da fare da intermediario tra il client e la logica applicativa dell'applicazione (implementata appunto dagli altri microservizi).
- Il microservizio Payment comunica col microservizio Booking per poter registrare i pagamenti assieme alle informazioni relative a ciascun volo prenotato.
- Il microservizio Flights Management comunica col microservizio Booking perché, per prenotare un volo o per aggiungerlo/modificarlo, sono necessarie sia le informazioni strettamente correlate con Booking (i.e. lista dei voli disponibili e dei posti liberi per ciascun volo), sia le informazioni strettamente correlate con Flights Management (i.e. prezzo aggiuntivo per la selezione dei posti a sedere e per la selezione dei servizi extra).
- Il microservizio Booking comunica col microservizio Suggestions poiché quest'ultimo deve disporre dello storico dei prezzi di ciascun volo già effettuato; dunque, Booking deve aggiornare volta per volta Suggestions sui prezzi di tutti i voli in modo tale che Suggestions possa mantenere il proprio archivio aggiornato.
- Ciascun microservizio gira all'interno di un container per facilitare il deploy dell'applicazione. Perciò, la comunicazione tra microservizi si traduce in una comunicazione tra container.

C. Descrizione delle scelte progettuali

Comunicazione tra i microservizi: la comunicazione tra i vari microservizi avviene mediante RPC (Remote Procedure Call) oppure mediante una coda di messaggi. In particolare, si è scelto di inserire una coda di messaggi nell'interazione tra il microservizio Payment e il microservizio Booking nel momento in cui devono essere registrate le informazioni relative al pagamento e alla relativa prenotazione effettuata da un certo utente. In tutti gli altri casi, si è scelto di utilizzare una comunicazione basata su RPC poiché si tratta di interazioni basate su richiesta-risposta dove il servizio client, per poter proseguire correttamente la sua esecuzione, deve in ogni caso attendere un riscontro da parte del servizio server.

Microservizi stateless e stateful: per migliorare la scalabilità dell'applicazione, tutti i microservizi, eccetto Suggestions, sono stati implementati in maniera stateless, ovvero in modo tale che il database non sia incluso all'interno dei microservizi stessi, bensì in un ambiente esterno. Per rendere l'unico microservizio stateful (Suggestions) più scalabile e tollerante ai guasti, si è deciso di replicarlo, introducendo più precisamente due repliche che corrispondono a due diversi container funzionanti. In particolare, si ha una replica primaria e una replica secondaria. La replica primaria è l'unica a comunicare direttamente con gli altri microservizi ed è quindi quella che risponde alle query e riceve per prima gli aggiornamenti. Quando riceve nuovi aggiornamenti (i.e. nuovi dati da mandare in persistenza), li invia a sua volta alla replica secondaria, in modo tale da averne una copia di back-up.

Design pattern adottati: per migliorare il disegno dell'architettura a microservizi, sono stati applicati alcuni design pattern, tra cui:

- Il pattern **saga**, per gestire agevolmente le transazioni che coinvolgono più microservizi, dato che si tratta di transazioni che insistono su molteplici tabelle appartenenti a servizi diversi. In particolare, i microservizi coinvolti nel pattern saga sono Payment e Booking per le operazioni di store delle informazioni relative al pagamento e alla prenotazione effettuata dall'utente. Il pattern è stato implementato in maniera decentralizzata (i.e. utilizzando un approccio basato sulla coreografia) per evitare il collo di bottiglia e il single point of failure e poiché sono coinvolti due soli microservizi (per cui è sufficientemente semplice coordinare i due servizi senza un componente centralizzato). È questo il motivo per cui viene utilizzata una coda di messaggi tra Payment e Booking: di fatto, per avere una corretta implementazione del pattern saga distribuito, i microservizi devono comunicare direttamente tra loro sfruttando un sistema a code di messaggi, che sia essa una semplice message queue (come nel nostro caso) o un sistema publish-subscribe (dove il sistema publish-subscribe sarebbe risultato più adeguato nel caso in cui fosse esistito un numero maggiore di microservizi coinvolto nel pattern saga).
- Il pattern **log aggregation**, per mantenere in modo centralizzato lo storico di tutte le operazioni che sono state eseguite all'interno del sistema, anche con lo scopo di effettuare delle attività di monitoraggio. Questo pattern risulta essere anche un aiuto importante per la fase di debugging dell'applicazione.

