Musify

A serverless music application

Valerio Moroni: valeriomoroni95@gmail.com

ABSTRACT

Il serverless computing ha ottenuto un ruolo di spicco nel paradigma di progettazione di applicazioni, sia perché si riduce il server management, essendo il server stateless, sia perché è un modello pay-as-you-go. In questo articolo verrà trattato questo paradigma utilizzato in un'applicazione di streaming musicale, descrivendone architettura ed implementazione.

INTRODUZIONE

Parallelamente allo stabilirsi del serverless computing come nuovo paradigma di sviluppo nel cloud, le app di streaming musicale hanno completamente rivoluzionato il mondo della musica, introducendo importantissime novità tra le quali il passaggio sempre più evidente da "copia fisica" a "copia digitale", che sta comportando notevoli benefici (oltre che guadagni) a produttori ed artisti. La facilità d'utilizzo di queste applicazioni, unita alla molto più semplice e rapida diffusione delle produzioni per i facenti parte del settore musicale, mi hanno spinto a cimentarmi nell'implementare una applicazione di streaming musicale in modo serverless che consente, oltre ad avere notevoli vantaggi in termini di costi, di semplificare l'intero processo di sviluppo e di deploy. L'obiettivo del progetto è quello di sviluppare un'applicazione in grado di fornire agli utenti una piattaforma di streaming musicale comoda, pratica ed intuitiva.

Musify è un'applicazione web che permette di riprodurre, scaricare e cercare canzoni tra quelle presenti all'interno di un sistema di storage. L'utilizzo dell'app è favorito dall'intuitività dell'interfaccia grafica, che tenta di riprodurre quella di una tipica piattaforma di streaming digitale. Il lato server è ospitato interamente da Amazon Web Service, configurato utilizzando le risorse customizzabili fornite.

KEYWORDS - *serverless, lambda, dynamoDB, s3, api gateway, aws, iam role.*

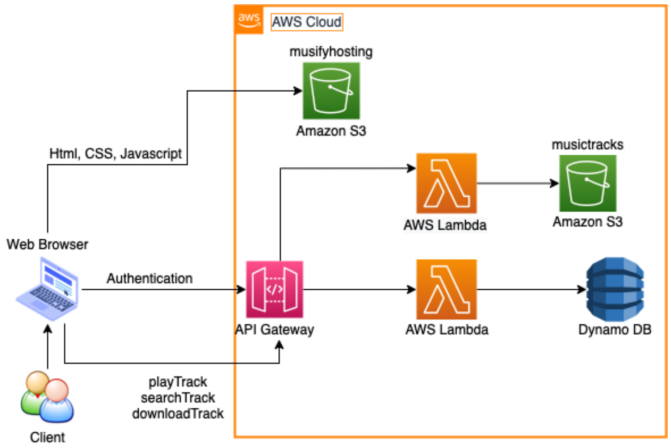
ARCHITETTURA

L'applicazione Musify è composta da un lato client ed un lato server, presentati più avanti nel seguito della trattazione. I linguaggi scelti per implementare l'applicazione sono stati Python e Javascript: Python è stato utilizzato per implementare le funzioni Lambda, cuore pulsante dell'applicazione, mentre Javascript è stato sfruttato per lo sviluppo dell'interfaccia web e per richiamare i servizi AWS lato client. Nelle varie pagine html utilizzate, infatti, oltre al CSS, sono stati richiamati numerosi script e funzioni, triggerati in base all'evento in questione nella rispettiva pagina web.

Il back-end è costituito da una parte totalmente residente sui server appartenenti ad Amazon, in particolare nella regione della Northern Virginia (con codice us-east-1). I servizi AWS utilizzati durante il deployment sono i seguenti:

- *Amazon S3,* per fornire all'applicazione un sistema persistente di storage in cui far risiedere i brani musicali presenti nell'app, oltre che come mezzo di hosting statico dell'intera applicazione. Queste due caratteristiche sono state sviluppate in due bucket S3 distinti.
- *AWS Lambda,* che è il cuore dell'intera logica serverless utilizzata nell'applicazione. Riceve eventi in input, passando attraverso API Gateway, ed effettua la computazione tramite richieste a DynamoDB o ad S3.
- *Amazon DynamoDB,* database non relazionale utilizzato per tener traccia dei dati degli utenti dell'applicazione.
- *Amazon API Gateway,* per creare un punto di accesso diretto al back-end per tutte le funzioni Lambda.

