MegAlexa

Arricchitore di skill di Amazon Alexa

Manuale dello Sviluppatore

Gruppo ZeroSeven

Versione	0.2.0
Data Redazione	2019-03-29
Redazione	Mirko Franco
	Stefano Zanatta
Verifica	Matteo Depascale
Approvazione	Ludovico Brocca
Uso	Esterno
Distribuzione	Prof. Tullio Vardanega
	Prof. Riccardo Cardin
	Gruppo ZeroSeven
	Zero12 s.r.l.
Email di contatto	zerosevenswe@gmail.com

Registro delle modifiche

Versione	Data	Descrizione	Autore	Ruolo
0.2.0	2019-04-10	Approvazione documento per rilascio RQ	Ludovico Brocca	Responsabile
0.2.0	2019-04-10	Verifica documento	Mirko Franco	Verificatore
0.1.1	2019-04-09	Stesura §5, §4.1	Stefano Zanatta	Progettista
0.1.0	2019-04-09	Verifica §2.1, §3, §2.3	Gian Marco Bratzu	Progettista
0.0.6	2019-04-08	Stesura §3.2.1, §2.3	Stefano Zanatta	Progettista
0.0.5	2019-04-05	Stesura capitolo §3	Mirko Franco	Progettista
0.0.4	2019-04-05	Stesura capitolo §2.1	Mirko Franco	Progettista
0.0.3	2019-03-30	Stesura capitolo §1	Mirko Franco	Progettista
0.0.2	2019-03-30	Stesura struttura documento	Mirko Franco	Progettista
0.0.1	2019-03-29	Creato documento	Mirko Franco	Progettista

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Scopo del documento

Il presente documento vuole essere una guida introduttiva del software *MegAlexa*, indirizzata agli sviluppatori che volessero adattarlo o estenderlo. Vengono spiegate le tecnologie interessate, l'architettura in dettaglio e le possibilità di estensione.

1.2 Scopo del prodotto

Lo scopo del progetto è lo sviluppo di un applicativo Mobile in grado di creare delle routine personalizzate per gli utenti gestibili tramite $Alexa_G$ di $Amazon_G$. L'obiettivo è la creazione di una $skill_G$ in grado di avviare $workflow_G$ creati dagli utenti fornendogli dei $connettori_G$.

1.3 Glossario

Al fine di evitare ogni ambiguità di linguaggio e massimizzare la comprensione dei documenti, i termini tecnici, di dominio, gli acronimi e le parole che necessitano di essere chiarite, sono riportate nel *Glossario v4.0.0*. Ogni occorrenza di vocaboli presenti nel Glossario è marcata da una "G" maiuscola in pedice.

Capitolo 2

Procedura di installazione

2.1 Requisiti di Sistema

MegAlexa è composta da un'applicazione compatibile con la maggior parte dei dispositivi *Android*$_G$ e da una skill Alexa.

2.1.1 Applicazione

L'applicazione è compatibile con tutti i dispositivi Android con versione 4.4 o superiore. Per poter modificare e ampliare la app i seguenti requisiti devono essere soddisfatti:

- **IDE Android Studio**$_G$**:** necessaria per eseguire e testare la app nel corso del suo sviluppo;
- **Git**$_G$**:** necessario per effettuare il `clone` della repository e il versionamento successivo del codice;
- **Gradle:** per il download automatico delle dipendenze e la compilazione del codice (consigliata versione 4.10.0 o superiore).

2.1.2 Skill

La skill è compatibile con tutti i dispositivi Amazon Echo.

2.2 Installazione App

Per installare la app è obbligatorio seguire i seguenti passi:

1. **Acquisire la repository:** eseguire il comando `git clone` seguito dal seguente URL: *https://github.com/sgt390/ProgettoSweCodice.git*;

2. **Registrare il dispositivo:** accedere alla console di sviluppo offerta da amazon con le credenziali inviate dai membri del gruppo e registrare una key univoca per la applicazione denominata *MegAlexa*[1];

3. **Posizionamento della key:** una volta aperto Android Studio, creare una cartella nel percorso `/MegAlexa/app/src/main`, nominarla `assets`, creare un file di testo nominato `api_key.txt` e posizionarlo nella cartella appena creata;

4. **Gradle Sync:** a questo punto, se la procedura è stata eseguita correttamente, Gradle dovrebbe scaricare in automatico le dipendenze per l'avvio della app, nel caso in cui ciò non avvenga, eseguire il comando `./gradlew build` nella cartella di root del progetto;

5. **Compilazione ed esecuzione:** la compilazione può avvenire mediante il comando `./gradlew build` oppure mediante la pressione dell'icona a forma di martello presente in alto su Android Studio (vedi figura 2.1), l'esecuzione avviene alla pressione del tasto run presente nell'IDE(vedi figura 2.1).
   Alla prima esecuzione verrà richiesta l'installazione di una versione Android per l'emulatore: scegliere l'opzione più gradita e continuare.
   In alternativa, è possibile eseguire l'applicazione su un dispositivo Android predisposto correttamente(per maggiori dettagli visitare https://developer.android.com/training/basics/firstapp/running-app).

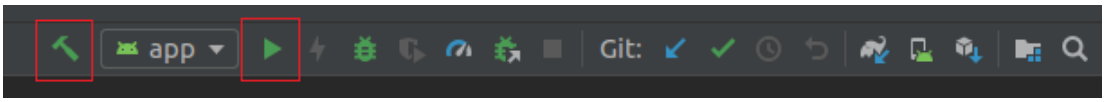

Figura 2.1: *Tasti Build e Run*

2.3 Installazione Skill

Per ognuno dei sequenti comandi è richiesta l'installazione del package manager **npm**. Installazione della skill e delle sue dipendenze:

[1] https://developer.amazon.com/loginwithamazon/console/site/lwa/overview.html

- clonare la repository attraverso il comando *git clone https://github.com/sgt390/MegAlexaSkill/*;
- eseguire il comando *npm install* per installare automaticamente le dipendenze.

Pubblicare la skill in AWS Lambda:

- installare il programma 7z e inserire il suo eseguibile tra le variabili di sistema;
- installare e configurare aws-cli[2];
- da terminal, eseguire il comando *npm run publish-lambda*.

Eseguire i test di unità:

- da terminal, eseguire il comando *npm run unitTest*.

Eseguire i test di integrazione:

- da terminal, eseguire il comando *npm integrationTest*.

[2] https://aws.amazon.com/it/cli/

Capitolo 3

Tecnologie utilizzate

3.1 Amazon Web Service

Amazon Web Service è una piattaforma di cloud computing sicura che offre servizi di calcolo, memorizzazione, distribuzione di contenuti e altre funzionalità per aiutare il business ad essere scalabile e crescere con facilità.
AWS_G fornisce infatti prodotti e servizi per costruire applicazioni, anche sofisticate, in modo flessibile, scalabile, economico e con un'ottima resistenza ai guasti.

3.1.1 AWS DynamoDB

Amazon $DynamoDB_G$ è un database che supporta i modelli di dati di tipo documento e di tipo chiave-valore che offre prestazioni di pochi millisecondi a qualsiasi livello. Si tratta di un database multi master, multi regione e completamente gestito che offre sicurezza integrata, backup, ripristino e cache in memoria per applicazioni Internet. DynamoDB può gestire oltre 10 trilioni di richieste al giorno e supporta picchi di oltre 20 milioni di richieste al secondo.