Discovery service: è stato implementato un servizio di Discovery che permette ai microservizi dell'applicazione di sapere qual è il numero di porta su cui gli altri microservizi sono in ascolto. Così si evita di scrivere in modo hard-coded il numero di porta del microservizio destinazione per ciascuna interazione. Il servizio di Discovery è stato implementato in modo decentralizzato con l'utilizzo di due Discovery Server distinti.

Meccanismi di sicurezza: infine, sono stati adottati i seguenti meccanismi di sicurezza:

- **Encryption**: i messaggi scambiati tra il front-end e il servizio Registration vengono cifrati per ottenere la confidenzialità dei dati. Effettivamente, le informazioni che tali microservizi si scambiano sono le più sensibili del sistema, poiché tra queste figurano le password e il numero delle carte di credito degli utenti. Poiché i dati vengono anche memorizzati nella base di dati per conseguire la persistenza, per evitare che eventuali attacchi al database risultino eccessivamente devastanti, viene cifrato anche il contenuto della tabella associata alle informazioni degli utenti che, come già detto, sono le informazioni più sensibili.
- **Autenticazione dell'utente**: come già accennato in precedenza, l'utente, per poter usufruire dell'applicazione, ha bisogno di registrarsi al sistema. Tuttavia, è stato necessario adottare dei particolari meccanismi aggiuntivi che non consentano all'utente di bypassare il login / l'iscrizione giungendo direttamente a una qualunque schermata dell'applicazione semplicemente digitando il relativo URL sul browser.

II. Implementazione realizzata

A. Registration

Nel momento in cui l'utente deve registrarsi per la prima volta all'interno del sistema, deve compilare un form con lo username, la password scelta, la tipologia di utente (turista o compagnia aerea), l'eventuale compagnia aerea per conto della quale si vuole loggare e l'eventuale numero della carta di credito (utile nel caso in cui l'utente sia un turista). Una volta immessi questi dati, il microservizio front-end si occupa di inviarli a Registration mediante chiamata RPC, cosicché Registration possa memorizzarli all'interno della tabella del database che tiene traccia di tutti gli utenti iscritti al sistema.

Nel momento in cui l'utente deve accedere al sistema le volte successive alla prima, deve semplicemente inserire username e password. Dopodiché il front-end li invia a Registration mediante chiamata RPC, in modo tale che Registration confronti le credenziali immesse dall'utente con le informazioni memorizzate nel database: se corrispondono allora il login ha successo, altrimenti Registration dovrà comunicare (sempre tramite RPC) al front-end che si è verificato un errore, cosicché il front-end potrà restituire al client una pagina di errore.

B. Booking

Il caso d'uso proprio del turista è l'acquisto dei biglietti aerei. In particolare, il turista deve inserire la data del volo che vuole prenotare, l'aeroporto di partenza e l'aeroporto di arrivo. Dopodiché, il front-end invia i dati a Booking, il quale va a recuperare tutti i voli che matchano con le informazioni immesse dall'utente e li restituisce a sua volta al front-end. A questo punto, il turista deve selezionare il volo che preferisce e deve comunicare al sistema se vuole scegliere i posti a sedere (ed eventualmente quali) e di quali servizi aggiuntivi vuole usufruire. Se non vuole scegliere i posti a sedere e i servizi aggiuntivi, deve solo limitarsi a selezionare il numero di biglietti da acquistare (chiaramente è possibile acquistare più biglietti insieme per agevolare la prenotazione dei voli per gruppi di persone). In caso contrario, all'utente devono essere mostrati i prezzi relativi ai posti a sedere e ai servizi extra: perciò, il microservizio Booking, affinché possa restituire al front-end questi prezzi, deve contattare a sua volta Flights Management mediante chiamata RPC, poiché le informazioni sono memorizzate all'interno delle tabelle detenute da Flights Management. Ricapitolando, il flusso logico completo per ottenere e mostrare all'utente i prezzi dei posti a sedere e i servizi extra è il seguente: il front-end invia una query a Booking poiché è quest'ultimo che si occupa del caso d'uso "acquisto biglietti"; Booking, a sua volta, per poter rispondere al front-end, invia una query a Flights Management; quest'ultimo va a leggere i prezzi sul database e li restituisce a Booking; infine, Booking invia i prezzi al front-end in modo tale che, in ultima istanza, vengano comunicati all'utente.