Nella pagina seguente possiamo trovare la rappresentazione dell'architettura dell'applicazione e dei servizi AWS utilizzati nelle fasi di sviluppo.



Continuiamo adesso la trattazione analizzando nel dettaglio ciascuno dei servizi utilizzati. Ciascuno di essi è stato utilizzato tramite la console di gestione fornita da AWS, che si è rivelata essere molto pratica, comoda ed intuitiva.

Amazon S3

S3 rappresenta uno dei principali servizi di storage distribuito di Amazon. Di vitale importanza all'interno dell'applicazione, costituisce il "luogo di residenza" delle tracce musicali e delle varie pagine web che formano l'app.

Come già accennato in precedenza, sono stati necessari due bucket: "*musictracksbucket*", in cui sono state salvate le 20 tracce musicali in formato mp3 utilizzate nel progetto, e *"musifyhostingbucket"*, dove sono state caricate le pagine html, i file CSS e javascript, dato che la sua unica funzione era quella di "hostare" in maniera statica l'applicazione web.

La configurazione di entrambi i bucket è avvenuta tramite console: per quanto riguarda *musictracksbucket*, le impostazioni di configurazione predefinite sono state modificate solo negli aspetti indispensabili al funzionamento dell'app. Affinché le tracce fossero disponibili e riproducibili da parte di qualsiasi utente registrato e loggato nell'applicazione, le autorizzazioni del bucket sono state modificate garantendone l'accesso pubblico (tramite l'apposita spunta presente durante la configurazione nella console di gestione), aggiungendo una policy che potesse consentire la lettura degli oggetti (il GetObject) da parte degli utenti. Non è stata modificata alcuna impostazione riguardante l'Access Control List, ma l'applicazione è stata studiata in modo tale da non permettere l'accesso (e quindi la lettura degli oggetti salvati) agli utenti non presenti all'interno della tabella creata in DynamoDB, che verrà spiegata accuratamente in seguito. A seguire viene riportata la policy di accesso al bucket sopra discussa:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "PublicRead",
      "Effect": "Allow",
      "Principal": "*",
      "Action": [
        "s3:GetObject",
        "s3:GetObjectVersion"
      ],
      "Resource": "arn:aws:s3:::musictracksbucket/*"
    }
  ]
}
```

"*musifyhostingbucket*", invece, ha avuto un ruolo cruciale solo al termine del deployment dell'applicazione: creato e configurato insieme al primo bucket, ospita i file che rappresentano la versione finale dell'app. La configurazione predefinita è stata modificata, anche qui, solo ove necessario: sono state lasciate tutte le impostazioni di default, eccetto la spunta per attivare l'hosting di siti web statici (per ottenere l'endpoint). A livello di autorizzazioni, è stato impostato l'accesso pubblico al bucket ed è stata scritta la policy seguente:

```
{
  "Version": "2012-10-17",
  "Id": "Policy1635151832813",
  "Statement": [
    {
      "Sid": "Stmt1635151821984",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::musifyhostingbucket/*"
    }
  ]
}
```

AWS Lambda

Come già accennato in precedenza, il servizio Lambda costituisce il cuore pulsante dell'applicazione serverless. Sono state implementate 5 funzioni Lambda, tutte quante scritte in Python 3.x, anche se all'inizio dello sviluppo erano 6, ma quest'ultima non avrebbe potuto compiere il suo lavoro a causa di un problema di librerie esterne, che verrà approfondito

più avanti nella trattazione. Vediamo adesso ciascuna funzione, incentrandoci sui dettagli architetturali per poi, in "Implementazione", concentrarci sui dettagli implementativi.

- *autentication_func:* è stata la prima funzione implementata ed entra in gioco nel momento della registrazione. Si occupa di prendere i dati dell'utente in input, ricevuti come evento, e scriverli nella tabella di controllo degli utenti in DynamoDB.
- *loginFunction:* viene triggerata durante il login ed è responsabile di verificare, esaminando le varie voci presenti in DynamoDB, se l'utente in questione (identificato per mezzo di email e password) è presente nel sistema. In caso affermativo ritorna un Json tra i cui campi sono presenti le informazioni dell'utente trovato, in caso contrario ritorna un Json con il campo "Items" vuoto.
- *obtainTracks:* altra funzione fondamentale, viene triggerata al caricamento della pagina principale dell'applicazione ("app.html") una volta effettuato il login. Serve a recuperare tutte le tracce presenti nel bucket S3 e viene utilizzata per costruire dinamicamente la tabella contenente la tracklist.
- *searchByKey:* questa funzione viene triggerata ogniqualvolta si digiti qualcosa all'interno della searchbar e si prema il bottone per effettuare la ricerca. L'input viene "parsato" in Json ed inviato come evento alla funzione. Una volta ricevuto, viene effettuato un "title" dell'input (essendo le key del bucket scritte con le maiuscole iniziali, sia per quanto riguarda l'artista che il nome della canzone) e, di conseguenza, si esaminano in un loop tutte le chiavi del bucket s3 che contengono l'occorrenza cercata. Durante il ciclo, ad ogni match, viene aggiunta la traccia in questione ad un array (inizializzato come vuoto ad inizio funzione) che verrà ritornato dalla funzione a fine loop.
- *downloadTrack:* come si può ben intuire dal nome, questa funzione è responsabile del download in locale delle varie tracce a seguito della pressione del pulsante di download. Viene infatti passato il nome dell'artista e della canzone come evento alla Lambda che, di conseguenza, genera un presigned URL per scaricare il brano in questione e lo ritorna.

Amazon DynamoDB

Amazon DynamoDB è un database NoSQL completamente gestito, serverless, chiave-valore. Ha avuto un ruolo molto importante nell'applicazione, essendo il servizio grazie al quale si tiene traccia degli utenti che usufruiscono dell'applicazione. È stata, infatti, creata una sola tabella denominata "UserTable", configurata con una chiave di partizione ed una di ordinamento. La prima è rappresentata da "emailID", la stringa contenente l'email dell'utente inserita in fase di registrazione, la seconda è invece rappresentata dalla password.

Amazon API Gateway

È un servizio completamente gestito che consente di creare API, nel caso in questione API RESTful. Servono come "porta d'entrata" al back-end e vengono direttamente collegate a ciascuna delle funzioni Lambda, come trigger di queste ultime.
È stata creata una API di nome "DatabaseAPI", di tipo REST con endpoint regionale. È stata poi costruita una fase con 5 risorse, ognuna delle quali andava ad interfacciarsi con la rispettiva funzione Lambda per cui è stata generata. Per ogni risorsa si andava poi ad aggiungere un metodo (o GET o POST) in base alla funzionalità richiesta. Per ciascuno dei metodi, successivamente, si andavano ad abilitare i CORS (Cross Origin Resource Sharing, meccanismo che consente di richiedere risorse limitate su una pagina Web da un altro dominio esterno al dominio da cui è stata servita la prima risorsa).

IMPLEMENTAZIONE

Di seguito vengono riportati aspetti essenziali dell'implementazione non descritti nel paragrafo riguardante l'architettura. Per ciascun servizio utilizzato, infatti, verranno sottolineati gli aspetti più tecnici e di configurazione.