3.1.2 AWS Lambda

AWS Lambda$_G$ consente di eseguire codice senza dover effettuare il provisioning né gestire il server. Le tariffe sono calcolate in base ai tempi di elaborazione.
Con Lambda, è possibile eseguire codice per qualunque tipo di applicazione o di servizio back-end, senza alcuna amministrazione. Una volta caricato il codice Lambda si prende carico delle azioni necessarie per eseguirlo e ricalibrarne le risorse con la massima disponibilità. É possibile configurare il

codice in modo che venga attivato automaticamente da altri servizi AWS oppure che venga richiamato direttamente da qualsiasi app Web o mobile.

3.1.3 AWS API Gateway

AWS *API Gateway*$_G$ è un servizio completamente gestito che semplifica la creazione, la pubblicazione, la manutenzione e la protezione delle API su larga scala. Con semplicità è possibile creare e configurare API REST che fungano da "porta di ingresso" per le applicazioni, per consentire l'accesso ai dati, alla logica di business o alle funzionalità dai propri servizi back-end. API Gateway gestisce tutte le attività di accettazione ed elaborazione relative a centinaia di migliaia di chiamate ad API simultanee, inclusi gestione del traffico, controllo di accessi e autorizzazioni, monitoraggio e gestione delle versione delle API. Gateway non prevede alcuna tariffa minima né investimenti iniziali. Vengono addebitati solo i costi di chiamate API ricevute e i volumi di dati trasferiti in uscita e con il modello tariffario a scaglioni di API Gateway potrai ridurre i costi al variare dell'utilizzo delle API.

3.1.4 AWS CloudWatch

AWS CloudWatch$_G$ è un servizio di monitoraggio e gestione creato per gli sviluppatori, operatori di sistema, ingegneri responsabili del sito e manager IT. CloudWatch fornisce dati e analisi concrete per monitorare le applicazioni, capire e rispondere ai cambiamenti di prestazioni a livello di sistema, ottimizzare l'utilizzo delle risorse e ottenere una visualizzazione unificata dello stato di integrità operativa.

3.1.5 Kotlin

Kotlin$_G$ è un linguaggio di programmazione general purpose, multi-paradigma, open source sviluppato dall'azienda di software JetBrains.
Kotlin si basa sulla JVM (Java Virtual Machine) ed è ispirato ad altri linguaggi di programmazione tra i quali Scala e lo stesso Java, mentre ulteriori spunti sintattici sono stati presi da linguaggi classici, come il Pascal e moderni come Go o F#.
Il vantaggio maggiore rispetto a Java risulta essere la sintassi più leggibile, insieme alla possibilità di eseguire file Java all'interno di progetti Kotlin (visto che entrambi utilizzano la JVM per produrre il bytecode).

3.1.6 XML

XML è un metalinguaggio per la definizione di linguaggi di markup, ovvero un linguaggio marcatore basato su un meccanismo sintattico che consente di definire e controllare il significato degli elementi contenuti in un documento o in un testo.
Android fornisce gli schema necessari per comporre widget e posizionarli sullo schermo, separando ulteriormente la parte logica da quella grafica.

3.1.7 Node.js

La skill è sviluppata attraverso un progetto *Node.js*$_G$. Tutte le classi (apparte index.js) sono scritte in *TypeScript*$_G$, ma vengono compilate in Javascript prima di essere trasferite in AWS Lambda.

3.1.8 Npm

Gestore di package per node.js. Permette l'esecuzione di comandi personalizzati (come descritto in §2.3).

3.1.9 7z

Permette di comprimere i file da linea di comando. È richiesto dalla procedura di build della skill.

3.2 Dipendenze esterne

3.2.1 Alexa Skill

Tutte le dipendenze principali si trovano nel file *package.json*, mentre quelle secondarie (cioè le dipendenze delle dipendenze) stanno nel file package-lock.json.
Dipendenze principali:

- **ask-sdk:** (Alexa Skill Kit) permette di comunicare direttamente con i servizi Amazon Alexa;
- **axios:** chiamate HTTP. Utilizzato principalmente in WorkflowService per le chiamate REST e nei connettori;
- **openweather-apis:** informazioni riguardanti il tempo atmosferico;

- **rss-parser:** trasformazione di un file RSS in un testo più leggibile;
- **twitter:** comunicazione con i servizi Twitter.

Dipendenze utili solo allo sviluppatore (test e compilazione):

- **mocha:** framework per i test in Javascript;
- **chai:** funzionalità aggiuntive mocha;
- **@types/node:** definizioni dei tipi di Node.js per *TypeScript*$_G$;
- **@types/mocha:** adattatore mocha per TypeScript;
- **@types/chai:** adattatore chai per TypeScript;
- **@types/chai-as-promised:** dipendenza aggiuntiva di chai per valutare le *promise*;
- **typescript:** permette la compilazione da TypeScript a JavaScript.

3.2.2 App Android

Tutte le dipendenze si trovano nel file *build.gradle(Module:app)*.

Dipendenze principali:

- **kotlin-stdlib-jdk7** Libreria standard di *Kotlin*$_G$ che si appoggia automaticamente alla versione più aggiornata;
- **Anko** Libreria utilizzata per le chiamate asincrone;
- **openweather-apis:** informazioni riguardanti il tempo atmosferico;
- **rss-parser:** trasformazione di un file RSS in un testo più leggibile;
- **twitter:** comunicazione con i servizi Twitter;
- **Lifecycle:** permettono un'implementazione semplice al pattern observer;
- **AppCompactActivity:** Libreria più stabile per le interfacce grafiche su Kotlin;
- **JUnit** framework per i test in Kotlin;
- **login-with-amazon-sdk.jar** Utilizzato per i servizi di login forniti da Amazon.

Capitolo 4

Architettura

4.1 Skill

La skill è ospitata nel sito https://developer.amazon.com/alexa/ (è necessario un account developer Amazon per accedervi). La skill è rappresentata da un JSON, configurabile attraverso l'UI fornito dalla piattaforma Alexa developer.