C. Payment

Dopo aver immesso tutte le informazioni necessarie per prenotare il volo, l'utente vedrà una schermata con i dettagli di pagamento dei biglietti. Da qui può decidere di confermare il pagamento. In tal caso, il front-end deve contattare il microservizio Payment per inviargli tutti i dettagli relativi al pagamento e alla prenotazione effettuata. A tal punto interviene il pattern saga dei microservizi: è necessario definire una transazione con semantica all-or-nothing che effettui un'operazione di store sia delle informazioni strettamente relative al pagamento effettuato dall'utente (e.g.

subtotale del pagamento), sia delle informazioni strettamente correlate al volo che è stato prenotato (e.g. quali posti a sedere sono stati occupati dalla prenotazione). Di conseguenza, è necessario che Payment e Booking si coordinino per portare a termine la transazione.

D. Suggestions

Il microservizio Booking, tramite RPC, invia al microservizio Suggestions i prezzi dei voli che si sono tenuti in passato. In particolare, per ciascun volo, si tiene traccia dell'intero storico dei prezzi al variare del tempo (più precisamente un valore per ogni giorno in cui era potenzialmente possibile prenotare quel volo). Tali dati concorreranno poi a formare il training set per un algoritmo di machine learning che ha come scopo predire quanti giorni prima di un volo può risultare conveniente effettuare la prenotazione dal punto di vista del prezzo (infatti, sappiamo che i prezzi dei biglietti aerei variano giorno dopo giorno). In particolare, per ogni volo V, si calcola la media aritmetica M di tutti i prezzi di V in funzione del tempo; dopodiché, per ogni volo V e per ogni data D in cui era possibile prenotare V, si calcola la differenza Δ tra la data del volo e D; inoltre, per ogni V e per ogni Δ, si effettua il confronto tra il prezzo P del volo V nella relativa data D e il prezzo medio M: se $P < M$, allora si assume che sarebbe convenuto acquistare i biglietti per il volo V a Δ giorni dal volo, altrimenti no. Questa informazione binaria (yes/no) sarà proprio l'attributo da predire nell'algoritmo di machine learning. Poiché i classificatori non sono in grado di gestire le stringhe che possono assumere un valore qualsiasi, all'interno del training set non si ha un attributo relativo all'identificatore del volo, bensì un attributo relativo alla compagnia aerea (che di fatto può essere Ryanair, EasyJet o ITA, ovvero può assumere un valore fra tre possibili). In definitiva, il training set è definito dalla tabella Table 1 riportata di seguito.

TABLE I. TRAINING SET

#giorni rimanenti al volo (Δ)	**Compagnia aerea**	**Conveniente**
		{yes, no}

Nel momento in cui è stato selezionato il volo da prenotare, prima di procedere col pagamento, se richiesto dall'utente, il front-end contatta anche Suggestions per ottenere dei suggerimenti su quando convenga procedere effettivamente col pagamento. Perciò, il front-end invia a Suggestions le informazioni relative al volo selezionato (i.e. la compagnia aerea e il numero Δ di giorni rimanenti alla data del volo). A tal punto Suggestions costruisce un testing set come quello riportato nella tabella Table 2.

TABLE II. TESTING SET

#giorni rimanenti al volo	**Compagnia aerea (fixed)**	**Conveniente (value to predict)**
Δ	Compagnia aerea X	{yes, no}
Δ-1	Compagnia aerea X	{yes, no}
…	…	…
2	Compagnia aerea X	{yes, no}
1	Compagnia aerea X	{yes, no}

Per generare i valori booleani da predire per le istanze del testing set, si ricorre a un particolare classificatore già esistente: Random Forest. Dopo che il classificatore ha generato tutte le istanze col valore predetto, si considera quella col numero di giorni rimanenti al volo maggiore tra tutte quelle che hanno "Conveniente" = yes. Da tale numero di giorni rimanenti si può ottenere facilmente la corrispettiva data, e questa sarà esattamente la data consigliata per effettuare la prenotazione. Di conseguenza, in ultima battuta, Suggestions restituisce al front-end tale data in modo tale che all'utente appaia un messaggio indicante appunto la data in cui è suggerita la prenotazione. Se però nel testing set non c'è alcuna istanza con "Conveniente" = yes, viene selezionata la data odierna come la più conveniente per prenotare, sia perché così viene massimizzata la probabilità di trovare il numero di posti liberi desiderato, sia perché è più verosimile avere un andamento del prezzo dei biglietti tendenzialmente crescente nel tempo piuttosto che viceversa.