autentication_func: è stata implementata in ambiente Python 3.7 ed, essendo triggerata da API Gateway, tra le autorizzazioni ha automaticamente "lambda: InvokeFunction", come qualsiasi funzione Lambda del progetto. Un altro aspetto fondamentale associato a Lambda è il ruolo IAM, che definisce con quali altri servizi AWS la funzione può interagire. Tramite l'apposita console, dunque, è stato creato un ruolo per consentire alla funzione di interagire con DynamoDB, denominato "*lambda_dyndb_role*". Quest'ultimo contiene varie autorizzazioni, che vengono elencate di seguito: DynamoDBFullAccess (in modo da poter ottenere qualsiasi permesso di interazione con DynamoDB), AWSLambdaBasicExecutionRole (per poter abilitare i log della funzione su CloudWatch) ed

AWSLambdaFullAccess (affinché la funzione abbia tutti i permessi abilitati).

loginFunction: è stata sviluppata in ambiente Python 3.8, ha associato come trigger API Gateway e come ruolo lo stesso della funzione di autenticazione, quindi *lambda_dyndb_role.*

obtainTracks: sviluppata anch'essa in ambiente Python 3.8, triggerata sempre tramite API Gateway, ha associato un nuovo ruolo d'esecuzione, chiamato "*essetrerole"*. Quest'ultimo contiene i seguenti permessi: S3FullAccess (per consentire il pieno accesso al servizio s3), APIGatewayInvokeFullAccess (inserito successivamente per problemi di sicurezza dovuti ai CORS), AWSLambdaBasicExecutionRole ed AWSLambdaFullAccess.

searchByKey: sviluppata in ambiente Python 3.8, stesso trigger delle altre funzioni, ruolo d'esecuzione chiamato "*fullaccesslambda"*, in cui ci sono gli stessi permessi di *essetrerole* meno AWSLambdaBasicExecutionRole.

downloadTrack: sviluppata in ambiente Python 3.7, ha la stessa configurazione delle altre funzioni viste finora, con ruolo d'esecuzione *essetrerole*.

Per creare, configurare e gestire i servizi AWS è stato impiegato "Boto3", che sarebbe l'SDK (Software Development Kit) di AWS per Python, che ha consentito una facile interazione nel codice con gli altri servizi AWS utilizzati. Non è stata necessaria l'installazione dell'SDK perché, scrivendo le funzioni dalla console AWS, era necessaria una semplice importazione ("import boto3").

Veniamo adesso ad i dettagli implementativi di API Gateway, in particolare all'analisi delle varie risorse create.

/gettracks: ha associato un metodo GET e presenta la seguente richiesta d'integrazione: lambda, nella regione us-east-1, in particolare è collegata alla funzione *obtainTracks*. Il timeout utilizzato è quello predefinito e non è stato aggiunto nessun modello di mappatura; la risposta d'integrazione è stata lasciata predefinita, così come la risposta del metodo (in caso di successo lo stato HTTP è 200 e verrà lasciato predefinito anche nelle altre risorse).

/loginreal: ha associato un metodo POST. Per quanto riguarda la richiesta del metodo, è stato inserito un validatore impostato su "convalida parametri di stringa query ed intestazione"; sono stati poi aggiunti, come parametri della stringa di query, "emailID" e "password", necessari per la funzione Lambda di login. Per quanto riguarda la richiesta d'integrazione, è stata associata la Lambda "*loginFunction"*, con timeout predefinito. A modelli di mappatura è stato aggiunto "application/json". La risposta d'integrazione e la risposta del metodo sono state lasciate predefinite.

/searchbykey: ha associato un metodo POST. A richiesta metodo è stato aggiunto il parametro "stringToSearch"; come richiesta d'integrazione è stata collegata la funzione Lambda "*searchByKey"* ed il timeout è stato lasciato predefinito. Tra i modelli di mappatura è stato aggiunto "application/json". La risposta d'integrazione e la risposta del metodo sono state lasciate predefinite.

/**trackdownload:** ha associato un metodo POST. Come parametri di stringa query URL è stata aggiunta la "key", necessaria per la funzione lambda associata alla richiesta d'integrazione, "*downloadTrack"*. Le altre impostazioni sono state lasciate predefinite, come descritto precedentemente per altre risorse.

/userslogin: ha associato un metodo GET. Per quanto riguarda la richiesta del metodo, è stato aggiunto un validatore per convalidare i parametri di stringa query ed intestazioni e sono stati aggiunti dei parametri della stringa di query URL obbligatori: "emailID", "password" e "username". Per quanto riguarda la richiesta d'integrazione, nonostante il nome possa risultare fuorviante, è stata associata la funzione lambda "*autentication_func"*.Il modello di mappatura è "application/json", tutte le impostazioni restanti sono state lasciate predefinite.