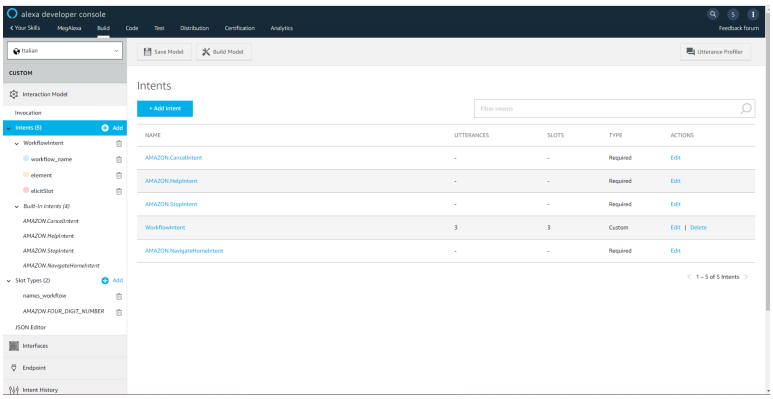

Figura 4.1: *Alexa developer platform*

La logica della skill si trova in una *AWS Lambda*$_G$, quindi è serverless. Questo significa che viene creata una nuova istanza della Skill ogni volta che arriva una richiesta da Alexa (ogni utente genera una istanza diversa). L'architettura della Skill si basa sul concetto di avere una skill leggera, che esegue poche operazioni ogni volta che questa viene invocata.
Una richiesta da parte di Alexa viene catturata dall'index, che la elabora e ritorna un risultato. Una richiesta è definita da un JSON contenente molteplici informazioni sullo stato del dialogo, l'utente, errori e molto altro. Una

dettagliata descrizione si trova sulla documentazione di Amazon AWS[1].
La lingua della skill viene impostata automaticamente a seconda della richiesta. Le frasi che Alexa può dire si trovano nei file di configurazione phrases-EN.json e phrases-IT.json.
Le informazioni più importanti contenute nel file JSON sono:

- **handlerInput.requestEnvelope.request.type:** rappresenta il tipo di richiesta: IntentRequest o LaunchRequest. LaunchRequest rappresenta la prima iterazione con la skill ("Alexa, apri MegAlexa"), IntentRequest rappresenta tutte le altre richieste;
- **handlerInput.requestEnvelope.request.intent.name:** contiene il nome della richiesta, definiti dove la skill si trova. I casi più comuni sono "WorkflowIntent", "StopIntent" e "HelpIntent";
- **handlerInput.requestEnvelope.request.slots:** contiene una lista di slot, cioè dei parametri che l'utente deve dire per continuare con il dialogo con Alexa;
- **handlerInput.requestEnvelope.request.attributesManager:** permette la gestione degli attributi di sessione. Servono per salvare delle variabili da una chiamata all'altra della skill (lambda).

Il seguente diagramma dei package descrive le dipendenze ad alto livello della Skill.

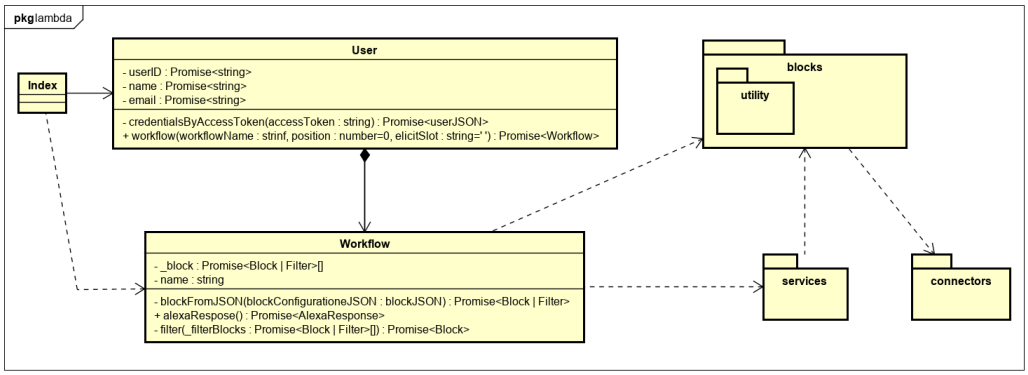

Figura 4.2: *Skill class diagram - Package*

[1] https://developer.amazon.com/docs/custom-skills/request-and-response-json-reference.html

La classe *User* rappresenta un singolo utente che sta utilizzando la skill. L'utente usa la classe *WorkflowService* per creare un *Workflow*. *Workflow* crea una lista di *Block*. Alcuni *Block* creano un *connector* per fare chiamate HTTP.
I blocchi che devono fare chiamate HTTP, vengono rappresentati come delle Promise, per questo motivo anche *Workflow* viene rappresentato in una promise in *User*.
Tutte le classi sono scritte in *TypeScript*$_G$. L'index è scritto in Javascript (per compatibilità con ask-sdk). Prima di fare il deploy in AWS Lambda, i file TypeScript vengono compilati in JavaScript.

4.1.1 Index

Index contiene gli handler, funzioni che catturano le richieste da parte di Alexa. Tutti gli handler sono definiti nella funzione "handler" e vengono eseguiti nell'ordine di dichiarazione.
Ogni singolo handler è formato da due parti:

- **canHandle:** verifica se quello è il giusto handler da eseguire. Il controllo viene fatto usando i parametri definiti in §4.1. Se nessun handler può gestire la richiesta, viene invocato *Error handler*;

- **handle:** se *canHandle* ritorna True, questa è la funzione che viene eseguita e deve ritornare una risposta comprensibile da Alexa.

4.1.2 Classe User

In *User* sono presenti i seguenti metodi:

- **credentialsByAccessToken:** prende come parametro un access token (fornito da Amazon Alexa) e ritorna una Promise contenente un JSON, il quale contiene username, email e userID. Lo userID è lo stesso di quello ottenuto attraverso il collegamento dell'applicazione all'applicazione *Android*$_G$, permettendo di autenticare l'utente nel database;

- **workflow:** usa WorkflowService per creare un Workflow a partire dalla sua rappresentazione in formato JSON.

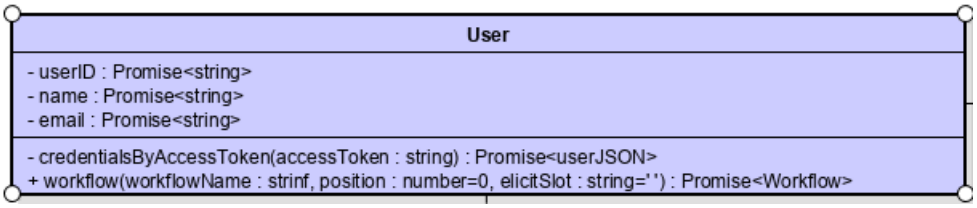

Figura 4.3: *Skill class diagram - User*

4.1.3 Classe Workflow

In *Workflow* sono presenti i seguenti metodi:

- **blockFromJSON:** crea un blocco a partire dalla sua rappresentazione in JSON;
- **alexaResponse:** crea la risposta sotto forma di Promise<string> a partire dalle risposte di ogni blocco;
- **filter:** rimuove tutti i *Filter* dalla lista di blocchi, e filtra i *Block* (chiamando l'apposito metodo nel *Block*) che seguono direttamente ogni Filter. Questo metodo non genera side-effect.

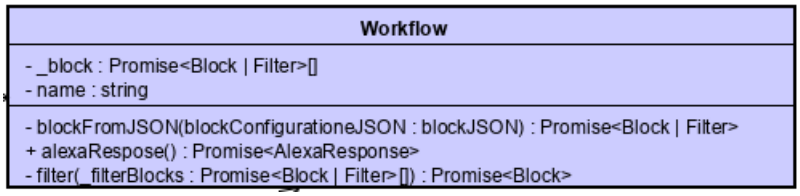

Figura 4.4: *Skill class diagram - Workflow*

4.1.4 Package services

Il package services contiene la classe WorkflowService, che si occupa di creare un *Workflow*, dopo aver ottenuto la sua rappresentazione JSON dal database (con una chiamata REST).

