

AtAVi

Definizione di Prodotto v1.0.0

Sommario

Questo documento le scelte progettuali effettuate dal gruppo Co.Code per la realizzazione del
progetto_g AtAVi.

Versione	1.0.0
Data di redazione	2017-02-02
Redazione	
Verifica	
Approvazione	
Uso	Esterno
Distribuzione	prof. Tullio Vardanega prof. Riccardo Cardin

Diario delle modifiche

Versione	Riepilogo	Autore	Ruolo	Data
----------	-----------	--------	-------	------

Indice

1	Introduzione	9
1.1	Scopo del documento	9
1.2	Scopo del prodotto	9
1.3	Glossario	9
1.4	Riferimenti	9
1.4.1	Riferimenti Normativi	9
1.4.2	Riferimenti Informativi	9
2	Standard di progetto	11
3	Architettura dell'applicazione	12
3.1	Descrizione generale	12
3.2	Endpoints client	13
3.3	Microservizi	14
3.3.1	Virtual Assistant	14
3.3.1.1	Descrizione	14
3.3.1.2	Endpoints	15
3.3.2	Notifications	15
3.3.2.1	Descrizione	15
3.3.2.2	Endpoints	16
3.3.3	Users	17
3.3.3.1	Descrizione	17
3.3.3.2	Endpoints	17
3.3.4	Rules	18
3.3.4.1	Descrizione	18
3.3.4.2	Endpoints	19
4	Diagrammi riassuntivi dei package	25
5	Diagrammi di sequenza	26
6	Specifiche delle Componenti	68
6.1	Back-end	68
6.1.1	Classi	68
6.1.1.1	ConversationsDAO	68
6.1.1.2	GuestsDAO	69
6.1.1.3	AdministrationWebhookService	71
6.1.1.4	AgentObservable	72
6.1.1.5	AgentObserver	73
6.1.1.6	Conversation	74
6.1.1.7	ConversationMsg	74
6.1.1.8	ConversationObservable	75
6.1.1.9	ConversationObserver	76
6.1.1.10	ConversationsDAODynamoDB	76
6.1.1.11	ConversationWebhookService	78
6.1.1.12	ErrorObservable	80
6.1.1.13	ErrorObserver	81
6.1.1.14	Guest	82
6.1.1.15	GuestObservable	83
6.1.1.16	GuestObserver	84
6.1.1.17	GuestsDAODynamoDB	84
6.1.1.18	Member	86
6.1.1.19	MembersSlackDAO	87
6.1.1.20	RuleObservable	88

6.1.1.21	RuleObserver	89
6.1.1.22	SNSEvent	90
6.1.1.23	SNSMessage	90
6.1.1.24	SNSRecord	91
6.1.1.25	STTParams	92
6.1.1.26	TaskObservable	94
6.1.1.27	TaskObserver	95
6.1.1.28	UserObservable	96
6.1.1.29	UserObserver	96
6.1.1.30	VAMessageListener	97
6.2	Back-end::APIGateway	98
6.2.1	Classi	99
6.2.1.1	Enrollment	99
6.2.1.2	VARequestAPIBody DA MODIFICARE DA MODIFICARE DA MODIFICARE DA MODIFICARE	99
6.2.1.3	VocalAPI	100
6.2.1.4	VocalLoginModuleConfig	104
6.3	Back-end::Auth	104
6.3.1	Classi	104
6.3.1.1	UsersDAO	104
6.3.1.2	SRUser	106
6.3.1.3	User	107
6.3.1.4	UsersDAODynamoDB	109
6.3.1.5	UsersService	111
6.3.1.6	VocalLoginModule	113
6.4	Back-end::Notifications	114
6.4.1	Classi	114
6.4.1.1	Action	114
6.4.1.2	Attachment	115
6.4.1.3	ConfirmationFields	117
6.4.1.4	NotificationChannel	118
6.4.1.5	NotificationMessage	119
6.4.1.6	NotificationService	120
6.4.1.7	Purpose	121
6.4.1.8	Topic	122
6.5	Back-end::Rules	123
6.5.1	Classi	123
6.5.1.1	RulesDAO	123
6.5.1.2	TasksDAO	125
6.5.1.3	Rule	126
6.5.1.4	RulesDAODynamoDB	129
6.5.1.5	RulesService	130
6.5.1.6	RuleTarget	132
6.5.1.7	RuleTaskInstance	133
6.5.1.8	Task	134
6.5.1.9	TasksDAODynamoDB	135
6.6	Back-end::STT	136
6.6.1	Classi	137
6.6.1.1	STTModule	137
6.6.1.2	STTWatsonAdapter	137
6.7	Back-end::Utility	138
6.7.1	Classi	138
6.7.1.1	MembersDAO	138
6.7.1.2	Error	140
6.7.1.3	LambdaContext	141
6.7.1.4	LambdaEvent	142

