Audio fingerprinting in WebAssembly per l'esecuzione in browser web

Davide Pisanò - Antonio Servetti

Introduzione

Negli ultimi anni si è notato un trend sempre crescente nell'utilizzo di JavaScript per la creazione di applicazioni desktop.

Il motivo principale dietro alla popolarità di questo ecosistema basato su JavaScript è la possibilità di utilizzare un'unica codebase che può essere eseguita su piattaforme molto diverse tra loro, problematica che è molto sentita nell'ambito mobile dove si hanno due piattaforme completamente diverse: Android e iOS.

Il trend di scrivere applicazioni in JavaScript è stato amplificato dalla crescente importanza del web come piattaforma per la distribuzione di applicazioni software. Software utilizzati quotidianamente da miliardi di utenti sono basati sul web e, per forza di cose, devono essere scritti in JavaScript.

Da qui la nascita delle cosiddette *Rich Internet Applications* (RIA), ovvero applicazioni web che offrono un'esperienza utente interattiva e avanzata simile a quella di un'applicazione desktop tradizionale. Oltre alla classica triade HTML + CSS + JavaScript le RIA possono fare uso di tecnologie più avanzate e recenti come WebAudio e WebAssembly. In sostanza il browser diventa un'interfaccia o un'astrazione della macchina sottostante, alla quale si può accedere utilizzando JavaScript.

Nello specifico WebAudio è un'API JavaScript avanzata che consente di manipolare e generare audio all'interno del browser. È stata progettata per consentire agli sviluppatori di creare RIA che includono funzionalità audio, come la registrazione, la riproduzione e l'elaborazione di suoni.

La necessità di scrivere applicazioni real time ha portato la necessità di dover eseguire codice ad alta efficienza, obiettivo non realizzabile completamente con un linguaggio interpretato quale JavaScript. Alla fine degli anni 2010 nasce quindi *WebAssembly* (Wasm): un formato di codice binario portabile che consente di eseguire codice di basso livello all'interno del browser web.

Sfruttando tutte queste tecnologie e un ecosistema ormai maturo, l'obiettivo della tesi è quello di discutere la realizzazione di un sistema per l'identificazione di audio: un utente sottopone uno spezzone di un brano audio di pochi secondi al sistema, il quale risponde col nome di quel brano. L'obiettivo principale è quello di eseguire l'algoritmo di identificazione su dispositivi eterogenei all'interno di un web browser, utilizzando Wasm e WebAudio.

Verrà discussa anche la realizzazione di una *second-screen application*, ovvero un applicazione o servizio progettato per essere utilizzato su un dispositivo separato, come smartphone o tablet, mentre si fruisce un altro contenuto su un dispositivo di *maggiore importanza*, come la televisione o lo schermo di un PC.

La tesi guiderà il lettore attraverso la realizzazione del sistema di identificazione tra i vari capitoli; saranno esaminate in maniera approfondita le molteplici sfaccettature implementative, corredate da presentazione di dati e grafici, al fine di consentire una comprensione completa delle motivazioni che hanno guidato le scelte dell'autore.

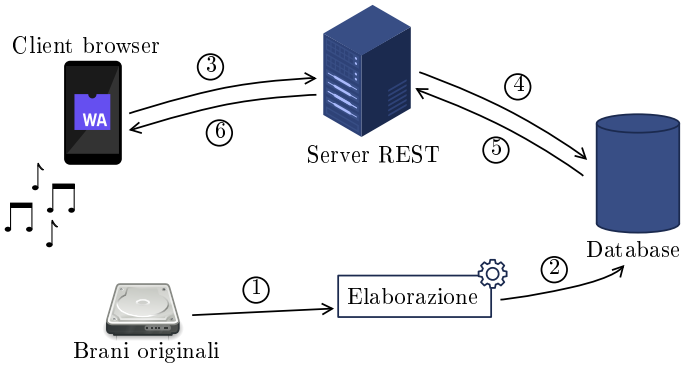

Figura 1: Schema architettura generale

Architettura generale

L'architettura di base del sistema (in figura 1), seppur ispirata al modello client/server, si discosta dalla tradizionale asimmetria tra i due attori, in cui il primo agisce come mero terminale passivo, limitandosi a interagire con l'API del server. La nuova soluzione adottata, invece, si propone di spostare parte della logica di business dal server al client, in un'ottica distribuita che avvicina la computazione all'utente e alleggerisce, al contempo, il carico sul server centrale, riducendo così i costi correlati.

