Relazione progetto SDCC

Luca Mastrobattista

13 novembre 2024

Indice

1 Descrizione dell'architettura

L'applicazione serverless sfrutta i seguenti servizi *AWS*:

- *AWS-lambda*
- *Amazon S3*
- *Amazon SQS*
- *Amazon RDS*

La *Figura 1* rappresenta graficamente le interazioni tra i servizi utilizzati.

1.1 AWS-Lambda

In accordo con il modello del *serverless computing*, questo servizio è l'astrazione principale per nascondere l'esistenza del server. Ogni funzionalità prevista dal progetto è sviluppata con una funzione differente. In tutto, si hanno 6 funzioni lambda, ognuna definita in un proprio file insieme alle sue funzioni di supporto. Il loro codice si trova all'interno della cartella `src/consumer/init_infrastructure/python_source`.

1.1.1 sign_up

Questa funzione è definita nel file `lambda_reg.py` e implementa il caso d'uso della registrazione. La funzione lambda prende in input un username e una password in chiaro, genera un sale di 256 bit randomicamente e invoca una funzione *HMAC* per ottenere un digest univoco per l'utente: vengono infatti combinati la sua password e il sale appena generato. Tutto questo viene fatto per evitare di memorizzare le password in chiaro nel database, mentre, per quanto riguarda la sicurezza nello scambio dei messaggi, la libreria utilizzata, *Boto3*, crea per default una sessione *TLS*, e si può quindi comunicare le credenziali in chiaro.
La funzione continua comunicando con *Amazon RDS*, invocando la *stored procedure* `sign_up` che prende in input il nome utente, la password e il sale. Se il *Trigger* non solleva eccezioni e la registrazione va quindi a buon fine, viene creata una folder nel bucket dei messaggi chiamata come l'username registrato: qui verranno salvati tutti i messaggi destinati a quell'utente.

1.1.2 log_in

Questa funzione è definita nel file `lambda_log.py`. La funzione riceve le credenziali di accesso in chiaro, cerca il sale relativo all'utente all'interno del database *RDS* tramite la *stored procedure* `get_salt` e genera il digest combinando questo sale con la password in input. Infine verifica che la stringa così ottenuta corrisponda al valore della password relativa all'utente salvata nel database invocando un'altra *stored procedure*: `log_in`.

1.1.3 users_list

Questa funzione, definita nel file `lambda_users_list.py`, sfrutta la *stored procedure* `get_user_list` per recuperare gli username correttamente registrati al servizio. La funzione, in seguito, li memorizza in una lista e la restituisce al client.

1.1.4 send_message

Una delle funzionalità chiavi dell'applicazione è quella di inviare messaggi. Questa funzionalità è offerta dalla funzione lambda `send_message`, definita nel file `lambda_send.py`. Questa funzione si appoggia al servizio *SQS*, in modo che l'invio del messaggio da parte del mittente e la ricezione da parte del destinatario siano disaccoppiati. Un messaggio applicativo, formato dai campi *mittente, destinatario, oggetto e testo* viene ricostruito a partire da un *messaggio SQS* che contiene un *body*, corrispondente al campo *testo*, e una seria di *messageAttributes*, tutti formato stringa, con chiavi *From, To, Object, Folder*. A prima vista sembra che i campi *To* e *Folder* reppresentino la stessa cosa, ma in realtà sono molto diversi: infatti il primo può essere un elenco di username che comparirà nel campo *To* dei messaggi quando un utente deciderà di leggerli, mentre il campo *Folder* è quello che specifica dove il messaggio deve essere salvato. Infatti, per realizzare la comunicazione *1-a-N*, un utente specifica una lista di di destinatari, per ognuno dei quali verrà inviato un messaggio *SQS* diverso dagli altri solo per l'attributo *Folder*. Questo attributo rappresenta quindi il vero destinatario del messaggio fisico, ed è quindi solo un'indicazione per la funzione lambda che specifica in quale cartella del *message-bucket-sdcc-20-21* debba essere salvato, mentre il campo *To* rappresenta tutti gli utenti che riceveranno quel messaggio.
Questa funzione può ricevere nel campo *To* anche destinatari non registrati ma, quando ciò accade, il messaggio viene semplicemente scartato dalla funzione lambda: infatti, prima di memorizzare il messaggio, viene effettuato un controllo sull'esistenza della cartella destinataria. Il messaggio viene salvato al suo interno solo se la cartella esiste, nella forma di file testuale a cui viene applicato un *tag* che lo marca come *nuovo*, cioè non letto.