6.7.1.5	LambdaIdEvent	142
6.7.1.6	LambdaResponse	143
6.7.1.7	PathParam	144
6.7.1.8	ProcessingResult	145
6.7.1.9	StatusObject	146
6.8	Back-end::VirtualAssistant	147
6.8.1	Classi	147
6.8.1.1	AgentsDAO	147
6.8.1.2	VAModule	148
6.8.1.3	WebhookService	149
6.8.1.4	Agent	150
6.8.1.5	AgentsDAO _{DynamoDB}	151
6.8.1.6	ApiAiVAAdapter	153
6.8.1.7	ButtonObject	154
6.8.1.8	Context	155
6.8.1.9	Fulfillment	155
6.8.1.10	Metadata	156
6.8.1.11	MsgObject	157
6.8.1.12	ResponseBody	159
6.8.1.13	VAEventObject	159
6.8.1.14	VAQuery	160
6.8.1.15	VAResponse	161
6.8.1.16	VAService	162
6.9	Client	164
6.9.1	Classi	164
6.9.1.1	ApplicationLocalRegistry	164
6.9.1.2	ConversationApp	166
6.9.1.3	IndexView	167
6.9.1.4	ObserverAdapter	169
6.10	Client::ApplicationManager	171
6.10.1	Classi	171
6.10.1.1	ApplicationRegistryClient	171
6.10.1.2	Application	172
6.10.1.3	ApplicationManagerObserver	173
6.10.1.4	ApplicationPackage	173
6.10.1.5	ApplicationRegistryLocalClient	174
6.10.1.6	Manager	175
6.10.1.7	State	177
6.11	Client::Logic	178
6.11.1	Classi	178
6.11.1.1	DataArrivedObservable	178
6.11.1.2	DataArrivedSubject	178
6.11.1.3	HttpError	179
6.11.1.4	HttpPromise	180
6.11.1.5	Logic	181
6.11.1.6	LogicObserver	182
6.12	Client::Recorder	183
6.12.1	Classi	183
6.12.1.1	Recorder	183
6.12.1.2	RecorderConfig	185
6.12.1.3	RecorderMsg	186
6.12.1.4	RecorderWorker	186
6.12.1.5	RecorderWorkerConfig	188
6.12.1.6	RecorderWorkerMsg	189
6.12.1.7	SpeechEndObservable	190
6.12.1.8	SpeechEndSubject	190

6.13 Client::TTS	191
6.13.1 Classi	191
6.13.1.1 Player	191
6.13.1.2 PlayerObserver	193
6.13.1.3 TTSTConfig	194
6.14 Client::Utility	195
6.14.1 Classi	195
6.14.1.1 BoolObservable	195
6.14.1.2 BoolObserver	196
6.14.1.3 BoolSubject	196
6.15 RxJS	197
6.15.1 Classi	197
6.15.1.1 Observable	197
6.15.1.2 Observer	198
6.15.1.3 Subject	199
6.16 Slack	200
6.16.1 Classi	200
6.16.1.1 WebClient	200
6.17 WatsonDeveloperCloud	201
6.17.1 Classi	201
6.17.1.1 SpeechToTextV1	201
7 Tracciamento	203
7.1 Tracciamento Classi-Requisiti	203
7.2 Tracciamento Componenti-Requisiti	206
A Design Patterns	211
A.1 Architetturali	211
A.1.1 Architettura a microservizi	211
A.1.2 Architettura event-driven	212
A.1.3 Client-side discovery	213
A.1.4 Data Access Object	214
A.1.5 Dependency Injection	215
A.2 Strutturali	215
A.2.1 Façade	215
A.2.2 Adapter	216
A.3 Creazionali	217
A.3.1 Module	217
A.4 Comportamentali	217
A.4.1 Observer	217
B Tecnologie utilizzate	218
B.1 Promise e Observable	218
B.1.1 Promise e bluebird	218
B.1.2 Observable e RxJS v5	219
B.2 AWS SDK per JavaScript in Node.js	219
B.2.1 Node.js	219
B.2.2 Request promise	219
B.3 Amazon Web Services	219
B.3.1 AWS Lambda	219
B.3.2 DynamoDB	220
B.3.3 API Gateway	221
B.3.4 SNS	221
B.3.5 CloudWatch	221
B.4 Serverless Framework	222
B.5 JWT	222

