POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Voice Recognition su Personal Assistant per sistemi Automotive

Amazon Alexa

Relatore
Prof. Maurizio Rebaudengo

Co-Relatore:
Prof. Sandro Cumani

Studente
Edoardo Fortunato
matricola: 228569

Supervisore aziendale
dott. ing. Giacomo Serre, dott. ing. Andrea Cavalieri

Luglio 2021

Sommario

In questa era tecnologica gli assistenti virtuali stanno avendo un ruolo sempre più rilevante, questo perchè hanno permesso di rivoluzionare e semplificare l'interfaccia uomo-macchina, attirando una quantità sempre maggiore di utenti finali, specialmente i meno esperti che hanno sempre visto con scetticismo il mondo tecnologico.
Inoltre i diversi ambiti in cui gli assistenti vocali possono essere utilizzati hanno aumentato l'interesse di numerose aziende, che hanno individuato nella possibilità di integrare questa tecnologia nei loro prodotti un'ulteriore opportunità di guadagno.
In base a tali considerazioni, questo elaborato ha lo scopo di fornire tutti i concetti e gli strumenti fondamentali che permettono di integrare, su qualsiasi dispositivo compatibile, l'assistente vocale Alexa di Amazon e di espanderne le funzionalità.
Verrà infine presentato un prototipo (PoC) con a bordo Android Automotive 10, in cui è presente una versione personalizzata per auto dell'assistente vocale Alexa, che permetterà all'utente di sfruttare tutte le funzionalità tipiche di un infotainment mediante comandi vocali, valutando infine le potenzialità di tale tecnologia e i limiti attuali.

Ringraziamenti

Il candidato ringrazia vivamente il team HMI e Speech di Marelli Europe SPA per i mezzi messi a disposizione e per le utili informazioni che hanno permesso di portare a terminare gli obiettivi prefissati.

Indice

Introduzione

Secondo una ricerca condotta da Ovum, gli assistenti virtuali installati nei vari dispositivi supereranno la popolazione mondiale entro il 2021, raggiungendo una quota complessiva di 7,5 miliardi di assistenti virtuali.
Questa previsione è giustificata dal fatto che gli assistenti vocali hanno trovato utilizzo in ogni tipo di settore come ad esempio la telecomunicazione, la vendita al dettaglio, l'assistenza sanitaria, l'educazione e gli smartphone. In forte aumento è anche l'interesse per l'esperienza interattiva in auto, si stima che quasi tutti gli automobilisti (il 95%) utilizzeranno un assistente vocale entro i prossimi tre anni.
Proprio nel settore automotive, ad aziende leader nelle innovazioni di intelligenza artificiale conversazionale, come Nuance Communications, si sono aggiunti negli anni numerosi competitors. Diversi produttori di auto hanno voluto fare da soli o quasi. Mercedes, cioè Daimler, ha lavorato proprio con Nuance Communications per realizzare il suo assistente vocale. Altre aziende tra cui Google con il suo Google Assistant e Apple con Siri hanno potuto sfruttare un certo vantaggio iniziale, integrando questa intelligenza artificiale nei sistemi di intrattenimento di bordo proprietari. In particolare Android Auto e CarPlay, appertenenti rispettivamente a Google ed Apple, sono ormai da anni molto di¨usi sulle auto e disponibili semplicemente collegando il proprio smartphone all'infotainment. Al contrario Amazon, Microsoft e tutti gli altri concorrenti, che stanno partecipando a questa lotta per il predominio dei propri assistenti vocali nel settore automotive, non hanno potuto sfruttare questo vantaggio iniziale poichè non predispongono di una head unit proprietaria alla pari di Android Auto o CarPlay. Amazon con la Alexa ha provato a salire a bordo dell'abitacolo con il suo Amazon Echo Auto, uno "scatolotto", acquistabile a parte, che si collega mediante bluetooth o AUX all'automobile e impostando tutto il necessario nell'applicazione uÿciale per smartphone oppure tramite pc, è possibile sfruttare tutti i servizi vocali associati ad Alexa stessa. Questo inevitabilmente è una della cause principali per la quale Amazon e gli altri contendenti si sono trovati penalizzati, senza

riuscire ad avere inizialmente quote di mercato significative in questo settore. Tutte queste proposte sono state accolte con soddisfazione generale da parte degli utenti ma allo stesso tempo è stato richiesto di migliorare e rendere ancora più potente tale tecnologia. Infatti sebbene sia possibile per mezzo dei comandi vocali riprodurre musica da servizi di streaming on demand, controllare le indicazioni stradali, e˙ettuare e gestire chiamate, ordinare cibo a domicilio oppure controllare a distanza i dispositivi facenti parte di un certo ecosistema, ci sono diversi limiti da superare.
Innanzitutto per usare Google Assistant o Siri bisogna in ogni caso passare per Android Auto o CarPlay e un cellulare, abbinato al sistema di bordo, con sopra installata un' applicazione opportuna. Questo non permette di avere l'assistente vocale come parte integrante dell'infotainment, ma piuttosto come funzionalità aggiuntiva, estranea al sistema. Nel caso di Alexa, come detto, addirittura l'utente è costretto a comprare un modulo aggiuntivo. Per questo motivo colossi come Google o Amazon hanno reso disponibile gratuitamente agli sviluppatori il software development kit (SDK) del proprio assistente vocale, in modo da poterlo integrare in maniera più profonda. L'utente , una volta salito in auto, potrà quindi sfruttare immediatamente le funzionalità vocali di Google Assistant o Alexa senza l'ausilio di dispositivi o moduli esterni e questa scelta, soprattutto per aziende come Amazon, sta cominciando a pagare. Audi alcuni mesi fa e Bmw più di recente stanno iniziando a integrare le funzionalità software di Alexa all'interno dei loro sistemi di intrattenimento di bordo.
Potenziare questa tecnologia significa anche migliorarla dal punto di vista del Natural-Language Understanding (NLU)[1] e renderla capace di comprendere comandi sempre più complessi per e˙ettuare operazioni sempre più elaborate. Mentre nel primo caso il compito spetta essenzialmente all'azienda proprietaria dell'assistente vocale e consiste nel ra˙orzare l'Engine per le operazioni di text-to-speech (TTS) e speech-to-text(STT), in modo da avere una migliore qualità nell'identificazione dell'argomento trattato nella frase che corrisponderà a una maggiore flessibilità nella pronuncia del comando vocale, nel secondo caso Amazon e Google, ad esempio, oltre a lavorare loro stesse per rendere disponibili nuovi comandi vocali, hanno deciso di o˙rire un'architettura modulare aperta alle terze parti tramite skill sviluppabili esternamente. L'obiettivo di questa proposta è di o˙rire la possibilità di creare un infotainment con un'assistente vocale personalizzato ed estendibile. Avendo a disposizione l'SDK e potendo sviluppare lei stessa o commissionando lo sviluppo di queste skill ai suoi partners, l'azienda automotive sarà in grado di presentare un prodotto unico rispetto agli altri

concorrenti del settore e trarre profitto su quelle skill proprietarie messe a disposizione come contenuti bonus a pagamento. Parallelamente l'azienda proprietaria dell'assistente vocale può raggiungere un ruolo primario nel settore poiché sempre più clienti utilizzano il suo prodotto e i servizi associati.
Per arrivare dunque a integrare un'intelligenza artificiale con un'architettura di questo tipo in un sistema di bordo, sono necessari diversi componenti software e hardware. Nel caso di Alexa, l'assistente vocale scelto per questo lavoro, si inizierà innanzitutto con l'analisi dei dispositivi compatibili con essa già presenti sul mercato e l'architettura generale che permette di esprimere un comando vocale, ricevere una risposta audio o video e creare nuove skill. Questo permetterà di individuare nello specifico quali particolari elementi occorrono per costruire e presentare un primo prototipo completo di tutte le caratteristiche tipiche di un infotainment che integri un assistente vocale come quello presentato da Amazon.

Capitolo 1

Dispositi

Nel novembre 2014 Amazon ha annunciato Alexa insieme ai suoi dispositivi Echo.
Successivamente vari OEM[1] hanno presentato i loro prodotti compatibili con Alexa. E' possibile suddividerli in 2 categorie principali :

- Dispositivi Connessi Alexa
- Dispositivi Alexa Built-In

1.1 Amazon Echo

Amazon Echo è la linea di smart speakers prodotti da Amazon. Ogni tipo di Amazon Echo, sebbene abbia caratteristiche hardware e software di˙eren ti dagli altri, è dotato di almeno un modulo di rete per potersi connettere ai servizi Alexa, un microfono attraverso il quale è possibile iniziare l'interazione vocale tramite la parola "Alexa" e un altoparlante per poter ascoltare la risposta. Entrando più nel dettaglio l'utente potrà chiedere ad Alexa di ascoltare musica in streaming o una webradio tra quelle disponibili, aggiungere attività alla "lista di cose da fare" e farsele notificare quando serve, chiedere informazioni sul traÿco, sul meteo o informazioni in tempo reale e controllare a distanza dispositivi compatibili e registrati nel cloud Alexa.

[1] Original Equipment Manufacturer

(a) Echo Dot

(b) Echo Plus

(c) Echo Show 10"

(d) Echo Show 5"

(e) Echo Spot

(f) Echo Flex

(g) Echo Studio

(h) Echo Auto

Figura 1.1: Amazon Echo Devices

(a) Echo Dot : Questo tipo di dispositivo è l'entry point della famiglia dei dispositivi Echo. Essendo un dispositivo piccolo, ideale per stare ad esempio sulla scrivania e riprodurre musica, è composto essenzialmente da uno speaker non di potenza elevatissima(un speaker da 1,6") e un microfono, oltre ad avere la capacità di connettersi a internet per sfruttare i servizi Alexa, i comandi vocali associati e controllare gli altri dispositivi di casa compatibili connessi al Cloud . Dalla quarta generazione, è integrato un orologio digitale.

(b) Echo Plus : L'Echo Plus si di˙erenzia dalla serie Dot innanzitutto per un Woofer al neodimio da 3" ed un tweeter da 0,8 che rende l'audio ottimo e

profondo e per questo può tranquillamente esser utilizzato come dispositivo principale per sentire la musica in casa. Oltre a possedere un sensore di temperatura, l'aggiunta principale a livello software consiste nell'hub Zigbee integrato che lo rende dunque anche un ottimo gestore di domotica, potendo controllare con la voce i dispositivi intelligenti compatibili con questo sistema tra cui lampadine, prese intelligenti, termostati, etc.

(c) Echo Show 10" : La serie Show è la serie di dispositivi Echo più avanzata dal punto di vista hardware e software. Qui troviamo un display da 10" con capacità touch e inoltre è possibile sfruttare tutte le skill che richiedono risposte con contenuto visivo, oltre che audio. E' possibile utilizzare anche i comandi vocali per avviare videochiamate, essendo fornito di una telecamera da 13MP. Novità dell'ultima generazione è la rotazione automatica dello schermo per seguire l'utente durante la videochiamata. Inoltre proprio per la presenza di un touch reattivo sono presenti delle app integrate tra cui Youtube e il browser firefox tramite le quali è possibile riprodurre video o musica o fare ricerca con i rispettivi comandi vocali. E' stato ulteriormente potenziato con 2 tweeter da 1" (25 mm) e un woofer da 3" (76 mm). Essendoci integrato, anche in questo caso, l'hub Zigbee, è possibile utilizzare questo tipo di dispositivo Alexa come cuore della domotica da porre in ambienti più aperti come il salone di casa.

(d) Echo Show 5" : L'Eco Show con display da 5" è particolarmente adatto come sveglia o radio da porre sul comodino. Possiede degli altoparlanti meno potenti e una telecamera frontale a risoluzione minore(1MP). Qui, rispetto al modello da 10", non c'è l'hub Zigbee integrato ma rimane la possibilità di utilizzare tutte le skill inerenti a contenuti visivi.

(e) Echo Spot : Ormai obsoleto, è stato sostituito dall'Echo Show 5".

(f) Echo Flex : Dispositivo di dimensioni estremamente ridotte, può essere fissato direttamente a un muro, agganciandolo a una presa elettrica. Possiede un altoparlante da 0,6", risultando inferiore addirittura a quello dell'Echo Dot, e per questo non è indicato per riprodurre musica. Nasce quindi con l'unico obiettivo di estendere la portata dei comandi Alexa agli ambienti più angusti come un bagno o un ripostiglio. La caratteristica peculiare di questo prodotto è la presenza di una porta USB che permette di collegarci ulteriori dispositivi di terze parti compatibili con Alexa come una luce notturna ed un sensore di movimento. La cosa

interessante è che una volta collegati i dispositivi verranno immediatamente riconosciuti dalla piattaforma Alexa permettendo di utilizzarli in interessanti routine.

(g) Echo Studio : E' stato progettato per poter o˙rire la migliore qualità sonora tra i dispositivi Echo. Per questo è particolarmente indicato per essere utilizzato come impianto audio connesso alla tv. In un unico cilindro, infatti, si trovano tre altoparlanti midrange da 2", un tweeter da 1" ed un woofer da 5,25". Il resto delle caratteristiche rimane pressochè uguale a quelle di un Amazon Echo Plus.

(h) Echo Auto : Dispositivo pensato per portare Alexa sulle automobili che ne sono sprovviste. Sfrutta la connettività a internet dello smartphone, con il quale dovrà essere abbinato via bluetooth, per sincronizzare i dati dall'app uÿciale Alexa presente sullo telefono e per poter usufruire dei servizi vocali. Non è dotato di altoparlanti perciò sfrutta le casse stereo dell'auto, connettendosi tramite ingresso aux o usb all'infotainment.

1.2 Dispositivi Connessi Alexa

A questa categoria appartengono tutti quei prodotti di terze parti tipici del mondo IoT che gli utenti possono controllare dai dispositivi Alexa. Non possedendo uno speaker e un microfono, ma solo un modulo di rete per connettersi a internet, l'utente, tramite l'applicazione uÿciale Alexa sullo smartphone, dopo essersi loggato, dovrà registrare il device. A questo punto il device entrerà a far parte del gruppo dei dispositivi Alexa associati all'utente e sarà possibile, attraverso uno di questi, purchè sia un Amazon Echo o un dispositivo Alexa Built-In, esprimere un comando vocale per gestirlo a distanza.
Un'ulteriore opzione per gestire questi dispositivi consiste nell'utilizzare anche direttamente l'app uÿciale Alexa per smartphone. Nello specifico è possibile far cambiare il loro stato oppure far sì che svolgano automaticamente delle routine, come ad esempio luci che si accendono, spengono o cambiano colore, interruttori che si attivano o disattivano, termostati dalla temperatura regolabile, telecamere di sorveglianza che iniziano a registrare o solo mostrare l'immagine, serrature che si aprono o chiudono oppure sveglie che impostano automaticamente un allarme.
Per le aziende interessate a creare questo tipo di prodotti, Amazon ha reso disponibili diversi developer tool. Per alcuni seguirà un approfondimento

nei capitoli successivi[2]. Ognuno di essi presenta una di˙eren te opzione di configurazione :

- Smart Home Skill API : l'azienda può creare una skill proprietaria e sfruttare il meccanismo di "account linking" durante il processo di attivazione da parte dell'utente, per registrare il dispositivo nel cloud Alexa. Con questo meccanismo sarà possibile controllare a distanza il prodotto con i comandi vocali corrispondenti.

- Alexa Connect Kit (ACK) : questa opzione è una scelta per quelle aziende che vogliono integrare Alexa a un costo fisso senza passare per lo sviluppo di skill ad hoc e la complessa gestione di WebService con i relativi problemi di sicurezza di rete.
  Il componente principale è un componente hardware system-on-module, chiamato ACK module, da integrare nel dispositivo. Questo modulo fornisce una connessione wifi e bluetooth. Al suo interno si trova il firmware utile a interfacciarsi con i servizi Alexa, a registrarsi nel cloud oltre che a comunicare con il microcontroller unit dell'host(HMCU) tramite UART (universal asynchronous receiver-transmitter) [3].

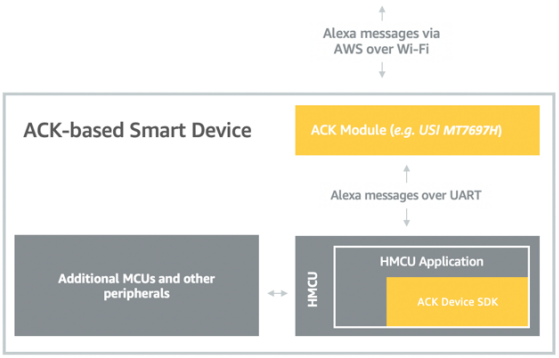


Figura 1.2: Architettura ACK

  L' ACK Device SDK fornisce il codice C sorgente da utilizzare nell'applicazione HMCU del device. In particolare vengono fornite le librerie per comunicare con l'ACK Module, gestire la registrazione, i cambiamenti di stato del device, l'interpretazione delle direttive che arrivano da Alexa e l'invio di nuovi eventi. Collegando delle periferiche sulle porte UART del dispositivo come ad esempio delle ventole o una lampadina, nell'applicazione HMCU sarà possibile in base alle direttive ricevute, comandare questi componenti con il codice specifico. L'utente dovrà soltanto registrare il dispositivo dall'app uÿciale Alexa per smartphone, dopodiché potrà gestire questi dispositivi direttamente dall'app uÿciale stessa oppure esprimendo un comando vocale verso un Amazon Echo o un dispositivo Alexa Built-In.

- Zigbee : è un protocollo di comunicazione wireless che permette di controllare e gestire a distanza i device compatibili[4]. Le aziende interessate possono integrare nei loro dispositivi il software che implementa questo standard, progettato per rendere facile lo sviluppo su microprocessori piccoli e a basso costo. Tra i prodotti che è possibile creare si trovano lampadine, sensori, interruttori, termostati,etc. L'utente sarà in grado di controllarli a distanza solo dopo averli collegati a un hub certificato. Nel caso di Alexa, l'hub Zigbee è già integrato negli Amazon Echo Plus ed Echo Show, quindi è possibile attraverso il comando vocale "Alexa scopri i miei device", scoprire e connettersi con i prodotti Zigbee presenti nelle vicinanze. Superata questa prima fase di configurazione, attraverso i classici comandi vocali di gestione della domotica che ci permettono l'accensione, lo spegnimento e la regolazione, sarà possibile interagire con questi dispositivi connessi.

- Alexa Gadgets Toolkit : è una soluzione particolarmente indicata per giocattoli, bottoni da quiz, orologi intelligenti, etc. Anche in questo caso, l'opzione non prevede uno sviluppo ulteriore di skill e protocolli da parte dell'azienda per poter connettere questi dispositivi ad Alexa, ma con il Tootlkit e il codice sorgente implementativo che ha reso disponibile Amazon, sarà possibile utilizzarli con le funzioni vocali semplicemente abbinandoli via bluetooth a un qualsiasi Echo. La particolarità di questa soluzione è che è possibile estendere le capacità di base del gadget con un'opportuna configurazione nel codice sorgente. Esso infatti, oltre ad essere capace di gestire le direttive standard che arrivano da Alexa, come ad esempio ricevere aggiornamenti dell'ora corrente o sveglie scattate sul dispositivo Echo, notifiche che lo informano di nuovi contenuti disponibili nel cloud Alexa o che il dispositivo Echo ha captato la parola di avvio "Alexa"[5], è in grado di rispondere a direttive custom, per le quali è richiesto lo sviluppo di skill ad hoc. Sarà quindi possibile implementare nuove funzionalità come, ad esempio, suonare dei determinati tasti di una tastiera in seguito a un opportuno comando vocale da parte dell'utente. Questi gadget, oltre a ricevere direttive, possono anche generare eventi, corrispondenti a dei cambiamenti di stato sul dispositivo, da inviare ad Alexa che restituirà in base a questo una risposta vocale mediante il dispositivo Echo abbinato. Anche in questo caso gli eventi sono estendibili e personalizzabili.

1.3 Dispositivi Alexa Built-In

A di¨erenza dei prodotti connessi ad Alexa, i prodotti "Built-In" contengono una versione personalizzata del software che si trova sugli altoparlanti intelligenti Echo e dispongono di un hardware completo per poter sostenere una conversazione adeguata.
Grazie all'Alexa Voice Service(AVS), un kit composto da API, documentazione e strumenti di sviluppo hardware/software, reso disponibile da Amazon, i vari OEM[2] hanno potuto presentare i propri prodotti personalizzati, capaci di ascoltare i comandi vocali dell'utente e reagire alle varie direttive che i servizi cloud Alexa inviano ai prodotti stessi. E' possibile trovare in commercio dispositivi di terzi come TV, speaker, radiosveglie, sistemi di navigazione per le auto e sistemi embedded IoT che dispongono delle stesse funzionalità dei prodotti ufficiali Amazon Echo.
Tra i componenti principali dell'AVS troviamo l'AVS Device SDK, basato principalmente su linguaggio C++. Questo SDK contiene tutte le librerie per integrare rapidamente le funzioni base di Alexa. È modulare e astratto, oltre che altamente personalizzabile, caratteristica chiave che ha permesso agli sviluppatori di poter creare delle versioni ottimizzate per i loro dispositivi. Amazon stessa, ad esempio, per andare incontro alle particolari esigenze dei produttori automotive, ha pubblicato una versione dell'SDK open source di Alexa per le auto che permette di sfruttare mediante la voce tutte le funzionalità tipiche di un infotainment basato su piattaforma Android o Linux e che sarà oggetto di studio nei capitoli successivi.

[2]Original Equipment Manufacturer

Capitolo 2

Alexa Voice Service

2.1 Che cos'è

Per poter arrivare a ottenere un prodotto Alexa "Built-in", Amazon ha messo a disposizione un kit di sviluppo che prende il nome di Alexa Voice Service, composto da API, documentazioni contenenti requisiti e linee guida da soddisfare, hardware di riferimento, estensioni e infine SDKs che riducono notevolmente i tempi di implementazione software necessari a utilizzare in pratica tutte le funzionalità di Alexa.
L'obiettivo principale di Amazon è rendere il processo di integrazione di Alexa libero e accessibile da tutti, oltre che più facile e intuitivo possibile, in modo da essere scelta da un numero sempre maggiore di aziende, interessate a portare questo tipo di AI sui loro prodotti. Ecco dunque che chiunque voglia iniziare questo tipo di operazione, dovrà, in base a quello che vuole ottenere, analizzare necessariamente quanto più dettagliatamente possibile i punti principale di questo kit di sviluppo.

2.2 Requisiti e Funzionalità

Prima di iniziare la progettazione e lo sviluppo è opportuno innanzitutto stabilire quali sono le funzionalità base che tutti i device "Built-In" devono o˙rire. Questo permetterà di individuare i requisiti software e hardware necessari per ottenere un prodotto certificato.

2.2.1 Impostazioni e Autenticazione

Per poter interagire con Alexa è necessario che gli utenti abbiano e˙ettuato la registrazione del proprio Device "Built-In". Dal punto di vista implementativo è obbligatorio dunque ottenere quello che viene chiamato "Login with Amazon (LWA) access token" che sarà inviato in ogni richiesta ad Alexa e permetterà di autorizzare ognuna di essa.
Esistono molteplici modi per autorizzare un prodotto e tutti si basano sull'OAuth, un protocollo di rete aperto e standard, progettato specificamente per lavorare con l'Hypertext Transfer Protocol (HTTP).

- Remote Authorization : usato con devices tipicamente senza display, come smart speaker o soundbar.[6],[7]
- Local Authorization : usato con app Android e iOS[8]
- Code-Based Linking : usato con dispositivi con strumenti di input limitati come televisioni, smartwatch, auto.

Sono 3 metodi simili che di˙eriscono essenzialmente per i tipi di parametri inviati nelle varie richieste POST HTTP. Per evitare prolissità, verrà analizzato il "Code-Based Linking", essendo il metodo utilizzato nel prototipo illustrato nella seconda parte. Per tutti gli altri metodi si può fare riferimento al link in Bibliografia[6][7][8].
Code-Based Linking : il seguente diagramma mostra come ottenere l'*access_token* and il *refresh_token* :

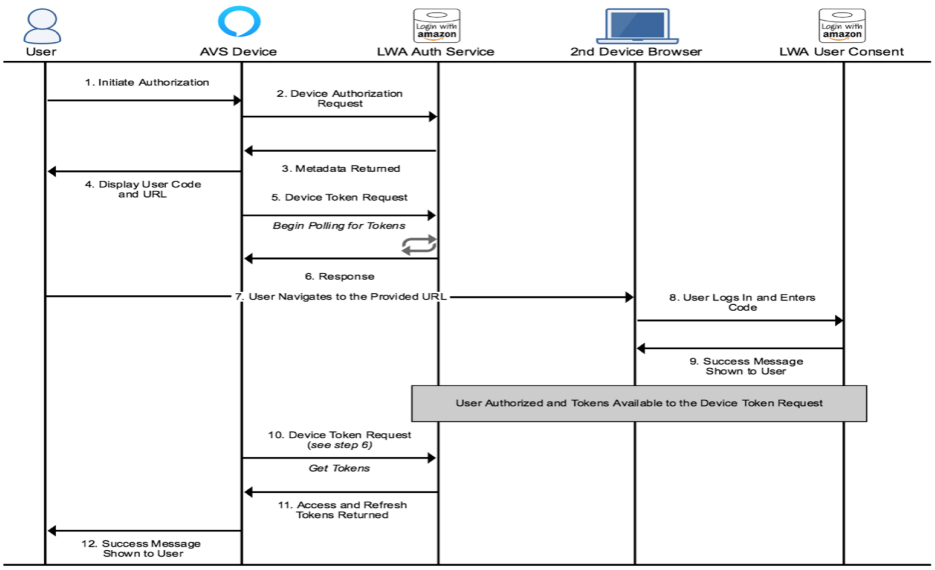


Figura 2.1: Code-Based Linking

(1) Il primo parametro essenziale da generare è il "ClientId" che lo sviluppatore può generare nella sua Alexa Dashboard. Dopo aver creato un prodotto, nella sezione Security Profile->Other devices and platforms, si definisce il "Client ID name" e infine con "Generate ID" si ottiene il parametro desiderato.

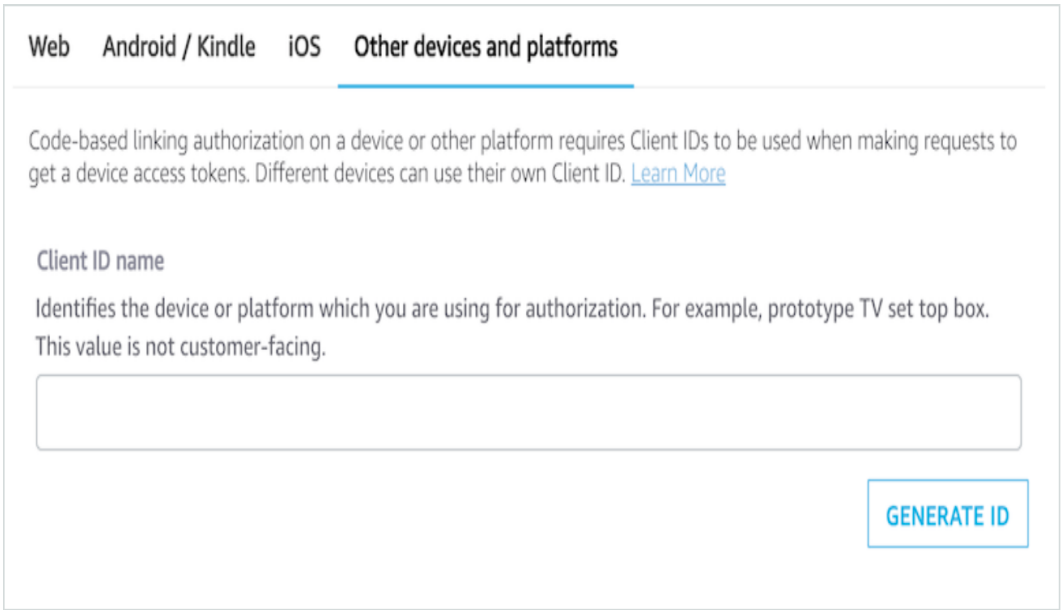


Figura 2.2: Sezione per generare il Client ID

(2) L'utente avrà a disposizione un "Display Button" per avviare la fase di login

(3) AVS device e˙ettua una "Device Authorization Request", una richiesta POST HTTP verso l'endpoint :
"*https://api.amazon.com/auth/O2/create/codepair*" per ottenere lo *user_code* e il *verification_uri*.
Inoltre la richiesta deve includere nel body le seguenti coppie chiave-valore formattate secondo il formato JSON:

- *Response_type* : *device_code*. Indica al servizio LWA che questa è una richiesta CBL
- *clientid* : qui si inserisce il "ClientId" generato al passo 1
- *scope* : *alexa:all* e indica i permessi dati al dispositivo
- *scopedata* : in questo caso a questo parametro corrisponde un oggetto JSON annidato *alexa:all*, nella quale si trovano due coppie chiave-valore:

- *productID* : è il "Device Type ID" che è possibile ricavare nel momento in cui si crea un prodotto nella dashboard Alexa
- *productInstanceAttributes* : è un oggetto JSON contente la coppia:
  - *deviceSerialNumber* : qui è possibile inserire un valore che identificherà univocamente il dispositivo come ad esempio il suo MACAddress

```
{
  "alexa:all": {
    "productID": "{{productId}}",
    "productInstanceAttributes": {
        "deviceSerialNumber": "{{deviceSerialNumber}}"
    }
  }
}
```

(4) Se la richiesta è andata a buon fine la risposta sarà così formatta :

```
{
    "user_code": "{{STRING}}",
    "device_code": "{{STRING}}",
    "verification_uri": "{{STRING}}",
    "expires_in": {{INTEGER}},
    "interval": {{INTEGER}}
}
```

- *user_code* : questo codice va inserito dall'utente al link puntato dal *verification_uri* per completare la fase di login, dunque va mostrato a schermo obbligatoriamente.
- *device_code* : il codice è utilizzato dall'AVS Device per e˙ettuare una *Device Token Request* che consentirà di ottenere l'access token.

E' un parametro molto importante e dunque va tenuto segreto.

- *verification_uri* : l'URL a cui l'utente deve collegarsi per inserire lo *user_code*. Inizialmente a questo URL gli verrà richiesto di effettuare il classico login con le sue credenziali dell'account Amazon e successivamente potrà inserire il codice.
- *expires_in* : il tempo di validità, in secondi, dei codici ricevuti.
- *interval* : l'intervallo, in secondi, tra una *Device Token Request* e l'altra.

(5) E' necessario mostrare all'utente sul display *user_code* e il *verification_uri*

(6) A questo punto l'AVS Device inizia una richiesta ciclica POST HTTP (*Device Token Request*) per ottenere l'*access_token*. Il numero di secondi tra una richiesta e l'altra sarà pari al parametro *interval* e il ciclo terminerà nel momento in cui si otterrà una risposta contenente il dato desiderato.
In particolare la ***Device Token Request*** dovrà essere struttura come segue :

- Endpoint : *https://api.amazon.com/auth/O2/token*
- Header con le seguenti coppie chiave-valore :
  - *Host: api.amazon.com*
  - *Content-Type: application/x-www-form-urlencoded*
- Body con le seguenti coppie chiave-valore :
  - *grant_type* : il valore corrispondente sarà il *device_code*
  - *device_code* : qui si troverà il *Device Authorization Request*
  - *user_code* : qui andrà inserito lo *user_code* precedentemente ottenuto, che l'utente dovrà inserire all'URL *verification_uri*

(7) Durante la fase di richiesta ciclica dell'*access_token* la risposta da parte dell'LWA Auth Service può essere :

- *authorization_pending*
- *slowdown*
- *invalid_code_pairuser*
- *invalid_client*

- *unauthorized_client*

(8) A questo punto lo user, tramite un secondo device, mediante il quale è più facile inserire caratteri di input, naviga verso *verification_uri*

(9) L'utente e˙ettuerà il login con le credenziali del suo account Amazon e procederà quindi a inserire lo *user_code*

Figura 2.3: Schermata per registrare il prodotto

(10) Se lo *user_code* sarà corretto, verrà restituito un messaggio di successo

(11) Dall'altra parte, l'AVS Device, impegnato nel ciclo della *Device Token Request*, riceverà una risposta di successo e dunque terminerà il ciclo.

(12) La risposta avrà la seguente struttura :

```
{
  "access_token": "{{STRING}}",
  "refresh_token": "{{STRING}}",
  "token_type": "bearer",
  "expires_in": {{INTEGER}}
}
```

- *access_token* : il token di accesso per validare ogni richiesta ad Alexa

- *refresh_token* : da presentare nei parametri della richiesta POST HTTP per ottenere un nuovo *access_token*, quando quest'ultimo non è più valido
- *expires_in* : il numero di secondi di validità dell'*access_token*

(13) A questo punto l'AVS Device notificherà alll'utente su schermo che la fase di registrazione è andata a buon fine ed è possibile da quel momento chiedere qualcosa ad Alexa. Quando l'utente spegnerà l'AVS Device non sarà necessario e˙ettuare nuovamente l'intera procedura ma sarà utilizzato il *refresh_token* per ottenere un nuovo *access_token*. La chiamata POST HTTP avrà quindi i seguenti parametri :

- Endpoint : *https://api.amazon.com/auth/O2/token*
- Body con le seguenti coppie chiave-valore :
  - *grant_type : refresh_token*
  - *refresh_token* : il refresh token
  - *client_id* : il ClientId

Se tutto andrà a buon fine si otterrà la risposta con la struttura come al punto (12)

2.2.2 Voice-Initiated e Touch-Initiated Products

Gli AVS Devices si dividono in *Voice-Initiated Products* e *Touch-Initiated Products*.
Per entrambi i tipi è obbligatorio utilizzare suoni predefiniti e avvisi visivi che notifichino l'utente riguardo ai passaggi di stato di Alexa durante la conversazione(*Idle*, *Listening*, *Thinking*, *Speaking*). Inoltre l'utente dovrà essere in grado di interrompere la conversazione in qualsiasi momento con il comando vocale o touch opportuno ma anche poter impostare il microfono nello modalità "privacy", disabilitandolo.
La di˙erenza tra questi due tipi consiste nel metodo utilizzato per iniziare una conversazione con Alexa.
Nei *Touch-Initiated Products* si sfrutta il *"tap-to-talk"* e l' *"hold-to-talk"*. Nel primo caso, e˙ettuando un "tap" su un bottone touch o fisico, si pronuncia direttamente il comando e si attende la risposta da Alexa. Nel secondo caso l'utente dovrà tenere premuto sul bottone, pronunciare il comando e rilasciare il bottone stesso, a quel punto si aspetta la risposta.