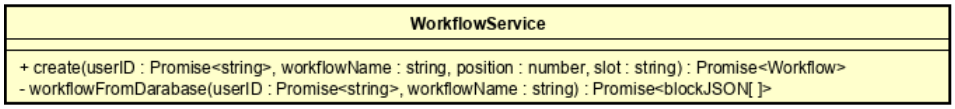

Figura 4.5: *Skill class diagram - WorkflowService*

4.1.5 Package blocks

Il package blocks contiene tutti i blocchi della skill e il package utils.
I blocchi implementano tutti l'interfaccia *Block*, alcuni implementano anche *Filterable* e *ElicitBlock*.
Un blocco Filterable può essere rappresentato come una lista, quindi deve permettere di ritornare una versione di se stesso con meno elementi.
Un blocco ElicitBlock richiede all'utente dei parametri aggiuntivi per poter essere eseguito.

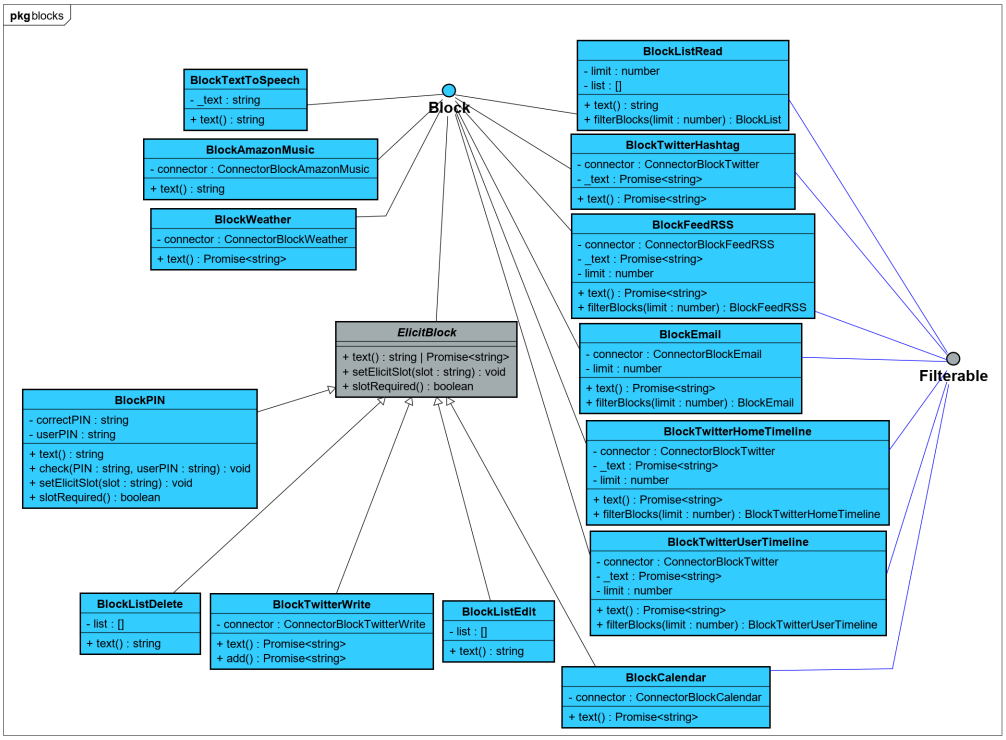

Figura 4.6: *Skill class diagram - Blocks package*

4.1.5.1 Package blocks.utils

Il package *blocks.utils* contiene le interfacce utili ai *Block* (ElicitBlock e Filterable).

4.1.6 Package connectors

Il package *connectors* contiene i connettori utilizzati dai blocchi.
Un $Connector_G$ permette al blocco di ottenere le informazioni che gli servono

da internet. Per esempio, BlockWeather (un blocco che rappresenta il meteo) chiamerà una libreria per conoscere il meteo di una certa zona.
Ogni *Connector* deve processare il risultato e trasformarlo nel testo che Alexa dovrà ripetere.

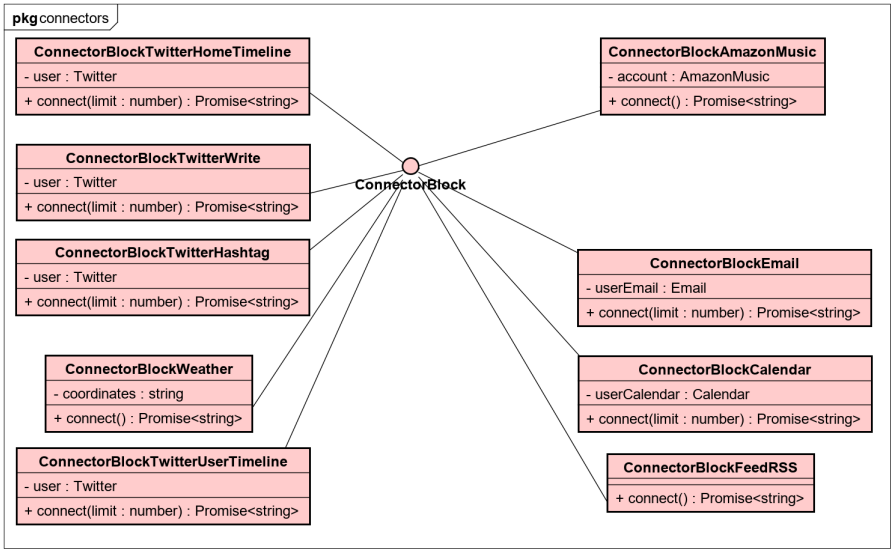

Figura 4.7: *Skill class diagram - blocks package*

4.2 App

L'app si trova nella repository GitHub https://github.com/sgt390/ProgettoSweCodice.
I pattern che abbiamo utilizzato sono:

- **ModelViewViewModel** utilizzato per la gestione e scambio dati tra Model e View;
- **Builder** utilizzato per il Model (Workflow);
- **Singleton** utilizzato per il Model (Workflow);
- **Facade** utilizzato per le connessioni ad API esterne;
- **Strategy** utilizzato per il service;
- **Template Method** utilizzato per il service.

Nella seguente immagine viene descritta l'architettura dell'app. La struttura è definita da tre cartelle:

- **model** contiene il modello delle delle classi sviluppate;
- **ui** definisce la user interface in cui sono contenute le classi grafiche dell'applicazione;
- **util** in cui sono contenute le funzionalità utilizzate per le chiamate esterne all'applicazione e il design pattern per lo scambio di informazioni tra modello e interfaccia grafica.