E. Flights Management

I casi d'uso propri della compagnia aerea sono l'aggiunta di un nuovo volo, l'aggiornamento del prezzo di un volo attualmente disponibile, l'aggiornamento dei prezzi per la selezione dei posti a sedere e l'aggiornamento dei prezzi dei servizi aggiuntivi.

Per quanto riguarda i primi due casi d'uso, che riguardano direttamente i voli, l'utente deve immettere le informazioni necessarie (id del volo, data, aeroporto di partenza, aeroporto di arrivo, orario di partenza, orario di arrivo e prezzo base nel caso di aggiunta di un nuovo volo, semplicemente id del volo e nuovo prezzo base altrimenti). Dopodiché, il front-end manda tramite chiamata RPC tali informazioni a Flights Management, poiché è il microservizio che si occupa dei casi d'uso propri delle compagnie aeree. Tuttavia, le informazioni sui voli non sono memorizzate all'interno del database relativo a Flights Management bensì all'interno del database relativo a Booking; per questa ragione, Flights Management dovrà trasferire a sua volta i dati immessi in input verso il microservizio Booking, ancora tramite chiamata RPC. Solo allora Booking sarà in grado di finalizzare la memorizzazione dei dati all'interno del database.

Per quanto riguarda invece l'aggiornamento dei prezzi per la selezione dei posti a sedere e l'aggiornamento dei prezzi dei servizi aggiuntivi, la compagnia aerea deve semplicemente compilare un form con i nuovi prezzi da immettere nel sistema. Dopodiché, questi prezzi vengono presi a carico dal front-end che li invia a Flights Management mediante chiamata RPC. Infine, Flights Management effettua lo store dei prezzi aggiornati.

F. Log aggregation

Inizialmente, il server di log centralizzato crea il proprio file di log per registrare gli eventi che verranno generati. Successivamente, tramite le due repliche di Discovery Server, recupera le porte su cui i microservizi dell'applicazione che devono essere monitorati sono in ascolto. Una volta recuperate le porte su cui i microservizi sono in ascolto, il server di log stabilisce le connessioni con tali microservizi in modo da poter poi richiedere i dati da inserire all'interno dei file di log. Una volta stabilite le connessioni con i microservizi, il server crea un file di log per ognuno di questi microservizi in modo da potervi scrivere le informazioni di logging. A questo punto, il server inizia un ciclo infinito in cui richiede periodicamente le informazioni di logging ai

microservizi dell'applicazione per poi scriverle all'interno dei loro rispettivi file di log. Nel momento in cui un microservizio riceve la richiesta delle informazioni di logging dal server, invia al server le informazioni presenti all'interno del proprio file di log che non sono ancora state inviate. Nel momento in cui il contenuto da inviare ha una dimensione elevata, il microservizio lo suddivide in chunk che verranno poi inviati al server. Il server di log va a scrivere nell'ordine corretto i dati che riceve andandoli ad appendere all'interno del file di log relativo al microservizio da cui sta ricevendo le informazioni.

Nell'implementazione è stato affrontato anche il problema relativo all'overflow. Infatti, viene utilizzata una variabile che memorizza al suo interno il numero di byte che sono stati inviati dal microservizio verso il server di log. Grazie a questa variabile è possibile inviare al server solamente le nuove informazioni poiché, di fatto, si tiene traccia dei dati che sono stati già inviati.

G. *Discovery Service*

All'avvio dell'applicazione, ogni microservizio è a conoscenza solamente della porta su cui lui stesso è in ascolto. Tuttavia, la logica dell'applicazione prevede la comunicazione sincrona o asincrona tra i microservizi. Di conseguenza, un microservizio qualsiasi, che necessita di comunicare con altri microservizi per implementare nell'insieme uno specifico caso d'uso, deve recuperare le informazioni relative alla porta e all'indirizzo IP per poter contattare tali microservizi target. Per quanto riguarda l'indirizzo IP del container su cui ciascun microservizio è in esecuzione, è possibile sfruttare il nome del microservizio all'interno della rete creata da Docker Compose. Per il recupero del numero di porta dei microservizi, è stato implementato un'architettura di discovery distribuita. Più precisamente, essendo l'applicazione di piccole dimensioni, abbiamo utilizzato due Discovery Server distinti che espongono ai microservizi un'interfaccia definita da:

- **GET**: permette a un microservizio di recuperare la porta relativa a un altro specifico microservizio.
- **PUT**: permette a un microservizio di registrare la porta su cui esso offre il servizio.