Nei *Voice-Initiated Products* invece, prima di pronunciare il comando, è necessario usare quella che viene chiamata *"Wake Word"* e che nei prodotti predefiniti corrisponde alla parola "Alexa". Quindi questo tipo di prodotto dovrà obbligatoriamente supportare la funzionalità "*cloud-based wake word verification*".

Cloud-based Wake Word Verification
Permette di migliorare il riconoscimento della *Wake Word* in modo da ridurre le false identificazioni causate da sorgenti non umane che possono pronunciarla o dalla parole simili a quella di Alexa come "Alexis", "Alex".
Il Device dovrà e˙etturare lo streaming dell'audio catturato dal microfono verso l'AVS(Alexa Voice Service). Il microfono rimarrà aperto fino a quando Alexa non chiuderà lo stream perchè il comando è stato riconosciuto e dunque una risposta è pronta oppure quando si esaurisce il tempo limite massimo di cattura, scadendo di fatto il timeout.
Una possibile realizzazione implementativa è riportata di seguito :

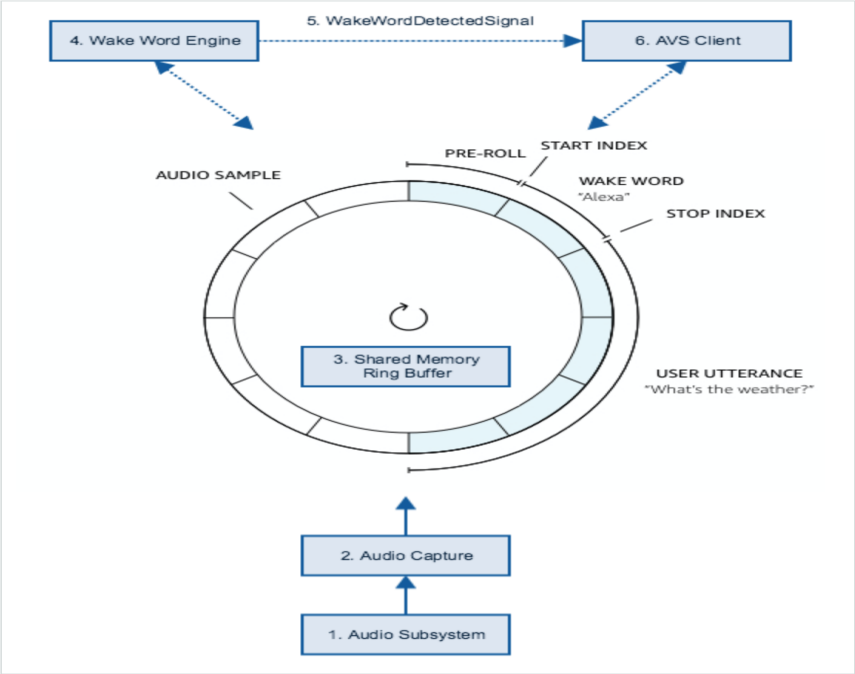


Figura 2.4: Architettura Wake Word Detection

(1) La cattura del comando vocale da parte del microfono deve essere fatto seguendo le specifiche PCM o OPUS :

PCM	Opus
16bit Linear PCM	16bit Opus
16kHz sample rate	16kHz sample rate
Single channel	32k bit rate
Little endian byte order	Little endian byte order

All'AVS viene inviato uno stream audio in un "multipart message". Nella prima parte sarà presente un oggetto JSON, contenete tutti i dati preliminari per identificare la struttura dell'audio inviato, e nella seconda parte saranno presenti i bytes catturati del comando vocale.
Per ridurre la latenza, l'audio catturato viene suddiviso e inviato in blocchi di 320bytes, ogni chunk contiene 10ms di registrazione.

(2) **Audio Subsystem** Un middle-ware, come ad esempio Advanced Linux Sound Architecture (ALSA). Permette di avere a disposizione delle API per chiedere al driver del device di aprire uno stream da cui ricevere dei campioni audio secondo delle specifiche indicate.

(3) **Audio Capture** Il processo che chiede all'Audio Subsytem di ricevere dei campioni audio dal microfono e li scrive nello *shared memory ring buffer*.

(4) **Shared Memory Ring Buffer** L'audio da inviare ad Alexa deve avere un formato specifico per permettere all'Alexa Engine di interpretare il comando :

- *pre-roll* : ha una durata di 500ms e serve all'Alexa Engine per calibrare il rumore ambientale.
- *wake word* : dalla fine del pre-roll, inizia la wake word che avrà una certa durata e permetterà all'AVS di e˙ettuare il "cloud-based wake word verification"
- *comando vocale* : dopodichè inizieranno i dati audio del comando vocale vero e proprio. Se il "cloud-based wake word verification" fallisce, verrà scartato tutto.

Questo stream, dal punto di vista implementativo, è rappresentato come uno *Shared Memory Ring Buffer*, una struttura "thread safe" a bu˙er circolare. Sarà accessibile contemporaneamente da diversi processi e dovrà dunque supportare gli accessi concorrenti di almeno uno scrittore(Audio Capture) e due lettori(Wake Word Engine e AVS Client), evitando di sovrascrivere nuovi campioni audio in zone di memorie non ancora lette.

(5) **Wake Word Engine** Sul device deve essere presente un Engine locale capace di riconoscere esclusivamente la parola "Alexa". Questo componente legge e processa i campioni audio scritti nello *shared memory ring buffer*, cercando la wake word. Se viene trovata, vengono registrati 2 parametri fondamentali che saranno comunicati all'AVS client e inseriti nei campi dell'oggetto JSON nel "multipart message", ossia *startIndexInSamples* and *endIndexInSamples*. Essi rappresentano rispettivamente l'indice di inizio e fine della wake word nello stream audio.

(6) **WakeWordDetectedSignal** E' un segnale inviato dall'*Wake Word Engine* all'AVS Client per avvisarlo che la wake word è stata trovata e dunque bisogna inviare l'audio catturato all'Alexa Voice Service.Il segnale includerà i paramentri *startIndexInSamples* e *endIndexInSamples*.

(7) **AVS Client** Il client che riceve il *WakeWordDetectedSignal* crea il "multipart message", inserisce nell'oggetto JSON i paramentri *startIndexInSamples* e *endIndexInSamples* e invia lo stream audio comprendente il pre-roll(500ms di audio precedente a *startIndexInSamples*, ossia 8000 campioni poichè la frequenza di campionamento del segnale audio deve avvenire a 16kHz come da specifiche) e i campioni audio della wake word e del comando e˙ettiv o.

2.2.3 Audio e Video

Alexa ha bisogno di ricevere degli stream audio per poter interpretare il comando vocale dell'utente ed elaborare una risposta. Dunque l'AVS Device dovrà essere predisposto di un microfono per ricevere l'input, uno speaker per riprodurre la risposta audio e di una connessione per inviare il comando vocale ad Alexa. Inoltre nella risposta è possibile che siano presenti anche contenuti video, quindi se si vorrà visionarli, bisognerà avere anche un display a disposizione.
Se nel caso dei contenuti video non sono specificati particolari requisiti hardware, oltre ai classici display dotati di risoluzioni standard(720p,1080p,etc),

per quanto riguarda l'audio Amazon fornisce delle ulteriori indicazioni.
Per ottimizzare quella fase chiamata *Automatic Speech Recognition*(ASR), ossia la conversione del comando vocale dell'utente in testo, in modo da elaborare successivamente una risposta opportuna, ogni AVS Device comunica ad Alexa, durante l'invio del comando vocale dell'utente, il profilo ASR che stanno utilizzando. Questi profili sono definiti in base al modo che si sta utilizzando sull'AVS Device per iniziare una conversazione con Alexa (sezione 2.2.2)

	Hold-to-talk	Tap-to-talk	Voice-initiated (Wake Word)	
Listening Range	0 to 0.3 m (1 ft)	0 to 0.9 m (3 ft)	0 to 0.9 m (3 ft)	0 to 2.75 m (9 ft)
ASR Profile	"CLOSE_TALK"	"NEAR_FIELD"	"NEAR_FIELD"	"FAR_FIELD"
Speech endpointing	Device	Alexa	Alexa	Alexa

- **Listening Range** : il range della distanza ottimale entro il quale si consiglia di esprimere il comando vocale
- **ASR Profile** : i profili ASR comunicati ad Alexa
- **Speech endpointing** : definisce il modo in cui lo stream audio è chiuso e da chi

2.2.4 Media

E' possibile chiedere ad Alexa di riprodurre media da varie sorgenti come USB o servizi streaming o chiedere di impostare determinati profili di equilizzazione, oltre che abbassare o alzare il volume degli speaker. Se per le sorgenti locali i codec e i container, che il dispositivo deve supportare, dipendono dalla piattaforma e dal sistema operativo, nel caso si sorgenti streaming Amazon dichiara dei requisiti obbligatori. Tra i vari media service troviamo

Amazon Music, TuneIn, iHeartRadio, Audible and Flash Briefing. I formati associati a ognuno di essi sono elencati nella figura di seguito :

MSP	Streaming Format	Playlist Extension	Container	Codec	Bit Rate(s)
Amazon Music[1]	HLSv4 (Muxed with MPEG-TS)	M3U8	MPEG-TS with ADTS	AAC LC, HE-AAC	256 Kbps (AAC)
Amazon Music[2]	Progressive	N/A	MP4	AAC	Up to 256 kbps
Amazon Music[3]	Progressive	N/A	MP3	MP3 (VBR), MP3 (CBR)	Up to 320 kbps
iHeartRadio	HLSv1	M3U8, PLSv2	N/A	AAC, AAC LC, HE-AAC	Up to 320 kbps
SiriusXM	HLS	M3U8	ADTS	AAC LC, HE-AAC	256 kbps (AAC LC), 23/64/96 kbps (HE-AAC)

[1] - Prime and Unlimited music will be provided in this audio format.
[2] - Music owned by the customer, either purchased or uploaded Amazon's Cloud Locker Service, may be streamed in this audio format.
[3] - Music owned by the customer, either purchased or uploaded Amazon's Cloud Locker Service, may be streamed in this audio format.

Amazon inoltre prevede che l'AVS Device abbia i seguenti comportamenti:

- Nel momento in cui l'utente avvia una conversazione per impartire un comando ad Alexa, il contenuto multimediale in esecuzione dovrà passare nello stato di "pausa" e riprendere automaticamente alla fine.
- L'utente dovrà essere in grado di fermare o mettere in pausa la sorgente riprodotta mediante un comando vocale oppure bottoni fisici o presenti a schermo.

2.2.5 Avvisi e Notifiche

Con Alexa è possibile impostare, cancellare o interrompere avvisi. A questa categoria appartengono timer, sveglie e promemoria. Anche in questo caso dovranno essere presenti bottoni per interrompere gli allarmi. Alla categoria notifiche, invece, appartengono tutte quelle abilità che permettono di chiedere

ad Alexa di notificare ad esempio quando ci sono nuove notizie o cambiamenti di stato di servizi di cui vogliamo tenere traccia. In questo caso i requisiti necessari sono tutti di natura visiva. Verranno approfonditi nelle sezioni successive.

2.2.6 Chiamate e Messaggi

Gli utenti possono chiedere di e˙ettuare chiamate e inviare messaggi tra dispositivi Alexa ed è necessario che essi siano sempre opportunamente informati sullo stato della comunicazione. Per far questo sono previsti degli avvisi sonori ben definiti, disponibili nel pacchetto implementativo, da utilizzare obbligatoriamente per notificare ad esempio un messaggio disponibile o una chiamata in entrata. Nel caso sia presente un display sul device, è previsto anche l'utilizzo di avvisi visivi. Esistono 2 tipi di implementazione, messi a disposizione da Amazon come estensioni integrabili ed è possibile sceglierle a piacere : *Multicolor LEDs* e *Voice Chrome*.

2.2.7 Bluetooth

E' possibile e˙ettuare il pairing Bluetooth con altri device esterni. Questa funzionalità è utile quando si hanno speaker, soundabar e auto e si vuole riprodurre musica o controllare chiamate telefoniche da sorgenti esterne Bluetooth come telefoni o tablet, pronunciando i comandi vocali corrispondenti. L'AVS Device, svolgendo il ruolo di ricevitore, dovrà necessariamente aver implementato nel suo sistema i profili Bluetooth necessari :

- *Advanced Audio Distribution Profile (A2DP-SNK)* : profilo ricevitore che consente di ricevere un segnale audio stereofonico attraverso lo standard Bluetooth da una sorgente che dispone del profilo *A2DP-Source*

- *Audio/Video Remote Control Profile(AVRCP)* : utilizzato per inviare direttive(ad esempio, avanzamento veloce, pausa, riproduzione) da un dispositivo di controllo (ad esempio, un auricolare stereo,un sistema infotainment,etc,etc) a uno di destinazione (ad esempio, un PC con lettore multimediale,un telefono).

- *Hands-free Profile(HFP)* : per gestire le chiamate telefoniche

2.3 AVS Device SDK

L'Alexa Voice Service Device SDK è un insieme di librerie C++ ed APIs che permette agli sviluppatori di velocizzare notevolmente la fase di implementazione software di tutti quei requisiti necessari a sfruttare le funzionalità di Alexa sui dispositivi Built-In, illustrate nel capitolo precedente.
La struttura di questo SDK è astratto ed estremamente modulare, formato da diversi componenti, ognuno adibito a svolgere un determinato compito nella catena di interazione con Alexa.

2.3.1 Interfacce, Eventi e Direttive

Alexa Voice Service si basa sul concetto di ***Interfaccia***. Ogni Interfaccia corrisponde a una funzionalità dell'AVS Device ed è composta da un gruppo di eventi e direttive, elementi fondamentali che permettono di costruire il dialogo con Alexa e ricevere delle risposte.

- **Evento** : messaggio inviato all'AVS per notificare ad Alexa che qualcosa sta accadendo sul Device. Il più comune è quello che notifica che l'utente ha appena iniziato la conversazione ed è in procinto di esprimere il comando

- **Direttiva** : è il messaggio di risposta, legato all'*evento*, inviato da Alexa verso l'AVS Device All'interno di questo messaggio ci sono tutte le informazioni che danno indicazioni sulle azioni da svolgere sul dispositivo come ad esempio riprodurre un contenuto streaming oppure un allarme oppure semplicemente l'audio di risposta di Alexa.

Essendo un concetto molto importante, è necessario quindi sapere le Interfacce presenti e le funzionalità corrispondenti :

Interfaccia	Descrizione
Alert	Interfaccia utilizzata per impostare,eliminare e interrompere sveglie e allarmi
AudioActivityTracker	Interfaccia utilizzata per informare Alexa su qual è l'ultimo componente che ha utilizzato il canale audio

AudioPlayer	Interfaccia utilizzata per gestire e controllare uno stream audio inviato da Alexa
Bluetooth	Interfaccia utilizzata per gestire il dispositivo Bluetooth accoppiato con l'AVS Device, come uno speaker o uno smartphone.
EqualizerController	Interfaccia utilizzata per regolare le impostazioni dell'equalizzatore dell'AVS Device come i bassi, gli acuti o il bilanciamento
InputController	Interfaccia utilizzata per selezionare o cambiare una sorgente di input del dispositivo Built-In, come ad esempio "HDMI 1", "VGA" o "TV"
Notifications	Interfaccia utilizzata per fornire all'utente il contenuto audio e video di una nuova notifica
PlaybackController	Interfaccia utilizzata per gestire la coda della fonte multimediale in riproduzione tramite GUI o bottoni fisici
Settings	Interfaccia utilizzata per gestire le impostazioni di Alexa come la lingua.
Speaker	Interfaccia utilizzata per controllare il volume dei contenuti multimediali inviati da Alexa
SpeechRecognizer	Interfaccia utilizzata per gestire il comando vocale dell'utente.
SpeechSynthesizer	Interfaccia utilizzata per gestire la risposta vocale in arrivo dall'AVS.
System	Interfaccia utilizzata per mandare informazioni ad Alexa riguardo lo stato del prodotto e quali interfacce sono supportate.
TemplateRuntime	Interfaccia utilizzata per i contenuti visivi.
VisualActivityTracker	Interfaccia utilizzata per informare Alexa qual è l'interfaccia che sta utilizzando il display.

2.3.2 Architettura

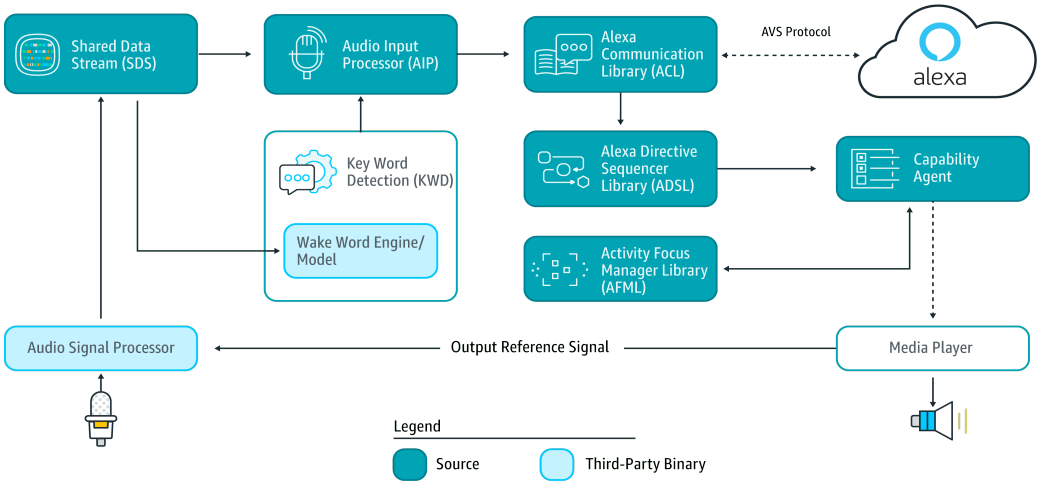


Figura 2.5: Architettura AVS Device SDK

Il diagramma illustra i vari componenti dell'SDK e come essi interagiscono tra loro :

- **Audio Signal Processor(ASP)** : non è un componente interno all'SDK ma è un firmware per il Digital Signal Processor. Il suo compito principale è di creare un singolo stream audio anche in presenza di molteplici microfoni e successivamente passare i campioni catturati allo Share Data Stream (SDS) quando sono richiesti dall'SDK. Inoltre migliora l'audio catturato dal microfono con tecniche come Acoustic Echo Cancellation (AEC), soppressione del rumore, beam forming ed eliminazione di interferenze che possono arrivare dagli speaker.

- **Share Data Stream** : interagisce sia con l'Audio Signal Processor che con il Wake Word Engine(WWE). E' un ring bu˙er sul quale operano in concorrenza 1 scrittore (ASP) e 2 lettori(AIP e WWE)e nel quale si trovano i campioni audio del microfono.

- **Wake Word Engine (WWE)** : è anch'esso un componente esterno all'SDK, quindi da integrare con librerie esterne. Il WWE implementa il "*Cloud-based wake word verification*" come illustrato nel capitolo 2.2.2, monitorando costantemente lo SDS in cerca della wake word "Alexa".

- **Audio Input Processor (AIP)** : il suo compito consiste nel leggere i campioni audio dall'SDS, mandarli all'ACL per essere inviati all'AVS. Si attiva in 3 casi :

(1) Tap-to-talk : quando l'utente utilizza il Tap-to-talk, l'AIP viene avvisato e invia i campioni audio nel SDS a partire dall'indice al quale si è arrivati a scrivere nel bu˙er in quel momento.

(2) Wake Word Detection : se è attivo il Wake Word Engine, nel momento in cui arriva il segnale di inizio lettura perchè la wake word è stata trovata, l'AIP inizia a leggere i dati nell'SDS secondo il formato preroll - wake word - comando vocale illustrato nel metodo "*Cloud-based wake word verification*".

(3) Speech directive : se nel dialogo, l'AVS aspetta un'ulteriore risposta dall'utente per esaudire la richiesta del comando iniziale, l'AIP verrà notificato di ciò e dunque inizierà a prelevare i campioni dall'SDS per spedirli all'AVS.

- **Alexa Communications Library (ACL)** : gestisce la connessione tra il device e L'AVS. In particolare si occupa di :
  - Stabilire e mantenere attiva una connessione HTTP/2 sicura e persistente.
  - Inviare eventi secondo il formato dei *multipart-message*, contenenti oggetti JSON e contenuti audio.
  - Inoltrare le risposte/direttive all'*Alexa Directive Sequencer Library*.
  - Gestire la disconnessione e le riconnessioni in caso di caduta accidentale del collegamento con l'AVS.
- **Alexa Directive Sequencer Library (ADSL)** : Poichè una riposta può essere formata da molteplici direttive, scritte secondo il formato JSON e incapsulata in *messaggi multipart/mixed*, il compito principale dell'ADSL è quello di e˙ettuar e il *parsing* di questi messaggi, analizzare le direttive e individuare a quali interfacce appartengono. Stabilite le interfacce di appartenenza, l'ADSL potrà inoltrare ogni direttiva al *Capability Agent* appropriato.
- **Capability Agents** : il loro compito si articola in 2 fasi :
  - Ricevere la direttiva appropriata dall'ADSL
  - Leggere l'azione contenuta nel payload della direttiva e eseguirla

  Ogni Capability Agent corrisponde a un'Interfaccia AVS. Quindi ad esempio se l'utente chiederà ad Alexa di riprodurre un brano musicale, sarà attivata la Capability Agent che si occupa di gestire la direttiva

"*AudioPlayer*", caricando la canzone nel mediaplayer locale e mandandolo in riproduzione.
La tabella di seguito mostra le corrispondenze con le Interfacce più comuni :

AVS Interface	Capability Agent	Descrizione
AudioPlayer	**AudioPlayer**	Gestione e controllo dello stream audio inviato da Alexa
Alert	**Alert**	Impostazione, eliminazione e interruzione di sveglie e allarmi
Bluetooth	**Bluetooth**	Gestione del dispositivo Bluetooth accoppiato con l'AVS Device
DoNotDisturb	**DoNotDisturb**	Gestione della funzionalità "*Non disturbare*".
EqualizerController	**Equalizer**	Regolazione delle impostazioni dell'equalizzatore dell'AVS Device.
InteractionModel	**InteractionModel**	Gestione delle Alexa Routines.
Notifications	**Notifications**	Gestione del contenuto delle notifiche in arrivo dall'AVS.

PlaybackController	**PlaybackController**	Gestione dei controlli della fonte multimediale in riproduzione tramite GUI o bottoni fisici.
Multi-room Music	**Multi-room Music**	Gestione del Multi-room Music.
Speaker	**SpeakerManager**	Controllo del volume dei contenuti multimediali inviati da Alexa.
SpeechRecognizer	**Audio Input Processor**	Gestione del comando vocale dell'utente.
SpeechSynthesizer	**SpeechSynthesizer**	Gestione della risposta vocale in arrivo dall'AVS.
System	**System**	Invio delle informazioni ad Alexa riguardo lo stato del prodotto e quali interfacce sono supportate.
TemplateRuntime	**TemplateRuntime**	Gestione dei contenuti visivi della risposta.

- **Activity Focus Manager Library (AFML)** : gestisce l'ordine di esecuzione delle direttive e seleziona quale Capability Agent deve avere il controllo sull'output. Ad esempio, se è in riproduzione uno stream audio e scatta un timer, il Capability Agent che gestisce le sveglie e i timer,

chiede il controllo all'AFML. Se questo Capability Agent gestisce una funzionalità prioritaria rispetto a quella attualmente in atto, l'AFML gli concederà immediatamente il controllo, altrimenti lo metterà in attesa. In questo caso poichè i timer hanno priorità maggiore su un contenuto multimediale, verrà messo in pausa lo stream audio e il Capability Agent che gestisce i timer potrà riprodurre l'avviso sonoro senza attesa. Terminato l'avviso audio, verrò notificato all'AFML che l'azione legata ai timer è completa e quindi verrà consegnato nuovamente il controllo al modulo che gestisce lo stream audio, riprendendo la riproduzione del contenuto multimediale.
Per gestire la priorità degli input e output audiovisivi, nell'SDK è stato definito il concetto di ***Channel***.

Channels

A tal proposito sono definiti 3 tipi di canali :

- **Dialog channel** : utilizzato quando l'utente o Alexa sta parlando.
- **Alerts channel** : utilizzato quando scatta una sveglia o un timer.
- **Content channel** : utilizzato per riprodurre un contenuto multimediale locale o in arrivo da un servizio streaming.

Ogni Interfaccia è mappata su un canale e solo un'interfaccia alla volta può essere attiva su quel canale. Di seguito è riportato il mapping :

Channel	Interfaces
Dialog	SpeechRecognizer, SpeechSynthesizer
Alerts	Alerts
Content	AudioPlayer

Un canale può avere 2 stati :

- **Foreground** : il contenuto audio al suo interno deve essere riprodotto in primo piano.
- **Background** : quando il canale passa in "*secondo piano*", il contenuto audio del canale deve essere messo in pausa oppure mixato con volume attenutato.

Nel caso in cui due o più interfacce, divenute attive, richiedono il passaggio in foreground del loro canale di appartenenza, il canale portato in primo piano sarà quello a priorità maggiore. La priorità segue il seguente ordine :

(1) *Dialog*

(2) *Alerts*

(3) *Content*

Sulla base di queste nozioni, l'AFML decide man mano a quale Capability Agent dare il controllo dell'output audio. A titolo d'esempio viene illustrato il seguente caso :

(1) Il Capability Agent "*Audio Player*" sta riproducendo sul *Content Channel* un contenuto multimediale

(2) L'utente chiede ad Alexa : "che tempo fa" e viene restuita una risposta audio e video. In questo caso si attivano rispettivamen-te e in sequenza i Capability Agents : *Audio Input Processor* e *SpeechSynthesizer*.

(3) I Capability Agent che operano sul *Dialog channel* richiedono dunque all'AFML l'attivazione del loro canale d'appartenenza ed essendo gestori di un canale a massima priorità gli verrà immediatamente concessa. Nel frattempo il contenuto del "*Content channel*" verrà messo in pausa fino a quando il dialogo con Alexa non sarà terminato.

(4) Quando l'interazione si chiude, verrà notificato il tutto all'AFML che porrà il "*Dialog channel*" in bakgroung e il "*Content channel*" in foreground, restituendo il controllo al Capability Agent "*Audio-Player*. Ecco che a questo punto verrà ripresa la riproduzione del contenuto all'interno del "*Content channel*".

- **MediaPlayer** E' un componente esterno. L'SDK fornisce un wrapper all'interno del quale si possono utilizzare le API per sfruttare i servizi del lettore multimediale della piattaforma come ad esempio GStreamer su Linux/UNIX oppure Android Media Player per il mondo Android.

2.3.3 HTTP/2

Essendo un servizio cloud, tutta l'infrastruttura di Alexa funziona mediante scambio di messaggi in rete tra client(l'AVS Device) e il server(Alexa Voice

Service).
Il protocollo di rete utilizzato per la comunicazione è un'estensione dell'HTTP, ossia l'*HTTP/2*. La semantica rispetto alla versione HTTP/1 è la stessa e non sono state apportate modifiche alle funzionalità o ai concetti principali come i metodi HTTP, i codici di stato, gli URI e i campi delle intestazioni. La principale differenza consiste nel fatto che tutte le comunicazioni vengono eseguite all'interno di una singola connessione TCP in grado di portare un numero qualsiasi di flussi/stream bidirezionali.
All'accensione del dispositivo, dopo aver ottenuto l'*Access Token*, è necessario stabilire una singola connessione persistente HTTP/2 con l'AVS che verrà utilizzata per lo scambio e la gestione di direttive, eventi, stream, ping e timeout, mediante i diversi metodi di chiamata tipici dell'HTTP.
Ogni chiamata HTTP avrà uno dei seguenti Base URLs, in base alle regioni d'appartenenza:

- **Asia**(Australia, Japan, New Zealand) : *https://avs-alexa-fe.amazon.com*
- **Europa**(Austria, France, Germany, India, Italy, Spain, United Kingdom) : *https://avs-alexa-eu.amazon.com*
- **North America**(Canada, Mexico, United States) : *https://avs-alexa-na.amazon.com*

Per stabilire correttamente la connessione HTTP/2 è obbligatorio eseguire i seguenti due passi :

(1) Entro 10 secondi dall'apertura della connesione HTTP/2 con l'AVS, si crea un **downchannel stream**, effettuando una chiamata HTTP GET.

```
:method = GET
:scheme = https
:path = /{{API version}}/directives
authorization = Bearer {{YOUR_ACCESS_TOKEN}}
```

Se la richiesta è andata a buon fine, il *downchannel stream* rimane in uno stato "*half-closed*" lato client e aperto lato AVS. Verrà usato per inviare tutte quelle direttive che non hanno una corrispondente richiesta vocale quindi timer, notifiche push, allarmi o messaggi.

(2) Dopo aver stabilito il *downchannel stream*, il device deve sincronizzare lo stato dei suoi componenti con l'AVS e comunicare quali interfacce supporta. Verrà dunque creato un nuovo stream sulla connessione HTTP/2 esistente e inviata una richiesta POST HTTP, il cui contenuto sarà un multipart-message di un "*SynchronizeState event*"[9].
In questo caso e per i successivi, gli stream aperti dal client si chiameranno **event stream** e verranno chiusi nel momento in cui si riceverà una risposta(direttiva) da Alexa.

Sincronizzando correttamente i suoi componenti con l'AVS, il Device sarà in grado infine di inviare qualsiasi evento e ricevere le direttive corrispondenti. E' importante tener a mente che ogni nuovo evento sarà inviato sempre su un nuovo stream e sarà chiuso nel momento in cui la direttiva associata verrà ricevuta.
La connessione TCP inoltre va tenuta attiva per non rischiare di chiuderla per inattività, è quindi necessario inviare un *PING frame* ogni 5 minuti verso l'AVS se la connessione è in stato di IDLE.

```
:method = GET
:scheme = https
:path = /ping
authorization = Bearer {{YOUR_ACCESS_TOKEN}}
```

2.4 Interaction Model

Come illustrato precedentemente, il protocollo utilizzato per inviare eventi e ricevere direttive è l'HTTP/2 con il quale si inviano *multipart-message* su degli stream. Per comprendere meglio la struttura di questi messaggi e la sequenza di richieste scambiate con l'AVS durante un'interazione, è utile a questo punto approfondire ed analizzare quelli che sono gli scenari più comuni.
Innanzitutto la struttura generale di una richiesta HTTP/2 contente un "multipart-message" sarà così composta :

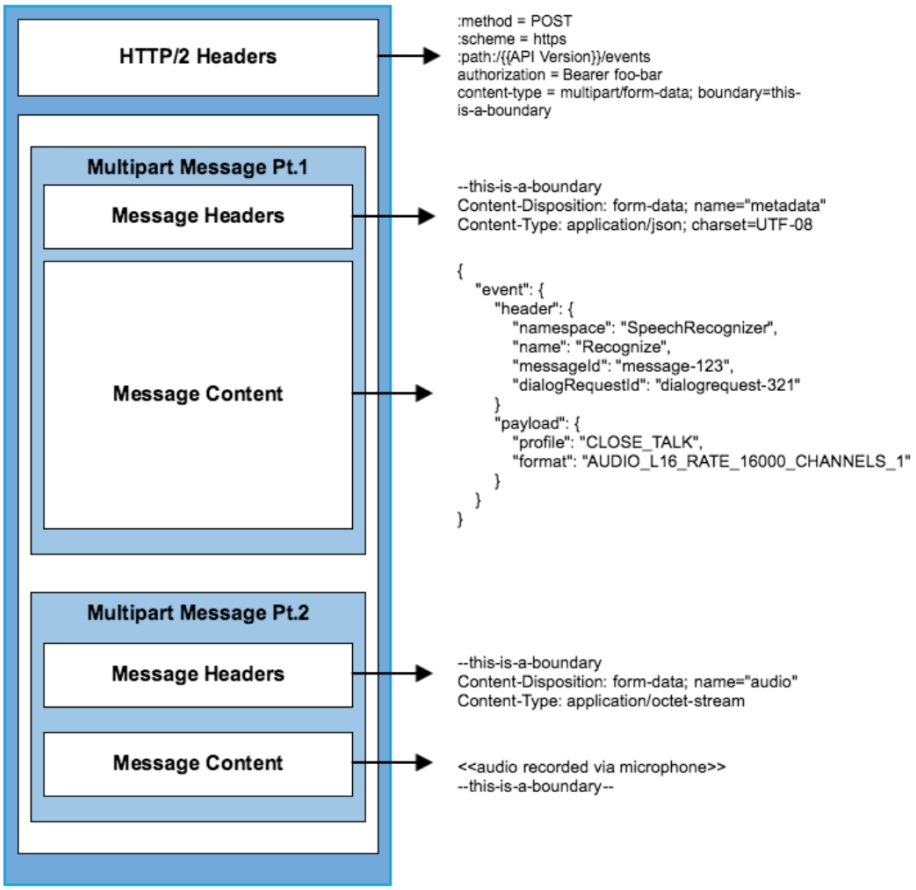


Figura 2.6: Messaggio HTTP/2 "multipart-message"

Ogni chiamata dovrà quindi avere :

- **HTTP/2 Headers**
  - Method : *POST* per tutti gli eventi inviati all'AVS, usando il path degli eventi, mentre *GET* per stabilire un *downchannel stream*, usando il path delle direttive.
  - Scheme : *https*
  - Paths : il path relativo da aggiungere ai BaseUrls. Per gli eventi : "*/v20160207/events*" mentre per le direttive : "*/v20160207/directives*".
  - Authorization : a questo campo corrisponde l'*AccessToken*.
  - Content Type : descrive i dati contenuti nel *body*. *multipart/form-data* per indicare all'AVS che questo è un multipart-message mentre con *boundary* si indica il carattere di delimitazione tra l'HTTP/2 Header, JSON object e i bytes dell'audio.