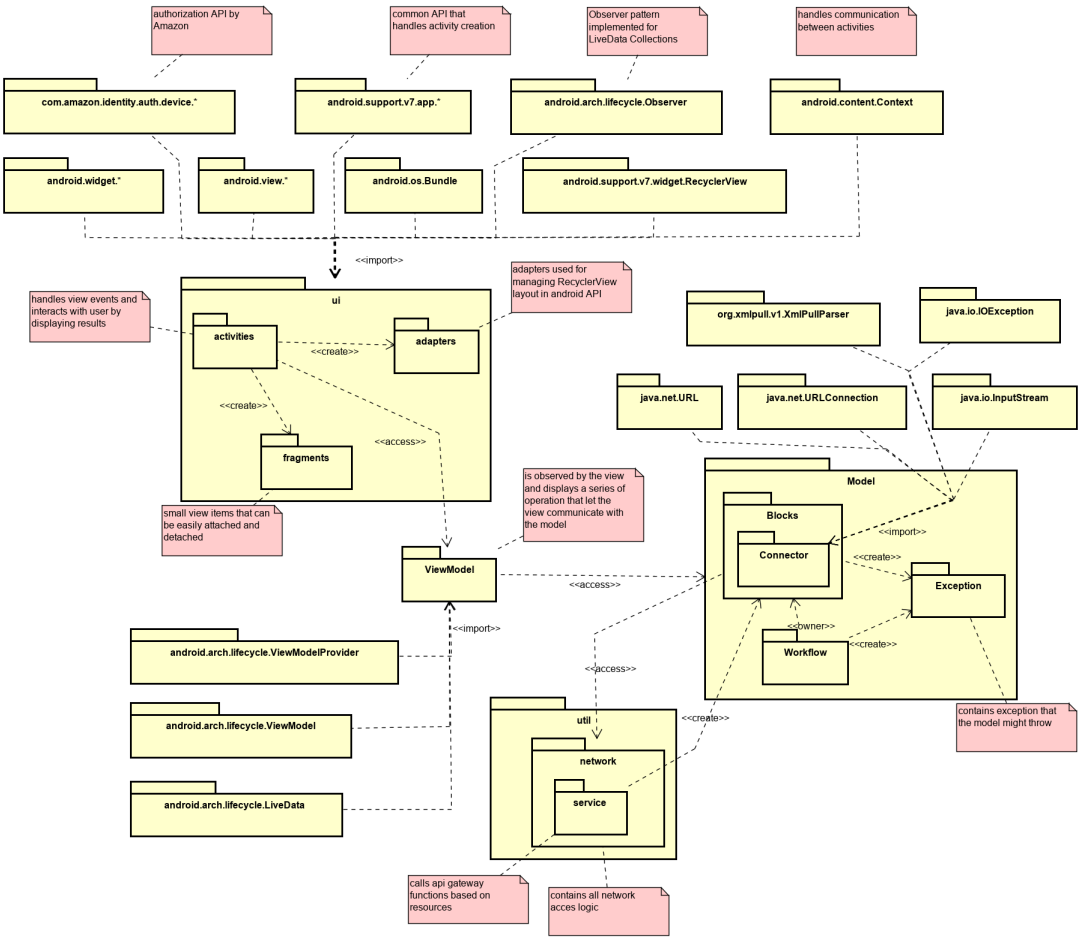

Figura 4.8: *App class diagram - blocks package*

La seguente immagine descrive la progettazione del pattern MVVM. Il ViewModel ha lo scopo di utilizzare l'istanza di Megalexa per fornire i contenuti personali dell'utente all'interfaccia grafica.

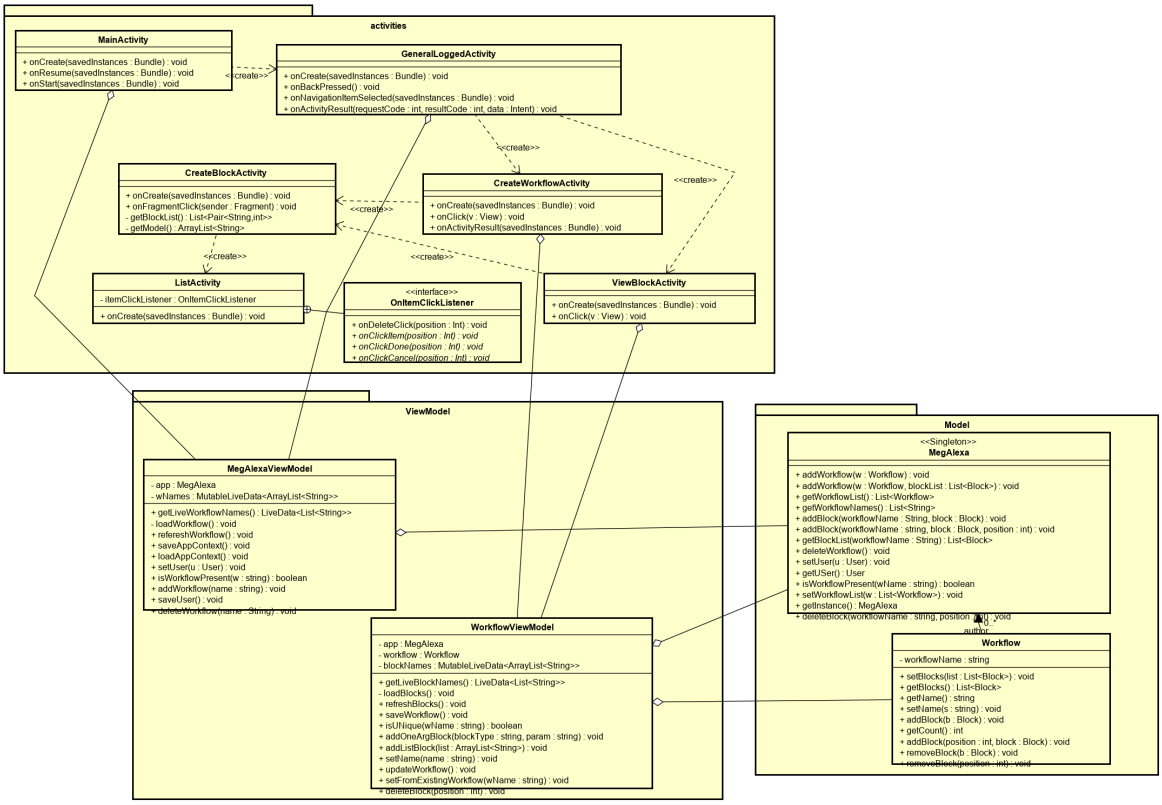

Figura 4.9: *App class diagram - MVVM*

La seguente immagine descrive la progettazione dei blocchi. Tutti i blocchi implementano dall'interfaccia Block che ha come unico metodo getInformation(). I blocchi per i quali è necessario filtrare la quantità di risultati possono estendere la classe BlockFilter.