B.6	Web Speech API	222
B.7	Speaker recognition	222
B.8	STT IBM Watson	222
B.9	api.ai	222
B.10	HTML5	223
B.11	CSS3	223

Elenco delle figure

1	Architettura a microservizi e API Gateway	13
2	Back-end:: ConversationsDAO	68
3	Back-end:: GuestsDAO	70
4	Back-end::AdministrationWebhookService	71
5	Back-end::AgentObservable	72
6	Back-end::AgentObserver	73
7	Back-end::Conversation	74
8	Back-end::ConversationMsg	75
9	Back-end::ConversationObservable	75
10	Back-end::ConversationObserver	76
11	Back-end::ConversationsDAO _{DynamoDB}	77
12	Back-end::ConversationWebhookService	78
13	Back-end::ErrorObservable	81
14	Back-end::ErrorObserver	82
15	Back-end::Guest	83
16	Back-end::GuestObservable	83
17	Back-end::GuestObserver	84
18	Back-end::GuestsDAO _{DynamoDB}	85
19	Back-end::Member	86
20	Back-end::MembersSlackDAO	87
21	Back-end::RuleObservable	88
22	Back-end::RuleObserver	89
23	Back-end::SNSEvent	90
24	Back-end::SNSMessage	91
25	Back-end::SNSRecord	92
26	Back-end::STTPParams	93
27	Back-end::TaskObservable	94
28	Back-end::TaskObserver	95
29	Back-end::UserObservable	96
30	Back-end::UserObserver	97
31	Back-end::VAMessageListener	98
32	Back-end::APIGateway::Enrollment	99
33	Back-end::APIGateway::VARequestAPIBody DA MODIFICARE DA MODIFICA- RE DA MODIFICARE DA MODIFICARE	100
34	Back-end::APIGateway::VocalAPI	100
35	Back-end::APIGateway::VocalLoginModuleConfig	104
36	Back-end::Auth:: UsersDAO	105
37	Back-end::Auth::SRUser	106
38	Back-end::Auth::User	107
39	Back-end::Auth::UsersDAO _{DynamoDB}	109
40	Back-end::Auth::UsersService	111
41	Back-end::Auth::VocalLoginModule	113
42	Back-end::Notifications::Action	115
43	Back-end::Notifications::Attachment	116
44	Back-end::Notifications::ConfirmationFields	117
45	Back-end::Notifications::NotificationChannel	118

46	Back-end::Notifications::NotificationMessage	119
47	Back-end::Notifications::NotificationService	120
48	Back-end::Notifications::Purpose	122
49	Back-end::Notifications::Topic	123
50	Back-end::Rules:: RulesDAO	123
51	Back-end::Rules:: TasksDAO	125
52	Back-end::Rules::Rule	126
53	Back-end::Rules::RulesDAODynamoDB	129
54	Back-end::Rules::RulesService	130
55	Back-end::Rules::RuleTarget	133
56	Back-end::Rules::RuleTaskInstance	134
57	Back-end::Rules::Task	135
58	Back-end::Rules::TasksDAODynamoDB	135
59	Back-end::STT:: STTModule	137
60	Back-end::STT::STTWatsonAdapter	137
61	Back-end::Utility:: MembersDAO	139
62	Back-end::Utility::Error	140
63	Back-end::Utility::LambdaContext	141
64	Back-end::Utility::LambdaEvent	142
65	Back-end::Utility::LambdaIdEvent	143
66	Back-end::Utility::LambdaResponse	143
67	Back-end::Utility::PathIdParam	144
68	Back-end::Utility::ProcessingResult	145
69	Back-end::Utility::StatusObject	146
70	Back-end::VirtualAssistant:: AgentsDAO	147
71	Back-end::VirtualAssistant:: VAModule	148
72	Back-end::VirtualAssistant:: WebhookService	149
73	Back-end::VirtualAssistant::Agent	151
74	Back-end::VirtualAssistant::AgentsDAODynamoDB	152
75	Back-end::VirtualAssistant::ApiAiVAAadapter	153
76	Back-end::VirtualAssistant::ButtonObject	154
77	Back-end::VirtualAssistant::Context	155
78	Back-end::VirtualAssistant::Fulfillment	156
79	Back-end::VirtualAssistant::Metadata	157
80	Back-end::VirtualAssistant::MsgObject	158
81	Back-end::VirtualAssistant::ResponseBody	159
82	Back-end::VirtualAssistant::VAEventObject	160
83	Back-end::VirtualAssistant::VAQuery	161
84	Back-end::VirtualAssistant::VAResponse	162
85	Back-end::VirtualAssistant::VAService	163
86	Client::ApplicationLocalRegistry	165
87	Client::ConversationApp	166
88	Client::IndexView	168
89	Client::ObserverAdapter	169
90	Client::ApplicationManager:: ApplicationRegistryClient	171
91	Client::ApplicationManager::Application	172
92	Client::ApplicationManager::ApplicationManagerObserver	173
93	Client::ApplicationManager::ApplicationPackage	174
94	Client::ApplicationManager::ApplicationRegistryLocalClient	174
95	Client::ApplicationManager::Manager	175
96	Client::ApplicationManager::State	177
97	Client::Logic::DataArrivedObservable	178
98	Client::Logic::DataArrivedSubject	179
99	Client::Logic::HttpError	180
100	Client::Logic::HttpPromise	180
101	Client::Logic::Logic	181