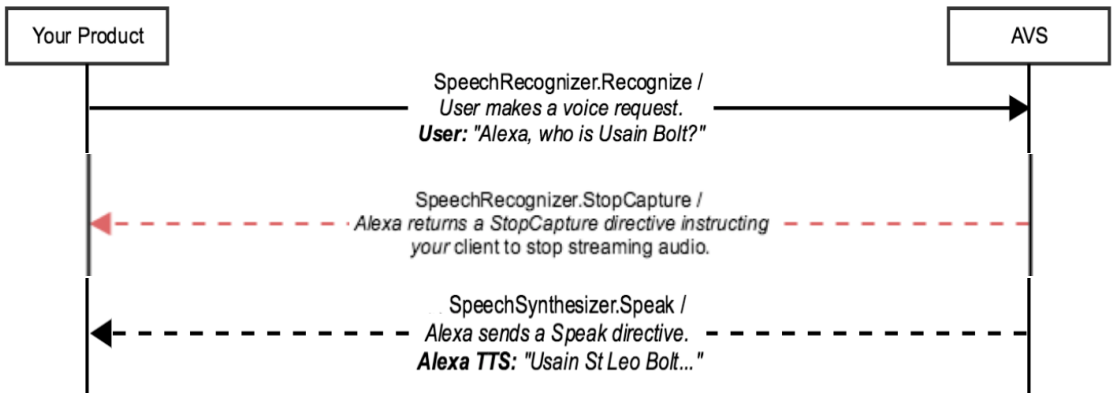
- **HTTP/2 Multipart Messages**

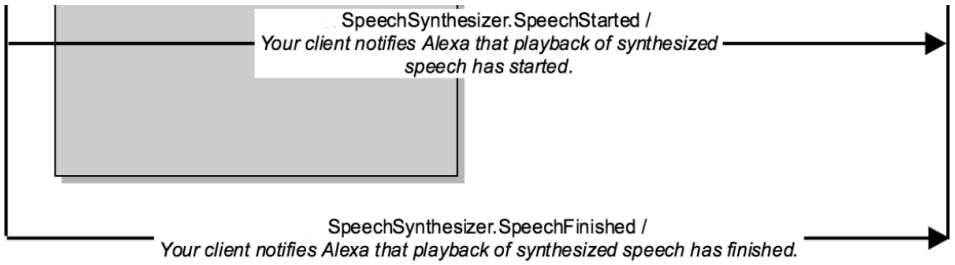
  – Message Header : indica con "*Content-Disposition* e "*Content-Type*" se nel corpo del messaggio ci sarà un JSON Object oppure i bytes audio registrati dal microfono

  – Message Content : contenuto vero e proprio. In base ai parametri del Message Header, sarà possibile trovare o i dati testuali del JSON Object che indicano il tipo di evento e i parametri che lo descrivono oppure i bytes audio.
  Nel caso del JSON Object, il parametro "**dialogRequestId**" è molto importante. Viene creato dal device nel momento in cui l'utente inizia una nuova conversazione e serve a legare le direttive mandate dall'AVS nella sessione corrente. Nel momento in cui viene iniziata una nuova conversazione e dunque si crea un nuovo *dialogRequestId*, il vecchio *dialogRequestId* non avrà più validità e dovranno essere scartate tutte quelle direttive pendenti legate alla vecchia conversazione. Può anche succedere che sia l'AVS ad iniziare una nuova conversazione, in quel caso nella direttiva sarà presente un campo "*NewDialogRequest*" al quale corrisponderà un nuovo dialogRequestId e dunque il device dovrà segnarlo come dialogRequestId corrente.

2.4.1 SpeechRecognizer e SpeechSynthesizer

Si passerà ora ad analizzare il caso in cui l'utente chieda qualcosa ad Alexa e la direttiva restituita sia semplicemente una risposta audio.





(1) Quando l'utente avvierà l'interazione, verrà attivato il Capability Agent "*Audio Input Processor*". Una volta ottenuto il focus dall'AFML, richiederà i campioni audio del microfono e dunque l'ACL invierà un messaggio HTTP/2, strutturato come illustrato precedentemente, contenente quindi un multipart-message, in cui sarà presente l'oggetto JSON che descriverà l'evento "***SpeechRecognizer.Recognize Event***" e lo stream audio del comando vocale. Il JSON Object dell'evento sarà quindi così rappresentato :

```
{
  "context": [
    // Use an array of context objects to communicate the
    // state of all client components to Alexa. See Context for details.
  ],
  "event": {
    "header": {
      "namespace": "SpeechRecognizer",
      "name": "Recognize",
      "messageId": "{{STRING}}",
      "dialogRequestId": "{{STRING}}"
    },
    "payload": {
      "profile": "{{STRING}}",
      "format": "{{STRING}}",
      "initiator": {
        "type": "{{STRING}}",
        "payload": {
          "wakeWordIndices": {
            "startIndexInSamples": {{LONG}},
            "endIndexInSamples": {{LONG}}
          },
          "wakeWord": "{{STRING}}",
          "token": "{{STRING}}"
        }
      }
      "startOfSpeechTimestamp": "{{STRING}}"
    }
  }
}
```

E' possibile notare che a ogni "namespace" appartiene un "name". Questo perchè il namespace descrive il contesto dell'azione, mentre il name specifica il tipo di azione. Ecco perchè il namespace "SpeechRecognizer" è inerente al comando vocale dell'utente, "Recognize" specifica che l'utente ha pronunciato il comando e il messaggio lo contiene.
Inoltre tra i parenti fondamentali fin qui non menzionati, rientra il campo *initiator*, utilizzato per indicare come l'utente ha iniziato la conversazione, se con "Wake Word", "Tap-to-talk" oppure "Hold-to-talk". In questo caso vista anche la presenza dei parametri "*startIndexInSamples*" e "*endIndexInSamples*", si evince che la conversazione è iniziata utilizzando la wake word Alexa.

(2) L'AVS manda sul "downchannel stream" una direttiva "**SpeechRecognizer.StopCapture**" per avvisare il device che ha capito il comando e sta preparando la risposta

(3) A questo punto viene inviata una o più direttive. In questo caso viene inviata una "**SpeechSyntesizer.Speak**" per indicare al client che la risposta contiene una risposta audio da riprodurre negli speaker.

```
:status = 200
content-type = multipart/related; boundary=this-is-a-boundary;
type="application/json"

--this-is-a-boundary
Content-Type: application/json; charset=UTF-8

{
    "directive": {
        "header": {
            "namespace": "SpeechSynthesizer",
            "name": "Speak",
            "messageId": "ewq-321",
            "dialogRequestId": "dialogRequestId-321"
        },
        "payload": {
            "url": "cid:1234",
            "format": "AUDIO_MPEG"
            "token": "kr17447380422"
        }
    }
}
```

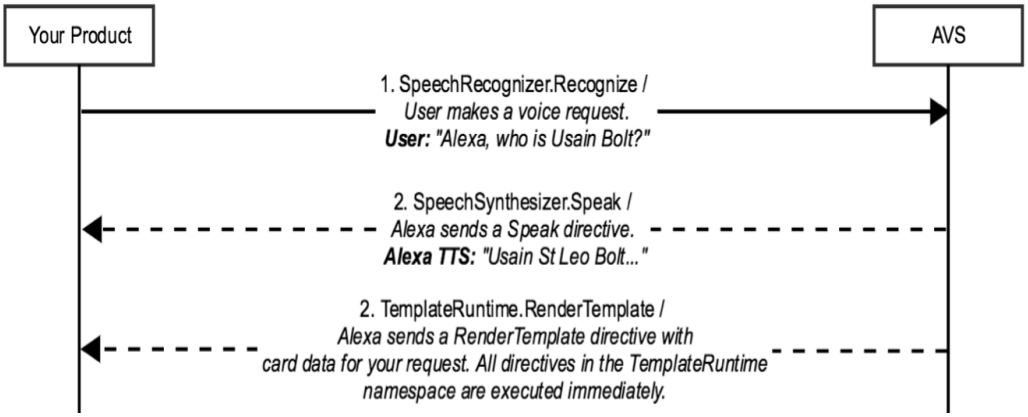
```
--this-is-a-boundary
Content-Type: application/octet-stream
Content-ID: <1234>

{{binary audio attachment}}
--this-is-a-boundary--
```

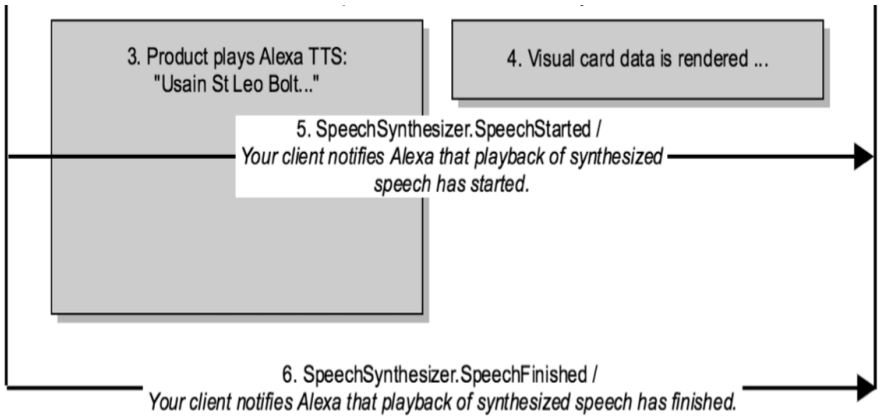
(4) L'ADSL consegnerà la direttiva ricevuta al Capability Agent "*SpeechSyntesizer*" e inizierà a riprodurre la risposta. Nel frattempo verrà inviato un evento "**SpeechSyntesizer.SpeechStarted**" per avvisare l'AVS che la risposta audio è in fase di riproduzione.

(5) Terminata la riproduzione, il device avvisa l'AVS che la risposta è stata riprodotta completamente mediante un evento "**SpeechSyntesizer.SpeechFinished**" e lo stream si chiude.

2.4.2 DisplayCards

Molte volte la risposta di Alexa oltre a contenere una direttiva "*SpeechSyntesizer.Speak*", può anche avere un contenuto video come ad esempio un'immagine o anche un video stream a seconda della richiesta dell'utente. In questo caso si utilizzano le direttive dell'Interfaccia "*TemplateRuntime*". Sarà compito del programmatore implementare la rappresentazione a video del contenuto visivo.
Ecco come può essere rappresentata l'interazione alla domanda "Alexa, chi è Usain Bolt?" :





In questo caso, la sequenza è pressoché identica a quella della sezione precedente, se non per il fatto che viene ricevuta in più una direttiva "**RenderTemplate**" che verrà presa in carico dal Capability Agent "*TemplateRuntime*" ed eseguita direttamente, e se possibile in contemporanea alla direttiva "*Speak*", in un thread di˙eren te.
Esistono 4 tipi di "*RenderTemplate*" e saranno indicati dal parametro "*type*". Ogni tipo darà indicazioni su come saranno organizzati e in quali campi i dati che si trovano nel JSON Object. Questi 4 tipi sono :

- **BodyTemplate1** : nel JSON Object ci saranno i dati per il titolo, il sottotitolo, un testo associato e un'icona del servizio web che ha fornito l'informazione.

- **BodyTemplate2** : nel template ci sarà un testo e la URL dal quale scaricare l'immagine associata.

- **ListTemplate1** : a questo template sono associati tutti quelle info tipiche di una lista della spesa o una lista di cose da fare in un dato giorno.

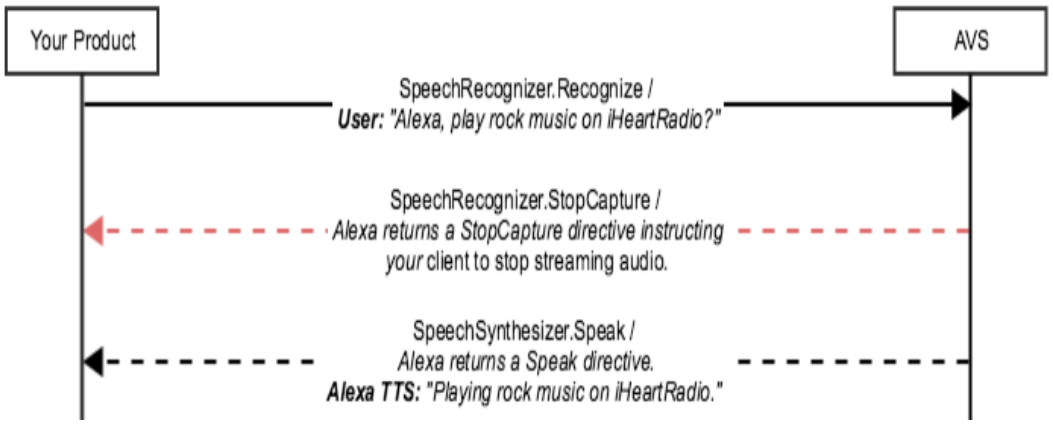
- **WeatherTemplate** : è il tipico template usato per rappresentare delle informazioni legate alle previsioni del tempo, quindi ci saranno per ogni giorno, la previsione, l'icona associata e il nome del giorno stesso.

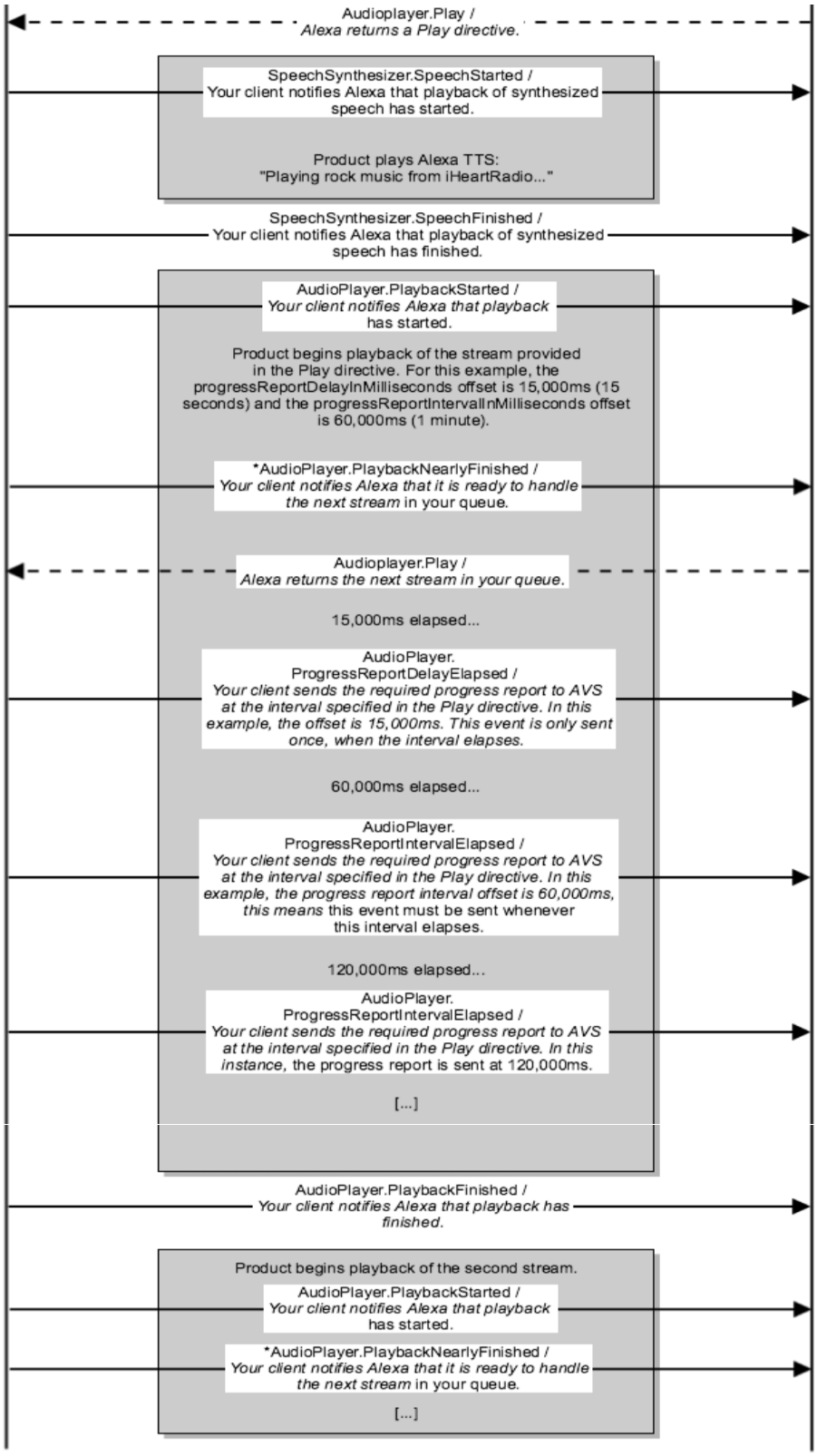
Di seguito è riportato l'esempio di un *BodyTemplate1* :

```
{
  "directive": {
    "header": {
      "namespace": "TemplateRuntime",
      "name": "RenderTemplate",
      "messageId": {{STRING}},
      "dialogRequestId": {{STRING}}
    },
    "payload": {
      "token": "{{STRING}}",
      "type": "bodyTemplate2",
      "title": {
        "mainTitle": "Who is Usain Bolt?",
        "subTitle": "Wikipedia"
      },
      "skillIcon": null,
      "textField": "Usain St Leo Bolt, OJ, CD born 21 August 1986..."
      "image": {
        "contentDescription": "{{STRING}}",
        "sources": [
          {
            "url": "https://example.com/usain_bolt.jpg",
            "size": "LARGE"
          }
        ]
      }
```

2.4.3 AudioPlayer

Questo è il caso in cui l'utente chiede di riprodurre una playlist da un music provider, ad esempio : "Alexa, riproduci musica rock da Amazon Music".




Audioplayer.Play /
Alexa returns a Play directive.
SpeechSynthesizer.SpeechStarted /
Your client notifies Alexa that playback of synthesized speech has started.
Product plays Alexa TTS:
"Playing rock music from iHeartRadio..."
SpeechSynthesizer.SpeechFinished /
Your client notifies Alexa that playback of synthesized speech has finished.
AudioPlayer.PlaybackStarted /
Your client notifies Alexa that playback has started.
Product begins playback of the stream provided in the Play directive. For this example, the progressReportDelayInMilliseconds offset is 15,000ms (15 seconds) and the progressReportIntervalInMilliseconds offset is 60,000ms (1 minute).
*AudioPlayer.PlaybackNearlyFinished /
Your client notifies Alexa that it is ready to handle the next stream in your queue.
Audioplayer.Play /
Alexa returns the next stream in your queue.
15,000ms elapsed...
AudioPlayer.
ProgressReportDelayElapsed /
Your client sends the required progress report to AVS at the interval specified in the Play directive. In this example, the offset is 15,000ms. This event is only sent once, when the interval elapses.
60,000ms elapsed...
AudioPlayer.
ProgressReportIntervalElapsed /
Your client sends the required progress report to AVS at the interval specified in the Play directive. In this example, the progress report interval offset is 60,000ms, this means this event must be sent whenever this interval elapses.
120,000ms elapsed...
AudioPlayer.
ProgressReportIntervalElapsed /
Your client sends the required progress report to AVS at the interval specified in the Play directive. In this instance, the progress report is sent at 120,000ms.
[...]
AudioPlayer.PlaybackFinished /
Your client notifies Alexa that playback has finished.
Product begins playback of the second stream.
AudioPlayer.PlaybackStarted /
Your client notifies Alexa that playback has started.
*AudioPlayer.PlaybackNearlyFinished /
Your client notifies Alexa that it is ready to handle the next stream in your queue.
[...]

(1) La sequenza *SpeechRecognizer.Recognize* - *SpeechRecognizer.StopCapture* - *SpeechSyntesizer.Speak* - *SpeechSyntesizer.SpeechStarted* - *SpeechSyntesizer.SpeechFinisched* è la medesima di ogni comando vocale.

(2) In più il device riceve una direttiva "**AudioPlayer.Play**" che serve per istruirlo su come riprodurre lo stream audio.

```
{
  "directive": {
    "header": {
      "namespace": "AudioPlayer",
      "name": "Play",
      "messageId": "42941f13-90ed-4d9e-8159-xxxxxxxx",
      "dialogRequestId": "req:a345fgh598383xxx""
    },
    "payload": {
      "playBehavior": "REPLACE_ALL",
      "audioItem": {
        "audioItemId": "test1.as-ct.v1.XYZ-ABCDE-FGHIJ#ACRI#url#ACRI#0f6bcd24-
f621-555a-822c-1111111:1",
        "stream": {
          "url": "https://opml.radiotime.com/Tune.ashx?
serial=SAMPLE&formats=aac,mp3&partnerId=SAMPLE",
          "streamFormat": "AUDIO_MPEG",
          "offsetInMilliseconds": 0,
          "expiryTime": "2016-09-13T18:22:49+0000",
          "progressReport": {
            "progressReportDelayInMilliseconds": 15000,
            "progressReportIntervalInMilliseconds": 900000
          },
          "token": "test1.as-ct.v1.XYZ-ABCDE-FGHIJ#ACRI#url#ACRI#0f6bcd24-f621-
555a-822c-1111111:1"
        }
      }
    }
  }
}
```

- **playBehavior** : indica cosa fare della cosa di riproduzione locale da quando è ricevuta la direttiva "*AudioPlayer.Play*". Sono supportati 3 comportamenti :
  - REPLACE_ALL : la coda di stream locale deve essere totalmente cancellata e bisogna iniziare a riempirla nuovamente con lo stream della direttiva ricevuta.
  - ENQUEUE : lo stream dell'audio deve essere messa in fondo alla coda di stream locale.
  - REPLACE_ENQUEUED : tutti gli stream nella coda vanno scartati ad eccezione di quelli che si sta riproducendo. In coda a questo va messo lo stream ricevuto nella direttiva.
- **audioItemId** : token che identifica l'audio stream.
- **url** : l'URL dal quale poter scaricare lo stream audio.
- **streamFormat** : formato dello stream audio.
- **offsetInMilliseconds** : o˙set in millisecondi da quale iniziare la riproduzione dello stream audio.
- **expiryTime** : il Timestamp in cui l'URL diventerà invalido.
- **progressReport** : oggetto che contiene 2 parametri
  - *progressReportDelayInMilliseconds* : o˙set in millisecondi dall'inizio della riproduzione dopo il quale bisognerà inviare l'evento "*AudioPlayer.ProgressReportDelayElapsed*". Verrà inviato una sola volta.
  - *progressReportIntervalInMilliseconds* : o˙set in millisecondi dall'inizio della riproduzione dello stream dopo il quale dovrà essere inviato periodicamente un evento "*AudioPlayer.ProgressReportIntervalI*". Questo evento dovrà essere inviato a intervalli. I millisecondi di attesa tra un intervallo e l'altro saranno pari a "progressReportIntervalInMilliseconds".

(3) Il device invia gli eventi "**AudioPlayer.PlaybackStarted**" e "**AudioPlayer.PlaybackNearlyFinished**" per notificare all'AVS rispettivamente l'inizio della riproduzione dello stream e che è pronto a ricevere un altro stream, e quindi una direttiva "*AudioPlayer.Play*", da aggiungere alla coda di riproduzione per ridurre la latenza di riproduzione tra uno stream e l'altro.

(4) Dopo "progressReportDelayInMilliseconds" verrà inviato il primo report con l'evento "**AudioPlayer.ProgressReportDelayElapsed**". Serve per informare Alexa sullo stato di riproduzione dello stream.

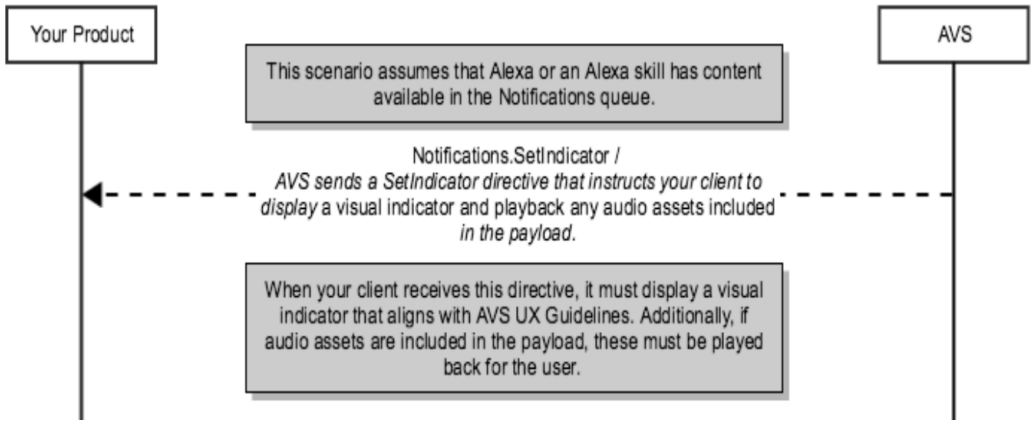
(5) Dopodichè da "progressReportIntervalInMilliseconds" e ad intervalli della stessa durata verrà inviato il report event "**AudioPlayer.ProgressReportIntervalI**". Serve per informare Alexa sullo stato di riproduzione dello stream.

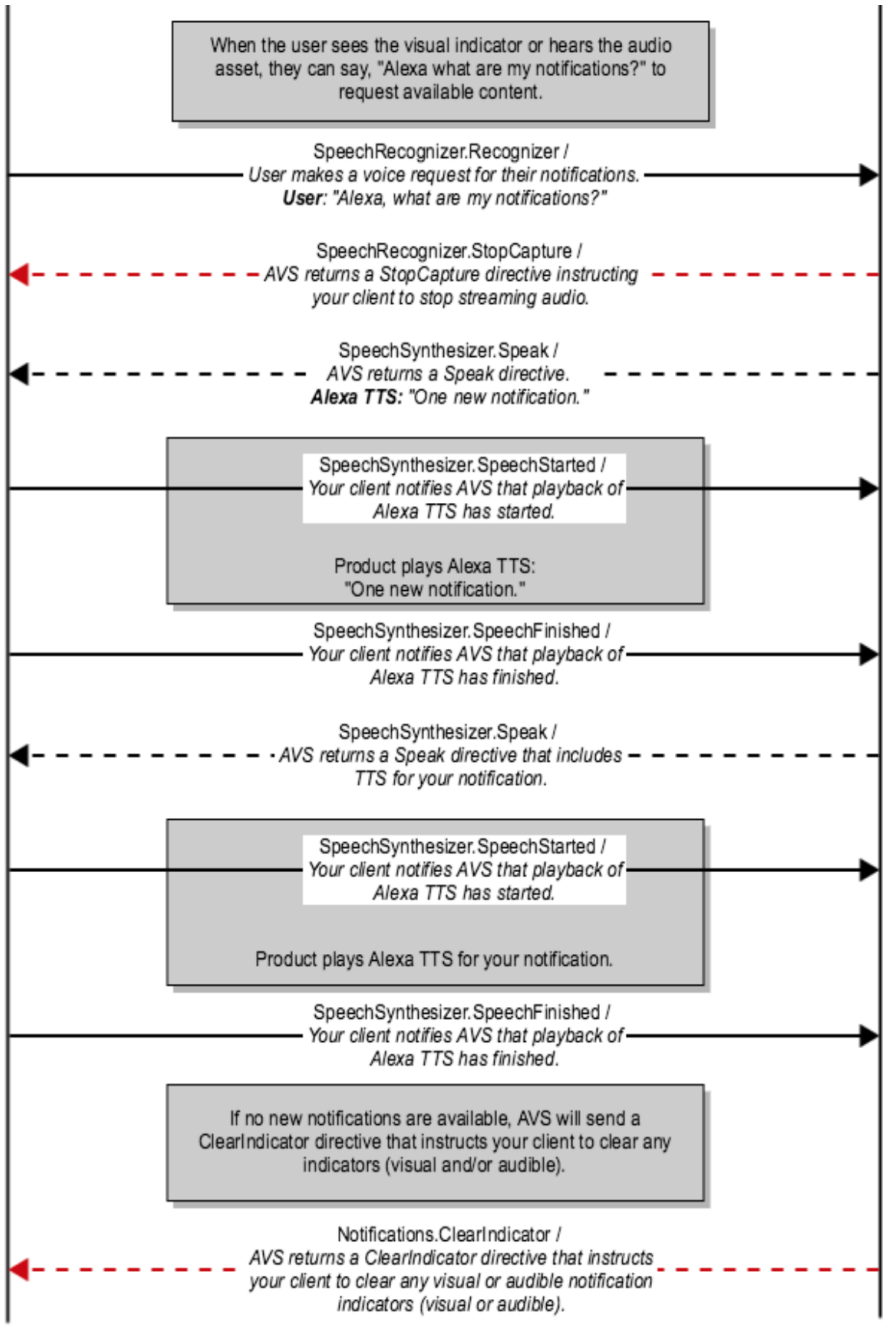
(6) Verrà notificata la conclusione di fine riproduzione dello stream con l'evento "**AudioPlayer.PlaybackFinished**"

(7) Si procede dal punto 3 in poi fino a svuotare la coda di stream da riprodurre.

2.4.4 Notifications

Nel caso delle notifiche, l'interazione non è generata inizialmente da una richiesta dell'utente :





(1) L'utente attiva le notifiche per un dato argomento come news o messaggi in arrivo tramite l'App Alexa.

(2) La skill Alexa genera la notifica che va ad inserirsi nella "*notifications queue*" del cloud Alexa.

(3) A questo punto l'AVS genera una direttiva "**SetIndicator**" dell'interfaccia "*Notifcations*" verso tutti i device collegati e in linea, associati all'account dell'utente. Su ogni dispositivo verranno mostrati degli avvisi visivi e sonori, segnalando la presenza di nuove notifiche.

```
{
    "directive": {
        "header": {
            "namespace": "Notifications",
            "name": "SetIndicator",
            "messageId": "{{STRING}}"
        },
        "payload": {
            "persistVisualIndicator": {{BOOLEAN}},
            "playAudioIndicator": {{BOOLEAN}},
            "asset": {
                "assetId": "{{STRING}}",
                "url": "{{STRING}}"
            }
        }
    }
}
```

(4) Per recuperare la notifica che è presentata come una direttiva "*SpeechSyntesizer.Speak*", l'utente potrà chiedere : "Alexa quali sono le mie notifiche?", iniziando a questo punto la classica conversazione illustrata nella sezione 2.4.1.

(5) Appena il device invia l'evento "*SpeechSyntesizer.SpeechFinished*", l'AVS invierà, sul downchannel stream, la direttiva "**Notifcations.ClearIndicator**" per comunicare al device di eliminare tutti gli indicatori di avviso di nuove notifiche.

```
{
    "directive": {
        "header": {
            "namespace": "Notifications",
            "name": "ClearIndicator",
            "messageId": "{{STRING}}"
        },
        "payload": {
        }
    }
}
```

Capitolo 3

Alexa Skill Kit

Grazie all'AVS SDK, è possibile sfruttare già un ampio set di comandi vocali messi a disposizione da Amazon per e˙ettuare chiamate, messaggi, riprodurre musica, news, impostare sveglie,etc.
La piattaforma Alexa non è stata tuttavia pensata come un ecosistema chiuso. Amazon, infatti, oltre a lavorare lei stessa per espandere questo set di comandi, ha lasciato la libertà di implementare nuove funzionalità anche a terzi, distribuendogli i necessari strumenti di supporto.
Se con Alexa Voice Service infatti si dispone di tutto ciò che serve per portare le funzionalità di Alexa su qualsiasi dispostivo, concentrandosi sul prodotto, Amazon ha pensato di fornire anche un software development framework, con il nome di **Alexa Skill Kit**, che permette di implementare quelle che vengono chiamate **Skills**, ossia nuove funzionalità in grado di espandere le capacità di Alexa, integrando all'occorrenza i propri servizi.

3.1 Skills

Fino a questo punto, ci si è limitati ad analizzare l'interazione con Alexa solo dal punto di vista del dispositivo, so˙ermandosi esclusivamente sull'analisi degli eventi inviati e delle direttive ricevute. Per lo sviluppo delle skills, occorre fare un passo ulteriore e capire i vari passi eseguiti da Alexa per interpretare il comando vocale dell'utente, eseguire la richiesta e restituire una risposta attinente con contenuti di varia natura. L'architettura, sia per le di skill native create da Amazon, sia per le skill sviluppate da terzi, rimane la medesima :

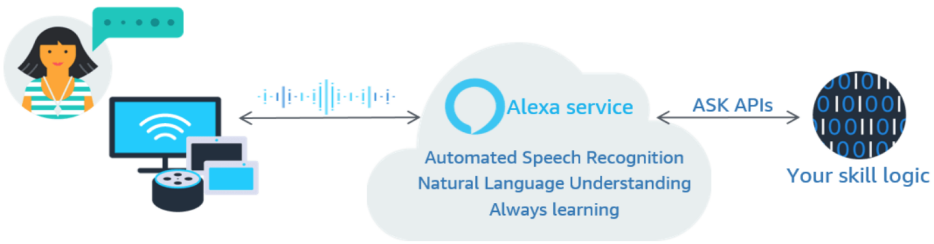


Figura 3.1: Architettura utilizzata per interpretare i comandi vocali

(1) L'utente fa una richiesta ad Alexa e viene mandato l'evento "*SpeechRecognizer.Recognize*" che contiene l'audio del comando pronunciato dall'utente.

(2) A questo punto grazie ai seguenti processi, eseguiti in concomitanza nel cloud, è possibile richiamare la funzione associata alla skill, inerente alla richiesta :

- **Automatic Speech Recognition** : e'ettua la conversione "**Speech-to-Text (STT)**", ossia trasforma il suono del comando vocale nel testo corrispondente dato che gli algoritmi per la comprensione semantica agiscono su un testo e non sull'audio.
- **Natural Language Understanding(NLU)** : permette di decodificare la semantica di una frase in senso lato o senso stretto. Questo corrisponde a una maggiore libertà nel pronunciare un comando inerente a un argomento, facendo percepire all'utente una conversazione più naturale. Ad esempio per chiedere informazione sul meteo, non sarà necessario esprimere esattamente il comando : "che tempo fa fuori?" ma sarà possibile anche un :"Alexa, com'è fuori?".