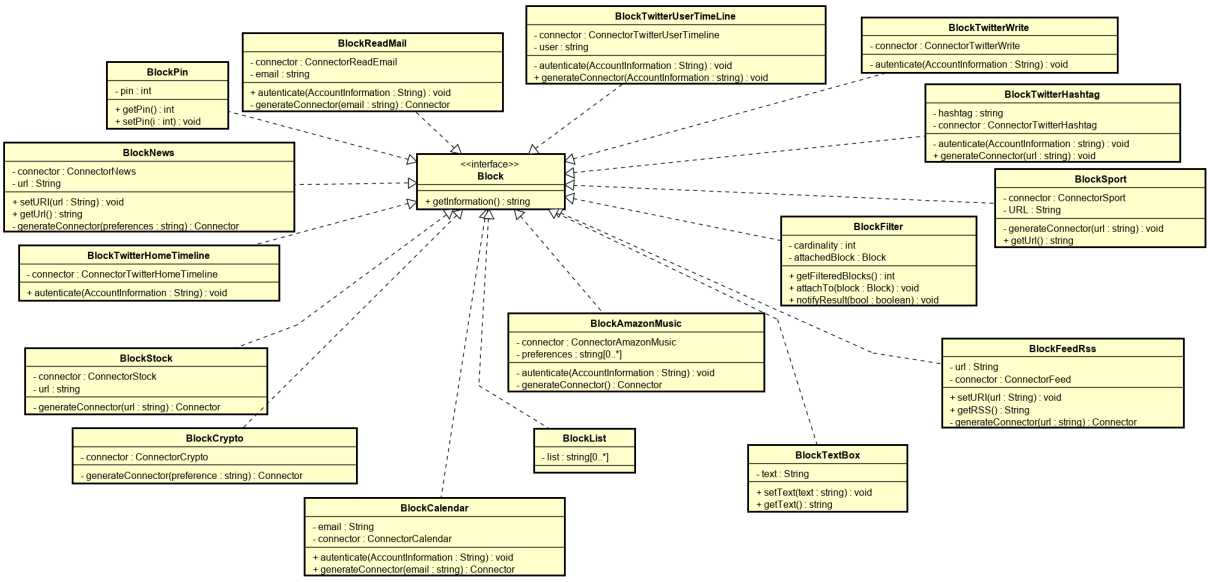

Figura 4.10: *App class diagram - Blocks*

La seguente immagine descrive la progettazione dei service. I service sono utilizzati per le chiamate REST e per la conversione dei blocchi tramite i metodi:

- **convertToJSON(t:T)** utilizzato per la conversione delle informazioni ottenute dall'utente in JSON e successivamente salvate in database;
- **convertFromJSON(J:JSONObject)** utilizzato per la conversione delle informazioni ottenute dal database e successivamente mostrate all'utente.

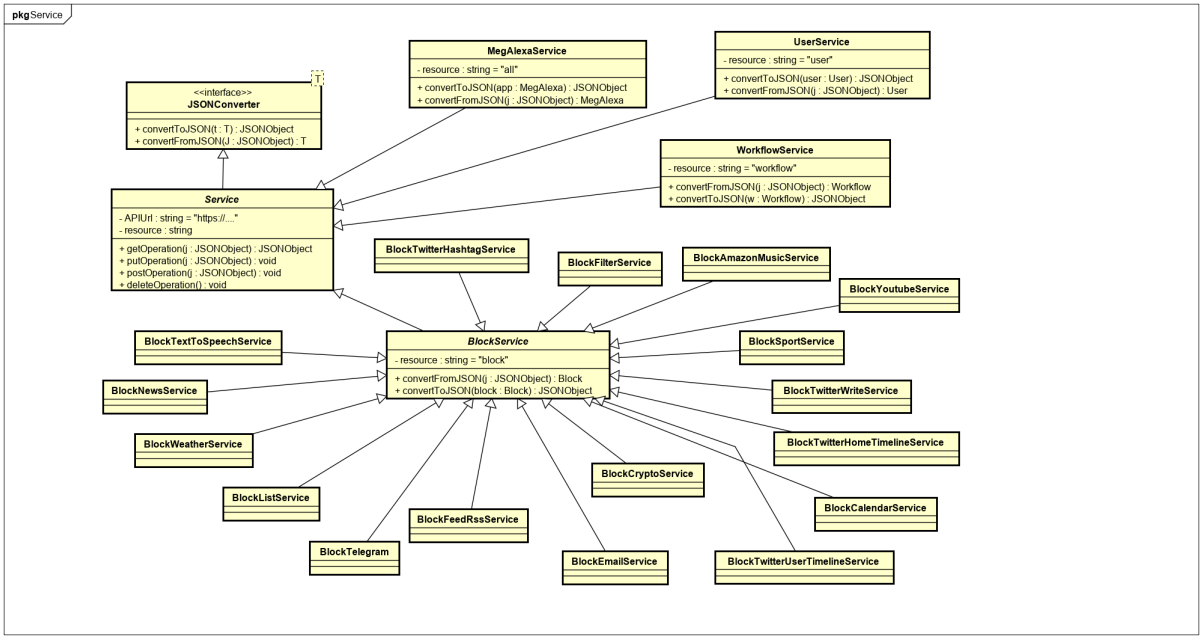

Figura 4.11: *App class diagram - Service*

La seguente immagine rappresenta la progettazione dell'UI. Un'activity è un'interfaccia utente mediante la quale si consultano dati o si immettono input. L'applicazione utilizza le seguenti activities:

- **MainActivity:** permette all'utente di autenticarsi;
- **GeneralLoggedActivity:** permette all'utente di visualizzare i propri workflow;
- **CreateWorkflowActivity:** permette all'utente di creare un nuovo workflow;
- **ViewBlockActivity:** permette all'utente di visualizzare e modificare i blocchi presenti in un singolo workflow;
- **CreateBlockActivity:** permette all'utente di aggiungere nuovi blocchi.

Gli adapters sono componenti che si occupano della rappresentazione grafica dei dati e dell'interazione con essi, per ogni elemento di una lista.