Scomposizione dell'architettura

L'architettura (in figura 1) può essere scomposta come segue:

1. Si inizia dai brani originali, la canzone nella sua interezza, salvata su una memoria di massa. La canzone è sottoposta ad una algoritmo di *fingerprinting*, in cui vengono estratte alcune features caratterizzanti.
2. Le features estratte vengono memorizzate all'interno di un database insieme al nome della canzone alla quale appartengono.
3. Si immagini quindi che, ad un certo punto, un client voglia avviare il processo di riconoscimento di un brano: viene registrato uno spezzone audio di pochi secondi e viene innescata la stessa procedura di *fingerprinting* al punto 1 sul client, ma in questo caso le features estratte vengono inviate ad un endpoint REST.
4. Il server REST cerca di individuare delle similarità tra le features già presenti nel database e quelle appena inviategli dal client.
5. Se la ricerca ha successo, il server REST estrae dal database il nome della corrispondenza migliore.
6. Se la ricerca ha successo, il server REST invia al client il nome della corrispondenza migliore.

Si noti, anzitutto, che la parte più intensiva dal punto di vista computazionale è l'estrazione delle features. In altre parole, il momento di maggior carico computazionale si verifica in due fasi:

- *Lato server*: solo nella fase iniziale che porta al popolamento del database, durante l'analisi dei brani originali (ovvero fasi 1 e 2)
- *Lato client*: nell'estrazione delle features della registrazione del brano da riconoscere

Quindi, l'operazione più onerosa per il server viene eseguita una sola volta: all'atto del fingerprinting dei brani originali. Sarà poi il client a farsi carico dell'operazione di fingerprinting per l'identificazione del singolo brano.

Il server REST ha una duplice funzione:

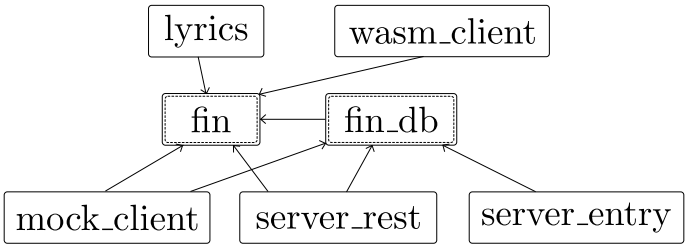

Figura 2: Schema organizzazione codice

- Presentare i dati nel formato corretto sia lato client che lato database, fungendo quindi da una sorta di relay e disaccoppiando la rappresentazione interna dei dati a quella esposta al client
- Individuare similarità con le features già presenti nel database

Organizzazione del codice

Il codice sorgente del sistema è suddiviso nei seguenti componenti (figura 2):

- La libreria *fin*, deputata all'estrazione delle features (ovvero fare *fingerprinting*) dei brani in esame.
- La libreria *fin_db* preposta all'interazione con il database, svolgendo i compiti di inserimento e ricerca delle features.
- L'eseguibile *mock_client*, riservato esclusivamente ad attività di testing, il quale riceve in input un segmento noto di un brano, al fine di verificare la corretta identificazione del brano stesso.
- L'eseguibile *server_entry*, in grado di elaborare i brani completi per estrarne le features, per poi memorizzarle nel database insieme al nome del brano associato.
- L'eseguibile *server_rest* che espone l'endpoint REST per l'individuazione dei brani: riceve le features del segmento audio estratte dal client, effettua una ricerca di un brano compatibile all'interno del database e, in caso di esito positivo, restituisce al client il nome del brano individuato.
- L'eseguibile[1] *wasm_client*, ovvero il client, in grado di acquisire il segmento audio tramite microfono del client, estraendone le features per poi inviarle a *server_rest*.
- L'eseguibile[2] *lyrics* è la second-screen application, costruita sulla base di wasm_client, aggiunge la possibilità di visualizzare il testo sincronizzato (in tempo reale) del brano riconosciuto.

[1] In realtà l'eseguibile è la RIA, contenente HTML, il modulo Wasm e il codice JavaScript necessario al caricamento del modulo Wasm

[2] Anche in questo caso si ha a che fare con una RIA