(3) Il cloud Alexa invia dunque un JSON object alla skill, contente tutti quei parametri e quei dati necessari per eseguire le azioni richieste.

(4) Dopo aver e'ettuato tutte le azioni necessarie, grazie alle ASK API, si possono restituire all'uscita della skill tutte quelle direttive appartenenti alle interfacce illustrate nei capitoli precedenti, tra cui AudioPlayer, SpeechSyntesizer, TemplateRuntime, etc.
Nel caso dello SpeechSyntesizer.Speak, verrà applicato infine il processo di sintesi vocale del contenuto testuale indicato nella API, ossia la conversazione **Text-to-Speech(TTS)**.

3.2 Voice Interaction Models

Ad ogni skill Alexa è associato un **Interaction Model** che definisce le parole e la frase che l'utente può usare per richiamarla e fornirle ulteriori informazioni. Esistono due tipi di Interaction Model :

- **Pre-built voice interaction model** : è definito un insieme fisso di comandi per ogni tipo di skill.

- **Custom voice interaction model** : in questo è lo sviluppatore a definire la frase usata per il comando e ulteriori parole per fornire informazioni aggiuntive alla skill.

3.3 Skill Types

Nell'Alexa Skill Kit le skills sono divise per categorie in base al loro Interaction Model e a quello che è possibile fare con esse. Per ognuna è definita l'interfaccia di appartenenza e la direttiva corrispondente che sarà possibile inviare al dispositivo grazie all'ASK APIs. Dall'altra parte il device dovrà avere il Capability Agent per comprendere le direttive inviate dalla skill e poter comunicare con essa, inviandole gli eventi associati dell'interfaccia.

Tipo	Voice Model	Interfaccia	Descrizione
Automotive	Pre-built and Custom	**VehicleData, Navigation, Management**	Adatta per il settore automotive
Business	Pre-built	**Calendar, MeetingClient-Controller, Reserva-tion.Room**	Permette di controllare gli eventi nel calendario e prenotare delle "meeting rooms"

Cooking	Pre-built	**TimeController, TemperatureController**	Permette di controllare gli elettrodomestici della cucina
Custom	Custom	JSON Object definito dallo sviluppatore	Permettono la comunicazione con i propri servizi.
Education	Pre-built	**Profile.Student, Grade.Course**	Permette a studenti e insegnati di gestire i dati dei corsi, recuperare informazioni sulla classe o sul singolo studente.
Entertainment Device	Pre-built	**KeypadController, ChannelController**	Permette di controllare i device di intrattenimento come SmartTv e Speaker
Flash Briefing	Pre-built	**Feed**	Permette agli utenti di accedere a notizie e aggiornamenti
Games	Custom	**APL, Alexa Web API for Games**	Permette di gestire giochi con la voce e restituire contenuti visivi in base ai comandi ricevuti
List	Custom	**List Events in Alexa Skills, Shopping and To-Do Lists**	Permette di gestire la lista della spesa o cose da fare.
Music, Radio, and Podcast	Pre-built	**Media.Playback, Media.PlayQueue**	Permette di controllare un contenuto audio inviato da qualsiasi servizio streaming

Networking and Wi-Fi	Pre-built	**ConnectedDevice**	Permette di gestire la connessione internet dei propri device
Smart Home	Pre-built	**PowerController, ThermostatController**	Permette di gestire gli smart home device come termostati e lampadine
Smart Home Security	Pre-built	**LockController, SecurityPanelController**	Permette di gestire gli smart home device di sicurezza come telecamere di sorveglienza, serrature e sensori di movimenti
Video	Pre-built	**PlaybackController, VideoRecorder**	Permette di controllare e riprodurre contenuti video di qualsiasi device dotato di display

3.4 Skill Development

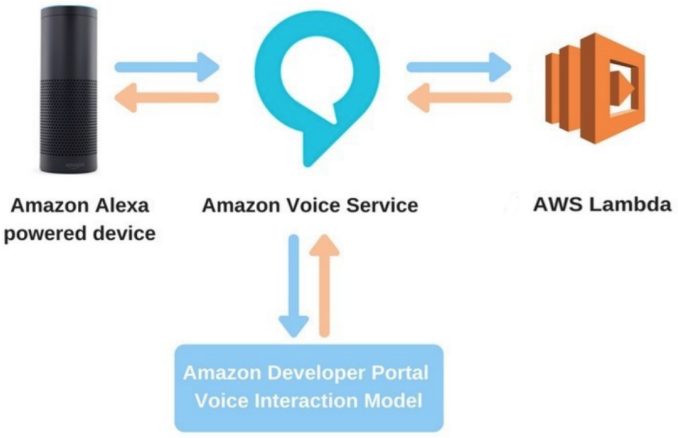


Figura 3.2: Architettura generale per lo sviluppo di una skill

Il funzionamento di ogni skill si basa su un servizio cloud chiamato **Amazon Voice Service**, grazie al quale è possibile eseguire tutti quegli algoritmi necessari al funzionamento di una IA come Alexa, quindi ad esempio le conversioni TTS e STT, l'ASR, l'NLU, la gestione delle connessioni con i clients,l'invio delle direttive,etc.
Quando si sviluppa una skill, il primo passaggio consiste nella definizione di un "*Voice Interaction Model*", in modo da istruire l'engine Alexa a riconoscere la struttura del dialogo della skill. Di fatto si andrà ad e˙ettuare il *deploy* dei metadati, formattati seguendo lo standard JSON. Per far questo Amazon fornisce agli sviluppatori uno strumento a linea di comando, l' **ASK CLI**, oppure una console di sviluppo semplificata, con il nome di **Amazon Developer Portal**.[10]

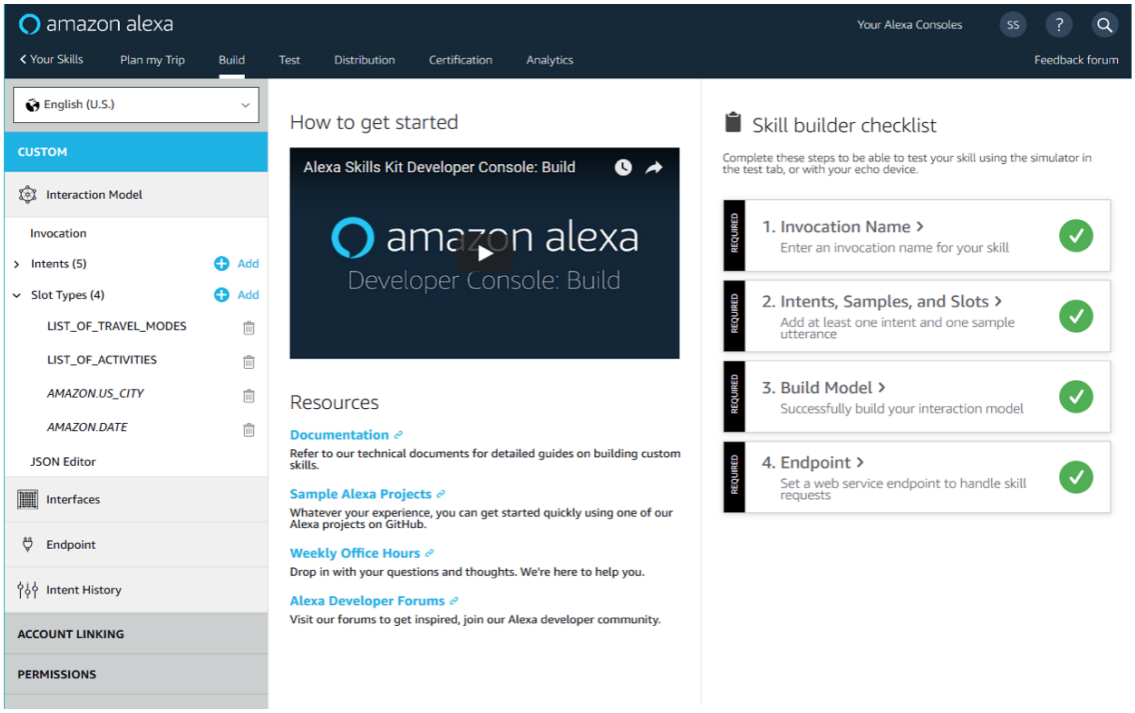


Figura 3.3: Amazon Developer Portal

E˙ettuato il *deploy*, Alexa sarà in grado di riconoscere i comandi della skill ma ad ognuno di essi non corrisponde ancora nessuna azione. Per implementare quindi la logica associata a ogni fase del dialogo, occorre legare la skill con un Web Service, sul quale sarà presente l'**ASK SDK**, che permette di gestire la comunicazione con l'Amazon Voice Service ed eseguire il codice corrispondente al comando. Anche in questo caso Amazon fornisce uno servizio cloud della suite **Amazon Web Services**, chiamato **AWS Lambda**, che

permette di eseguire il codice per qualsiasi tipo di servizio back-end (*Alexa Skill Kit, WebSockets, WebService, etc..*) senza alcuna amministrazione. Ci penserà la piattaforma ad eseguirlo in modo ottimizzato, curandosi del bilanciamento del carico in base al traÿco in entrata e richieste ricevute. Lo sviluppatore dovrà semplicemente configurare l'ambiente nell'AWS Console, implementare il codice adeguato e integrare le librerie necessarie a creare il servizio back-end.
Nel caso della skill Alexa, occorre selezionare il servizio *Alexa Skill Kit* che permette di caricare le librerie dell'ASK SDK e creare il servizio automaticamente. Inoltre è necessario collegare la skill con l'AWS Lambda. Per far questo, si utilizza un parametro univoco identificativo presente nell'AWS Console, ossia l'**Amazon Resource Names (ARN)**. Nell'Amazon Developer Portal, durante la fase di configurazione della skill, sarà possibile andare a inserire questo parametro, definendolo come endopoint collegato alla skill.

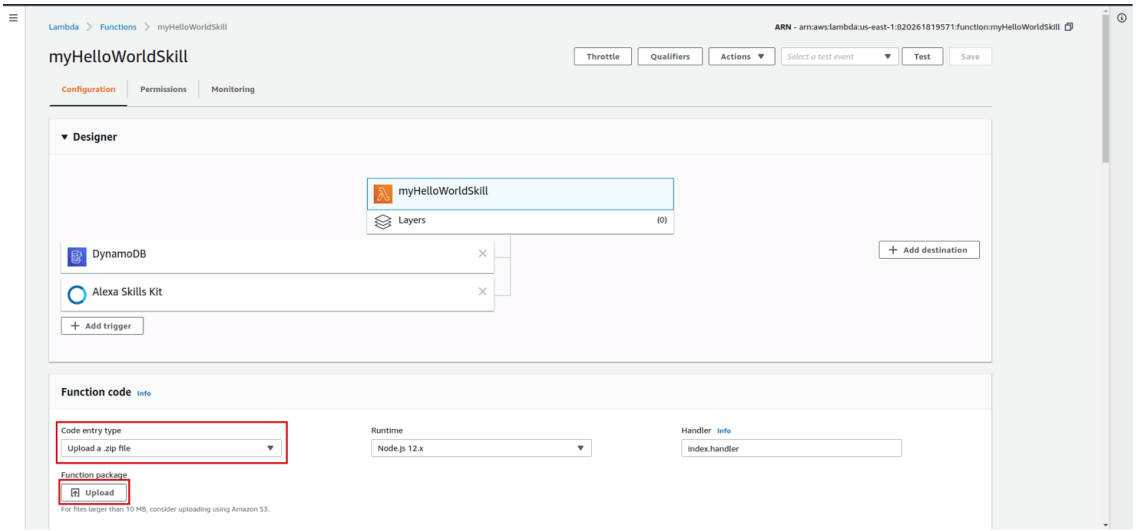


Figura 3.4: AWS Console

Quando tutto è configurato, il processo per quella determinata skill sarà il seguente :

(1) L'utente pronuncia il comando, verrà inviato l'evento *SpeechRecognizer.Recognize* con annesso l'audio catturato dal microfono

(2) L'Amazon Voice Service analizza l'audio ed e˙ettua lo STT. Avendo a questo punto a disposizione il testo del comando dell'utente e basandosi sui metadati di tutte le Skill presenti, in particolare sul comando di

invocazione che sarà univoco tra tutte le skill, cerca una corrispondenza con una di esse.

(3) In questo caso specifico, verrà trovata la corrispondenza con la skill creata dallo sviluppatore e a questo punto si richiama la sua logica associata nell'AWS Lambda.

(4) Eseguito il codice della funziona nell'AWS Lambda e sfruttando le ASK API, viene ritornata una risposta di testo all'Amazon Voice Service che conterrà le indicazioni su quale direttive inviare e il loro relativo contenuto

(5) L'Amazon Voice Service prepara la risposta JSON con tutte le direttive, formattate secondo lo standard, in modo che il device le capisca e le invia al client.

3.4.1 Account Linking

Il meccanismo di Account Linking permette di identificare un utente Alexa in un altro dominio. Per esempio, si supponga che attraverso una skill si voglia gestire, con il comando vocale opportuno, un proprio device compatibile con Alexa o effettuare la prenotazioni di un servizio taxi. In questo caso, per compiere l'azione nella funzione Lambda associata, occorrerà effettuare delle richieste HTTP autenticate verso un altro WebService. Queste richieste conterranno un parametro identificato, chiamato **Access Token**, per identificare l'utente appartenente al dispositivo che si vuole controllare o per risalire ai suoi dati di pagamento nel caso del servizio di prenotazioni taxi. Il meccanismo di Account Linking, in fase di attivazione di una skill, permetterà di ottenere questo parametro obbligatorio per i seguenti tipi di skill : Smart Home, Video, Meeting, Education. Quando l'utente abilita la skill dallo Store si troverà davanti la schermata seguente in cui gli sarà richiesto di inserire username e password per utilizzare quel determinato servizio.

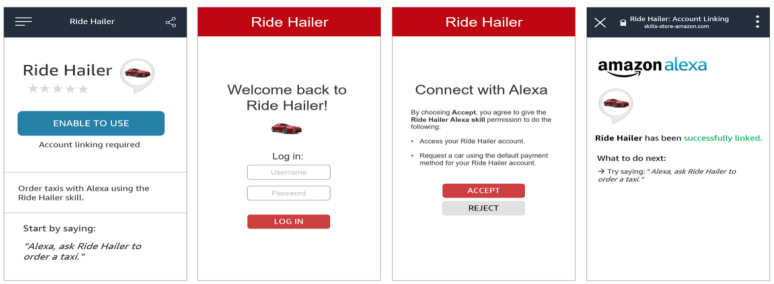


Figura 3.5: Schermate dell'app uÿciale Alexa per abilitare una skill

Il seguente flowchart descrive ciò che avviene dietro a ogni azione dell'utente

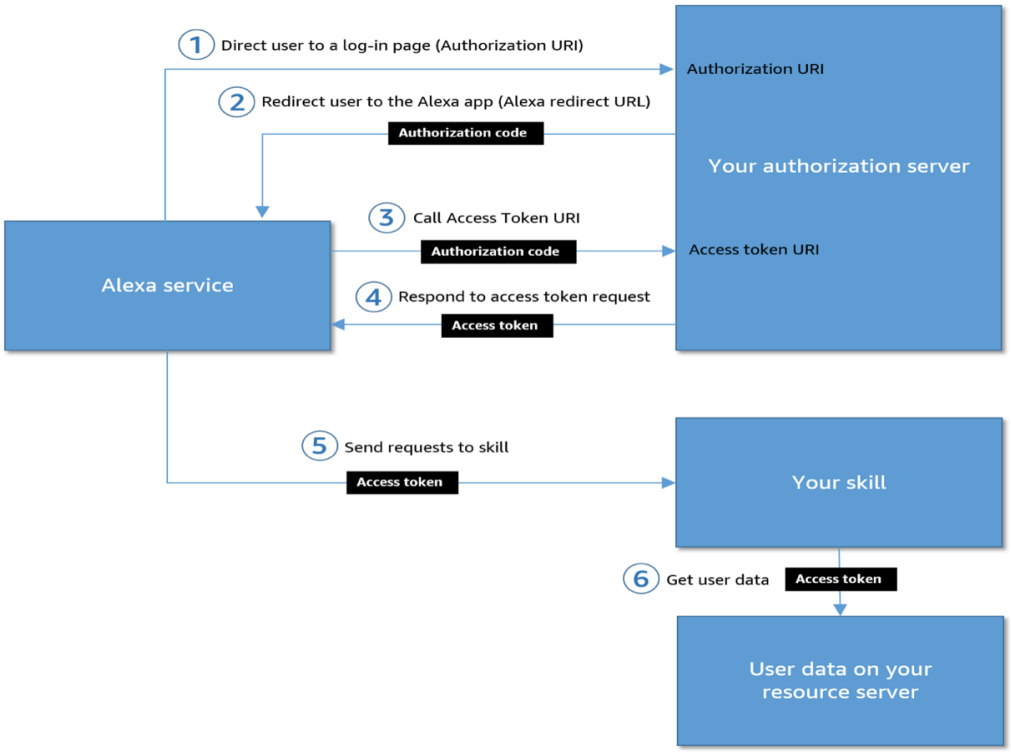


Figura 3.6: Oauth2 Authentication per Alexa

(1) L'utente clicca su "ENABLE TO USE", Alexa Service accede all'**Authorization URI**. L'**Authorization Server** presenterà la schermata di login in cui bisognerà inserire username e password per accedere al servizio.

(2) Quando l'utente inserisce le sue credenziali corrette e accetta i termini, l'*Authorization Server* reindirizza l'utente all'**Alexa redirect URL**, mostrandogli la pagina di successo e restituisce ad Alexa Service l'**Authorization Code**.

(3) Alexa Service richiederà l'*Access Token* all'*Authorization Server*, effettuando una chiamata HTTP GET al link l'**Access Token URI** con i parametri :

- Authrization Code
- ClientId
- ClientSecret

L'*Access Token URI* sarà utilizzato anche per il refreshing dell'*Access Token* se scaduto.

(4) A questo punto tutte le volte che l'utente e˙ettua una richiesta alla skill, alla funzione Lambda associata sarà passato come parametro anche l'*Access Token* per sfruttare i servizi del WebService per il quale sono previste richieste autenticate.

(5) Di seguito la schermata di impostazione dell'Account Linking in fase di creazione di una skill :

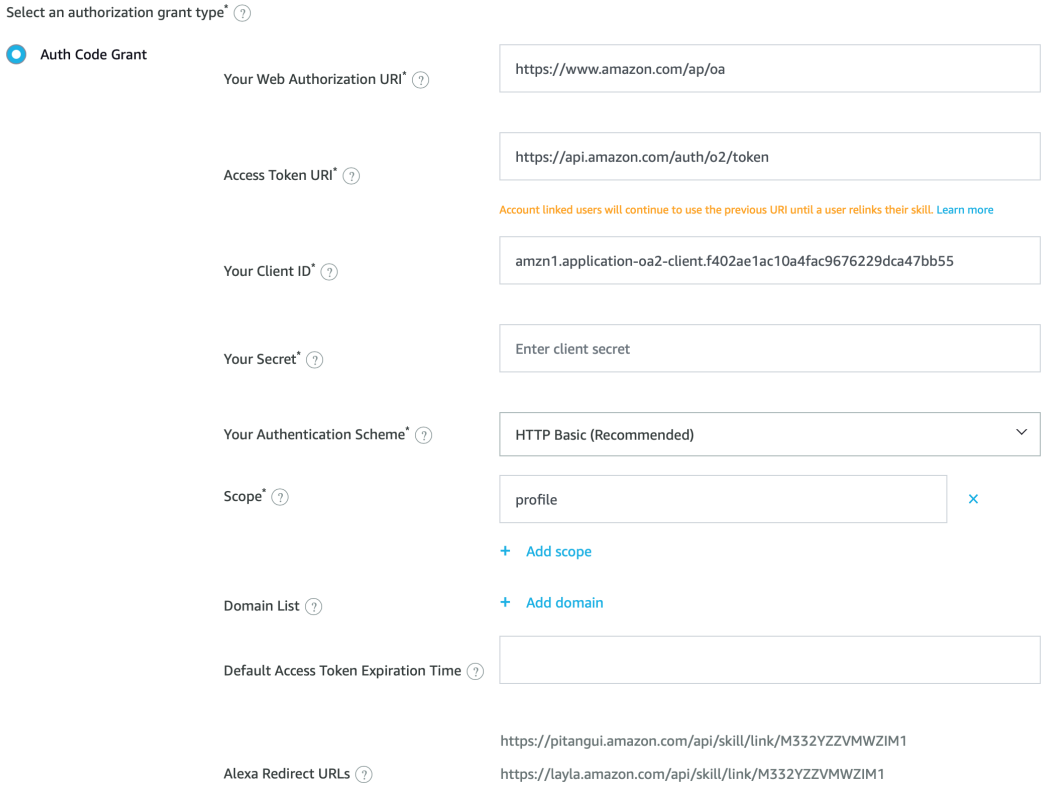


Figura 3.7: Impostazioni Account Linking

3.4.2 Smart Home Skills

Queste skill sono utilizzate quando si vuole controllare un device a distanza con Alexa. Inoltre tutti gli altri tipi di skill derivano da questo. Il Voice Interaction Model è già definito, dunque i comandi non sono creati dallo sviluppatore, ma fissati a priori. In particolare sarà possibile chiedere ad esempio :

- Alexa, abbassa la luce del bagno
- Alexa, imposta le luce dell'abitacolo al 50%
- Alexa, imposta la luce della camera da letto sul rosso
- Alexa, rendi il soggiorno di un bianco caldo.
- Alexa, accendi/spegni nome dispositivo
- Alexa, imposta la potenza della ventola al 40%
- Alexa, accendi la mia giornata(per attivare un gruppo di dispositivi)
- Alexa, imposta il termostato a 20
- Alexa, metti in pausa il microonde
- Alexa, imposta il ciclo di lavaggio su cotoni

La seguente figura mostra l'architettura dietro al funzionamento di una Smart Home Skill, dalla quale è possibile anche capire i passi necessari per una sua implementazione e sviluppo :

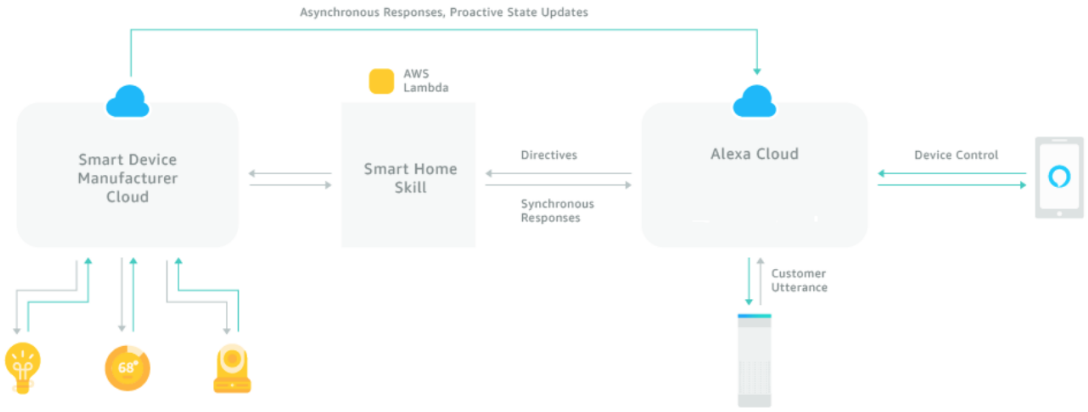


Figura 3.8: Architettura Smart Home SKill

(1) Come prima cosa, si accede all'Alexa Developer Console e si crea la skill, selezionado Smart Home Skill e impostando il nome della skill. Dopodichè si imposta l'ARN endpoint, in modo da collegare la skill con la relativa AWS Lambda, e l'Account Linking, obbligatorio per questo genere di skill.

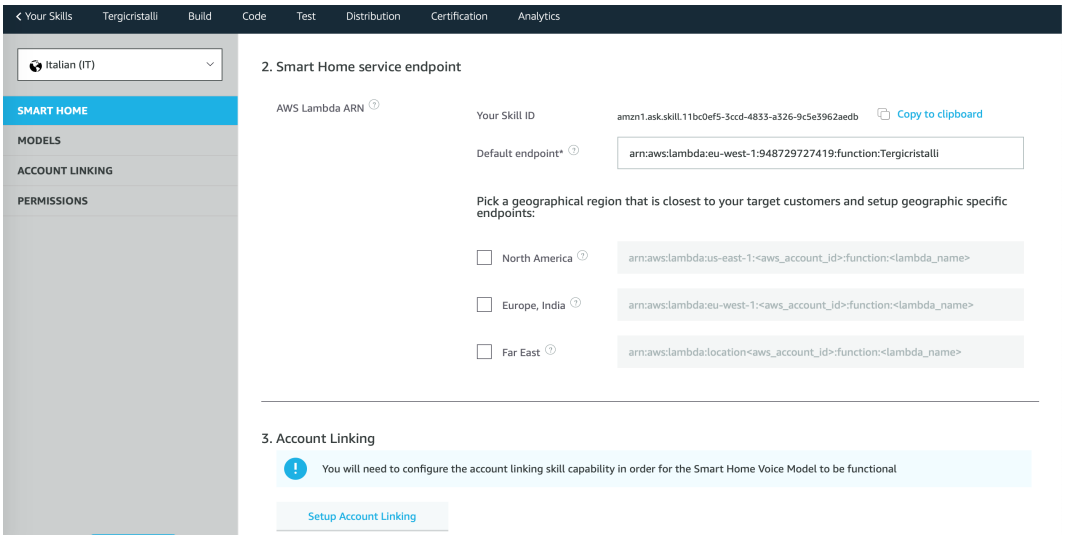


Figura 3.9: Alexa Console

(2) Nell'AWS Console, una volta selezionata il servizio Lambda, si crea la funzione della skill, il trigger Alexa Smart Home, la si lega allo Skill Model con il parametro *Skill ID* e infine si caricano le librerie della ASK SDK per utilizzare le APIs che permettono di comunicare con Alexa Cloud e dunque con il dispositivo. Risulterà infine come in figura 3.9. **Alexa Cluod** avrà 2 compiti :

- Comunicare con il device dal quale l'utente ha dato il comando. Quindi accettare l'evento *SpeechRecognizer.Recognize* e analizzare cosa è stato chiesto. Infine comunicare al device che la skill ha eseguito il compito, inviandogli una direttiva *SpeechSyntesizer.Speak*
- Comunicare con l'AWS Lambda. In questo caso la comunicazione avviene nuovamente mediante eventi e direttive. Quando Alexa Cloud riconosce cosa l'utente vuole, invia una direttiva, che darà indicazioni all'AWS Lambda riguardo a ciò che l'utente ha chiesto e sarà dunque richiamata la funzione che gestisce quel determinato comando. Dopo che la skill ha gestito il comando e controllato il device desiderato, viene avvisato Alexa Cloud, inviando un evento in maniera **sincrona** dall'interno della skill stessa oppure in modo **asincrono** dalla funzione di gestione del dispositivo all'interno del web service.

(3) Il compito dell'utente a questo punto sarà quello di andare a registrare un account e i dati del proprio dispositivo sul sito del produttore che

gestisce quei determinati dispositivi. Dopodichè attiverà la skill dall'App Uÿciale Alexa, e˙ettuando l'Account Linking.

(4) Per poter identificare e iniziare a comandare e˙ettiv amente il device, occorrerà cercare i dispositivi dall'App Uÿciale o chiedere ad Alexa "Scopri i miei dispositivi".
Inizierà dunque la fase di **discovery**, una fase comune per tutte le Smart Home Skill. Alexa, infatti, invierà all'AWS Lambda una direttiva *Alexa.Discovery* :


```
{
    "directive": {
        "header": {
            "namespace": "Alexa.Discovery",
            "name": "Discover",
            "payloadVersion": "3",
            "messageId": "879f8019-a57e-4de4-96a6-2451841847b0"
        },
        "payload": {
            "scope": {
                "type": "BearerToken",
                "token": "Atza|IwEBIIx1ETuEEKEcANhqkWHiFRT8IjQyeDAPFwB4KebcOW-
a1iooJ2HOxl_k74VQvRlbSzRgaIu6WwTfVoWPoWneCJkFHj0efdr4f2EK3olQVxCetKv3Z9U6q9u650r3vcD-
0KMLkcPqhmB7LQwVEDZrBJ3PW2BVPo89P3Ckyobxd7pV7_fdnmwHhLDaQfzXbeV7f9SwShVfghdQlMSF1gu_pxMAbxzPnO4olMDnPdgthoF4hrSKSuMNg9L2V
oDVesSLofRhZmhqUT8FycUm-ggELGJoXvopSQIhGsdlrTiYBITc7OsWRHbqwRS7RF0uohizIW-s311LAE_RRYxMNlAXiOvZlK1ZYPf59JoUi-
H7RTHB6L0DKsrc3RDAT3YsSsDh8Tvah7xzcnTv69SCtzsPkmpgwVYi4aUhuUaWiJtl0ORihQ"
            }
        }
    }
}
```


Nella funzione dell'AWS Lambda per una Smart Home Skill verrà riconosciuta la direttiva, e˙ ettuando il parsing del JSON Object.

```
exports.handler = function (request, context) {

    if (request.directive.header.namespace === 'Alexa.Discovery' && request.directive.header.name === 'Discover') {
        log("DEBUG:", "Discover request",  JSON.stringify(request));
        handleDiscovery(request, context, "");
    }
    else if (request.directive.header.namespace === 'Alexa.PowerController') {
        if (request.directive.header.name === 'TurnOn' || request.directive.header.name === 'TurnOff') {
            log("DEBUG:", "TurnOn or TurnOff Request", JSON.stringify(request));
            handlePowerControl(request, context);
        }
    }

    function handleDiscovery(request, context) {

        var payload = getDevices(request);//richiesta per ottenere i dati dei devices associati all'utente dall'Web Service
        var header = request.directive.header;
        header.name = "Discover.Response";
        log("DEBUG", "Discovery Response: ", JSON.stringify({ header: header, payload: payload }));
        context.succeed({ event: { header: header, payload: payload } });
    }
```

Figura 3.10: Funzione AWS Lambda di una Smart Home Skill

La funzione *getDevices(request)* andrà a fare una richiesta HTTP verso il Web Service del produttore per ottenere i dispositivi associati all'utente. L'*Access Token*, ottenuto durante l'Account Linking e contenuto nella direttiva *Discovery*, sarà fornito durante la richiesta per identificare l'utente. Con la funzione *context.succeed* si invia l'evento di risposta all'Alexa Cloud. L'evento dovrà avere il seguente formato :

```
{
  "event": {
    "header": {
      "namespace": "Alexa.Discovery",
      "name": "Discover.Response",
      "payloadVersion": "3",
      "messageId": "<message id>"
    },
    "payload": {
      "endpoints": [
        {
          "endpointId": "<unique ID of the endpoint>",
          "manufacturerName": "Sample Manufacturer",
          "description": "Smart Light by Sample Manufacturer",
          "friendlyName": "Living Room Light",
          "additionalAttributes":  {
            "manufacturer" : "Sample Manufacturer",
            "model" : "Sample Model",
            "serialNumber": "<the serial number of the device>",
            "firmwareVersion" : "<the firmware version of the device>",
            "softwareVersion": "<the software version of the device>",
            "customIdentifier": "<your custom identifier for the device>"
          },
          "displayCategories": ["LIGHT"],
          "cookie": {},
          "capabilities": [
            {
              "type": "AlexaInterface",
              "interface": "Alexa.PowerController",
              "version": "3",
              "properties": {
                "supported": [
                  {
                    "name": "powerState"
                  }
                ],
                "proactivelyReported": true,
                "retrievable": true
              }
            },
```

```
        {
          "type": "AlexaInterface",
          "interface": "Alexa.BrightnessController",
          "version": "3",
          "properties": {
            "supported": [
              {
                "name": "brightness"
              }
            ],
            "proactivelyReported": true,
            "retrievable": true
          }
        },
        {
          "type": "AlexaInterface",
          "interface": "Alexa",
          "version": "3"
        }
      ]
    }
```

In questo evento i 3 parametri principali sono :

- **endpointId** : deve essere univoco per identificare il dispositivo nell'Alexa Cloud e sul WebService per poter richiamare azioni, mediante chiamate HTTP, sul quel particolare device.
- **firendlyName** : anche questo è un parametro che deve essere univoco. Viene utilizzato per esprimere nel comando vocale il nome del dispositivo che l'utente vuole comandare per permettere ad Alexa di individuarlo.
- **capabilities** : l'array delle capabilities permette ad Alexa di capire quale interfacce sono abilitate sul device e dunque quale tipo di *skill pre-built* è possibile attivare. Ad esempio inviando l'interface *Alexa.PowerController*, saranno abilitati i comandi di spegnimento e accensione del dispositivo, quindi l'utente potrà chiedere : "Alexa, spegni friendlyName". Un altro esempio riguarda le **Entertainment Device Skills**. Inviando nelle capabilites l'interfaccia *Alexa.ChannelController*, Alexa capirà che con quel dispositivo è possibile cambiare canale e dunque saranno abilitati i comandi come : "Alexa, metti su Rai 1 sulla 'friendlyName'".