I microservizi dell'applicazione sono stati suddivisi in due gruppi differenti in modo tale da inviare le richieste di GET e di PUT a uno specifico Discovery Server. Più precisamente:

- La prima replica del Discovery Server risulta essere la replica di default per i microservizi Booking, Management e Suggestions.
- La seconda replica del Discovery Server risulta essere la replica di default per i microservizi Payment e Registration.

Per diffondere le informazioni relative ai microservizi, le due repliche di Discovery Server si scambiano periodicamente le informazioni. Esse utilizzano delle cache per memorizzare le informazioni ricevute dai microservizi. In questo modo, si evita un numero eccessivo di richieste al servizio DynamoDB per eseguire le operazioni di GET. Tuttavia, nel momento in cui le due repliche di Discovery si scambiano le informazioni, esse accedono al database su Cloud per recuperare le informazioni aggiornate. Allo startup, il Discovery Server non è a conoscenza di alcuna informazione relativa ai microservizi. Le due repliche apprendono le informazioni nel momento in cui ricevono una richiesta di PUT, che permette ai microservizi dell'applicazione di registrare la propria porta, oppure nel momento in cui si manifesta uno scambio di informazioni non nulle tra le due repliche. I microservizi, quando vengono avviati, registrano la propria porta sul Discovery Server di default. Inizialmente, i microservizi sono a conoscenza di un solo Discovery Server. Tuttavia, nel momento in cui registrano la propria porta, ottengono l'informazione relativa anche all'altra replica di Discovery Server. In questo modo, nel caso in cui la replica di default non dovesse rispondere alle richieste del microservizio, quest'ultimo tenta di contattare l'altra replica di Discovery Server.

III. LIMITAZIONI RISCONTRATE

A. *Funzionalità "coming soon"*

A causa della quantità limitata di tempo a disposizione per lo sviluppo di Fly on the Cloud, sono state implementate solo le funzionalità fondamentali che rendono l'applicazione effettivamente utilizzabile. Di seguito è riportato un elenco di funzionalità che possono essere potenzialmente inserite in ipotetiche release future dell'applicazione.

- **Cancellazione della prenotazione di un volo**: molto spesso le compagnie aeree offrono la possibilità di annullare una prenotazione e di rimborsare i relativi biglietti. Questa funzionalità non è attualmente presente in Fly on the Cloud ma può essere aggiunta definendo nel pattern saga la cancellazione di un'istanza dalla tabella Pagamento e l'aggiornamento di un'istanza della tabella PostiOccupati e di un'istanza della tabella Volo.
- **Modifica di ulteriori informazioni di un volo**: nell'implementazione attuale di Fly on the Cloud, ciascuna compagnia aerea, quando avvia il caso d'uso associato alla modifica di un volo, può aggiornare soltanto il prezzo del volo. Perciò, può essere una buona idea introdurre la possibilità di modificare anche altre informazioni relative al volo, come ad esempio l'orario di partenza e di arrivo, in modo tale da mantenere i dati dell'applicazione correttamente aggiornati anche a seguito di eventuali cambi di programma.
- **Possibilità di leggere le informazioni relative ai pagamenti**: è possibile permettere alle compagnie aeree di visualizzare i dettagli di pagamento di una certa prenotazione, in modo tale da conoscere quali posti a sedere devono essere assegnati a ciascuna persona e quali sono gli eventuali servizi aggiuntivi selezionati da ogni utente.
- **CHAP e integrità dei messaggi**: oltre ai meccanismi di sicurezza descritti nella sezione "Descrizione delle scelte progettuali" del presente documento, è possibile introdurre anche CHAP (per l'autenticazione dei container) e una funzione hash crittografica che permetta di conseguire l'integrità dei messaggi.