1.1.5 read_messages

Questa funzione è definita nel file `lambda_read.py` e implementa il caso d'uso della lettura dei messaggi. Può essere invocata per leggere tutti i messaggi o soltanto quelli che non sono ancora stati mai letti. I messaggi vengono ordinati in base al *timestamp* di creazione, in modo da visualizzare per primi i messaggi più recenti. Una volta visualizzato un messaggio, si può decidere se continuare a leggerne altri, se interrompere, se rispondere al solo mittente, se rispondere al mittente e a tutti gli altri destinatari o se eliminare il messaggio corrente. Questo è l'unico modo previsto per eliminare un messaggio: è necessario che sia prima stato visualizzato. La funzione che gestisce la richiesta di risposta ai messaggi è la stessa che gestisce l'invio di un nuovo messaggio: ciò che cambia è il passaggio di parametri alla funzione che la realizza lato client. La funzione che recupera i messaggi dal bucket S3 è la stessa sia nel caso di lettura dei soli messaggi non letti, sia nel caso di lettura di tutti i messaggi; la differenza è che nel primo caso si faranno dei controlli aggiuntivi sul *tag* dei file salvati filtrando solo quelli *nuovi*.
La gestione dei tag dei messaggi è cruciale: si è scelto di aggiornare il *tag* di un messaggio a *non nuovo* solo quando questo messaggio è stato effettivamente visualizzato. Per evitare di invocare troppe volte una eventuale funzione lambda per modificare i tag dei messaggi, si è fatta la seguente scelta progettuale: è stata implementata una nuova classe che gestisce la lista dei messaggi lato client. In particolare, ogni elemento della lista è una sottoclasse della classe `Message`, definita in `producer/functionalities/Message.py`. Tra i vari attributi aggiuntivi della

sottoclasse, c'è la chiave del messaggio, ossia il nome nel bucket S3, e due attributi booleani che tengono traccia se il messaggio viene letto o cancellato. L'idea è quella di memorizzare tutte le richieste di eliminazione o lettura dei messaggi e di propagare i cambiamenti solo al termine del caso d'uso, invocando quindi una opportuna funzione lambda solo una volta. La creazione della nuova classe lista, inoltre, permette di *nascondere* all'utente i messaggi che sono stati eliminati: supponiamo di avere, ad esempio, una lista di 3 elementi `[A, B, C]` e di eliminare il messagio in posizione 1. Richiedendo di nuovo la lettura del messaggio in posizione 1, verrà mostrato il messaggio `C`, anche se la lista continua ad essere `[A, B, C]`.
Al termine del caso d'uso verrà quindi invocata la funzione `delete_and_mark` per propagare i cambiamenti sul bucket. Questa funzione è impostata anche come *handler* per segnali di tipo `SIGINT` per tutta la durata di questo caso d'uso.