(5) Quando il dispositivo è registrato nel WebService e in Alexa, l'utente può a questo punto comandarlo esprimendo un comando da qualsiasi device Alexa abilitato. Nel caso del PowerController, alla richiesta : "Alexa, accendi l'auto", Alexa Cloud invierà una direttiva *Alexa.PowerController*

```
{
  "directive": {
    "header": {
      "namespace": "Alexa.PowerController",
      "name": "TurnOn",
      "messageId": "<message id>",
      "correlationToken": "<an opaque correlation token>",
      "payloadVersion": "3"
    },
    "endpoint": {
      "scope": {
        "type": "BearerToken",
        "token": "<an OAuth2 bearer token>"
      },
      "endpointId": "<endpoint id>",
      "cookie": {}
    },
    "payload": {}
  }
}
```

Il corpo di questa direttiva verrà analizzata come in 3.10 e al campo *namespace* si troverà *Alexa.PowerController*, quindi con la funzione "*handlePowerControl* si gestirà la richiesta. In particolare, recuperato l'endpointId, l'*access token* e l'azione dalla direttiva, si e˙ettue rà una richiesta POST autenticata al WebService che avendo aperta una connessione persistente full-duplex, utilizzando ad esempio una tecnologia WebSocket, potrà inviare l'azione al device e comandarlo. Ricevendo una risposta a˙ermativ a dal WebService, la skill dovrà inviare un evento di risposta all'Alexa Cloud per avvisarla che l'azione è stata eseguita con successo.

```
{
  "event": {
    "header": {
      "namespace": "Alexa",
      "name": "Response",
      "messageId": "<message id>",
      "correlationToken": "<an opaque correlation token>",
      "payloadVersion": "3"
    },
```

```
    "endpoint": {
      "scope": {
        "type": "BearerToken",
        "token": "<an OAuth2 bearer token>"
      },
      "endpointId": "<endpoint id>"
    },
    "payload": {}
  },
  "context": {
    "properties": [
      {
        "namespace": "Alexa.PowerController",
        "name": "powerState",
        "value": "OFF",
        "timeOfSample": "2017-02-03T16:20:50.52Z",
        "uncertaintyInMilliseconds": 500
      }
    ]
  }
}
```


(6) Alexa Cloud invierà una direttiva *SpeechSyntesizer.Speak* al device dal quale è arrivato il comando e l'utente potrà ascoltare la risposta a˙ermativa di Alexa.

3.4.3 Custom Skills

Questo tipo di skill è particolarmente indicato nel caso in cui si vogliano o˙rir e le funzionalità di un servizio come ad esempio prenotazione di biglietti per un evento, richiesta di informazioni riguardo al traÿco, statistiche di visione di un contenuto streaming. La principale di˙erenza con le Smart Home Skill sta nel Voice Interaction Model. Non si ha più un *pre-built model* ma un *custom model*, quindi sarà compito dello sviluppatore definire il dialogo e i comandi. Inoltre in questo caso il Linking Account non è un requisito necessario, poiché potendo liberamente definire la conversazione con Alexa, non è obbligatorio fornire anche il nome del device nella richiesta. Ecco perchè non è prevista neanche la fase iniziale di *discovery* che permette di andare a comunicare all'Alexa Cloud i dati dei dispositivi associati all'utente, in particolare il *friendlyName*. Lo "*User Interaction Flow*" risulta dunque più semplificato. Infatti non si parla più di direttive inviate da Alexa Cloud alla funzione Lambda che dovrà rispondere in modo *sincrono* o *asincrono*, inviando lo specifico evento, anche se tuttavia i principi di funzionamento e il metodo di implementazione rimangono pressoché identici. Così come per le Home

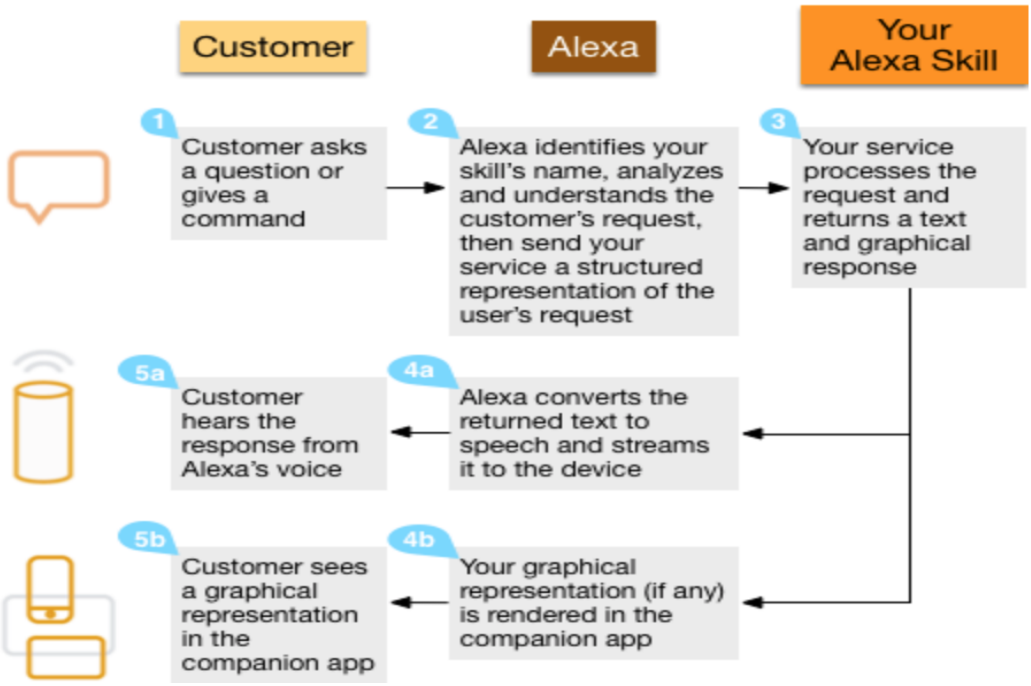


Figura 3.11: Architettura Custom Skill

Skills, per le Custom Skills occorre definire nell'Alexa Developer Console l'Interaction Model e successivamente legarla a una funzione Lambda in cui si implementano i passaggi necessari per servire la richiesta dell'utente. A questo punto :

(1) L'utente esprime un comando

(2) Alexa Service individua la Custom Skill e invia una *richiesta* alla funzione Lambda, strutturata con tutti i dati necessari a comprendere cosa ha chiesto l'utente.

(3) Nella funzione Lambda, si gestisce la *richiesta* e viene inviata una *risposta* appropriata, formattata secondo lo standard JSON, nella quale sarà presente del contenuto audiovisivo. In questo caso l'ASK SDK mette a disposizione delle API per aiutare lo sviluppatore a implementare queste risposte.

(4) Alexa Service e˙ettua la conversione TTS di ciò che è stato indicato nella risposta audio della skill sotto forma di testo e invia un multipart message con le direttive appropriate per essere comprese e riprodotte sul Device.

Intents, Slots, Dialog Dopo aver definito il nome della skill e il tipo di modello nella Alexa Developer Console, lo sviluppatore definirà il dialogo. Esso è composto da vari componenti :

- **Invocation Name** : è la frase che invoca la skill e con la quale si inizia la conversazione con essa.
- **Intent** : il nome dell'azione richiesta alla skill.
- **Sample utterances** : a ogni intento corrisponde una o più frasi esatte che l'utente può utilizzare per indicare alla skill ciò che vuole. Deve essere strutturata in modo da risultare più esaustiva possibile
- **Slot** : in ogni intento è possibile utilizzare in un punto della frase una parola piuttosto che un'altra per dare indicazioni sul contenuto. Questo insieme di valori utilizzabili in un singolo punto è chiamato *slot*. E' possibile avere valori **predefiniti** da Amazon, come giorni della settimana, giorni del mese, etc oppure valori **custom**, quindi definiti dallo sviluppatore.

Il *Voice Interaction Modell* sarà rappresentato da una struttura JSON e potrà essere modificato o tramite la procedura guidata nell'Alexa Console oppure modificando direttamente questa struttura. Di seguito è riportata una skill che gestisce il canale di una piattaforma livestream al quale l'utente può richiedere quante visualizzazione hanno avuto i suoi video in totale o in un determinato mese e a quanto ammontano i suoi guadagni. Anche in questo caso la skill prevederà l'utilizzo dell'Account Linking perchè per potere e˙ettuare richieste al WebService che gestisce questi dati, sarà necessario fornire nelle chiamate HTTP un parametro che identifichi l'utente.

```
{
    "interactionModel": {
        "languageModel": {
            "invocationName": "mio gestore statistiche",
            "intents": [
                {
                    "name": "AMAZON.CancelIntent",
                    "samples": []
                },
                {
                    "name": "AMAZON.HelpIntent",
                    "samples": []
                },
                {
                    "name": "AMAZON.StopIntent",
                    "samples": []
                },
                {
                    "name": "AMAZON.NavigateHomeIntent",
                    "samples": []
                },
```

```
22              {
23                  "name": "TotaleViews",
24                  "slots": [
25                      {
26                          "name": "azioni",
27                          "type": "azione"
28                      }
29                  ],
30                  "samples": [
31                      "{azioni} il totale delle
                          visualizzazione dei miei video"
32                  ]
33              },
34              {
35                  "name": "MonthViews",
36                  "slots": [
37                      {
38                          "name": "azioni",
39                          "type": "azione"
40                      },
41                      {
42                          "name": "mese",
43                          "type": "AMAZON.Month"
44                      }
45                  ],
46                  "samples": [
47                      "{azioni} quante visualizzazione ci
                          sono state a {mese}"
48                  ]
49              }
50          ],
51          "types": [
52              {
53                  "name": "azione",
54                  "values": [
55                      {
56                          "name": {
57                              "value": "conteggiare"
58                          }
59                      },
60                      {
61                          "name": {
62                              "value": "contare"
63                          }
64                      },
65                      {
66                          "name": {
67                              "value": "calcolare"
68                          }
69                      }
70                  ]
71              }
```

Questa skill ha come Invocation Name "mio gestore statistiche", dunque potrà essere richiamata con "Alexa lancia/chiedi/apri mio gestore statistiche". In ogni Custom Skill sono definiti vari *intenti built-in*, come ad esempio "Amazon.HelpIntent" che sarà richiamato nel momento in cui l'utente chiede aiuto alla skill. A questi *intenti* predefiniti sono stati aggiunti due *intenti custom* il cui nome sarà utilizzato nella funzione Lambda per identificare il

tipo di azione richiesta dall'utente. In particolare :

- **TotalViews** : serve per calcolare il numero totale di visualizzazioni ricevute nel canale livestreaming dell'utente. All'interno della frase del comando è stato definito un solo Slot. Quindi l'utente potrà utilizzare la frase "<azioni> il totale delle visualizzazione dei miei video", dove ad "azioni" può corrispondere una parola che fa parte del gruppo "azione" : calcolare, conteggiare, contare.

- **MonthViews** : serve per calcolare il numero totale di visualizzazione ricevute nel canale livestreaming dell'utente per un dato mese. In questo caso la frase del comando corrisponderà a <azioni> quante visualizzazione ci sono state a <mese>. Anche qui i valori possibili dello slot "azioni" rimangono i medesimi mentre per "mese" i valori corrisponderanno ai mesi dell'anno.

A questo punto definito l'Account Linking e il collegamento all'endpoint con l'ARN della funzione Lambda, si può procede a implementare la logica della Custom Skill, definendo innanzitutto un gestore di eventi :

```
const Alexa = require('ask-sdk-core');

const LaunchRequestHandler = {
  canHandle(handlerInput) {
    const request = handlerInput.requestEnvelope.request;
    return request.type === 'LaunchRequest';
  },
  handle(handlerInput) {
    return handlerInput.responseBuilder
      .speak('Certo!Come posso aiutarti?')
      .reprompt()
      .getResponse();
  },
};

const MonthViewsHandler = {
  canHandle(handlerInput) {
    const request = handlerInput.requestEnvelope.request;
    return request.type === 'IntentRequest'
        && request.intent.name === 'MonthViews';
  },
  handle(handlerInput) {
    var accessToken = handlerInput.requestEnvelope.context.System.user.accessToken;
    const speakOutput = calculateMonthView( handlerInput.requestEnvelope.request,accessToken);
    return handlerInput.responseBuilder
      .speak(speakOutput)
      .getResponse();
  },
};
```

```
30
31  const TotaleViewsHandler = {
32    canHandle(handlerInput) {
33      const request = handlerInput.requestEnvelope.request;
34      return request.type === 'IntentRequest'
35          && request.intent.name === 'TotaleViews';
36    },
37    handle(handlerInput) {
38      var accessToken = handlerInput.requestEnvelope.context.System.user.accessToken;
39      const speakOutput = calculateTotalView( handlerInput.requestEnvelope.request,accessToken);
40      return handlerInput.responseBuilder
41        .speak(speakOutput)
42        .getResponse();
43    },
44  };
45
46  const HelpHandler = {
47    canHandle(handlerInput) {
48      const request = handlerInput.requestEnvelope.request;
49      return request.type === 'IntentRequest'
50        && request.intent.name === 'AMAZON.HelpIntent';
51    },
52    handle(handlerInput) {
53      const requestAttributes = handlerInput.attributesManager.getRequestAttributes();
54      return handlerInput.responseBuilder
55        .speak("Dammi un comando")
56        .reprompt()
57        .getResponse();
58    },
59  };
60  const ExitHandler = {
61    canHandle(handlerInput) {
62      const request = handlerInput.requestEnvelope.request;
63      return request.type === 'IntentRequest'
64        && (request.intent.name === 'AMAZON.CancelIntent'
65          || request.intent.name === 'AMAZON.StopIntent');
66    },
67    handle(handlerInput) {
68      const requestAttributes = handlerInput.attributesManager.getRequestAttributes();
69      return handlerInput.responseBuilder
70        .speak(requestAttributes.t('STOP_MESSAGE'))
71        .getResponse();
72    },
73  };
74
75  const SessionEndedRequestHandler = {
76    canHandle(handlerInput) {
77      const request = handlerInput.requestEnvelope.request;
78      return request.type === 'SessionEndedRequest';
79    },
80    handle(handlerInput) {
81      console.log(`Session ended with reason: ${handlerInput.requestEnvelope.request.reason}`);
82      return handlerInput.responseBuilder.getResponse();
83    },
84  };
85
86  const ErrorHandler = {
87    canHandle() {
88      return true;
89    },
90    handle(handlerInput, error) {
91      console.log(`Error handled: ${error.message}`);
92      console.log(`Error stack: ${error.stack}`);
93      const requestAttributes = handlerInput.attributesManager.getRequestAttributes();
```

```
94      return handlerInput.responseBuilder
95        .speak(requestAttributes.t('ERROR_MESSAGE'))
96        .reprompt(requestAttributes.t('ERROR_MESSAGE'))
97        .getResponse();
98    },
99  };
100
101 const skillBuilder = Alexa.SkillBuilders.custom();
102
103 exports.handler = skillBuilder
104   .addRequestHandlers(
105     LaunchRequestHandler,
106     MonthViewsHandler,
107     TotaleViews,
108     HelpHandler,
109     ExitHandler,
110     SessionEndedRequestHandler,
111   )
112   .addErrorHandlers(ErrorHandler)
113   .withCustomUserAgent('sample/basic-fact/v2')
114   .lambda();
115
```

L'utente per utilizzare le capacità della skill avrà 2 scelte :

(1) Aprire la skill con il comando di apertura : "Alexa lancia/chiedi/apri mio gestore statistiche". Alexa riconoscerà quindi il tipo di comando e dunque invierà una richiesta nella quale sarà indicato come "type" **LaunchRequest**.

```
{
  "type": "LaunchRequest",
  "requestId": "string",
  "timestamp": "string",
  "locale": "string"
}
```

Grazie alle API *speak()* e *reprompt()*, verrà chiesto all'utente cosa vuole fare, mandando una direttiva *SpeechSyntesizer.Speak* e si rimarrà in attesa di una risposta.

(2) Un'ulteriore opzione consiste nell'aprire la skill invocando direttamente l'*intento* richiesto come ad esempio : "Alexa, chiedi al mio gestore statistiche di calcolare quante visualizzazioni ci sono state a gennaio". In questo caso nella richiesta di Alexa inviata alla funzione lambda ci sarà alla voce type il valore **IntentRequest** e alla voce name il nome dell'*intento* **MonthViews**, con i relativi valori utilizzati dall'utente.

Inoltre sarà presente anche il campo *token* nell'oggetto *System.user* per poter effettuare la richiesta del dato desiderato al WebService che gestisce il servizio visualizzazioni. Infine ricevuto il dato con la funzione *calculateMonthView*, grazie all'API *speak()* si invia la risposta all'utente.


```
{
    "type": "IntentRequest",
    "requestId": "amzn1.echo-api.request.6a5309d4-ec15-4dec-8fd9-1f907a9a359b",
    "locale": "it-IT",
    "timestamp": "2021-02-03T22:02:13Z",
    "intent": {
        "name": "MonthViews",
        "confirmationStatus": "NONE",
        "slots": {
            "azioni": {
                "name": "azioni",
                "value": "calcolare",
                "resolutions": {
                    "resolutionsPerAuthority": [
                        {
                            "authority": "amzn1.er-authority.echo-sdk.amzn1.ask.skill.7f06b930-1dae-4745-8cb1-7de30af66dda.azione",
                            "status": {
                                "code": "ER_SUCCESS_MATCH"
                            },
                            "values": [
                                {
                                    "value": {
                                        "name": "calcolare",
                                        "id": "9b64ddab9cf410832e2e30da7eb99c97"
                                    }
                                }
                            ]
                        }
                    ]
                },
                "confirmationStatus": "NONE",
                "source": "USER",
                "slotValue": {
                    "type": "Simple",
                    "value": "calcolare",
                    "resolutions": {
                        "resolutionsPerAuthority": [
                            {
                                "authority": "amzn1.er-authority.echo-sdk.amzn1.ask.skill.7f06b930-1dae-4745-8cb1-7de30af66dda.azione",
                                "status": {
                                    "code": "ER_SUCCESS_MATCH"
                                },
                                "values": [
                                    {
                                        "value": {
                                            "name": "calcolare",
                                            "id": "9b64ddab9cf410832e2e30da7eb99c97"
                                        }
```

```
                                }
                            ]
                        }
                    ]
                }
            }
        },
        "mese": {
            "name": "mese",
            "value": "gennaio",
            "confirmationStatus": "NONE",
            "source": "USER",
            "slotValue": {
                "type": "Simple",
                "value": "gennaio"
            }
        }
    }
  }
}
```

Gli *intenti pre-built* sono gestiti quando l'utente dà dei comandi predefiniti durante la conversazione. La Request inviata alla funzione Lambda segue la corrispondenza IntentRequest - Comando:

- AMAZON.HelpIntent -> Aiuto!
- AMAZON.ExitHandler -> Esci! o Stop!
- AMAZON.SessionEndedRequestHandler -> Inviata quando la conversazione è finita
- AMAZON.ErrorHandler -> se si verifica un errore durante la conversazione

Capitolo 4

Alexa Auto

Come accennato nella fase introduttiva, l'obiettivo ultimo di questo trattato, data anche la natura dell'azienda presso la quale si è svolta la tesi, consiste nell'integrare un'assistente vocale come Alexa su un prototipo di Infotainment. Si è posto inoltre l'accento sull'importanza di una fase di studio preliminare dei concetti principali e dell'architettura che stanno alla base di essa, qualsiasi sia il tipo di dispositivo sul quale si vuole portare questa tecnologia.
In questo capitolo si analizzeranno le soluzione adottate per conseguire l'obiettivo del lavoro.

4.1 Strumenti

4.1.1 Hardware

La piattaforma hardware utilizzata è la Snapdragon Automotive Development Platform (ADP) basata sul processore Snapdragon SA8155P e fornita da Qualcomm. Tale piattaforma permette l'accesso all'infotainment automobilistico ad alte prestazioni di QTI[1], piattaforma di assistenza alla guida avanzata per lo sviluppo, il test, l'ottimizzazione e la presentazione di soluzioni di infotainment per veicoli di nuova generazione[11]. La seguente figura mostra l'hardware in questione[2] :

[1]Qualcomm® Technologies, Inc.

[2]fonte : `https://cdn.lantronix.com/wp-content/uploads/img/adp-855-new.jpg`

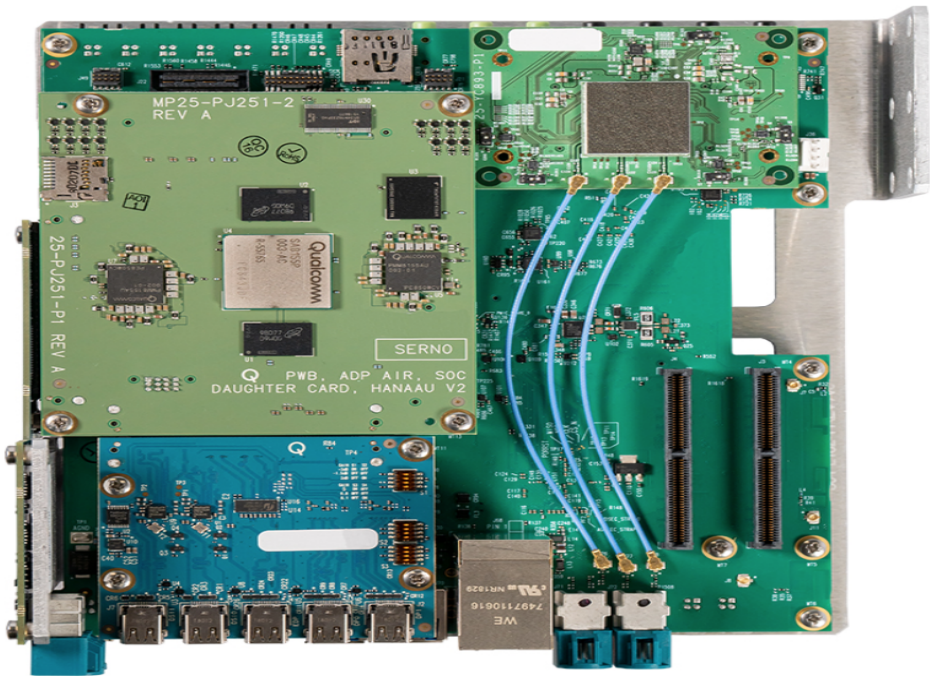
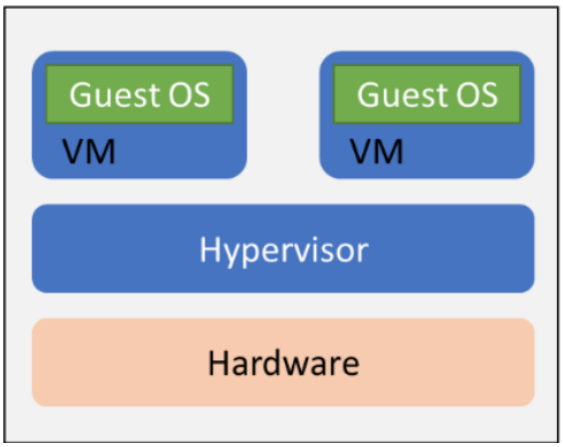


Figura 4.1: ADP8155

4.1.2 Software

A livello software la piattaforma è gestita da un **QNX Hypervisor**, una tecnica di virtualizzazione hardware ideale per ospitare ed eseguire contemporaneamente più sistemi operativi guest su un singolo host, andando a condividere filesystem, device I/O, connessioni, CAN Bus, microfoni e speaker. Con questa soluzione è possibile isolare i componenti critici da quelli non critici, andandoli ad eseguire in sistemi guest di˙eren ti. In questo modo si facilita la loro gestione e l'ottenimento dei certificati di sicurezza per coloro che ne necessitano. Ad esempio, tipicamente il cluster e il SO Infotaiment sono eseguiti separatamente poichè per il primo sono richiesti dei requisiti di sicurezza per la gestione di avvisi critici o di velocità, mentre in un In-Vehicle Infotainment solitamente non sono richiesti o soddisfatti. Come Sistema Operativo



Infotainment è presente **Android Automotive 10**[12], personalizzato successivamente da Marelli. Sebbene la sua trattazione completa esuli dagli obiettivi di questo trattato, Android Automotive è stato progettato da Google con l'idea di o˙rire un sistema operativo dotato di un insieme di servizi, interfacce ed API tipiche per l'accesso diretto alle proprietà di un veicolo. A livello applicativo, le tipiche apps di un sistema Infotainment tra cui radio, media, navigazione e impostazioni del veicolo sono state personalizzate a partire da quelle native mentre altre sono state progettate da capo dagli sviluppatori Marelli. Le seguenti immagini mostrano il Sistema Operativo, dopo le personalizzazioni, sul quale è stata integrata Alexa :

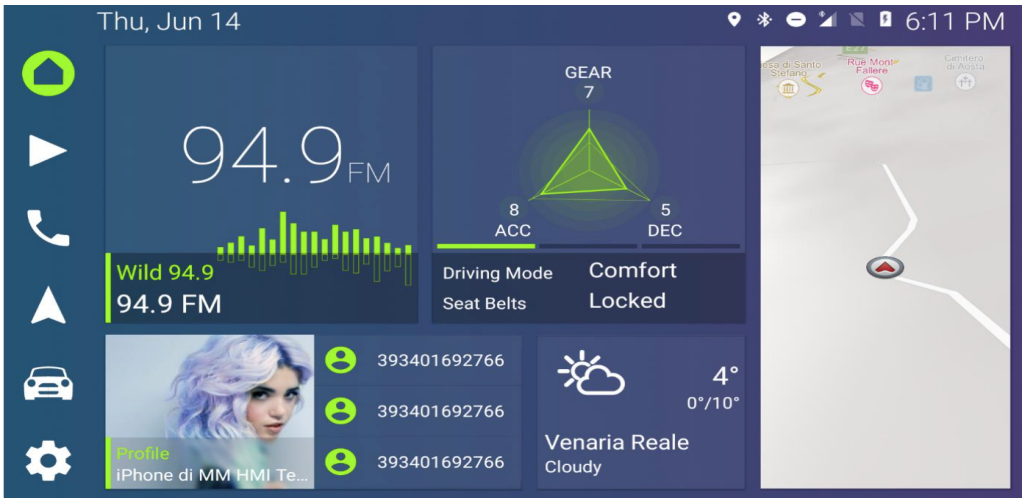


Figura 4.2: Overview

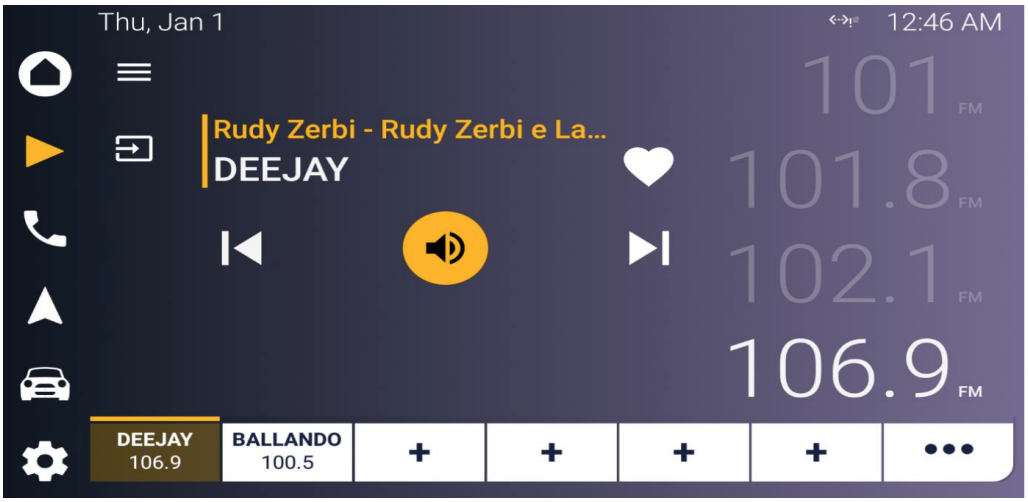


Figura 4.3: Radio

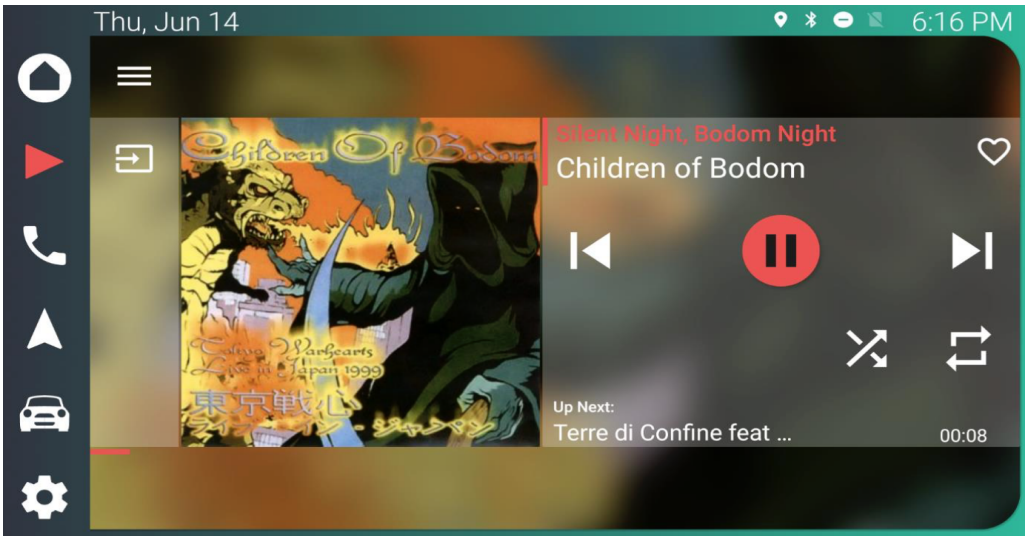


Figura 4.4: Media

Figura 4.5: Phone

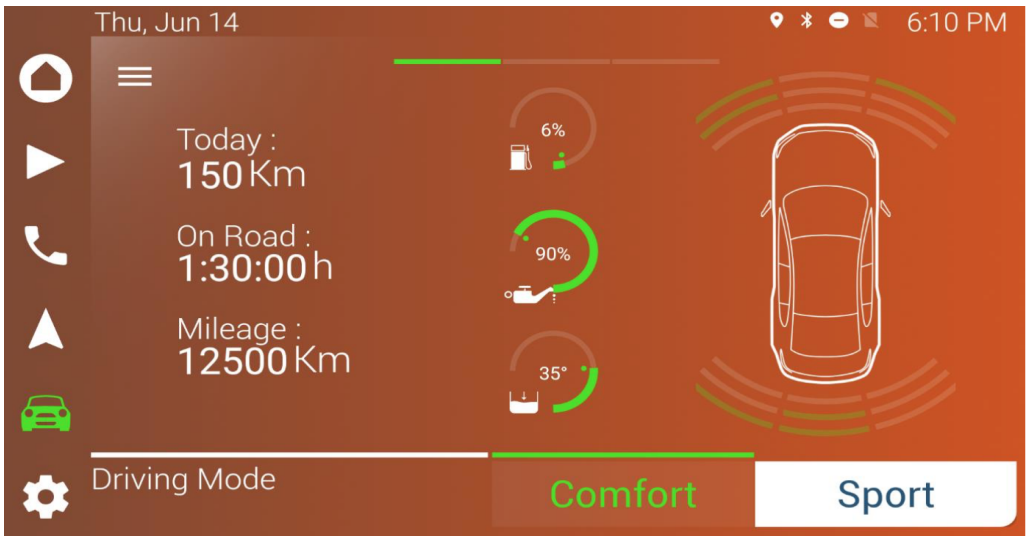


Figura 4.6: MyCar

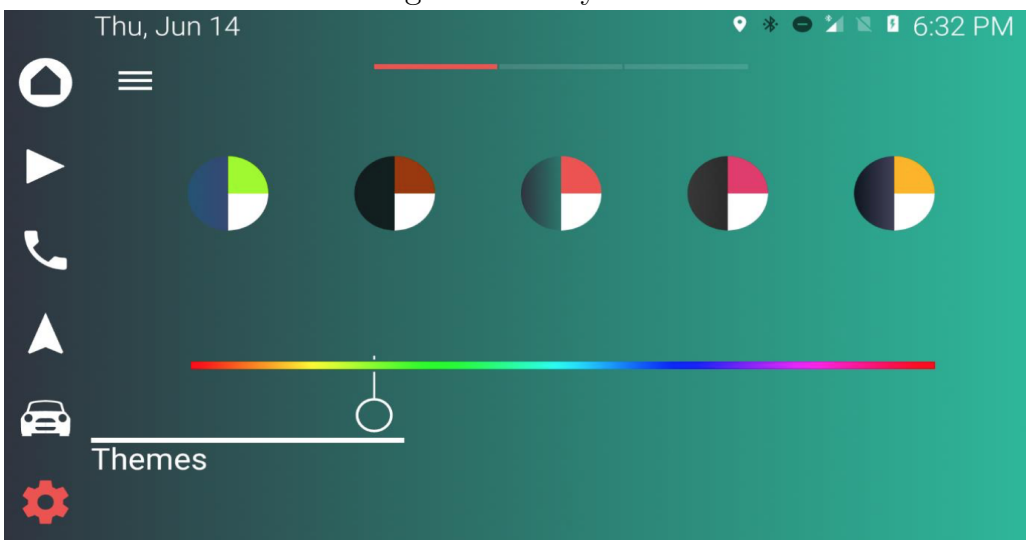


Figura 4.7: Preferences

4.1.3 Toolchain

Avendo come target di test una piattaforma Qualcomm dotata di Android Automotive su cui sono presenti app native e personalizzate che o˙rono le

funzionalità tipiche di un Infotainment, si è scelto di integrare Alexa, sviluppando una app dedicata. La scelta di sviluppare una app dedicata è giustificata dal fatto che Android permette la comunicazione tra applicazioni mediante il metodo degli **intenti**, una forma di messaggistica gestita dal sistema operativo con cui una componente di un'applicazione può richiedere l'esecuzione di un'azione da parte di un'altra componente di una seconda applicazione.[13].
L'IDE uÿciale per lo sviluppo di app Android porta il nome di **Android Studio**, basato su IntelliJ. E' un potente editor che eredita gli strumenti per sviluppatori di Intellij e ne aggiunge di altri specifici per l'ambiente Android. In particolare o˙re un sistema di compilazione flessibile basato su Gradle, supporto per C++ e NDK e possibilità di inviare e modificare il codice dell'app in esecuzione senza riavviare l'app. Integra inoltre due componenti molto importanti della toolchain di android chiamati "**adb**"[14] e "**logcat**"[15]. Entrambi strumenti utilizzabili da riga di comando, il primo permette di comunicare con il device con opportuni comandi tra cui installare una app mentre il secondo permette il dump e la lettura dei messaggi di sistema, inclusi gli errori critici ma anche i messaggi inviati dalle varie app mediante la classe *Log*. Ogni progetto di Android Studio contiene diversi moduli contenti file sorgenti e file di risorse e la loro organizzazione visiva nell'IDE dipende dal tipo selezionato. La vista predefinita è la "*Android project*" nel quale si possono vedere :

- **manifests** : contiene l'AndroidManifest.xml, un file nel quale si trovano tutti i dettagli dell'app
  - il nome del package
  - il codice della versione
  - il nome della versione
  - la versione del SDK Android minima per l'utilizzo dell'applicazione (minSdkVersion)
  - l'immagine che rappresenta l'applicazione sul dispositivo (icon)
  - il nome dell'applicazione (label)
  - il nome delle activity presenti nell'applicazione richiamabili dall'esterno
- **java** : contiene i file sorgenti Java, inclusi il codice dei JUnit test

- **res** : contiene i file sorgenti non inerenti al codice della app ma ad esempio all'XML layout, *UI strings* e *bitmap images*
- Sotto **Gradle Scripts** ci sono i file ".gradle" che descrivono tutti i componenti necessari al tool Gradle per la fase di *build* dell'app come librerie, path dell'SDK Android, i vari file che costituiscono l'app,etc.