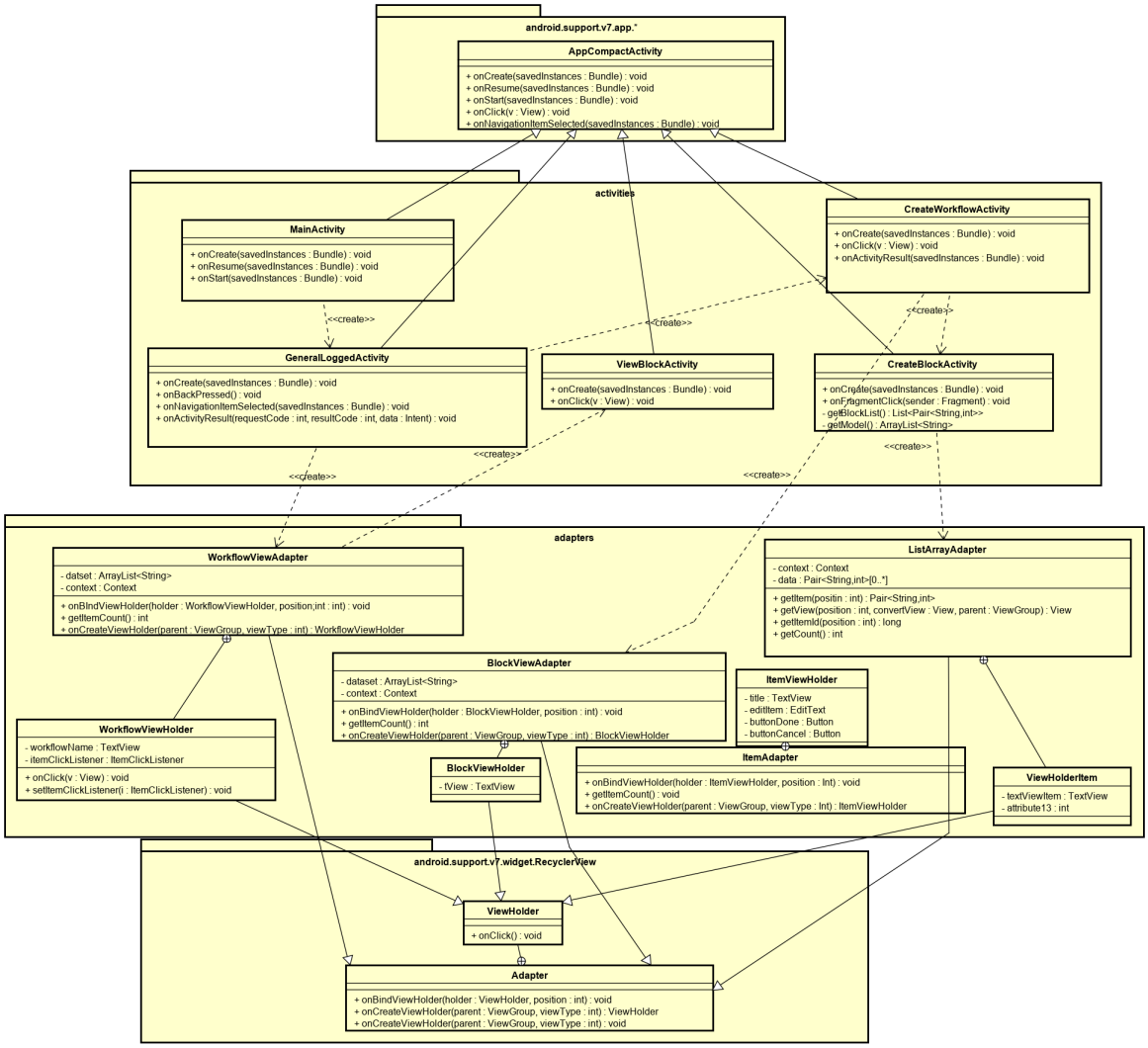

Figura 4.12: *App class diagram - UI*

La seguente immagine descrive la progettazione del ViewModel.

- **MegAlexaViewModel:** si occupa di gestire le informazioni tra interfaccia e modello relativo ai workflow;
- **WorkflowViewModel:** si occupa di gestire le informazioni tra interfaccia e modello relativo ai blocchi;

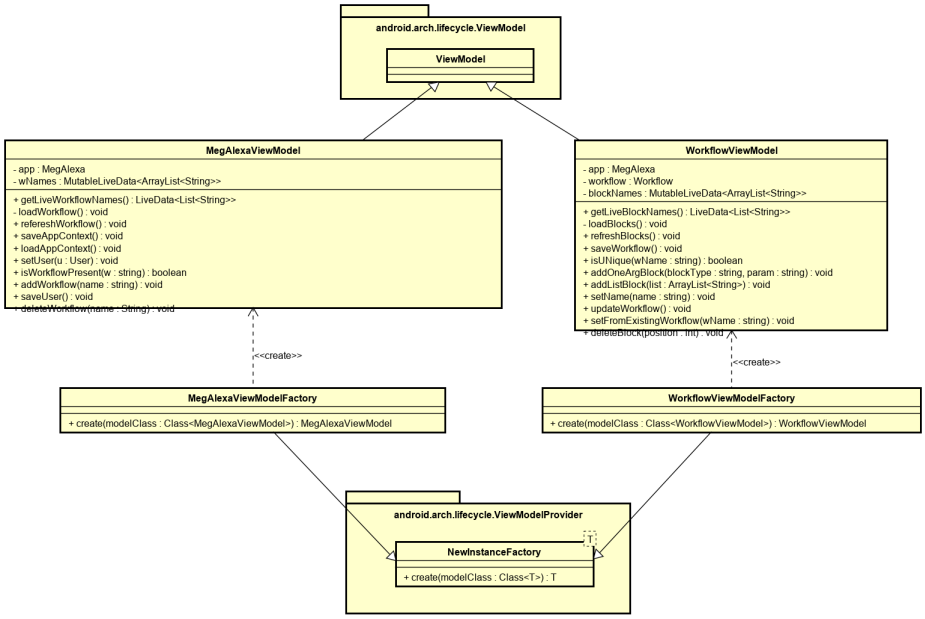

Figura 4.13: *App class diagram - ViewModel*

La seguente immagine descrive la progettazione dei blocchi che implementano i connettori. I connettori sono utilizzati per la gestione e validazione dei dati ottenuti da chiamate esterne all'applicazione.

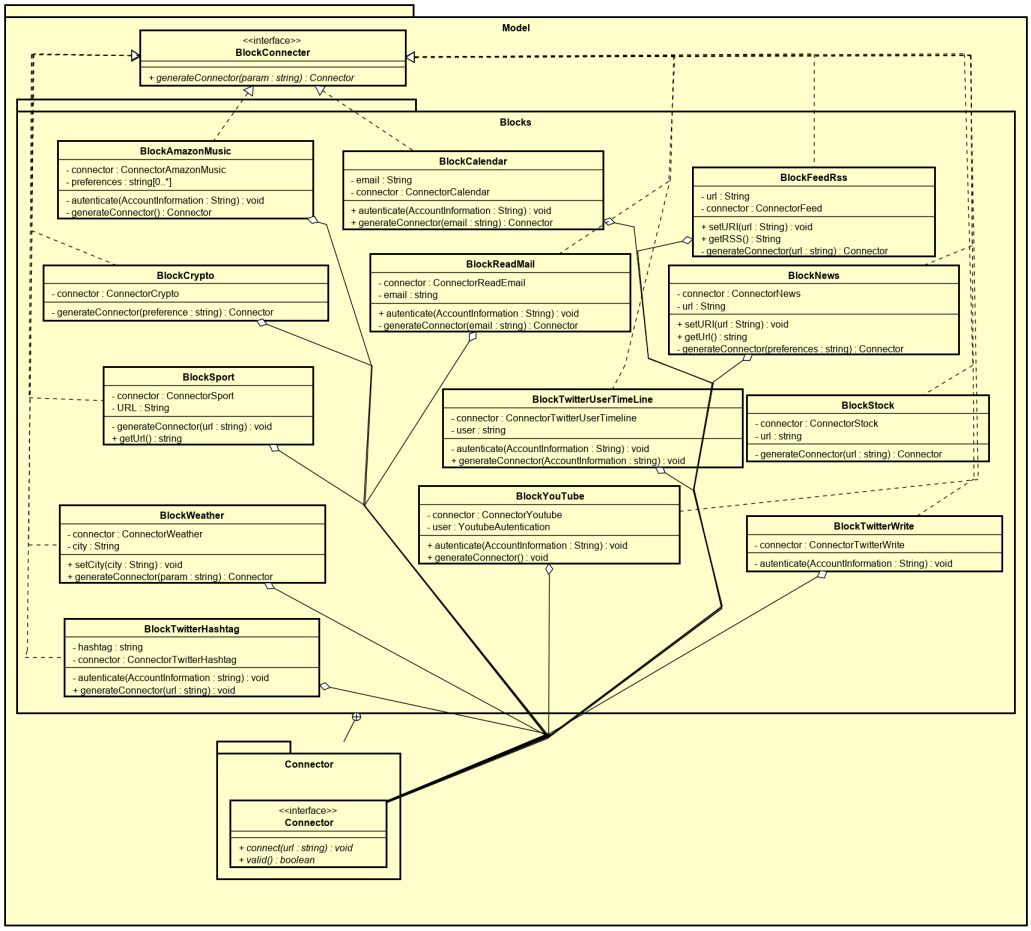

Figura 4.14: *App class diagram - BlockConnection*

La seguente immagine descrive la progettazione del Model.