1.1.6 delete_and_mark

L'invocazione di questa funzione è asincrona rispetto all'effettiva richiesta dell'utente. Questa ha lo scopo di gestire le eliminazioni e i cambiamenti dello stato dei messaggi di un utente, modificandone il *tag* in *non nuovo* o eliminandolo dal bucket. Riceve in input un dizionario formato da coppie *chiave-valore* in cui la chiave rappresenta l'identificativo del messaggio nel bucket mentre il valore rappresenta l'azione da eseguire su quel messaggio. In particolare, se il valore sarà `mark`, il *tag* del messaggio verrà modificato in *non nuovo*, se invece è `del` il messaggio verrà eliminato. Il vantaggio dell'esecuzione asincrona di questa funzione è una migliore *user experience*, perché non ci sono tempi da attendere dopo ogni lettura o eliminazione, ma soltanto alla fine del caso d'uso. Inoltre, poiché il costo del servizio *AWS lambda* dipende anche dal numero di richieste che le funzioni ricevono, questa soluzione è anche più economica.

1.2 Amazon S3

Per il corretto funzionamento dell'applicazione e per un corretto disaccoppiamento tra *back-end* e *front-end* è stato necessario creare 2 bucket differenti:

- `message-bucket-sdcc-20-21`, che manterrà traccia dei messaggi ricevuti per ogni utente;
- `source-bucket-sdcc-20-21`, che invece ha lo scopo di mantere il codice sorgente per le funzioni lambda.

Non sarebbe stato problematico, in termini funzionali, memorizzare il tutto nello stesso bucket; tuttavia per ogni eventuale aggiornamento al codice sorgente, si andrebbe a lavorare sullo stesso bucket dei messaggi degli utenti, senza separare l'ambiente di sviluppo dai dati che l'applicazione mantiene.

1.3 Amazon SQS

Questo servizio è utilizzato come input per la funzione lambda `send_message`. In questo modo, anche se per un qualche motivo non dovesse essere possibile invocare la funzione lambda instantaneamente, i messaggi inviati resterebbero nella coda per un tempo massimo di 14 giorni. Inoltre, essendo una funzione chiave dell'applicazione, utilizzare questo servizio permette una maggiore scalabilità e un maggiore disaccoppiamento tra l'invio dei messaggi e l'effettiva ricezione.

1.4 Amazon RDS

Questo servizio è utilizzato solo per memorizzare gli utenti correttamente iscritti all'applicazione. Esiste una sola tabella, con 3 colonne e un *trigger* di controllo di tipo *BEFORE_ INSERT* con cui si controlla che l'username che si sta inserendo non esista già. Tutte le operazioni sulla tabella vengono effettuate dalle funzioni lambda tramite invocazione di *stored procedures*, in modo da evitare attacchi di tipo *SQL injection*. Le procedure salvate permettono di:

- recuperare il sale di utente: necessario in fase di login per ottenere il corretto digest della password;
- recuperare la lista degli utenti del sistema: questa operazione serve a realizzare il relativo caso d'uso;
- effettuare il login: si passano username e password criptata per verificare che i dati siano effettivamente corretti;
- registrarsi: si passano username, password e sale per essere inseriti nella tabella. Questo è lo scenario in cui interviene il trigger della tabella.

Se una *stored procedure* viene interrotta, una eccezione viene restituita alla funzione lambda che la gestisce interrompendo la sua esecuzione e comunicando l'errore al client.

2 Limitazioni

Essendo un'applicazione che interagisce molto con l'utente, tutte le invocazioni delle funzioni lambda sono sincrone, ad eccezione della `send_message`: infatti, tutte le altre funzioni lambda restituiscono qualcosa al client. La lettura dei messaggi, inoltre, prevede che i messaggi vengano ordinati in base al timestamp. Per farlo, viene invocata la funzione `list_objects_v2()` della classe `boto3.client`. Questa funzione può recuperare fino a un massimo di 1000 oggetti, quindi se un utente ne conserva di più potrebbe non visualizzare quelli più recenti.
La creazione del database relazione RDS richiede molto tempo, pari circa a 5 minuti.