Nel caso della Alexa app, il progetto si presenta come segue :

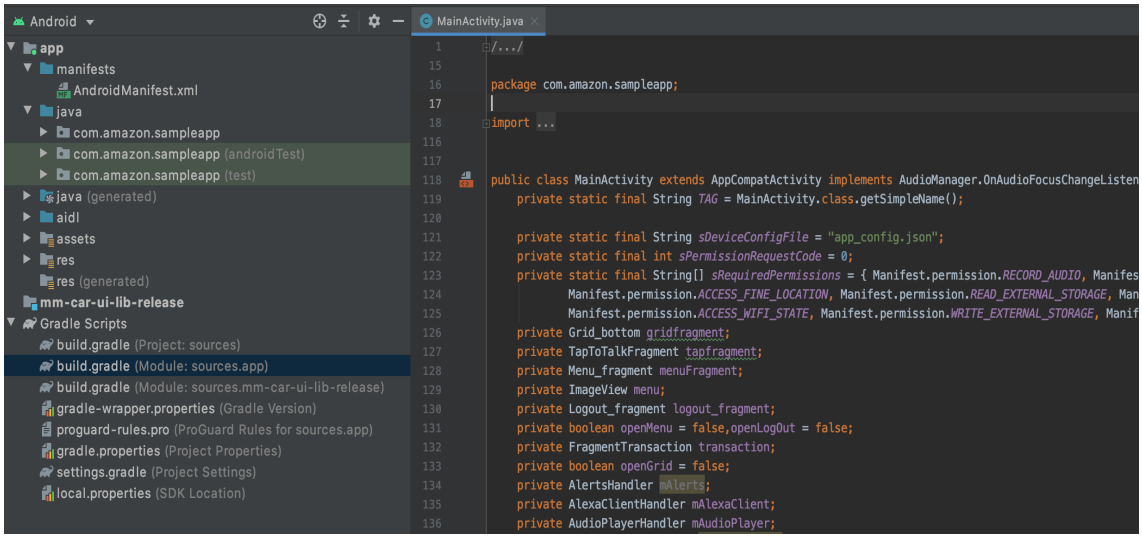


Figura 4.8: Progetto Alexa in Android Studio

4.2 Alexa Auto SDK

Amazon fornisce un *Alexa Auto SDK* che si basa sull"SDK generale descritto nel capitolo 2.3 ma con una riorganizzazione dei moduli e delle personalizzazioni in modo da aiutare gli sviluppatori a integrare Alexa negli autoveicoli. E' anch'esso modulare e astratto. Le librerie sono in linguaggio C++ ma è fornita anche una versione Java. Ogni interfaccia Java è stata costruita e mappata direttamente sulla sua controparte in C++ grazie al framework **JNI**[16]. Vengono perciò esposte delle interfacce per poter personalizzare le varie funzioni richieste in base allo specifico target, quindi audio input, riproduzioni di sorgenti, controllo per le chiamate, navigazione e impostazioni del veicolo. In altri termini, questo SDK fornisce tutto il necessario a livello implementativo per poter avviare, gestire una conversazione e ricevere direttive da Alexa ma sarà compito dello sviluppatore implementare il codice

di ogni interfaccia per lo specifico target, oltre che creare un'interfaccia grafica che o˙ra una "human experience" adeguata e rispettosa dei requisiti imposti da Amazon.

4.2.1 UX Design

Anche se Amazon fornisce, insieme all'SDK, una app predefinita con una interfaccia grafica molto semplice, è stato necessario ripensarla completamente. Il motivo sta nel fatto che il prototipo, con una prima versione di Alexa, è stato presentato al CES di Las Vegas 2020, ma Amazon ha voluto prima valutare la bontà del lavoro fatto sull'app Alexa e accertarsi che le linee guida per comporre la nuova interfaccia grafica fossero state seguite. Partendo quindi dalla struttura dell'app predefinita, dalla quale è stato possibile comprende gli eventi legati a ogni componente e seguendo la documentazione che fornisce un set di icone, colori e layout indicati[17], l'app è così composta :

MainActivity : utilizzando essenzialmente un display 10" 16:9, l'interfaccia si presenterà in landscape ed è composta da tre zone orizzontali :

- un *RelativeLayout* superiore in cui sarà presente :
  - un *ImageView* sul quale sarà registrato un evento click che aprirà una sezione per cambiare lingua
  - un *ImageView* che riporterà il logo uÿciale Alexa fornito da Amazon
  - un *ImageView* sul quale sarà registrato un evento click per e˙ettuare il logout e tornare alla sezione del *Login*
- un *FrameLayout* nella zona centrale dell'applicazione. E' una porzione dinamica, infatti in essa saranno mostrati i contenuti in base a ciò che ha chiesto l'utente o che è stato selezionato sull'interfaccia, sostituendo il contenuto vecchio con quello nuovo. Per poter cambiare dinamicamente il contenuto è stato definito un Id con valore '*centerCard*'.
- nella parte inferiore è stato definito un *FrameLayout* nel quale si alterna la visibilità di 2 *Fragment* che compariranno in alternanza quando la conversazione è iniziata oppure no. Sotto questi frammenti sarà visibile un ulteriore *FrameLayout*, nel quale si andrà a integrare l'estensione **VoiceChrome**, disponibile su richiesta, per implementare gli *Alexa attention states* relativi ai vari stati della conversazione : *Listening, Thinking, Speaking.*

Di seguito uno parte del file XML che implementa la vista :

```
24          <RelativeLayout...>
75
76          <LinearLayout...>
82
83          <FrameLayout
84              android:id="@+id/centerCard"
85              android:layout_width="match_parent"
86              android:layout_height="500dp">
87              <!--android:layout_height="495dp">-->
88
89
90              <FrameLayout...>
97
98              <TextView...>
109             <ImageView...>
118
119         </FrameLayout>
120         <LinearLayout...>
126         <FrameLayout
127             android:id="@+id/fragmentContainer"
128             android:layout_width="match_parent"
129             android:layout_height="80dp">
130             <FrameLayout
131                 android:id="@+id/voice_chrome_view_container"
132                 android:layout_width="match_parent"
133                 android:layout_height="12dp"
134                 android:layout_gravity="bottom"/>
135         </FrameLayout>
136     </LinearLayout>
137 </RelativeLayout>
138 </FrameLayout>
```

Figura 4.9: XML MainActivity

Login Il metodo di login scelto è il Code-Based Linking. Secondo le indicazioni di Amazon è necessaria una *card* di Login che verrà mostrata nel *FrameLayout* centrale.(figura 4.10). Oltre al logo Alexa e alla frase di informazione, sul componente *TextView* con id : *viewLogin* è stato registrato un "ClickListener", in questo modo se l'utente e˙ettuerà il click si va avvierà la fase di login.
Ricevuto l'authorization code, si mostrerà una seconda vista, riportando il codice e l'URL al quale l'utente deve collegarsi e inserirlo. E' stato definito l'*id code* sull'ultima *Textview* per poter cambiare dinamicamente il suo contenuto(figura 4.11, figura 4.12).
Nel momento di avvio dell'applicazione, se non sarà presente il Refresh Token nella struttura dati persistente dell'applicazione, la vista dell'applicazione ri-sulterà come in figura 4.13. Se invece il Refresh Token è già presente oppure la fase di login è terminata l'app mostrerà la Home.

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:windowBackground="@{colorsViewModel.backgroundGradientColors}"
    android:id="@+id/centerFrame">

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:orientation="vertical"
        android:layout_centerInParent="true">
        <ImageView
            android:layout_width="200dp"
            android:layout_height="120dp"
            android:layout_gravity="center"
            app:srcCompat="@drawable/vertical_rgb_color_whitetext">
        </ImageView>
        <TextView
            android:id="@+id/message"
            android:layout_width="687dp"
            android:layout_height="wrap_content"
            android:fontFamily="sans-serif"
            android:text="Alexa allows you tu use your voice to hear the news,\n check w
            android:textAlignment="center"
            android:textColor="@color/cblText"
            android:textSize="25sp"
            android:paddingBottom="20dp">
        </TextView>

        <TextView
            android:id="@+id/viewLogin"
            style="@style/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="Sign in with Amazon"
            android:fontFamily="sans-serif"
            android:textColor="@color/cblBackground">
```

Figura 4.10: CBLLogin XML

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:windowBackground="@{colorsViewModel.backgroundGradientColors}"
    android:id="@+id/centerFrame">

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:orientation="vertical"
        android:layout_centerInParent="true">
```

Figura 4.11: CBLCode XML

```
<ImageView
    android:layout_width="200dp"
    android:layout_height="120dp"
    android:layout_gravity="center"
    app:srcCompat="@drawable/vertical_rgb_color_whitetext">
</ImageView>

<TextView
    android:id="@+id/message"
    android:layout_width="687dp"
    android:layout_height="wrap_content"
    android:fontFamily="sans-serif"
    android:textAlignment="center"
    android:textColor="@color/cblText"
    android:textSize="25sp"
    android:paddingBottom="20dp">

</TextView>

<TextView
    android:id="@+id/code"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:fontFamily="sans-serif-light"
    android:textColor="@color/cblCode"
    android:textSize="60sp"
    android:singleLine="true" />
</LinearLayout>
</FrameLayout>
```

Figura 4.12: CBLCode XML

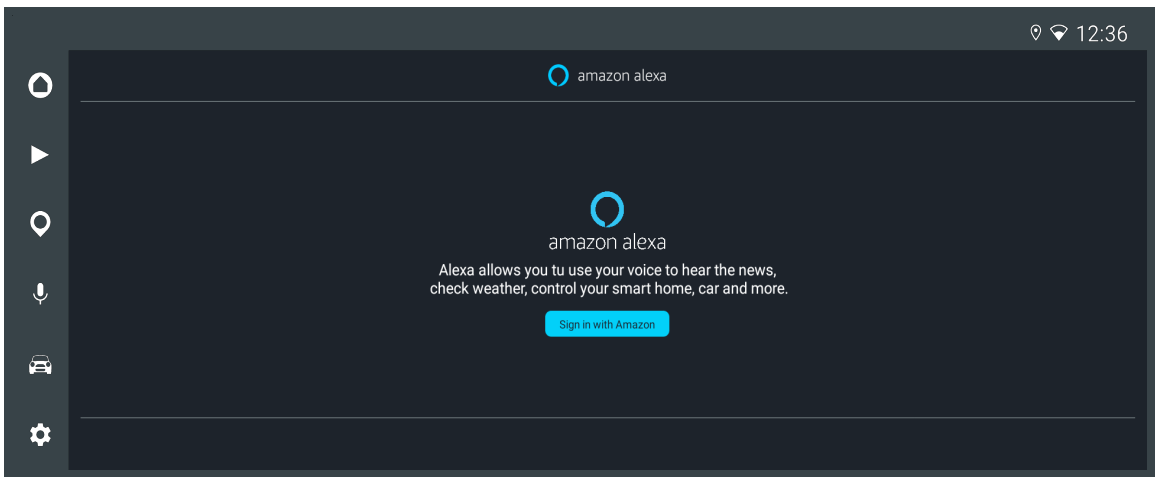


Figura 4.13: SignIn View

Home A login avvenuto, nella schermata di Home l'icona in alto a destra per il logout e in alto a sinistra per il cambio della lingua diventano visibili e cliccabili. Nella *FrameLayout* inferiore, comparirà il *Fragment* predefinto mediante transazione animata. Esso è composto da un *GridLayout* nel quale si trovano 5 celle *ImageView*. Su ogni *ImageView* è registrato un ClickListener per poter reagire ai click dell'utente. Selezionando la cella, verrà mostrato l'utlimo contenuto inviato da Alexa relativa alla sezione a cui appartiene l'*ImageView*. In figura 4.14 e 4.15 sono riportati rispettivamente uno stralcio dell'XML e il risultato della vista. Le celle sono così organizzate :

- **weather** : questa cella è legata al contenuto delle previsioni meteo. Quando ad esempio si chiede "che tempo fa?", Alexa invia una direttiva RenderTemplate di tipo WeatherTemplate, quindi verrà mostrato nel *FrameLayout* centrale il contenuto ricevuto.

- **music** : questa celle è legata ai metadati di un contenuto multimediale come una canzone o una webradio che Alexa sta inviando. In questo caso nel *FrameLayout* centrale viene mostrato il contenuto relativo a una direttiva *RenderPlayerInfo*.

- **alexaButton** : su questa cella è presente il ClickListener che permette l'avvio della conversazione con Alexa tramite il **TapToTalk**. In questo caso il *Fragment* corrente sparirà con una transazione animata verso il basso e comparirà il *Fragment* relativo alla gestione della conversazione. Al sui interno a sinistra c'è un *TextView* dal contenuto dinamico, utilizzato per mantenere l'utente al corrente riguardo allo stato del dialogo con Alexa(figura 4.16)Al centro, con un tap sull'*ImageView* si potrà interrompere la conversazione in ogni momento. Per informare l'utente sullo lo stato del dialogo con Alexa, nel momento in cui la conversazione inizierà, comparirà inoltre un FrameLayout sotto al *Fragment* per la conversazione in cui saranno emessi gli *Attention States* secondo le specifiche del **VoiceChrome**, un requisito visivo imposto da Amazon. A livello visivo, in termini pratici si potrà vedere una linea cromatica che cambia colore e animazione al cambio dello stato del dialogo(figura 4.17).

- **calendar** : questa cella fa parte dei contenuti relativi agli impegni nell'agenda, alla lista di cose da fare, al carrello della spesa ma anche alla ricerca dei *Point of Interest* (POI) più vicini alla luogo corrente. Nel caso dei POI verrà mostrato il contenuto della direttiva "LocalSearchListTemplate1", quindi una lista dei POI vicini, con il relativo nome, via e

distanza. In tutti gli altri casi, come ad esempio a seguito di richieste del tipo : "Quali sono gli impegni di oggi?" oppure "Mostrami la lista di cose da fare", verrà analizzato il contenuto della direttiva "ListTemplate1" e dunque mostrato un semplice elenco.

- **history** : a questa cella appartengono i contenuti dei *BodyTemplate1* e *BodyTemplate2*.

```
14      <GridLayout
15          android:id="@+id/bottomLayout"
16          android:layout_width="match_parent"
17          android:layout_height="match_parent"
18          android:columnCount="5"
19          android:orientation="horizontal"
20          android:rowCount="1">
21          <RelativeLayout...>
39
40          <RelativeLayout...>
58
59          <RelativeLayout
60              android:layout_width="0dp"
61              android:layout_height="match_parent"
62              android:layout_row="0"
63              android:layout_column="2"
64              android:layout_columnSpan="1"
65              android:layout_columnWeight="1"
66              android:orientation="vertical"
67              >
68
69              <ImageView
70                  android:id="@+id/alexaButton"
71                  android:layout_width="wrap_content"
72                  android:layout_height="wrap_content"
73                  android:layout_centerInParent="true"
74                  app:srcCompat="@drawable/ic_alexa" />
75          </RelativeLayout>
76
77          <RelativeLayout...>
95
96          <RelativeLayout...>
114
115     </GridLayout>
```

Figura 4.14: Fragment Principale XML

Figura 4.15: Fragment Principale e LogOut Views

```
<GridLayout  xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:background="#000000"
    tools:context=".controller.Fragment.TapToTalkFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnCount="5"
    android:orientation="horizontal"
    android:rowCount="1">
    <LinearLayout
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_row="0"
        android:layout_column="0"
        android:layout_columnSpan="1"
        android:layout_columnWeight="1"
        android:orientation="vertical"
        android:padding="7dp">
        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:id="@+id/stateDialog"
            android:fontFamily="@font/amazonember_it"
            android:textColor="#98A5A5"
            android:textSize="24dp"
            android:layout_marginTop="30dp" />
    </LinearLayout>

    <LinearLayout...>

    <RelativeLayout...>

    <LinearLayout...>

    <LinearLayout...>
</GridLayout>
```

Figura 4.16: FragmentDialogo XML

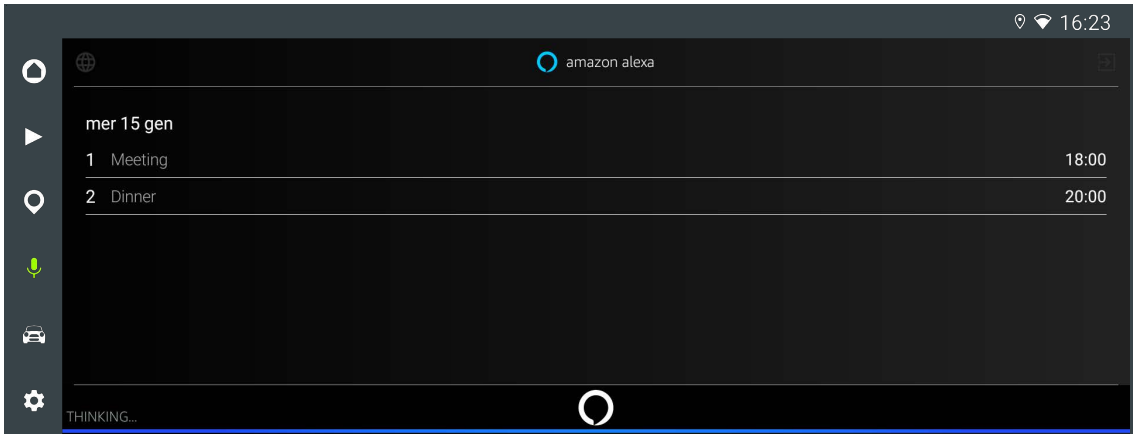


Figura 4.17: Listening View

Weather Per questa sezione è stato creato un *FrameLayout* che occuperà il *FrameLayout* centrale. Ogni elemento che lo compone è recuperato dal JSON Object inviato da Alexa. In particolare si trova :

- *Title e SubTitle* : sono 2 *TextView* uno sopra l'altro e riportano rispettivamente la località e il giorno

- Al centro si trova una *ImageView* relativa all'icona del tempo del giorno corrente e subito a lato le informazioni sulla temperatura attuale, tipo di tempo e temperatura massima e minima, tutte rappresentate con delle *TextView*.

- Nella parte bassa 7 celle contenenti le previsioni della successiva settimana, ognuna composta da icona, temperatura massima e minima.

In figura 4.18 è riportato uno stralcio dell'XML, mentre la figura 4.19 dimostra come risulta la *card* delle previsioni del tempo.

```
25
26          <TextView
27              android:id="@+id/mainTitle"
28              android:layout_width="wrap_content"
29              android:layout_height="wrap_content"
30              android:fontFamily="sans-serif"
31              android:lines="1"
32              android:textColor="@color/cardDivider"
33              android:textSize="30sp"
34
35             />
36
37          <TextView
38              android:id="@+id/subTitle"
39              android:layout_width="wrap_content"
40              android:layout_height="wrap_content"
41              android:layout_marginTop="4dp"
42              android:fontFamily="sans-serif-light"
43              android:lines="1"
44              android:textColor="@color/cardSubTitle"
45              android:textSize="20sp"
46              />
47
48      </LinearLayout>
49
50      <LinearLayout
51          android:layout_width="match_parent"
52          android:layout_height="160dp"
53          android:layout_marginTop="20dp"
54          android:orientation="horizontal"
55          android:layout_marginLeft="130dp"
56          android:layout_marginRight="130dp">
57
58          <ImageView
59              android:id="@+id/currentWeatherIcon"
60              android:layout_width="wrap_content"
61              android:layout_height="match_parent"
62              android:layout_gravity="center"
63              android:contentDescription="Current Weather Icon"
```

Figura 4.18: Weather XML

Figura 4.19: Weather View

Music Questa sezione è stata creata per mostrare graficamente le informazioni ricevute riguardo alla traccia che si sta riproducendo. Quando Alexa e˙ettua lo streaming di una traccia audio da un servizio musicale, vengono anche inviati i metadati della playlist, webradio, etc, mediante una direttiva *TemplateRuntime* di tipo *RenderPlayerInfo*.
La *card* Music si presenta come in figura 4.21, mentre la figura 4.20 mostra una sezione dell'implementazione XML. In questo caso ci sono tre sezioni :

- Un *LinearLayout* in cui si trova, in alto a sinistra, un *header* e un *headerSubtext1* che corrispondono rispettivamente al nome del servizio che sta fornendo il contenuto e a ulteriori informazioni legate al provider. In alto a destra è presente una *ImageView* che sarà il logo del provider.

- Nella sezione centrale ci saranno uno a fianco all'altro 2 *RelativeLayout*. Nel *RelativeLayout* di sinistra ci saranno 3 *TextView* che riporteranno il nome, l'artista e l'album per una traccia di una playlist oppure la frequenza, il nome della radio e la regione di appartenenza nel caso di una webradio. Subito sotto sono posti i *Buttons* per poter mettere in pausa, riprendere, passare alla traccia successiva oppure alla precedente e la *SeekBar* di avanzamento(figura 4.20). Nel *RelativeLayout* di destra sarò presente il logo della radio oppure l'immagine dell'album che si sta riproducendo.

La figura 4.21 mostra la vista nel caso in cui l'utente chieda di riprodurre radio105.

```
79          <RelativeLayout
80              android:layout_width="match_parent"
81              android:layout_height="150dp"
82              android:orientation="vertical">
83
84              <TextView
85                  android:id="@+id/title"
86                  android:layout_width="match_parent"
87                  android:layout_height="wrap_content"
88                  android:fontFamily="@font/amazonember_it"
89                  android:layout_alignParentTop="true"
90                  android:maxLines="2"
91                  android:scrollbars="vertical"
92                  android:gravity="center"
93                  android:textColor="#FFFFFF"
94                  android:textSize="50sp" />
95
96              <TextView...>
106
107             <TextView...>
117         </RelativeLayout>
118             <RelativeLayout
119                 android:layout_width="match_parent"
120                 android:layout_height="match_parent">
121                 <ImageView
122                     android:layout_width="wrap_content"
123                     android:layout_height="wrap_content"
124                     android:layout_centerInParent="true"
125                     app:srcCompat="@drawable/ic_pause_circle_outline_black_24dp">
126                 </ImageView>
127                 <ImageView...>
135                 <ImageView...>
143                 <SeekBar
144                     android:id="@+id/seek"
145                     android:layout_width="match_parent"
146                     android:layout_height="wrap_content"
147                     android:layout_alignParentBottom="true"
148                     android:progress="0">
149                 </SeekBar>
```

Figura 4.20: Music XML

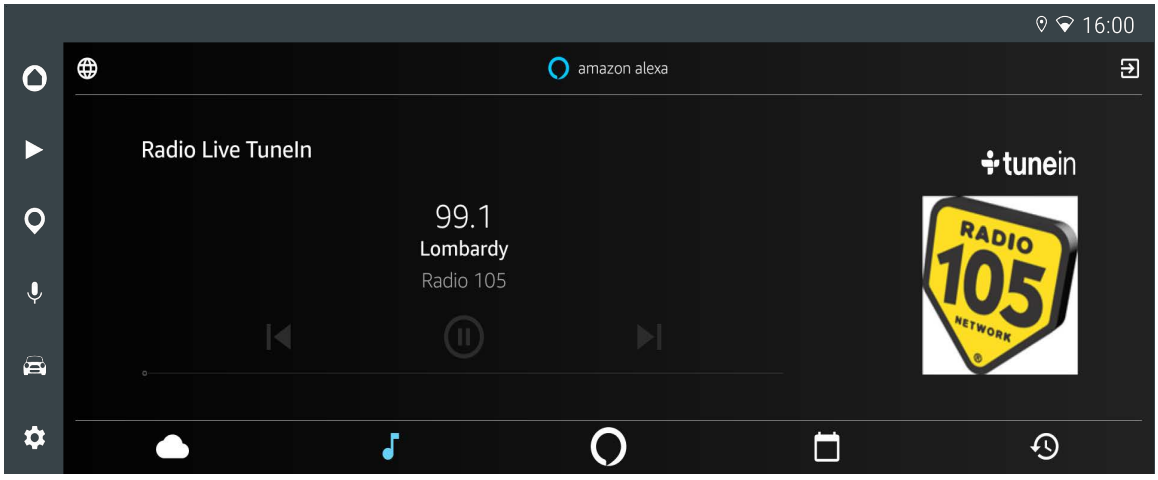


Figura 4.21: Webradio View

Calendar Questa sezione è dedicata alla presentazione della direttiva *ListTemplate1*, quindi per eventi nel proprio calendario o per mostrare la lista delle cose da fare, e per *LocalSearchListTemplate1*, ossia per la ricerca dei POI più vicini.
Per entrambe le tipologie, la struttura basilare è la medesima, ossia un semplice *LinearLayout* con all'interno uno *ScrollView*. Ogni riga della lista è composta da 2 *RelativeLayout* uno accanto all'altro. La differenza sta nei componenti che vanno a riempire questi due.

- ListTemplate1(figura 4.22)
  - Per il calendario In quello di sinistra verrà presentato il nome dell'evento e in quello di destra l'orario(figura 4.23).
  - Per "la lista di cose da fare" si riporterà a sinistra semplicemente l'azione registrata.
- LocalSearchListTemplate1 : in questo caso a sinistra verrà riportato il nome del POI e il suo indrizzo mentre a destra la distanza dalla posizione corrente dell'utente(figura 4.24, figura 4.25).

```
12      <RelativeLayout
13          android:layout_width="wrap_content"
14          android:layout_height="match_parent">
15          <TextView
16              android:id="@+id/index"
17              android:layout_width="wrap_content"
18              android:layout_height="wrap_content"
19              android:ellipsize="end"
20              android:fontFamily="sans-serif"
21              android:maxLines="1"
22              android:text="1"
23              android:layout_centerInParent="true"
24              android:textColor="@color/cblText"
25              android:textSize="30sp" />
26      </RelativeLayout>
27
28      <RelativeLayout...>
46      <RelativeLayout...>
```

Figura 4.22: ListTemplate1 XML

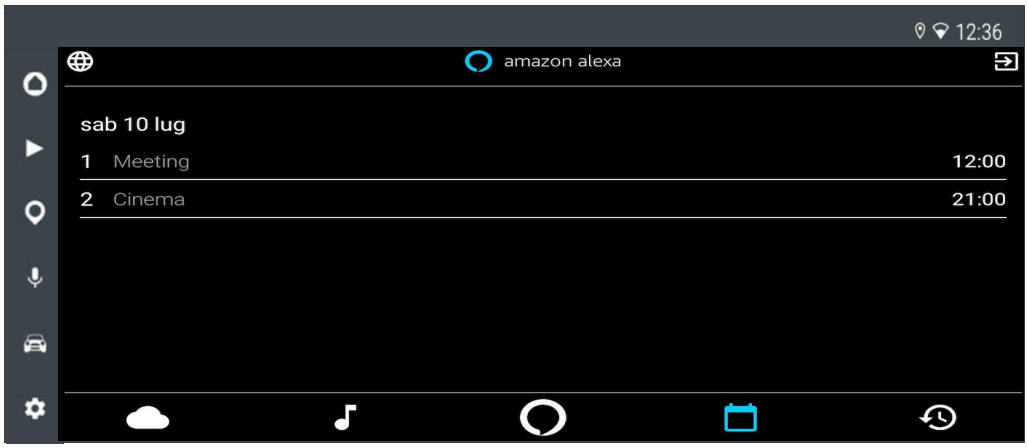


Figura 4.23: Calendar View

```
36                  android:layout_width="wrap_content"
37                  android:layout_height="wrap_content"
38                  android:orientation="horizontal">
39                  <TextView
40                      android:id="@+id/name"
41                      android:layout_width="wrap_content"
42                      android:layout_height="wrap_content"
43                      android:layout_marginStart="12dp"
44                      android:ellipsize="end"
45                      android:fontFamily="sans-serif"
46                      android:maxLines="1"
47                      android:textColor="@color/cblText"
48                      android:textSize="40sp" />
49              </LinearLayout>
50              <TextView
51                  android:id="@+id/address"
52                  android:layout_width="wrap_content"
53                  android:layout_height="wrap_content"
54                  android:layout_marginStart="12dp"
55                  android:fontFamily="sans-serif-light"
56                  android:maxLines="1"
57                  android:ellipsize="end"
58                  android:textColor="@color/cardSubTitle"
59                  android:textSize="28sp">
60              </TextView>
61          </LinearLayout>
62
63      </RelativeLayout>
64      <RelativeLayout
65          android:layout_width="match_parent"
66          android:layout_height="match_parent">
67          <TextView
68              android:id="@+id/dist"
69              android:layout_centerInParent="true"
70              android:layout_alignParentEnd="true"
71              android:layout_marginEnd="20dp"
72              android:layout_width="wrap_content"
73              android:layout_height="wrap_content"
```

Figura 4.24: LocalSearchListTempletate1 XML

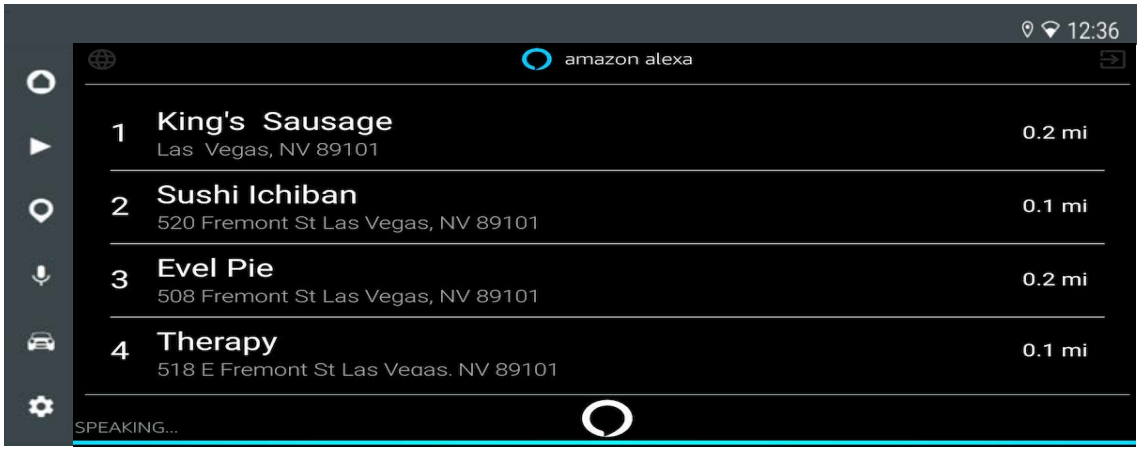


Figura 4.25: POI View

History Nell'ultima sezione si prendono in esame le direttive di tipo *BodyTemplate1* e *BodyTemplate2* del namespace *TemplateRuntime*. A queste direttive sono associati i metadati visivi che Alexa invia quando l'utente fa delle domande generiche del tipo "Chi è il presidente della repubblica" oppure "Parlami della seconda guerra mondiale". Entrambe le direttive contengono un testo informativo. La di˙erenza sta nel fatto che nella direttiva *BodyTemplate2* è presente anche un'immagine rappresentativa. Dovendo quindi renderizzare solo un'immagine, una descrizione e un titolo, la vista è organizzata come :

- Un *RelativeLayout* nella parte superiore dello schermo in cui saranno indicati il "mainTitle" al quale corrisponderà la domanda che l'utente ha fatto e un "subTitle" per riportare informazioni aggiuntive.
- Nella parte sottostante, il restante spazio è occupato da un *RelativeLayout* nel quale sono poste una a fianco all'altra un *TextView* nel quale ci sarà il testo informativo e un *ImageView* che mostrerà solo nel caso del *BodyTemplate2* l'immagine associata(figura 4.26, figura 4.27).

```
29          <RelativeLayout
30              android:layout_width="match_parent"
31              android:layout_height="wrap_content"
32              android:orientation="horizontal"
33              android:layout_marginBottom="20dp">
34              <TextView
35                  android:id="@+id/mainTitle"
36                  android:layout_width="wrap_content"
37                  android:layout_height="wrap_content"
38                  android:fontFamily="@font/amazonember_rg"
39                  android:layout_alignParentStart="true"
40                  android:textColor="#ffffff"
41                  android:textSize="38sp"/>
42              <TextView
43                  android:id="@+id/subTitle"
44                  android:layout_width="wrap_content"
45                  android:layout_height="wrap_content"
46                  android:textColor="#ffffff"
47                  android:textSize="32sp"
48                  android:fontFamily="@font/amazonember_it"
49                  android:layout_alignParentEnd="true" />
50          </RelativeLayout>
51          <RelativeLayout
52              android:layout_width="match_parent"
53              android:layout_height="match_parent"
54              android:orientation="horizontal">
55          <ScrollView
56              android:id="@+id/scroll_view"
57              android:layout_width="match_parent"
58              android:layout_height="wrap_content"
59              android:orientation="horizontal"
60              android:fadeScrollbars="false">
61              <TextView...>
71          </ScrollView>
72          <ImageView
73              android:id="@+id/icon"
74              android:layout_width="match_parent"
75              android:layout_height="match_parent">
76          </ImageView>
```

Figura 4.26: BodyTemplate XML

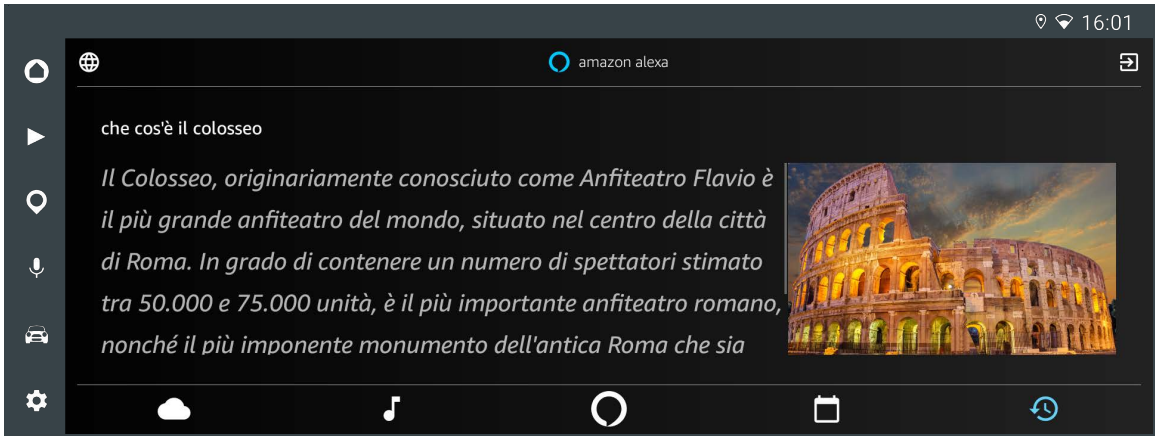


Figura 4.27: BodyTemplate2 View

4.2.2 Architettura

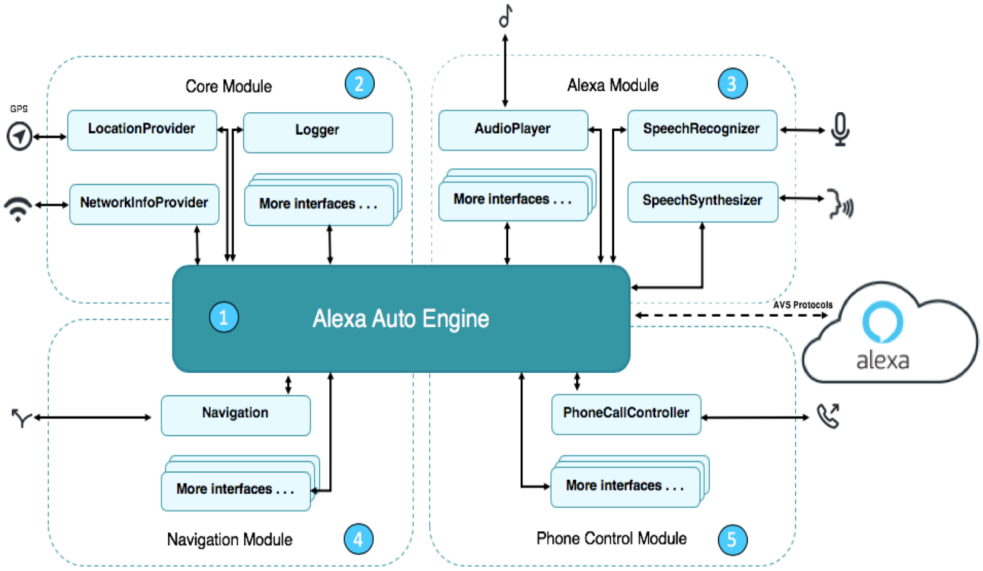


Figura 4.28: Architettura Alexa Auto SDK

La logica che comanda l'interfaccia grafica e l'interazione generale con Alexa è gestita da "Alexa Auto SDK" per cui, come detto precedentemente, sono definite delle interfacce che lo sviluppatore dovrà implementare. Anche in questo caso Amazon fornisce un'implementazione base di ognuna di esse, realizzando dunque una applicazione completa di esempio in modo da testare le skills associate, oltre che il funzionamento generale dell'Engine. Per poter realizzare l'applicazione presentata nel capitolo 4.2.1 si è partiti proprio da essa, modificandola a seconda delle particolari esigenze e dei problemi legati alla specifica piattaforma. Ogni componente ed evento è stato collegato all'interfaccia grafica totalmente riprogettata e sono state implementate le funzionalità mancanti come ad esempio quella della telefonata e della sincronizzazione dei contatti tra il telefono e Alexa. E' bene precisare che avendo definito degli step nei quali si sono posti degli obiettivi di funzionalità da raggiungere, in modo da garantire un grado di presentabilità sempre più stabile, la priorità di ogni modifica è dipesa da essi. In particolare questi step sono stati così definiti :

- **Step 1** : Login, Tap-to–talk interaction, meteo, multimedia (webradio, Spotify, Amazon Music, TuneIn), calendario, POI, domande generiche, navigazione.