Una volta creata l'API, è stato necessario deployarla ed associarla ad una fase, ossia un riferimento logico allo stato del ciclo vitale dell'API, nominabile a piacere. Senza queste operazioni, l'API non sarebbe invocabile e, quindi, non sarebbe possibile utilizzarla.

Front-end

Veniamo adesso alla descrizione di uno degli aspetti più laboriosi dell'applicazione, ovvero il front-end e tutti i dettagli relativi alla user experience.
Una volta sviluppate le funzioni lambda e settato correttamente l'intero ambiente di deployment dalla console AWS, infatti, è stato necessario pensare ad un

front-end che potesse interfacciarsi con la logica sottostante. Una volta trovato il template ed il relativo CSS, è stato necessario modificare numerose componenti. È stata innanzitutto creata la homepage, la pagina "*index.html*", scorrendo la quale è possibile trovare una breve descrizione di ogni funzionalità implementata. In alto a destra nella pagina è possibile trovare il menu, dal quale ci si può registrare o loggare. Cliccando su "Sign up" si viene reindirizzati alla pagina "*signup.html*", dove è stato inserito un form compilabile unitamente ad un bottone per inviare i dati. Associato a quest'ultimo, è stata implementata la prima funzione Javascript per poter effettuare la chiamata Lambda ad *autentication_func*, di nome "*signinUp()"*. Le varie funzioni Javascript implementate sono state raccolte in un file chiamato "*funcCaller.js"*, inserito nella cartella "assets/js". La funzione sopracitata si occupa di prendere i dati inseriti dall'utente ed inviarli tramite una XMLHttpRequest alla funzione lambda, passando per la risorsa di API Gateway "*userslogin*". Una volta inviati i dati, appare un alert per segnalarlo.
Parallelamente alla pagina "signup.html", in caso si cliccasse dal menu su "log in", si verrà reindirizzati ad un'altra pagina, chiamata "login.html". Anche qui è stato inserito un form in cui è possibile accedere con email e password: con la pressione del pulsante di login viene triggerata una nuova funzione Javascript, chiamata "*loggingIn()"*, responsabile di prelevare i dati inseriti, parsarli in un Json ed inviarli tramite metodo POST in una chiamata "ajax", utilizzando "jquery", all'URL di Api Gateway per richiamare la funzione lambda corrispondente associata al login. In caso di esito positivo, viene controllato il Json in risposta e, se il campo "Items" non è vuoto (altrimenti significa che l'utente non è stato trovato e, di conseguenza, viene restituito un errore, mostrato a schermo tramite un alert), viene prelevato lo username ed aperta la pagina "*app.html"*, chiave dell'intera applicazione. All'apertura viene immediatamente chiamata, durante il caricamento della pagina web, la funzione Javascript "*loadTracks()"*, utilizzata per caricare dinamicamente la tracklist ed inserirla in una tabella costruita ad hoc. Questa funzione, infatti, effettua tramite "$.getJSON" la chiamata all'URL tramite cui, passando sempre per API Gateway, si arriva alla funzione Lambda corrispondente. La risposta della funzione viene quindi parsata in un nuovo Json, creato appositamente, dividendo gli artisti dai nomi delle tracce, il quale viene dato in pasto ad un'altra importantissima funzione, incaricata di costruire dinamicamente la tabella: "*buildTable()"*. La responsabilità di quest'ultima è quella di ciclare, per tutta la lunghezza dei dati passatigli, aggiungendo dinamicamente dell'html che andrà a posizionare il nome dell'artista e della canzone nei campi corrispondenti della tabella. Oltre a questo, ad ogni iterazione, vengono anche aggiunti dinamicamente i bottoni "play" e "download" ai quali, sull'"onClick", viene associato rispettivamente il comportamento descritto dalle funzioni Javascript "*playTrack()"* e "*downloadTrack()"*, che verranno descritte successivamente.
Utilizzando la searchbar presente in alto a sinistra, sopra la scritta che riporta lo username con cui si è loggati, è possibile cercare tra le varie canzoni presenti nella tracklist. La ricerca, come già specificato, può avvenire per artista, per titolo canzone o per occorrenza. In caso si cercasse una singola lettera, infatti, verranno riportati tutti i brani che hanno almeno una parte del nome che inizia con quella lettera. La pressione del pulsante "search" triggera una funzione Javascript chiamata "*searchTrack()"*: prende in input la stringa immessa per la ricerca, la inserisce in un Json creato appositamente e poi, tramite una chiamata ajax, viene passata alla funzione lambda corrispondente. La risposta di quest'ultima, una volta che la chiamata è andata a buon fine, viene utilizzata in maniera simile alla *loadTracks()*: viene prima separato l'artista dal nome della canzone in un ciclo, viene inserito il risultato in un Json che viene inviato alla funzione *buildTable()* la quale, dinamicamente, aggiorna i campi mostrati nella tabella con quelli prodotti dal risultato della ricerca.