B. *Assenza dei Docker volumes*

La replicazione del microservizio Suggestions porta a un vantaggio importante: nel caso in cui una delle repliche (i.e. uno dei container) di Suggestions dovesse essere eliminata, lo stato del microservizio verrebbe comunque conservato dalle repliche rimanenti. Grazie a questa caratteristica, per

semplicità, si è deciso di non utilizzare un volume Docker per mantenere lo stato di Suggestions. Tale scelta lascia comunque spazio alla possibilità di perdere lo stato del microservizio nel caso in cui tutti i container relativi a Suggestions dovessero essere eliminati.

IV. PIATTAFORME E LIBRERIE USATE

A. *Descrizione delle piattaforme software*

- **Linguaggi di programmazione**: i microservizi front-end, Registration, Booking, Management e Payment sono stati scritti in Python, mentre il microservizio Suggestions è stato scritto in Java. Anche i servizi e i meccanismi ausiliari come il discovery e il logging sono stati implementati in Python.
- **Sviluppo del front-end**: il front-end, con la relativa logica di navigazione tra le pagine html dell'applicazione, è stato sviluppato con Flask.
- **Esecuzione del codice Java**: per generare il bytecode relativo al codice Java, si esegue una build con Maven. Questo tool viene utilizzato anche per eseguire il bytecode e, quindi, runnare il microservizio Suggestions.
- **Comunicazione RPC**: per implementare la comunicazione sincrona di RPC, è stato utilizzato il gRPC, che è perfetto quando si vogliono far comunicare più microservizi scritti con linguaggi di programmazione qualsiasi (eventualmente anche eterogenei, come nel nostro caso).
- **Comunicazione asincrona**: per implementare la comunicazione asincrona, sono state usate le code di messaggi, poiché abbiamo a che fare esclusivamente con comunicazioni di tipo one-to-one. Il tool sfruttato per mettere in piedi le code di messaggi è RabbitMQ.
- **Persistenza dei microservizi stateless**: i microservizi stateless non hanno un database implementato al loro interno, bensì fanno riferimento a un servizio esterno per memorizzare le informazioni. In particolare, è stato utilizzato DynamoDB, offerto da Amazon, per definire il database remoto. DynamoDB prevede che le tabelle che costituiscono la base di dati non siano relazionali, bensì formate da delle coppie {chiave, valore}.
- **Persistenza del microservizio stateful**: il microservizio Suggestions è perfetto per essere stateful, poiché il modo più comodo e opportuno per mantenere i suoi dati è mediante un file .arff. Di fatto, le informazioni proprie di tale servizio costituiscono proprio il training set per l'algoritmo di machine learning utilizzato per generare il suggerimento da mostrare all'utente sulla data in cui conviene maggiormente effettuare la prenotazione del volo.
- **Containerizzazione dei microservizi**: per essere in grado di incapsulare ciascun microservizio all'interno di un container, è stato utilizzato Docker.
- **Orchestrazione dei container**: poiché abbiamo deciso di far girare tutti i container Docker in locale, la loro orchestrazione può essere effettuata mediante Docker Compose.

B. *Descrizione delle librerie*

Le librerie importate e utilizzate nell'applicazione sono elencate all'interno della tabella Table 3.

TABLE III. LIBRERIE USATE

Libreria	**Scopo**
Boto3 (Python)	Operare col database remoto di DynamoDB offerto da Amazon.
Crypto (Pycryptodome – Python)	Implementare la cifratura dei dati per conseguire la confidenzialità delle informazioni più sensibili (in particolare quelle relative agli utenti, come le credenziali e il numero della carta di credito).
Flask (Python)	Sviluppare il front-end.
Flask Session (Python)	Supportare Flask per mantenere le informazioni specifiche della sessione di utilizzo dell'applicazione.
gRPC (grpcio e grpcio-tools per Python, io.grpc e protobuf per Java)	Realizzare il meccanismo di Remote Procedure Call (RPC) per la comunicazione sincrona tra i microserivizi.
RabbitMQ (pika - Python)	Realizzare le code di messaggi per la comunicazione asincrona tra i microservizi.
Weka (Java)	Sfruttare un classificatore di machine learning già esistente (i.e. Random Forest) per far sì che il microservizio Suggestions sia in grado di effettuare la predizione su quando possa essere conveniente prenotare ciascun volo.