- **Step 2** : Bluetooth, gestione delle telefonate, radio (AM/FM/SDARS/

DAB), impostazioni del veicolo (i.e. Controllo ventole e climatizzazione), notifiche, allarmi, gestione equalizzatore audio.

- **Step 3** : Sviluppo skill per espandere le funzionalità ed estensioni

Data la scelta di sfruttare le applicazioni già presenti sulla piattaforma e l'utilizzo degli *intenti* per comunicare con esse, alcuni moduli hanno richiesto modifiche più profonde mentre per altri è stato solo necessario comprendere il significato delle varie funzioni di richiamo e coordinarsi infine con gli altri team di sviluppo per ricevere indicazioni sugli *intenti* necessari a richiamare i servizi delle applicazioni.
Sulla base di queste premesse, si procede ad analizzare l'architettura di questo SDK.

(1) **Auto SDK Engine** : il motore runtime che gestisce tutti gli eventi e il flusso di interazione tra il device e Alexa Cloud. La struttura generale di questo SDK corrisponde a quanto visto per l'AVS Device SDK, quindi, sotto questo componente si trovano tutte le librerie in C++ che implementano l'ACL,l'ADSL,l'AFML,l'SDS, la definizione degli endpoints e dei channels, l'apertura e mantenimento di una connessione HTTP/2,etc. Quello che viene esposto sono i *Capabality Agents* e i loro moduli per poter personalizzare le azioni richieste a seconda della piattaforma su cui ci si trova o dell'interfaccia grafica che si vuole presentare. Ovviamente il tutto è rimappato sulla controparte JAVA con il framework JNI.

(2) **Core Module** : include la classe Engine e tutte le sue interfacce. Inoltre fornisce ai moduli esterni tutte quelle classi e funzioni base come audio input e output, logger, posizione del dispositivo e monitoraggio della rete.

- **Engine** : all'avvio occorre creare, configurare e avviare l'engine.
  La funzione *Engine.create()* restituisce un oggetto Engine.
  Dopodichè si procederà a configurarlo, chiamando il metodo *config* che accetta come parametro un array di oggetti *EngineConfiguration*. In ogni oggetto *EngineConfiguration* saranno presenti i dati del ClientId,ProductDSN,ProductId, parametri delle estensioni come il Local Voice Control, i path usati da SQLlite per salvare i dati permanenti dell'app, le informazioni sull'auto, le informazioni riguardo alle ventole, temperatura interna, etc. E'ettuata la configurazione, inizia la fase di registrazione delle istanze delle classi che implementano

le varie interfacce, utilizzando la funzione *registerPlatformInterface*, alla quale sarà passato come argomento l'istanza della classe stessa. Infine si avvia l'engine, richiamando il metodo *start()* per aprire una connessione persistente con il cloud e rimanere in attesa che l'utente faccia partire la fase di login, in modo da inviare successivamente l'evento Syncronize.State() per informare Alexa sui Capability Agents supportati e sul loro stato, oltre che lo stato generale dell'applicazione. A login e˙ettuato, l'engine è pronto ed è possibile iniziare a conversare con Alexa.

- **Property Manager Handler** : estende la classe *Property Manager* e permette di cambiare una determinata proprietà dell'applicazione in fase di runtime.
  Con il metodo *getPropriety* è possibile leggere una certa proprietà come la timezone impostata, mentre con il metodo *setProperty(com.amazon.aace.alexa.AlexaProperties.LOCALE,"en-CA")* si cambia ad esempio la lingua utilizzata. Quest'ultimo metodo è collegato infatti con il menù del cambio lingua che è possibile aprire dal pulsante posto in alto a sinistra nell'interfaccia grafica.
  Vanno implementati tuttavia 2 metodi che l'interfaccia espone :
  - *propertyStateChanged* : viene richiamato dall'engine quando è stata chiamata con successo la funzione *setProperty* ed è possibile notificare sull'interfaccia grafica il successo dell'azione.
  - *propertyChanged* : equivale al metodo precedente ma viene richiamato quando dall'applicazione uÿciale si cambiano delle proprietà come ad esempio la timezone. Al momento questi 2 metodi nell'app sviluppata richiamano un Dialog Alert.
- **LocationProviderHandler** : estende la classe *LocationProvider*. Tra i metodi più importanti da implementare si trova *getLocation*, una callback richiamata dall'engine per recuperare la posizione del dispositivo. E' possibile impostare anche una posizione fittizia con i metodi *setAltitude,setLatitude,setLongitude* e infine *enableMockLocation* nel caso in cui la piattaforma di test, come in questo caso, non abbia antenne GPS e quindi nessun provider di localizzazione disponibile, altrimenti, se la posizione fittizia non è abilitata, l'implementazione della applicazione di esempio permette recuperare le coordinate su una piattaforma generica Android.
- **NetworkInfoProviderHandler** : estende la classe *NetworkInfoProvider* e permette all'applicazione di monitorare i cambi di stato

della connessione di rete e comunicarla all'engine.
In particolar all'atto dell'istanziazione della classe, nel costrutto si registra un ricevitore di eventi inerenti ai cambi di rete. A ogni cambio di stato della rete, si legge il nuovo stato con il metodo Android *ConnectivityManager.getActiveNetworkInfo()*, lo si salva e con la funzione *networkStatusChanged* lo si comunica all'engine.
Tra i metodi da implementare c'è *getNetworkStatus*, callback richiamata dall'engine, la cui implementazione dipende dalla piattaforma ma in generale questo metodo ritorna l'ultimo stato salvato dal ricevitore che registra i cambi di stato della rete. Inoltre essendo un componente i cui servizi sono utilizzati da molti altri moduli dell'SDK, mediante l'Observer pattern[18], gli altri componenti si possono registrare nella sua lista di Observers per essere notificati tutti insieme quando questo componente registra i nuovi stati della rete.

- **AudioInputProviderHandler** : estende la classe AudioInputProvider. Fornisce agli altri componenti l'implementazione dell'*AudioInput* per un dato canale e sarà legata allo specifico target, in questo caso Android. Nel metodo *openChannel* per i canali *AudioInputType.VOICE* e *AudioInputType.COMMUNICATION* richiesti rispettivamente dai moduli SpeechRecognizer e AlexaComms, gli si fornisce lo stesso AudioInput perchè i canali sono condivisi.

- **AudioInputHandler** : estende l'interfaccia AudioInput. A ogni canale è associato un solo AudioInputHandler. Un modulo che prevede operazioni di "AudioRecording" e˙ettua una richiesta all'engine in modo che gli venga assegnato, in base al canale di input sul quale vuole operare, il suo AudioInputHandler.
L'implementazione dell'AudioInputHandler dipende dalla piattaforma. Quando viene costruito, viene ottenuto al suo interno un oggetto tipico Android chiamato *AudioRecord* che gestisce la registrazione da una sorgente audio, in questo caso il microfono. L'*AudioRecord* espone infatti il metodo *read* per ottenere i campioni audio dalla sorgente di registrazione.
Quando ad esempio sul modulo *SpeechRecognizer* viene richiamato il metodo *tapToTalk*, l'engine invoca la funzione *startAudioInput* all'interno dell'AudioInputHandler associato. Se più componenti richiedono contemporaneamente il controllo dello stesso canale, la funzione *startAudioInput* verrà richiamata una sola volta. In questa funzione, si avviano i seguenti 2 metodi dell'oggetto *AudioRecord*:

  - *startRecording* per iniziare l'e˙ettiv a registrazione
  - Viene avviato un thread che leggerà in loop i campioni audio tramite la funzione bloccante *read*. Durante l'operazione di lettura, ogni campione audio sarà successivamente scritto nel SDS, grazie alla funzione *write* dell'SDK. In questa funzione, a ogni scrittura sul ring bu˙er, verrà notificata la disponibilità di nuovi campioni all'engine che avviserà tutti i componenti interessati, pronti ora alla lettura.

  Quando tutti i componenti hanno concluso la loro azione di controllo, l'engine chiama il metodo *stopAudioInput* nell'oggetto *AudioInputHandler* per cessare l'operazione di registrazione.

- **AudioOutputProviderHandler** : estende l'interfaccia AudioOutputProvider. In questo caso, questo componente fornisce l'AudioOutputHandler che gestisce il canale inerente al componente che ha fatto richiesta.Ogni modulo per il quale sono previste funzioni di audio output ha un AudioOutputHandler associato.
  Il metodo *openChannel* riceve come parametro *AudioOutputType* per decidere che tipo di *AudioOutputhandler* restituire. I tipi corrispondono ai tipi di canali esposti nel capito 2.3.2 ossia : TTS, MUSIC, NOTIFICATION, ALARM, EARCON, COMMUNICATION.

- **AudioOutputHandler** : estende l'interfaccia AudioOutput. Per questa Interfaccia si ha un legame diretto con la piattaforma, dovendo quindi implementare i metodi :

  - *prepare* : istanzia e inizializza la classe Androd *MediaPlayer*, utilizzata per controllare la riproduzione di file e flussi audio / video, fornendogli il tipo di sorgente che può essere uno stream o un URI da cui e˙ettuare lo streaming.
  - *play* : manda in riproduzione il contenuto preparato per il mediaplayer.
  - *stop* : ferma la riproduzione
  - *pause* : mette in pausa la riproduzione
  - *resume* : riprende la riproduzione
  - *setPosition* : salta la riproduzione a un determinato timestamp
  - *getPosition* : restituisce il minuto corrente al quale si è arrivati a riprodurre la sorgente audio
  - *getDuration* : restituisce la durata della traccia

Non tutte le funzioni sono attivate ma dipende dal tipo di canale associato all'*AudioOutputHandler*.
Esse verranno richiamate dall'engine in base alla priorità dei channels. Se viene ad esempio mandato in riproduzione il canale del TTS, mentre si sta riproducendo una playlist, verrà invocato il metodo *pause* sull'AudioOutputHandler del canale MUSIC e mandato in riproduzione l'AudioOutputHandler del canale TTS. Terminato la riproduzione di quest'ultimo, sull'AudioOutputHandler del canale MUSIC verrà richiamato il metodo *resume*. Nel caso del channel TTS invece non sarà disponibile e dunque richiamato dall'engine *pause* o *resume* perchè il protocollo non lo prevede.
Per il channel MUSIC è possibile sfruttare le funzioni di *play*,*stop*, *pause*,*resume* e *setPosition* mediante dei bottoni presenti sulla interfaccia grafica che saranno abilitati in base alle istruzioni presenti nella direttiva *RenderPlayerInfo*.

- **AuthProviderHandler** : estende l'interfaccia AuthProvider. Gestisce i cambi di stato dell'autorizzazione con Alexa e salva i dati nel contesto dell'applicazione. Gli stati gestiti sono :
  - REFRESHED : per indicare che il login è avvenuto con successo e l'authToken è stato salvato
  - UNINITIALIZED : login non avvenuto

  Infine, dopo aver gestito i 2 stati, tramite la funzione *authStateChange* avvisa l'engine che a sua volta andrà a notificare il cambio di stato a tutti gli altri componenti in ascolto. Questa classe per recuperare le informazioni sullo stato del login si basa su un'altra classe chiamata **LoginWithAmazonCBL**, nella quale sono presenti le funzioni per poter preparare e completare la fase di login, secondo il metodo del Code-Based Linking. Poichè nell'applicazione di esempio queste funzioni sono già state implementate, è stato necessario collegare semplicemente i vari eventi generati da questa classe ai componenti della nuova interfaccia grafica.
  In particolare le funzioni più importanti sono :
  - *requestDeviceAuthorization* : questa funzione implementa i passaggi per ottenere lo *user_code* e il *verification_uri*. Una volta ottenuti i parametri verranno mostrati a video.
  - requestDeviceToken : viene utilizzata per ottenere l' *access_token*, il *refresh_token* e l'*expires_in*. Una volta ottenuti i parametri

si notifica l'evento a tutti i moduli in ascolto tra cui appunto l'*AuthProviderHandler*.

Per avviare questa sequenza di funzioni è possibile cliccare il pulsante in figura 4.13, a cui è collegato un ClickListener. Una volta azionato il pulsante, viene richiamata la funzione *authorize* all'interno della classe LoginWithAmazonCBL e dunque successivamente il metodo *requestDeviceAuthorization*. Inoltre nella figura 4.21 è possibile vedere in alto a destra la presenza di un ulteriore pulsante che servirà, una volta cliccato, a richiamare la funzione *deauthorize*, sempre all'interno di questa classe, per resettare l'applicazione, eliminare l'AuthToken e tornare allo stato iniziale (figura 4.13).
Quando invece si avvia l'applicazione e viene trovato un *refresh token* tra i dati persistenti dell'applicazione, la classe LoginWithAmazonCBL avvierà la funzione *startRefreshTimer* per ottenere automaticamente un nuovo AuthToken. In assenza di connessione internet, questa procedura non partirà e verrà restituito un errore all'utente.

(3) **Alexa Module** : a questo modulo appartengono tutti quei componenti che sfruttano le funzionalità del modulo Core, interagendo inoltre con l'engine stesso. Ne fanno parte quei Capability Agents che contengono delle funzioni richiamate dall'engine e che corrispondono a funzionalità come rendering dei contenuti visivi, equalizzatore audio, riproduzione delle sorgenti locali, allarmi, notifiche, "speech input e output" ma anche notificare lo stato generale di Alexa.

- **SpeechRecognizerHandler** : estende l'interfaccia SpeechRecognizer. Gestisce due compiti :
  - Il collegamento con la GUI per avviare il tapToTalk e holdToTalk. Per questa applicazione si è deciso di supportare solo il tapToTalk. In particolare l'alexaButton in figura 4.15 ha l' "eventClick" collegato con la funzione *onTapToTalk()* di questo componente. Questa funzione, richiamando il metodo *tapToTalk()*, avvisa l'engine che l'utente ha avviato la conversazione e dunque verranno e˙ettuati i passaggi necessari ad aprire il microfono.
  - Contemporaneamente alla prima parte, in ogni funzione d'avvio, ci sarà l'implementazione per riprodurre grazie a un *mediaplyer* Android i suoni di servizio START_TOUCH, START_VOICE,

END. Infatti al rientro della funzione *tapToTalk()*, tutti i altri canali saranno stati messi in pausa o fermati, eccetto quello relativo al EARCON sui cui ora si può riprodurre.

- **SpeechSynthesizerHandler** : estende l'interfaccia SpeechSynthesizer. Il suo compito è quello di gestire la riproduzione della risposta audio di Alexa. Tuttavia, dalla versione 2.0 dell'SDK, non è più necessaria un'implementazione dipendente alla piattaforma ma basterà solo registrarla nell'engine. A questo componente infatti sarà collegato un *AudioOutputHandler* che opera sul canale TTS e dunque si richiameranno le sue funzioni *play*, alla quale l'engine fornirà lo stream da riprodurre dopo aver interpretato la direttiva SpeechSyntesizer.Speak e infine *stop*.
- **AudioPlayerHandler** : estende l'interfaccia AudioPlayer. Quando l'engine prepara e manda in riproduzione un audio stream come una traccia di una playlist o una webradio, vengono richiamati i metodi *prepare* e *play* dell'AudioOutputHandler associato al canale MUSIC. Dopodichè l'engine utilizza l'AudioPlayerHandler per informare sullo stato di riproduzione dello stream in modo da poter gestire di conseguenza la GUI dell'applicazione. Viene perciò richiamata la funzione *playerActivityChanged* alla quale si passa il parametro che informa sullo stato del player. La sua implementazione è dunque dipendente dalla piattaforma.
- **PlaybackControllerHandler** : estende l'interfaccia PlaybackController. All'AudioPlayerHandler è strettamente legato un PlaybackControllerHandler. Quando infatti all'AudioPlayerHandler viene notificato ad esempio uno stato PlayerActivity.PLAYING, al suo interno si va a invocare la funzione *start()* del PlaybackControllerHandler. Nella funzione *start* verranno abilitati i pulsanti "previous", "pause" e "next" in figura 4.21 sui quali saranno collegati dei clickListener che andranno ad avvisare l'engine nel momento in cui uno di questi pulsanti verrà azionato. Sarà dunque possibile controllare la traccia e la playlist direttamente dalla GUI. Infine il componente AudioPlayerHandler aggiornerà la durata della traccia utilizzando la funzione *setTime* del PlaybackControllerHandler, impostando il valore sulla GUI e inizierà un ciclo che durerà fino al completamento della traccia per aggiornare lo stato della SeekBar.
  Per poter abilitare altri comandi sulla GUI come ad esempio lo "shu˜e" o il "repeat", quando si riceverà e leggerà la direttiva

RenderPlayerInfo si potranno richiamare ulteriori metodi *update* di questo componente.

– **AlexaSpeakerHandler** : estende l'interfaccia AlexaSpeaker. Questo componente gestisce i cambiamenti dei livelli audio sulla GUI, quindi è un componente la cui implementazione dipende dalla piattaforma. Alexa tiene traccia di 2 tipi di volumi del dispositivo : "ALEXA_VOLUME" che racchiude tutti quei volumi dei canali TTS, MUSIC, EARCON e "ALERTS_VOLUME" relativi ai canali di avvisi e allarmi.
Per controllare il loro livello esistono due modi :

  * L'utente chiede ad Alexa di abbassare il volume o rendere muto un determinato contenuto. In questo caso l'engine richiama la funzione *speakerSettingsChanged* di AlexaSpeakerHandler, alla quale verranno passati come parametri il tipo di SpeakerType che può essere "ALEXA_VOLUME" o "ALERTS_VOLUME" e quei parametri che indicano se si vuole rendere muto o solo impostare un nuovo livello. In questa funzione si andranno perciò ad aggiornare gli elementi grafici in base ai parametri ricevuti. L'engine nel frattempo andrà a richiamare uno dei metodi tra *volumeChanged* e *mutedStateChanged* degli AudioOutputHandler associati ai relativi canali per rispettivamente abbassare/aumentare il volume o impostare il muto.

  * E' possibile tramite GUI rendere disponibile dei "PushButton" e "SeekBar" medianti i quali si può regolare il volume e impostare il "mute" e l'"unmute" di un determinato tipo di volume. Questi eventi poi verranno collegati a 2 funzioni dell'AlexaSpeakerHandler ossia
  localAdjustVolume(SpeakerType,[0-10]) e
  localSetMute(SpeakerType,true/false) che andranno rispettivamente ad avvisare l'engine riguardo il nuovo livello di volume e di impostazione o meno del "mute" per un determinato canale. L'engine infine provvederà ad impostare i relativi AudioOutputHandler.

  Per scelta nell'applicazione sviluppata si è scelto di tenere come unica possibilità l'impostazione del volume tramite comando vocale da parte dell'utente.

- **AlertsHandler** : estende l'interfaccia Alerts. Al suo interno si trova l'implementazione per gestire i diversi AlertState tra cui ad esempio stop, completato, ready,etc ed avvisare l'utente con i relativi avvisi visivi sulla GUI. In particolare l'engine, dopo aver avviato l'AudioOutputHandler relativo al canale ALERT, richiama nell'AlertsHandler la funzione *alertStateChanged* per avvisarlo dello stato di un determinato avviso e in base al parametro AlertState sarà possibile capire quale elemento aggiornare o aggiungere sull'interfaccia grafica.
  Inoltre è possibile con le funzioni *onLocalStop* e *onRemoveAllAlerts*, collegate a dei PushButton sull'interfaccia grafica, eliminare uno o tutti gli allarmi creati tramite la GUI, informando l'engine con le funzioni *localStop* e *removeAllAlerts*.

- **NotificationHandler** : espande l'interfaccia Notifications. In questo componente si stabilisce l'implementazione per fornire un'indicazione visiva all'utente riguardo la presenza di nuove notifiche. L'engine avvisa questo componente mediante la callback *setIndicator*. Da qui sarà possibile chiedere ad Alexa quali sono le notifiche.

- **EqualizerControllerHandler** : estende l'interfaccia EqualizerController. Sebbene l'Applicazione non preveda implementazioni di tale caratteristica, in AlexaAutoSDK è fornita tramite questo modulo la possibilità di impostare l'equilizzatore audio del dispositivo, andando ad agire sui livelli delle frequenze supportate. A tal proposito viene definita la classe *EqualizerConfiguration* da cui ottenere la configurazione dei livelli e delle bande per poterla successivamente comunicare ad Alexa cloud. Di default tra le bande definite si trovano BASS,MIDRANGE e TREBLE, mentre i livelli min e max vanno da -8dB a +8dB. E' possibile comunque definire la configurazione tramite questa classe in base alle capacità del dispositivo.
  Per agire sulla loro impostazione esistono due modi :

  – L'utente chiede ad Alexa : "aumenta i bass". In questo caso, ricevuta la direttiva EqualizerController.SetMode, l'engine la interpreta e richiama il metodo *setBandLevels* di questo componente. Questa funzione, che riceve i parametri che danno informazioni sul tipo di frequenza da cambiare e il suo nuovo livello, conterrà il codice implementativo che dipenderà dalla piattaforma per poter eseguire l'azione.

  - E' possibile aggiungere sulla GUI delle SeekBar con le quali l'utente potrà direttamente interagire e cambiare dunque i livelli dall'interfaccia grafica. A livello implementativo, gli eventi su questi componenti grafici verranno collegati alle funzioni appropriate di questo componente. Se ad esempio l'utente andrà a impostare un nuovo livello per i BASS, verrà richiamata la funzione appropriata e, una volta recuperato il tipo di frequenza e il nuovo livello, si andrà a informare l'engine con la funzione *localSetBandLevels* di ciò che è avvenuto. A questo punto l'engine comunica il cambiamento di stato ad Alexa cloud e verrà richiamato il metodo *setBandLevels* del punto precedente.

- **LocalMediaSourceHandler** : estende l'interfaccia LocalMediaSource. Permette di gestire la riproduzione di un LocalMediaPlayer specifico della piattaforma (BLUETOOTH, USB, AM_RADIO, FM_RADIO, SATELLITE_RADIO, LINE_IN, COMPACT_DISC, SIRIUS_XM, DAB). In questo componente perciò è stata definita la logica per poter interpretare quale sorgente locale l'utente vuole avviare, lanciando successivamente, mediante un *intento*, quell'applicazione della piattaforma che gestisce lo specifico riproduttore. Esistono dunque funzioni che saranno richiamate dall'engine in base a come l'utente ha pronunciato il comando :

  - *play(ContentSelector selector, String payload)* : verrà richiamato quando l'utente chiederà di riprodurre una "radio local media source"(AM_RADIO, FM_RADIO, SIRIUS_XM, DAB) in base al suo ContentSelector che può essere per frequenza, preset o nome canale. Richiamando la funzione *getSource()* si recupererà il tipo di sorgente che si vuole avviare, mentre in *selector* sarà presente l'indicazione del tipo ContentSelector e nel payload le informazioni dettagliate del numero della frequenza, del nome del canale,etc. Infine questi parametri saranno utilizzati per lanciare l'*intento specifico*.
  - *playControl(PlayControlType controlType)* : in questo caso l'utente chiede di avviare o fermare la sorgente, senza dare ulteriori parametri ("Alexa riproduci bluetooth,USB,CDPlayer"). Il parametro *controlType* dà informazione sul tipo di azione che si vuole eseguire sul LocalMediaPlayer mentre *getSource()* restituirà il tipo di sorgente che l'utente ha chiesto di controllare. Di seguito è possibile vedere uno stralcio dell'implementazione :

```
@Override
public boolean playControl( PlayControlType controlType ) {
    switch( controlType ) {
        case PAUSE:
                nPause++;
                mLogger.postInfo( sTag, String.format( "playControl [source=%s,controlType=%s]", getSource(), controlType.toString()));
                setFocus();
                setPlaybackState("STOPPED");
            break;
        case RESUME:
                final PlayControlType playControlType = controlType;
                new Thread((Runnable) () → {
                        try {
                            Thread.sleep( millis: 250);
                            if(nPause < 2) {
                                nPause = 0;
                                mLogger.postInfo(sTag, String.format("playControl [source=%s,controlType=%s]", getSource(), playControlType.toString()));
                                setFocus();
                                setPlaybackState("PLAYING");
                                if (getSource().compareTo(Source.BLUETOOTH) == 0) {
                                    /*Intent mapIntent = mContext.getPackageManager().getLaunchIntentForPackage("com.example.mm_mediacontroller");
                                    mapIntent.putExtra("Type", "Bluetooth Audio");
                                    mapIntent.putExtra("Action", "PLAY");
                                    mapIntent.setType("text/plain");
                                    mContext.getApplicationContext().startActivity(mapIntent);*/
                                }
```

Indipendentemente dal valore di *controlType*, verrà sempre chiamata la funzione *setFocus()*. Questa funzione avvisa l'engine su quale sia l'ultima sorgente locale attiva sul canale audio in uscita. In questo modo, se, ad esempio, l'utente inizierà una nuova conversazione con Alexa, l'engine controllerà se c'è una sorgente locale in riproduzione e nel caso la interromperà. Inoltre sarà chiamato sempre la funzione *playControl(PlayControlType controlType)* con valore PAUSE nel parametro *controlType* che permetterà di richiamare nuovamente *setFocus()* e avvisare l'engine che la sorgente locale è effettivamente in pausa. Quando la conversazione dell'utente sarà terminata, verrà richiamata nuovamente *playControl(PlayControlType controlType)* ma questa volta con valore RESUME nel parametro *controlType* e riprenderà la riproduzione della sorgente corrispondente al valore dato dalla funzione *getSource()*, richiamando inoltre nuovamente *setFocus()*.

- **AlexaClientHandler** : estende l'interfaccia AlexaClient. E' utilizzata per riportare alla piattaforma i cambiamenti di stato del dialogo, della connessione e dell'autorizzazione in modo da abilitare o disabilitare di conseguenza determinate funzionalità o attivare cambiamenti di stato nell'interfaccia utente.
  In particolare le callback sono :

– *dialogStateChanged(final DialogState state)* : questa funzione viene richiamata nel momento in cui Alexa cambia i suoi stati : IDLE, LISTENING, THINKING, SPEAKING, EXPECTING. Ad ogni stato corrisponde una combinazione di colori nella barra della *Voice Chrome*. Nel caso, ad esempio, in cui venga notificato lo stato di IDLE, verrà mostrato il fragment sottostante, contenente il pulsante Alexa per iniziare la conversazione e tutte le altre icone delle varie sezioni, come nella figura 4.21, mentre per tutti gli altri stati ci sarà uno scambio con il fragment che riporta in basso a sinistra lo stato di Alexa e al centro il bottone per poter interrompere la conversazione con Alexa(figura 4.25). L'implementazione risulta come di seguito :

```
62    public void dialogStateChanged( final DialogState state ) {
63        mDialogState = state;
64        mLogger.postInfo( sTag,  message: "Dialog State Changed. STATE: " + state );
65        if ( mAutoVoiceChromeController != null ) {
66            if(( getConnectionStatus () == ConnectionStatus.CONNECTED ) ) {
67                mAutoVoiceChromeController.onStateChanged(
68                        convertToAutoVoiceChromeState( state ) );
69            } else {
70            // If Alexa is disconnected or pending and dialog state comes then AutoVoiceChrome should show red bar
71                        mAutoVoiceChromeController.onStateChanged( AutoVoiceChromeState.IN_ERROR );
72            }
73        }
74        mActivity.runOnUiThread( () → {
77            TextView t = (TextView) mActivity.findViewById(R.id.stateDialog);
78            switch (state){
79                case LISTENING:
80                    startConv = true;
81                    if(t != null)
82                        t.setText("LISTENING...");
83                    mActivity.findViewById(R.id.logout).setClickable(false);
84                    mActivity.findViewById(R.id.ty).setClickable(false);
85                    ImageView ty = mActivity.findViewById(R.id.ty);
86                    ImageView logout = mActivity.findViewById(R.id.logout);
87                    ty.setColorFilter(ContextCompat.getColor(ty.getContext(),
88                            R.color.drawerSection), android.graphics.PorterDuff.Mode.SRC_IN);
89                    logout.setColorFilter(ContextCompat.getColor(logout.getContext(),
90                            R.color.drawerSection), android.graphics.PorterDuff.Mode.SRC_IN);
91                    break;
92                case THINKING: {
93                    if(t != null)
94                        t.setText("THINKING...");
95                    break;
96                }
97                case SPEAKING:
98                        if(t != null)
99                    t.setText("SPEAKING...");
100                   break;
101               case EXPECTING:
102                   if(t != null)
103                       t.setText("EXPECTING...");
104                   break;
```

– *authStateChanged(final AuthState state, final AuthError error)* : questa funzione è utilizzata dall'engine per poter aggiornare lo stato di autorizzazione. Le combinazioni sono :

* *UNINITIALIZED* : lo stato iniziale. In questo stato si troverà l'applicazione all'apertura e verrà controllato se esiste un "refresh_token". Se non esiste, questo stato rimarrà inalterato.
* *REFRESHED* : questo stato indicato che l'operazione di login è avvenuta con successo, avendo ottenuto l'AuthToken.
* *EXPIRED* : token scaduto e va rinnovato.

* *UNRECOVERABLE_ERROR* : si sono verificati degli errori di rete o sul server.

L' implementazione prevede che sia semplicemente registrato il nuovo stato in una variabile di istanza.

