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—microservizio.

# Architettura e scelte progettuali

## 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.

## Descrizione dell’architettura

Come richiesto dalle specifiche, per lo sviluppo dell’applicazione è stata usata un’architettura a microservizi, dove ciascun microservizio mira a risolvere uno specifico sottoproblema relativo al funzionamento 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 comunica direttamente con tutti gli altri microservizi in modo tale da poter visualizzare a schermo tutte le informazioni necessarie e, in generale, 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 inviare a suggestions il prezzo di tutti i voli che si tengono nella giornata odierna in modo tale che suggestions aggiorni il proprio storico.

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.

## Descrizione delle scelte progettuali

La comunicazione tra i vari microservizi avviene mediante RPC (Remote Procedure Call) oppure mediante coda di messaggi. In particolare si è scelto di inserire una coda di messaggi nell’interazione tra booking e suggestions poiché qui è sufficiente che booking invii (attraverso un meccanismo di push) delle informazioni a suggestions, senza dover ricevere un messaggio di risposta: dunque, in questo contesto, una comunicazione one-to-one asincrona che offra disaccoppiamento temporale è adeguata. Un secondo caso in cui viene inserita una coda di messaggi è la comunicazione 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.

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 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 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.
* 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.

È stato inoltre implementato un meccanismo di discovery che permette a ciascun container di identificare il numero di porta sulla quale un altro container si è messo in ascolto, in modo tale che sia possibile instaurare un’interazione tra i container e, quindi, tra i microservizi. Tale meccanismo di discovery è stato implementato in modo decentralizzato, con due discovery server.

Infine, sono stati adottati i seguenti meccanismi di sicurezza:

* **Encryption**: i messaggi scambiati tra i container vengono cifrati per ottenere la confidenzialità dei dati.
* **Autenticazione dei container**: ogni volta che due container devono comunicare tra loro, seguono il protocollo di autenticazione CHAP (Challenge Handshake Authentication Protocol): sia A il container che si deve autenticare e sia B il container autenticatore. Allora, B invia ad A un nonce n, ed A dovrà rispondere con F(pwd, n), dove pwd è la password necessaria per A per autenticarsi, e F() è una funzione hash crittografica. La password è memorizzata in chiaro tra le variabili d’ambiente del container B. Quest’ultimo, per verificare la correttezza delle credenziali immesse da A, dovrà calcolare a sua volta F(real\_pwd, n) (dove real\_pwd è la password corretta memorizzata tra le variabili d’ambiente) e confrontare il valore ottenuto con quello ricevuto dal container A.
* **Autenticazione dell’utente**: come già detto 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.

# Implementazione realizzata

## 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 email, 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 regitration confronti le credenziali immesse dall’utente con le informazioni memorizzate nel database: se corrispondono allora il login ha successo, altrimenti regitration 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.

## 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, l’aeroporto di arrivo e il numero di biglietti da acquistare (chiaramente è possibile acquistare più biglietti insieme per agevolare la prenotazione dei voli per gruppi di persone). 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 che hanno un numero di posti disponibili non minore del numero di biglietti da acquistare, 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. Tuttavia, affinché possa restituire al front-end le informazioni (i.e. i prezzi) relative ai posti a sedere e ai servizi extra, il microservizio booking deve contattare a sua volta flights management mediante chiamata RPC, poiché tali 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 mostrati all’utente tramite una stampa a schermo.

## 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 inviarvi 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 una 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.

## Suggestions

Il microservizio booking, sfruttando una coda di messaggi, 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 D tra la data del volo e D; inoltre, per ogni V e per ogni D, 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 D giorni dal volo, altrimenti no. Questa informazione binaria (yes/no) sarà proprio l’attributo da predire nell’algoritmo di machine learning. Per praticità, all’interno del training set non si ha un attributo relativo all’identificatore del volo, bensì tre attributi relativi rispettivamente alla compagnia aerea, all’aeroporto di partenza e all’aeroporto di destinazione. In definitiva, il training set è definito dalla tabella Table 1 riportata di seguito.

1. Training set

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#giorni rimanenti al volo (D)** | **Compagnia aerea** | **Aeroporto partenza** | **Aeroporto arrivo** | **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, l’aeroporto di partenza, l’aeroporto di arrivo e il numero D di giorni rimanenti alla data del volo). A tal punto suggestions costruisce un testing set come quello riportato nella tabella Table 2.

1. Testing set

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#giorni rimanenti al volo** | **Compagnia aerea (fixed)** | **Aeroporto partenza (fixed)** | **Aeroporto arrivo (fixed)** | **Conveniente (value to predict)** |
| D | Compagnia aerea X | Aeroporto Y | Aeroporto Z | {yes, no} |
| D-1 | Compagnia aerea X | Aeroporto Y | Aeroporto Z | {yes, no} |
| … | … | … | … | … |
| 2 | Compagnia aerea X | Aeroporto Y | Aeroporto Z | {yes, no} |
| 1 | Compagnia aerea X | Aeroporto Y | Aeroporto Z | {yes, no} |

Per generare i valori booleani da predire per le istanze del testing set, si ricorre a un particolare classificatore: 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.

## 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 invece riguarda 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.

# Limitazioni riscontrate

## Database per service

Un design pattern per i microservizi che avevamo inizialmente considerato per il sistema è il **database per service**, che consiste nell’assegnare a ciascun microservizio un proprio database privato, a cui solo lui può accedere: di fatto, è un pattern semplice da realizzare e incentiva la protezione dei dati (le informazioni strettamente correlate a un certo microservizio non possono essere manipolate né tantomeno accedute da altri microservizi). Tuttavia, come descritto nella sezione IV del presente documento, per tutti i servizi stateless è stata utilizzata una tecnologia (DynamoDB) che non si sposa molto bene con l’idea di creare più di un database nell’ambito di un unico sistema software. Per questo motivo, è stato adottato un compromesso, che consiste nell’implementare un rilassamento del pattern database per service: in particolare, all’unico servizio stateful (suggestions) è stato assegnato un data store a parte (che, per praticità, non è un database vero e proprio, bensì un file in formato .arff), mentre agli altri microservizi è stato associato un sottoinsieme di tabelle del database definito in DynamoDB. Tale configurazione, in ogni caso, mira a emulare la logica che c’è dietro al database per service: è per questo motivo che è risultato possibile implementare il pattern saga all’interno del sistema.

# Piattaforme e librerie usate

## 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.
* **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).
* C**omunicazione 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 EC2, per definire il database remoto. DynamoDB prevede che le tabelle che costituiscono la base di dati siano non 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, che è il tool più famoso per la definizione delle immagini dei container e la creazione e l’avvio dei container stessi.
* **Orchestrazione dei container**: poiché abbiamo deciso di far girare tutti i container Docker in locale, la loro orchestrazione può essere effettuata mediante Docker Compose.

## Descrizione delle librerie

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

1. 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 (sia per Python che per Java) | 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. |