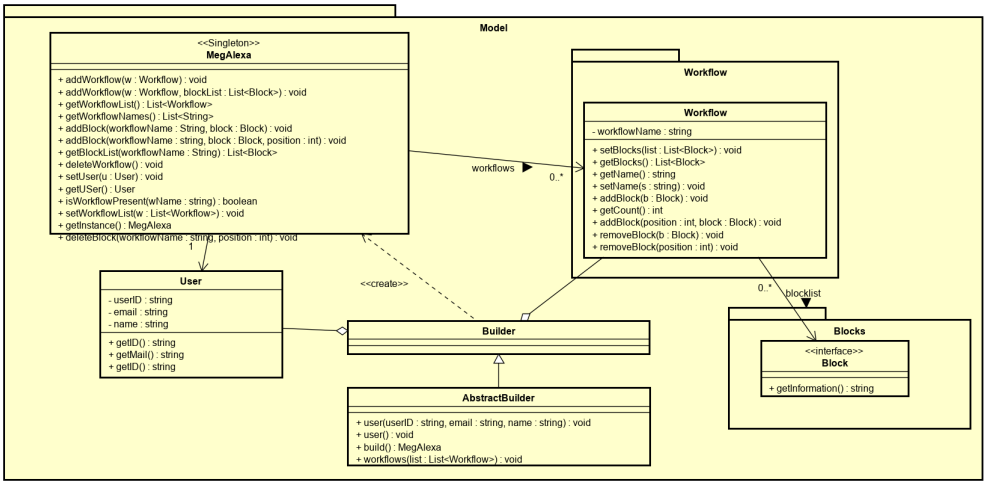

Figura 4.15: *App class diagram - Model*

Capitolo 5

Estensione delle funzionalità

In questo capitolo vengono descritti i passaggi per l'estensione del prodotto *MegAlexa*.

5.1 Estensione App

L'estensione della app avviene mediante l'aggiunta di nuovi blocchi.
Segue una lista di attività da attuare che descrive i passi da eseguire per un'estensione corretta:

- Definire un blocco che derivi dall'interfaccia `Block` posta nel package **blocks**;
- Se il blocco richiede una validazione tramite chiamata esterna, definire un connettore che estenda `Connector` nel package **connectors**;
- Se la chiamata esterna prevede l'utilizzo di una API particolare, definire tali interazioni incapsulandole in una nuova classe da posizionare nel package **network**;
- Definire le conversioni da JSON e a JSON dichiarando una nuova classe che derivi da `BlockService` nel package **service**, per la comunicazione con *API Gateway*$_G$;
- Ampliare la lista presente in `CreateBlockActivity` (package **activities**) in modo che presenti un elemento cliccabile in più;
- Definire un widget a scelta (activity o fragment) che rappresenti l'inserimento dei dati da parte dell'utente;

- Inserire il listener dell'elemento aggiunto alla lista nel widget appena creato;
- Nel caso in cui sia stato creato un fragment ampliare la funzione *onFragmentClick(sender)* in modo che registri e notifichi alle activity interessate le informazioni per l'aggiunta del blocco, in alternativa, se è stata creata una nuova activity, gestire l'interazione con essa mediante le funzioni *startActivityForResult(intent,context)* e *onActivityResult(requestCode,resultCode,data)*;
- Nella `CreateWorkflowActivity` e `ViewBlockActivity`, nelle funzioni *onActivityResult(requestCode,resultCode,data)* prelevare i dati passati dalle altre activity e chiamare il viewModel di conseguenza;
- Aggiornare il viewModel in modo che supporti l'aggiunta del blocco appena creato.

5.2 Estensione Skill (lambda)

La Skill può essere estesa in questi modi:

- aggiunta di un nuovo blocco §5.2.1;
- aggiunta di nuove frasi per Alexa §5.2.2;
- aggiunta di un nuovo metodo d'accesso al database §5.2.3;
- aggiunta di un nuovo sistema di autenticazione §5.2.4.

5.2.1 Nuovo blocco

Questa sezione verrà redatta quando il prodotto sarà completo.

5.2.2 Nuove frasi per Alexa

Il file phrases-EN.json e phrases-IT.json contengono tutte le frasi personalizzate che Alexa può dire. A ogni parola chiave (uguale per tutte le lingue) sono associate delle frasi (tradotte nella lingua relativa al file).
Per inserire una nuova frase, è necessario inserirla nel relativo oggetto associato alla parola chiave. E' importante inserire la frase tradotta per tutti i file phrases-LINGUA.json.
Per inserire una nuova parola chiave:

- inserire un oggetto contenente come chiave la **parola chiave** e come contenuto una lista di stringhe, contenenti le frasi;
- ripetere l'operazione precedente per ogni lingua;
- inserire un nuovo metodo nella classe *PhrasesGenerator.ts*, che ritorna la frase che Alexa deve dire;
- l'attributo statico **jsonPhrases** contiene il file *phrases-LINGUA.json* (la selezione della lingua viene fatta in automatico);
- chiamare questo metodo in modo statico dove è necessario.

Per inserire una nuova lingua:

- inserire un nuovo file *phrases-LINGUA.json*, contenente tutte le parole chiavi presenti negli altri file di configurazione e le relative frasi tradotte;
- nel metodo setLanguage aggiungere un case con la nuova lingua.

5.2.3 Nuovo metodo d'accesso al database

Tutti gli accessi al database, per quanto riguarda il download dei workflow, avvengono attraverso la classe **WorkflowService**, nel metodo *workflowFromDatabase*.
É possibile aggiungere una nuova funzione in **WorkflowService**, contenente un nuovo metodo d'accesso a un diverso database. Per esempio, può essere fatto attraverso una chiamata HTTP a un servizio REST, oppure usando un web socket. É necessario che il valore ritornato sia una lista di blocchi.
Lo schema del blocco si può trovare nella cartella *JSONconfigurations*, nel file *JSONconfiguration.ts*, con la chiave (**BlockJSON**).

5.2.4 Nuovo sistema di autenticazione

L'autenticazione, nella versione 1.0.0 della skill, avviene solo attraverso Amazon.
Il file contenente la richiesta dell'utente (dal Alexa) contiene anche un **accessToken**. Questo viene utilizzato dalla classe **User**, dal metodo *credentialsByAccessToken*, per estrarre l'**AmazonID** dell'utente (lo stesso salvato nel database).
Questo sistema è basato su oauth, anche se tutta la parte a basso livello è gestita da Amazon.
Per aggiungere un altro sistema di autenticazione basato su oauth:

- creare una nuova configurazione Oauth e ottenere i dati d'accesso (URI, clientID, secret...);
- aprire il pannello di controllo di Alexa [1];
- accedere alla skill MegAlexa;
- aprire la sezione "account linking";
- inserire i dati richiesti, generati da oauth;
- nella skill MegAlexa, modificare la funzione *credentialsByAccessToken*, nella classe **User**, così che sia compatibile con il nuovo servizio (ciò che cambia sarà l'ID e la chiamata GET per ottenerlo).

[1] https://developer.amazon.com/alexa/console/ask