– *connectionStatusChanged(final ConnectionStatus status, final ConnectionChangedReason reason)* : questa callback viene utilizzata per riportare gli stati della connessione : DISCONNECTED, PENDING, CONNECTED che indicano rispettivamente assenza di connessione con Alexa, fase di login in atto e connessione avvenuta con successo. Da questa funzione passa l'apertura della Home in fase di avvio. Infatti nel momento in cui l'applicazione viene avviata ed è presente un "refresh_token", l'engine inizierà la fase di refresh dell'AuthToken, notificando lo stato della connesione come PENDING. Quando l'AuthToken è ottenuto e questa funzione richiamata, se il ConnectionStatus risulta CONNECTED e l'AuthState, che sarà stato già aggiornato, equivale a REFRESHED allora si aprirà la Home. La funzione è stata implementata come in figura :

```
@Override
synchronized public void connectionStatusChanged( final ConnectionStatus status,
                                                  final ConnectionChangedReason reason ) {
    mConnectionStatus = status;
    if(status == ConnectionStatus.CONNECTED) {
        //connesso al cloud
        if(mAuthState == AuthState.REFRESHED) {
            mActivity.runOnUiThread(() -> {
                    mActivity.findViewById(R.id.wait1).setVisibility(View.INVISIBLE);
                    mActivity.findViewById(R.id.wait2).setVisibility(View.INVISIBLE);
                    mActivity.openFirstTimeGrid();

            });
            //connesso al LVC, è stato fatto il primo login e non ci hanno buttato fuori
        }else if( mActivity.getSharedPreferences( "com.amazon.sampleapp.SampleAppSharedPreferences",Context.MODE_PRIVATE ).contains("Re
            if ( mActivity.getSharedPreferences( "com.amazon.sampleapp.SampleAppSharedPreferences",Context.MODE_PRIVATE ).getString("Re
                mActivity.runOnUiThread(() -> {
                        mActivity.findViewById(R.id.wait1).setVisibility(View.INVISIBLE);
                        mActivity.findViewById(R.id.wait2).setVisibility(View.INVISIBLE);
                        mActivity.openFragmentGrid();
                });
        }
```

- **TemplateRuntimeHandler** : estende l'interfaccia TemplateRuntime. Da questo componente è possibile recuperare il contenuto visivo associato una determinata richiesta ed e˙ettuare il render sulla GUI in maniera opportuna. L'engine quindi andrà ad ad invocare due tipi di callback :

– *renderTemplate(String payload)* : da questa funzione è possibile capire il tipo di RenderTemplate che Alexa ha inviato e decidere di conseguenza in quale sezione dell'interfaccia andare a mostrare il contenuto. Dalla seguente implementazione è possibile vedere che nel momento in cui l'utente fa qualsiasi domanda di argomento generale come gli appuntamenti nel calendario, ricerca di un POI più vicino oppure che tempo fa in una città verrà passato a questa funzione il parametro "payload" nel quale sarà presente uno dei tipi : BodyTemplate1, BodyTemplate2, ListTemplate1, LocalSearchListTemplate1, TraÿcDetailsTemplate,WeatherTemplate e il contenuto associato.

```
63      @Override
64      public void renderTemplate( String payload ) {
65          try {
66              // Log payload
67              JSONObject template = new JSONObject( payload );
68              String type = template.getString( name: "type" );
69              Log.i(sTag,payload);
70              /**FORTUNATO**/
71              switch ( type ) {
72                  case "BodyTemplate1":
73                      viewAnswer.updateView( type: "BodyTemplate1",template);
74                      break;
75                  case "BodyTemplate2":
76                      viewAnswer.updateView( type: "BodyTemplate2",template);
77                      break;
78                  case "ListTemplate1":
79                      viewList1.updateView(template);
80                      break;
81                  case "WeatherTemplate":
82                      viewWeather.updateView(template);
83                      break;
84                  case "LocalSearchListTemplate1":
85                      viewSearchList1.updateView(template);
86                      break;
87                  case "TrafficDetailsTemplate":
88                      break;
89                  default:
90                      break;
91              }
92          } catch ( JSONException e ) {
93              mLogger.postError( sTag, e.getMessage() );
94          }
95      }
```

– *renderPlayerInfo(String payload)* : quando invece l'utente chiede di avviare una contenuto multimediale streaming come una webradio o una playlist da Amazon Music, Alexa oltre a inviare l'audio stream, invierà anche i metadati associati alla traccia che riporteranno il titolo, l'artista, l'album, nome della radio, icona del provider, immagine dell'album, ma anche i controlli che possono essere supportati dal PlaybackController. Queste informazioni per essere mostrate sulla GUI verranno passate nel "payload" della funzione, la quale provvederà ad interpretarlo in maniera opportuna :

```
97          @Override
98          public void renderPlayerInfo( String payload ) {
99              Log.i(sTag,payload);
100             /**FORTUNATO**/
101             try {
102                 JSONObject playerInfo = new JSONObject( payload );
103                 String audioItemId = playerInfo.getString( name: "audioItemId" );
104                 JSONObject content = playerInfo.getJSONObject( "content" );
105                 JSONObject audioProvider = content.getJSONObject( "provider" );
106                 String providerName = audioProvider.getString( name: "name" );
107                 viewPlayerInfo.updateView(content,providerName);
108             } catch ( JSONException e ) {
109                 mLogger.postError( sTag, e.getMessage() );
110             }
111         }
```

(4) **Navigation Module** : a questo modulo appartiene il NavigationHandler. In fase di configurazione dell'engine è possibile caricare dalla piattaforma, tramite un file Json.

- **NavigationHandler** : estende l'interfaccia Navigation. Al suo interno si può gestire l'avvio verso una destinazione ma anche riportare lo stato corrente della navigazione per abilitare altre skill come ad esempio cancellare la navigazione oppure sapere quanto manca per la destinazione corrente.
  - *setDestination(String payload)* : quando l'utente chiede ad Alexa di portarlo verso una certa destinazione, l'Engine riceve una direttiva StartNavigation e chiama questa callback. Nel parametro payload saranno disponibili le coordinate per quella destinazione da inserire nell'*intento* per avviare l'applicazione della piattaforma che gestisce la navigazione. Il codice seguente mostra l'*intento* utilizzato per avviare la navigazione con Google Maps:

```
@Override
public boolean setDestination( String payload ) {
    // Handle navigation to destination here
    try {

        Log.i(sTag,payload);
        JSONObject template = new JSONObject( payload );
        JSONObject destination = template.getJSONObject("destination");
        JSONObject coordinate = destination.getJSONObject("coordinate");

        /**FORTUNATO**/
        /**Here it is possible to handle turn-by-turn navigation and then to start for example google maps via intent.
        String latitude = coordinate.getString( name: "latitudeInDegrees");
        String longitude = coordinate.getString( name: "longitudeInDegrees");
        Uri gmmIntentUri = Uri.parse("google.navigation:q="+latitude+","+longitude+"&mode=d");
        Intent mapIntent = new Intent(Intent.ACTION_VIEW, gmmIntentUri);
        mapIntent.setPackage("com.google.android.apps.maps");
        mActivity.startActivity(mapIntent);

        return true;
    } catch ( JSONException e ) {
        mLogger.postError( sTag, e.getMessage() );
        return false;
    }
}
```

– *getNavigationState()* : l'engine richiama periodicamente questa callback per riportare lo stato della navigazione ad Alexa. Se si riporta uno stato vuoto le skill aggiuntive non saranno attivate, mentre nel caso si riporti un "full navigation state" esse saranno abilitate. Le scelte fatte prevedono per questa prima fase l'invio di una stringa vuota e dunque anche la funzione *cancelNavigation()* non sarà richiamata.

E' importante segnalare che per questo modulo Amazon prevede ulteriori funzionalità nelle future versioni dell'SDK.

(5) **CarControl Module** : questo modulo permette di utilizzare Alexa per controllare le funzionalità del veicolo ed è organizzato secondo la logica di una Smart Home Skill.
Nella fase iniziale di configurazione dell'engine è possibile infatti registrare quelli che sono chiamati **endpoints** del veicolo(che fisicamente rappresentano ventole, luci, fari, specchietti,sedili,finestrini,etc) e il loro relativo "friendlyName", utilizzato dall'utente per identificare l'oggetto che vuole controllare. Quando l'engine comunicherà la configurazione di questo modulo, Alexa cloud registrerà per quell'utente la presenza di vari dispositivi controllabili con i comandi tipici di una Smart Home Skill e Alexa Auto Client fungerà da proxy per inviare eventi e ricevere direttive per conto degli endpoints connessi.
In fase di definizione oltre a definire gli endpoints, vanno associate e comunicate le loro **capabilities** che stabiliscono i comandi abilitati per quel determinato componente e sono principalmente 4 :

- *Power Controller* : abilita il controllo per accendere e spegnere l'endpoint.
- *Toggle Controller* : abilita il controllo per accendere e spegnere una proprietà(es. sbrinatore) di un endpoint(parabrezza).
- *Mode Controller* : abilita il controllo per impostare una proprietà di un endpoint, presa da un insieme di valori dichiarati, ad esempio il colore rosso/verde/giallo di una lampadina.
- *Range Controller* : abilita il controllo di un endopoint per impostarlo su un determinato valore numerico o di quantità come basso, medio o alto.

Il veicolo può essere suddiviso in **zone**(zona guidatore, passeggero anteriore, passeggero posteriore sinistro, etc) e in ognuna di essi si andranno

a indicare gli endpoints secondo i loro *id*, presentati precedentemente, in modo da evitare ridondanze. Così facendo si potrà chiedere ad Alexa ad esempio : "apri il finestrino del guidatore".
Infine per ogni endpoints,capability e zone si definiscono gli **asset** che stabiliscono il sostantivo convenzionale usato per far riferimento a uno di essi. Si possono crearne di nuovi oppure usare quelli di default, ad esempio definendo "Alexa.Automotive.DeviceName.AirConditioner" come asset per l'endpoint "rear.ac" si userà la parola "aria condizionata" oppure "condizionatore d'aria" per indicarlo.

- **CarControlHandler** : estende l'Interfaccia CarControl. In questa classe dopo una prima inizializzazione in cui si va a definire con l'oggetto CarControlConfiguration la configurazione del veicolo, verranno richiamate le callback per poter gestire il tipo di impostazione richiesta dall'utente. In particolare per ogni funzione è possibile andare a identificare il controller per quel determinato endpoint e dunque interfacciarsi con i dispositivi del veicolo, utilizzando librerie per poter sfruttare lo standard Controller Area Network, noto anche come **CAN-bus**. Purtroppo trattandosi di un prototipo e dovendo avere a disposizione un veicolo, le funzionalità di questo modulo non sono state implementate. Di seguito sono mostrate le callback messe a disposizione :

```
    /// PowerController
    @Override
    public void turnPowerControllerOn(String endpointId) {...}

    @Override
    public void turnPowerControllerOff(String endpointId) {...}

    @Override
    public boolean isPowerControllerOn(String endpointId) throws Exception {...}

    /// ToggleController
    @Override
    public void turnToggleControllerOn(String endpointId, String controllerId) {...}

    @Override
    public void turnToggleControllerOff(String endpointId, String controllerId) {...}

    @Override
    public boolean isToggleControllerOn(String endpointId, String controllerId) throws Exception {...}

    /// RangeController
    @Override
    public void setRangeControllerValue(String endpointId, String controllerId, double value) {...}

    @Override
    public void adjustRangeControllerValue(String endpointId, String controllerId, double delta) {...}

    @Override
    public double getRangeControllerValue(String endpointId, String controllerId) throws Exception {...}

    /// ModeController
    @Override
    public void setModeControllerValue(String endpointId, String controllerId, String value) {...}

    @Override
    public void adjustModeControllerValue(String endpointId, String controllerId, int delta) {...}

    @Override
    public String getModeControllerValue(String endpointId, String controllerId) throws Exception {...}
}
```

(6) **Phone Call Controller Module** : con questo modulo si rendono disponibili le funzionalità per gestire le chiamate. L'utente, dopo aver collegato il telefono via Bluetooth alla piattaforma, potrà sincronizzare i contatti nella sua rubrica sull'Alexa Cloud ed effettuare telefonate. Tuttavia, è necessario che la piattaforma svolga il ruolo di ricevente e che quindi abbia abilitati i profili Bluetooth definiti nella sezione 2.2.7. Le due classi che fanno parte di questo modulo sono chiamate **PhoneCallControllerHandler**, utilizzata per gestire gli stati delle telefonate e **AddressBookHandler** che si occuperà di registrare i contatti del telefono accoppiato via Bluetooth sull'Alexa Cloud in modo da dare successivamente la possibilità all'utente di avviare le telefonate, pronunciando come destinatario il nome del suo contatto.

- **PhoneCallControllerHandler** : estende l'interfaccia PhoneCallController. In questo oggetto ci sono le callback da implementare, richiamate dall'engine in seguito a richieste dell'utente, come ad esempio : "chiama <numero>", "chiama <nome>", "riaggancia", "rispondi"(per telefonate in arrivo). Per la piattaforma specifica è stata implementate la funzione di chiamata per un contatto, lasciando alle fasi successive l'implementazione dei metodi restanti. Per far questo nel modulo è compreso un *BroadcastReceiver* nel quale si va a controllare il tipo del device che si è appena connesso o disconesso. Nel caso sia stato rilevato un nuovo device compatibile si richiama la funzione dell'AddressBookHandler per l'upload dei contatti e si avvisa l'engine, mediante la funzione *togglePhoneConnectionState(true)* che un telefono si è connesso, attivando dunque le skill per telefonare. Nel momento in cui il device si disconnette si procede ad avvisare l'engine e ad eliminare i contatti nell'Alexa Cloud.

```
/**FORTUNATO**/
/**Only one device can be connected to Engine for PhoneCallController**/
private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
    public void onReceive (Context context, Intent intent) {
        String action = intent.getAction();
        if (BluetoothDevice.ACTION_ACL_CONNECTED.equals(action)) {
            //TO DO check profile device connected
            if(device == null) {
                mAudioBluetoothConnected.start();
                togglePhoneConnectionState( enable: true);
                device = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
                address = device.getAddress();
                ((MainActivity) mActivity).getmAddressBook().updateContacts(device);
            }
```

```
        }else if (BluetoothDevice.ACTION_ACL_DISCONNECTED.equals(action)) {
                BluetoothDevice currentDevice = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
                if(currentDevice.getAddress().compareTo(address) == 0) {
                    getmAudioBluetoothDisconnected.start();
                    togglePhoneConnectionState( enable: false);
                    ((MainActivity) mActivity).getmAddressBook().updateContacts( device: null);
                    device = null;
                    address="";
                }
            }
        }
    };
```

Un controllo simile viene fatto anche nel momento in cui si istanzia la classe. A telefono connesso, l'utente quindi potrà chiedere ad Alexa di telefonare a un suo contatto. L'engine, dopo aver interpretato la direttiva associata, richiama la funzione *dial(String payload)*. Il parametro payload conterrà tutti i dati dell'utente destinatario, tra cui il numero. A questo punto nella funzione si andrà a richiamare tramite un *intento* l'applicazione della piattaforma che si occupa delle telefonate, passandole il numero a cui telefonare.

```
    @Override
    public boolean dial(String payload) {
        // Handling should not block the caller
        String callId = "";
        String calleeNumber = "";

        try {
            JSONObject obj = new JSONObject(payload);
            callId = obj.getString( name: "callId");
            calleeNumber = getCalleeDefaultAddressValue(obj);
        } catch (JSONException e) {
            mLogger.postError(sTag,  message: "Error parsing dial directive payload: "
                    + e.getMessage());
            return false;
        }

        mCallId = callId;
        mCurrentCallNumber = calleeNumber;
        mCallState = CallState.DIALING;
        logCallInfo( msg: "dial()");
        callStateChanged(CallState.DIALING, callId);

        startDialingCallTimer(sDialingToRingingDelay);
        mLocalCallStarted = true;
        /**FORTUNATO**/
        /*Start call*/
        Intent intent = new Intent(Intent.ACTION_CALL, Uri.parse("tel:" + mCurrentCallNumber));
        mActivity.startActivity(intent);
        //updateGUI();
        return true;
    }
```

- **AddressBookHandler** : estende l'interfaccia AddressBook. Si occupa di e˙ettuare l'upload dei contatti telefonici ad Alexa Cloud. La sua funzionalità verrà sfruttata come descritto precedentemente dal modulo *PhoneCallControllerHandler*. Nella funzione *updateContacts* si vanno a leggere o rimuovere i contatti in base al parametro passato. Con la funzione *addAddressBook* si informa l'engine che è possibile aggiungere nuovi contatti. L'engine a questo punto richiamerà

la callback *getEntries* e si andranno a leggere i contatti della rubrica.

```
/**FORTUNATO**/
public void updateContacts(BluetoothDevice device){
    /**to be implemented*/
    if(device == null) {
        Log.i(sTag,  msg: "Removing contacts from device name : ");
        /**Remove contacts phone from cloud Alexa**/
        currentDevice = null;
        removeAddressBook(sPhoneBookId);
    }else{
        Log.i(sTag,  msg: "Adding contacts from device name : " + device.getName());
        /**Adding contacts phone**/
        currentDevice = device;
        addAddressBook(sPhoneBookId,  name: "PhoneBook", AddressBookType.CONTACT);
    }

}
```

Alla funzione *getEntries* viene passato un parametro per poter identificare la sorgente di lettura, dato che è possibile leggere i contatti da diverse sorgenti quali file, oggetti di configurazione o device Bluetooth. Nel caso del telefono, la funzione *parseFileasPhoneContacts(factory)* andrà a fare l'upload dei contatti del telefono, utilizzando *addName* dell'oggetto factory.

```
@Override
public boolean getEntries(String contactsSourceId, IAddressBookEntriesFactory factory) {
    boolean success = false;
    if( contactsSourceId.equals( sContactsSourceId ) ) {
        success = parseFileAsContacts( mContactsDataPath, factory );
    } else if( contactsSourceId.equals( sNavigationFavoritesSourceId ) ) {
        success = parseFileAsNavigationFavorites( mNavigationFavoritesDataPath, factory );
    }else if(contactsSourceId.equals( sPhoneBookId ))
        success = parseFileasPhoneContacts(factory);

    mLogger.postInfo( sTag, String.format( "success status of adding contacts with ID: (%s)
    return success;
}
```

4.2.3 Local Voice Control Extension

Amazon permette agli sviluppatori di estendere le capacità di Alexa Auto SDK con delle estensioni da integrare mediante "*patch*". Tra queste troviamo AlexaComms, AmazonLite, Voice Chrome e Local Voice Control Extension. Nel caso specifico della applicazione sviluppata, oltre a integrare l'estensione VoiceChrome, che permette di implementare gli avvisi visivi secondo lo standard imposto da Amazon, si è deciso di integrare anche il Local Voice

Control.
Questa estensione riporta una funzionalità molto ricercata nel settore Automotive per un assistente vocale, ossia la possibilità di utilizzare Alexa in assenza di connessione. Con la versione 2.1 presente sul prototipo, tuttavia, non tutte le skill sono disponibili ma solo quelle inerenti all'intrattenimento (radio e riproduzione di sorgenti locali), navigazione(verso le destinazioni preferite comunicate in fase di avvio), telefono e controllo del veicolo. Con il passare del tempo e gli aggiornamenti di Alexa Auto SDK a nuove versioni, anche questa estensione sarà arricchita di nuove caratteristica. L'architettura con il LVC integrato diventa dunque come illustrato in figura :

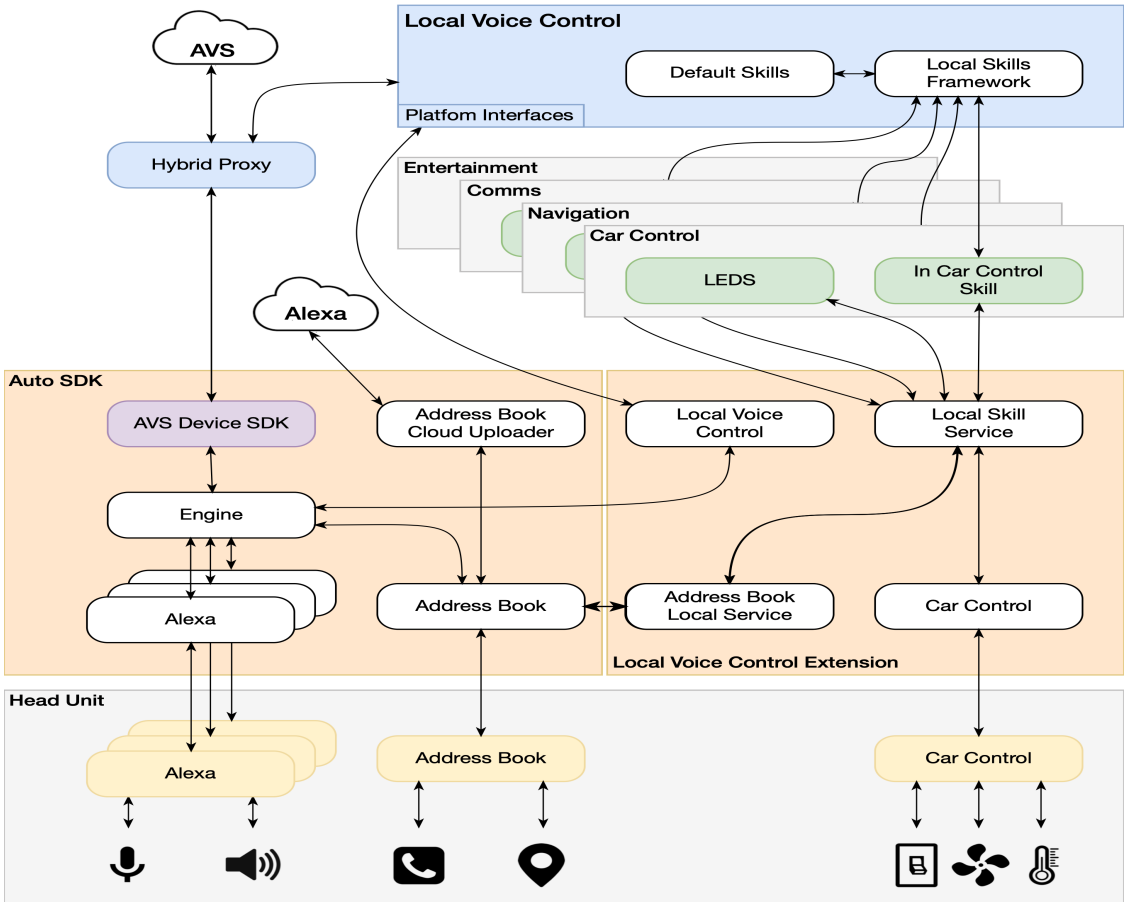


Figura 4.29: Architettura Local Voice Control

Il flusso di comunicazione non cambia, basandosi sempre sullo scambio di eventi e direttive, inoltre non vengono aggiunte nuove interfacce la cui implementazione dipenda dalla piattaforma. L'integrazione infatti si basa essenzialmente sull'applicazione di "patch" e sull'inserimento di nuove librerie

che vanno a modificare le funzionalità già presenti ma anche ad aggiungerne di nuove, in modo da poter comunicare con i nuovi componenti.

- **Local Voice Control APK** : per poter interagire a livello locale è ovviamente necessario un engine che sia in esecuzione sulla piattaforma e nel caso di Android, viene fornito un APK da avviare necessariamente. Questo *Local Voice Control APK* simula l'engine Alexa quindi sarà fornito di un Alexa Speech Recognition (ASR), Natural Language Understanding (NLU), Text-To-Speech (TTS) e dell'implementazione delle skill menzionate precedentemente, potendo perciò riconoscere eventi e rispondere con le direttive corrispondenti.

- **Hybrid Proxy** : questo componente si occupa dello scambio di eventi e direttive sia con Alexa Cloud che con il Local Voice Control Engine.

- **Local Voice Control Module Auto SDK** : poichè la comunicazione tra l'engine locale e Alexa Auto SDK avviene tramite *Inter-Process Communication(IPC)*, questo modulo si occupa di avviare l'APK Local Voice Control e inviargli il percorso dei sockets con i relativi permessi su cui l'engine locale potrà andare a comunicare con i vari componenti di Alexa Auto.

- **Local Skill Service Module Auto SDK** : per le funzionalità di controllo auto, di navigazione e telefono, occorre sincronizzare i dati dei vari moduli con le skills corrispondenti. Essendo in un contesto locale, questo componente si occupa dello scambio di dati tra i servizi delle "local skills" e il dispositivo.

- **Address Book Local Service** : questo modulo non espone interfacce da implementare ma si appoggia sull"*Local Skill Service* per comunicare i contatti e le destinazioni preferite alle *Local Skill Navigation e Communication*. Per recuperare i dati utilizzera la callback *getEntries* dell'AddressBookHandler.

- **Car Control** : questo modulo aggiunge la funzionalità al componente omonimo già presente sull'Auto SDK per poter comunicare la configurazione del veicolo anche alla skill Car Control dell'engine locale.

Per poter utilizzare questa funzionalità, si è scelto di obbligare l'utente a e˙ettuare il login con Alexa Cloud almeno una volta. Dalla seconda volta in poi, l'applicazione, vedendo la presenza di un AuthToken, nel caso di assenza di connessione, si collegherà al Local Voice Control. Verrà mostrata la Home

ma senza i pulsanti per cambiare lingua in alto a sinistra e per e˙ettuare il logout in alto a destra.
A titolo d'esempio il seguente diagramma mostra la sequenza di interazioni nei caso ci fosse un assenza di connessione di rete:

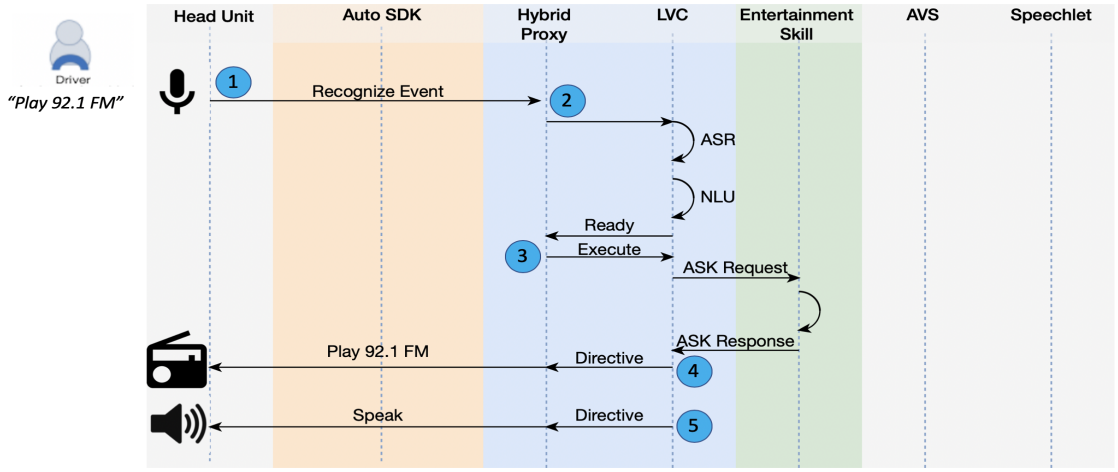


Figura 4.30: Diagramma interazione con estensione Local Voice Control

(1) L'utente chiede di riprodurre la frequenza radio 92.1FM

(2) L' Hybrid Proxy sa che non c'è connessione con l'AVS, manda perciò un evento Recognize solo all'LVC.

(3) LVC processa la richiesta (ASR+NLU), crea una direttiva e comunica all'HybridProxy di essere pronto

(4) L'Hybrid Proxy avvisa LVC che può inviare la direttiva

(5) La direttiva verrà interpretata dall'engine e passata all'interfaccia che si occupa di eseguire l'azione.

(6) LVC manda infine una direttiva Speak : "Playing 92.1 FM"

Nel caso in cui ci sia connessione con la rete, quello che cambia è che l'Hybrid Proxy sa che c'è connessione con Alexa Cloud, quindi manda a entrambi l'evento Recognize. Ricevendo infine una direttiva dalla rete, dirà al'LVC di non inviare nulla e dunque di scartare la risposta che ha costruito.

Conclusioni

Il mercato degli assistenti vocali è in rapida espansione e la gara che stabilirà il migliore è appena cominciata. Tutto è inevitabilmente legato ai numeri, quindi con il passare degli anni chi riuscirà a mettere il proprio assistente vocale sul maggior numero di dispositivi in commercio avrà inevitabilmente un ruolo predominante. Ecco perché la capacità di portare il proprio assistente vocale in tutti i settori possibili farà la differenza e chi offrirà il maggior numero di funzionalità e caratteristiche migliori, rendendo la conversazione sempre più naturale al punto di non riuscire più a distinguere tra persona reale e IA, sicuramente avrà maggiori possibilità di successo commerciale. Inoltre ci sono alte probabilità che aprire la propria tecnologia a sviluppatori terzi, dando loro la libertà di poterla integrare su qualsiasi dispositivo, senza passare dall'azienda proprietaria, ma utilizzando kit di sviluppo preconfigurati ed API pubbliche, come ha fatto Amazon con Alexa Voice Service e Alexa Skill Kit oppure Google con il suo Google Assistant SDK, si riveli una mossa vincente. Infatti, partendo con l'obiettivo di integrare Alexa su un prototipo di Infotainment basato su Android Automotive, è bastato Alexa Auto SDK, anche se è stato necessario lo studio approfondito della documentazione fornita nell'Alexa Voice Service per comprendere ogni suo componente. Sebbene possano nascere problemi con l'implementazione audio dipendente dalla piattaforma o con altri specifici componenti hardware o software, l'SDK ha fornito tutta la parte implementativa necessaria ad organizzare l'interazione, lasciando la libertà allo sviluppatore di utilizzare come meglio crede i moduli e il modo di integrazione con la piattaforma. Si può scegliere di creare un'applicazione Android che comunichi con le altre, come in questo caso, ma è anche possibile scegliere un livello di integrazione più profondo, rendendo accessibile Alexa da ogni parte del sistema e dunque come un servizio permanente. In questo caso sarebbe stato necessario agire sui sorgenti Android. Un ulteriore vantaggio è che appena Alexa Auto SDK viene montato sulla piattaforma, una serie di skill native sono già disponibili pronte all'uso, dovendo solo pianificare come presentare i contenuti visivi

delle risposte e implementare le azioni indicate nelle direttive. Questo ha permesso di realizzare la parte grafica e implementare le funzionalità tipiche come la telefonata, la navigazione o la riproduzione di varie sorgenti audio, rispettando i tempi prestabiliti per poterla presentare in eventi dedicati come al CES 2020.
Un altro importante fattore è dato indubbiamente dalla creazioni di nuove skill e tutti i competitors si stanno allineando per fornire gli strumenti necessari agli sviluppatori. Alexa Skill Kit, come è stato illustrato, non passa necessariamente per la creazione di un prodotto Built-In ad hoc. Infatti che siano Home Skills, sviluppate dalle aziende che presentano i loro prodotti compatibili, oppure Custom Skills, sviluppabili da chiunque, per poter fornire nuovi servizi vocali, ad esempio, sullo stato del traffico, sulla quantità di parcheggi disponibili in una certa area oppure ancora sull'acquisto guidato in uno store, si presuppone in ogni caso l'utilizzo di un dispositivo Alexa per poterle utilizzare e questo potrebbe essere un motivo in più per dominare il mercato degli assistenti vocali.
Tuttavia anche se sono strumenti già di per sè molto flessibili, Amazon e i gli altri concorrenti si stanno adoperando per renderli sempre migliori al fine di poter realizzare le caratteristiche più ricercate ancora non disponibili, come ad esempio nell'ambito automotive la possibilità di creare skill di diagnostica in cui, al verificarsi di un evento del veicolo, sia Alexa ad iniziare il dialogo. Ciò implicherebbe una ridefinizione del ruolo client-server e dunque del protocollo di comunicazione ma darebbe sicuramente un'impronta più umana e naturale a questo tipo di tecnologia.
In ogni caso, aldilà di chi avrà il monopolio in questo settore, nel futuro tecnologico gli assistenti vocali avranno un ruolo da protagonisti assoluti.

Bibliografia

[1] *Natural-language understanding*. URL: https://medium.com/@ODSC/getting-to-know-natural-language-understanding-f18a0dc5c97d.

[2] *Alexa Connected Device*. URL: https://developer.amazon.com/it-IT/alexa/devices/connected-devices/development-resources.

[3] *USI Development Kit for ACK*. URL: https://developer.amazon.com/en-US/docs/alexa/ack/usi-dev-kit-overview.html.

[4] *Smart Home Zigbee Support*. URL: https://www.google.it/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjX3pnc_ozxAhUKCewKHaMtANsQFjAAegQIAxAD&url=https%3A%2F%2Fzigbeealliance.org%2Fwp-content%2Fuploads%2F2019%2F11%2Fdocs-05-3474-21-0csg-zigbee-specification.pdf&usg=AOvVaw0UQdvxUu2ZDVoD9bB3cay4.

[5] *Alexa Gadgets Toolkit*. URL: https://developer.amazon.com/it-IT/docs/alexa/alexa-gadgets-toolkit/understand-alexa-gadgets-toolkit.html.

[6] *AuthorizeCompanionApp*. URL: https://developer.amazon.com/en-US/docs/alexa/alexa-voice-service/authorize-companion-app.html.

[7] *AuthorizeCompanionSite*. URL: https://developer.amazon.com/en-US/docs/alexa/alexa-voice-service/authorize-companion-site.html.

[8] *AuthorizeOnProduct*. URL: https://developer.amazon.com/en-US/docs/alexa/alexa-voice-service/authorize-on-product.html.

[9] *SynchronizeState*. URL: https://developer.amazon.com/en-US/docs/alexa/alexa-voice-service/system.html#synchronizestate.

[10] *Amazon Developer Console*. URL: https://developer.amazon.com/en-US/docs/alexa/devconsole/about-the-developer-console.html.

[11] *ADP8155*. URL: https://www.lantronix.com/products/sa8155p-automotive-development-platform/#tab-techspecs.

[12] *AndroidAutomotive*. URL: https://source.android.com/devices/automotive.

[13] *Android Intent*. URL: https://developer.android.com/reference/android/content/Intent.

[14] *ADB Tool*. URL: https://developer.android.com/studio/command-line/adb.

[15] *Logcat Tool*. URL: https://developer.android.com/studio/command-line/logcat.

[16] *JNI*. URL: https://developer.android.com/training/articles/perf-jni.

[17] *AVS UX Design*. URL: https://developer.amazon.com/en-US/docs/alexa/alexa-voice-service/ux-design-overview.html.

[18] *Observer Pattern*. URL: https://www.baeldung.com/java-observer-pattern.