3 Piattaforma software utilizzata

La creazione dell'infrastruttura sfrutta lo strumento software *Terraform*, con il quale è possibile definire l'infrastruttura come codice. Lo script in cui viene definita l'intera infrastruttura è `src/consumer/init_infrastructure/infrastructure.tf`.
L'applicazione, scritta in Python, sfrutta molte librerie:

- `Boto3` è senza dubbio la più importante: è la librearia con cui avvengono tutte le interazioni tra il client e i servizi AWS utilizzati;
- `mysql.connector`: libreria utilizzata per la connessione al database
- `hashlib, hmac, string, os`: librerie utilizzate per l'*encryption* delle password;

- `json`: per il passaggio di parametri alle funzioni lambda;
- `stdiomask.getpass`: questa funzione serve a prendere in input una password da *command line* senza che sia visibile in chiaro sullo schermo;
- `decouple`: viene utilizzata per recuperare i parametri di configurazione dell'account AWS dal file nascosto `.env` definito durante la configurazione dell'applicazione. Questa libreria è necessaria solo nel caso non si utilizzi il file `/.aws/credentials` per definire le credenziali di accesso ai servizi Amazon.
- `ast`: il valore di ritorno delle funzioni lambda è una stringa che rappresenta un dizionario. La funzione `ast.literal_eval()` permette di convertire una stringa che rappresenta un tipo di dato nel tipo di dato stesso, in questo caso in un dizionario;
- `signal, partial, sys`: per la gestione dell'*handler* dei segnali `SIGINT`. In particolare, `partial` estende il numero di parametri passabili all'*handler*; `sys.exit()` è invece invocata alla fine dell'*handler*, in modo che se viene catturato un `ctrl+c` il comportamento standard è comunque eseguito e il programma termini;
- `PyQt5`: libreria di supporto per la *GUI*;

4 Immagini

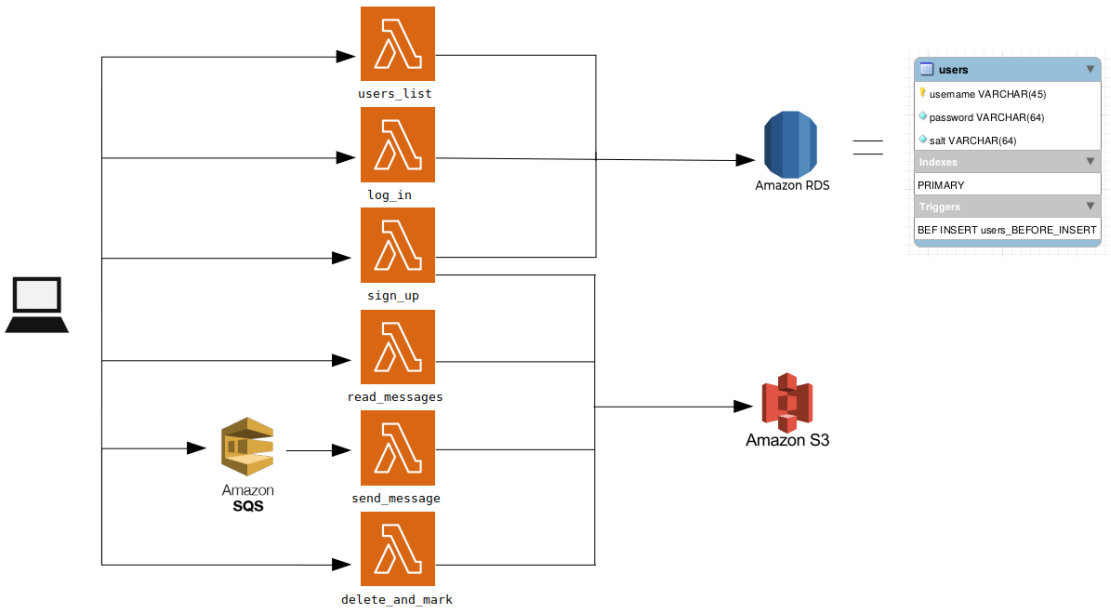

Figura 1: descrizione grafica dei servizi utilizzati.