Procediamo la trattazione con la descrizione di ciò che avviene in caso premessimo uno dei due pulsanti "play" e "download", partendo da quest'ultimo. Al click viene triggerata la funzione Javascript "*downloadTrack()"*, alla quale vengono passati il nome dell'artista e della canzone. Questi ultimi vengono inseriti in un Json creato appositamente ed inviati, tramite una nuova chiamata ajax, alla funzione Lambda incaricata di gestire il download dei brani. La risposta della Lambda contiene l'URL da cui è possibile scaricare direttamente il brano in locale, quindi viene semplicemente aperta una nuova finestra contenente l'URL ed il download inizia automaticamente.
Diverso è, invece, il comportamento che si ottiene con la pressione del pulsante play. Quest'ultimo triggera un'altra funzione Javascript chiamata "*playTrack()"* a cui vengono passati il nome dell'artista e della canzone, che vengono salvati nella memoria locale (*localStorage*). A seguito di tutto ciò, viene aperta una

nuova pagina chiamata "*player.html"*, responsabile della riproduzione dei brani. In questa pagina è contenuta la grafica del player più vari trigger per richiamare funzioni Javascript contenute nel file "*playerLogic.js"*, responsabili del comportamento del riproduttore musicale. All'apertura della pagina vengono azzerati i contatori del player e viene triggerata immediatamente una funzione chiamata "*getTracksWithUrl()"*. A quest'ultima viene affidato il compito di effettuare una chiamata alla funzione Lambda per ottenere la tracklist, come già accadeva in precedenza, ma questa volta i dati ricevuti dalla callback della funzione vengono elaborati in maniera diversa: oltre alla divisione "artista - nome canzone", viene aggiunto un terzo campo che è quello contenente l'URL della traccia, creato dinamicamente, dato che segue un path comune (*"https://musictracksbucket.s3.us-east-1.amazonaws.com/{object_key}*"). Una volta ritornata la tracklist, elaborati i dati come descritto ed inseriti in un Json, viene impostata una variabile globale chiamata "track_list" con i dati del Json e viene fatto partire un ciclo in cui si cerca esattamente l'indice della canzone avente come artista e titolo quelli passati tramite l'onClick sul tasto play; a questo punto viene chiamata la funzione *loadTrack(),* responsabile di azzerare i valori del player e caricare i dati della traccia in questione, nonché impostare il colore del background. Le altre funzioni presenti nel file sono invece responsabili di quello che avviene in caso di interazione con le varie componenti del player, quali i bottoni o le seekbar. Ogni funzione viene triggerata dall'html, tramite eventi "onClick" o "onChange", come per esempio l'interazione con il "seek_slider" lancia la funzione "*seekTo()"*.
Tornando indietro è possibile uscire dal player e riaprire la pagina "app.html", di nuovo. Da qui non resta che parlare di due script Javascript, "embeddati" nell'html, la funzione di logout e quella, semplicissima, che aggiorna la label col nome dello username loggato. Entrambe sfruttano la memoria locale "*localStorage"*, in particolare il logout (il cui pulsante è cliccabile dal menu dopo essersi loggati, ovviamente) la cancella e reindirizza all'homepage, la seconda invece setta una variabile nella pagina di login che viene recuperata nella pagina in questione e trascritta come label.

LIMITAZIONI

Durante lo sviluppo dell'applicazione si sono incontrate varie limitazioni che verranno descritte di seguito. Il primo vero limite, che ha causato molte perdite di tempo, è stato il debugging dell'app. È possibile, infatti, monitorare l'applicazione con la piattaforma CloudWatch di AWS, ma questa deve essere ben configurata con eventi ad hoc. L'IDE messo a disposizione da AWS Lambda non consente un debugging live, ma è necessario sempre creare "alert" all'interno del codice per vedere fin dove arriva il flusso esecutivo e, spesso, si è dovuto procedere a tentativi, scegliendo come opzione corretta non quella concettualmente più giusta, ma quella che realmente produceva il risultato atteso.
Il debugging più complesso è stato quello delle varie chiamate dal front-end alle funzioni Lambda: non disponendo dei log propri di API Gateway per limitazioni interne dovute all'account con AWS Educate, si è dovuti ricorrere all'esecuzione controllata delle varie fasi del ciclo di vita dell'app, riempiendo la console del browser con log che indicassero in quale funzione ci si trovasse e, soprattutto, vedere le varie risorse di rete per capire se ci fosse scambio di dati da parte delle varie funzioni Lambda.
Un altro aspetto cruciale è stato gestire i CORS. Come già accennato in precedenza, questi ultimi sono usati per determinare se la risorsa può essere accessibile o meno dal contenuto che opera all'interno dell'origine corrente: in caso negativo, la richiesta di accesso alla risorsa viene bloccata. Ogni richiesta cross-dominio, di conseguenza, veniva inizialmente bloccata e bisognava (questo per ogni singola chiamata alle funzioni Lambda) capire in che modo effettuare la richiesta, aggiungendo header che consentissero le chiamate "cross-origin" (a volte senza ottenere l'effetto sperato e, quindi, costringendo al cambio drastico nella modalità d'invocazione della Lambda e/o nel metodo http utilizzato tramite API Gateway).
Un altro importante aspetto è stato, infine, quello riguardante la gestione della sesta funzione Lambda, che avrebbe dovuto rappresentare la riproduzione del brano da S3. Essendo possibile tramite librerie esterne in locale, non è stato fattibile implementare il riproduttore (o quantomeno la funzione "play") direttamente da Lambda perché, per importare le librerie necessarie, sarebbe stato necessario effettuare innumerevoli passaggi aggiuntivi che avrebbero complicato inutilmente lo sviluppo.

ALTRI SOFTWARE O LIBRERIE UTILIZZATI

Per quanto riguarda l'utilizzo di altri software durante il deployment del progetto, ci si è serviti esclusivamente di Insomnia, un API client open source, per testare le chiamate REST direttamente e verificare se fossero funzionanti non solo tramite gli eventi di test interni ad AWS, propri della console Lambda.
Oltre all'utilizzo di questo software, ci si è appoggiati anche al download di un template html free che fosse utilizzabile e modificabile per il progetto: il tutto è stato possibile grazie al sito HTML5 UP, riportato nei riferimenti.
L'implementazione delle pagine HTML, del CSS e delle funzioni Javascript è stata effettuata su Visual Studio Code.
Il resto è stato implementato tutto in ambiente interno AWS, quindi non sono state necessarie altre librerie aggiuntive oltre a JQuery dalle Google APIs, per effettuare chiamate alle funzioni Lambda da codice Javascript.

SVILUPPI FUTURI

Come già scritto all'interno dell'applicazione, l'app si presenta molto incline ad un ingente numero di sviluppi futuri e miglioramenti. Primo fra tutti, l'aumento del numero di brani presenti all'interno del database, essendo 20 tracce un numero a malapena d'esempio per un'app musicale. In secondo luogo, sarebbe possibile dividere le canzoni per generi, per artista, per album (inserendo ed associando metadati a ciascun brano), dare la possibilità ad un utente di creare playlist interne all'app e creare un algoritmo di apprendimento automatico dei gusti dell'utente (magari sfruttando anche il machine learning) in modo da suggerirgli nuove canzoni da poter ascoltare e scoprire.
Un altro aspetto migliorabile è la sicurezza: un'applicazione di streaming musicale deve poter contare su un'autenticazione a due fattori, su un login più solido e robusto, su una modalità di accesso alle canzoni più valido e non tramite un bucket S3 pubblico.
Ultimo, ma non per importanza, i tempi di cold-start, problema generale di Lambda: ci sono approcci warm-up che consistono nel mantenere una funzione Lambda sempre attiva che fa da ping ad altre Lambda che devono essere tenute sempre pronte, in modo tale da non consentire all'architettura di chiudere e riavviare ogni volta i container usati per eseguirle.

REFERENCES

[1] www.aws.amazon.com.
[2] https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html
[3] https://docs.aws.amazon.com/s3/index.html
[4] https://docs.aws.amazon.com/lambda/index.html
[5] https://docs.aws.amazon.com/dynamodb/
[6] https://html5up.net
[7] https://medium.com/@shreyakatuwal/serverless-web-application-using-s3-dynamodb-api-gateway-and-aws-lambda-32a10b1a9d5e
[8] https://www.youtube.com/watch?v=XmdOZ5NSqb8
[9] https://highlandsolutions.com/blog/hands-on-examples-for-working-with-dynamodb-boto3-and-python