102	Client::Logic::LogicObserver	182
103	Client::Recorder::Recorder	183
104	Client::Recorder::RecorderConfig	185
105	Client::Recorder::RecorderMsg	186
106	Client::Recorder::RecorderWorker	187
107	Client::Recorder::RecorderWorkerConfig	189
108	Client::Recorder::RecorderWorkerMsg	189
109	Client::Recorder::SpeechEndObservable	190
110	Client::Recorder::SpeechEndSubject	191
111	Client::TTS::Player	192
112	Client::TTS::PlayerObserver	193
113	Client::TTS::TTSCConfig	194
114	Client::Utility::BoolObservable	195
115	Client::Utility::BoolObserver	196
116	Client::Utility::BoolSubject	196
117	RxJS::Observable	198
118	RxJS::Observer	199
119	RxJS::Subject	200
120	Slack::WebClient	201
121	WatsonDeveloperCloud::SpeechToTextV1	202
122	Architettura a microservizi	212
123	Pattern client-side discovery adattato alle esigenze del gruppo	213
124	Pattern client-side discovery	214
125	pattern Data Access Object (DAO)	215
126	pattern Façade	216
127	Pattern Adapter	217
128	Pattern Observer	218

Introduzione

Scopo del documento

Lo scopo di questo documento consiste nella definizione in dettaglio della struttura e funzionamento delle componenti del progetto AtAVi. Questo documento sarà usato come guida dai *Programmatori* del gruppo.

Scopo del prodotto

Si vuole creare un'applicazione web che permetta ad un ospite, in visita all'ufficio di Zero12, di interrogare un assistente virtuale per annunciare la propria presenza, avvisare l'interessato del suo arrivo sul sistema di comunicazione aziendale (*Slack_g*) e nel frattempo essere intrattenuto con varie attività.

Glossario

Allo scopo di evitare ogni ambiguità nel linguaggio e rendere più semplice e chiara la comprensione dei documenti, viene allegato il “*Glossario v1.0.0*”. Le parole in esso contenute sono scritte in corsivo e marcate con una ‘g’ a pedice (p.es. *Parola_g*).

Riferimenti

Riferimenti Normativi

- “*Norme di Progetto v2.0.0*”;
- “*Analisi dei Requisiti v2.0.0*”;

Riferimenti Informativi

- Design patterns:
 - strutturali: <http://www.math.unipd.it/~tullio/IS-1/2016/Dispense/E04.pdf>;
 - creazionali: <http://www.math.unipd.it/~tullio/IS-1/2016/Dispense/E05.pdf>;
 - comportamentali: <http://www.math.unipd.it/~tullio/IS-1/2016/Dispense/E06.pdf>;
 - architettonici:
 - * <http://www.math.unipd.it/~tullio/IS-1/2016/Dispense/E08.pdf>;
 - * <http://www.math.unipd.it/~tullio/IS-1/2016/Dispense/E07.pdf>;
 - * <http://microservices.io/patterns/microservices.html>;
 - * <http://microservices.io/patterns/data/event-driven-architecture.html>;
 - * <http://microservices.io/patterns/client-side-discovery.html>;
 - * https://en.wikipedia.org/wiki/Data_access_object.
 - * https://en.wikipedia.org/wiki/Representational_state_transfer
- Slide dell'insegnamento - Diagrammi delle classi: <http://www.math.unipd.it/~tullio/IS-1/2016/Dispense/E02a.pdf>;

- Slide dell'insegnamento - Diagrammi dei packages: <http://www.math.unipd.it/~tullio/IS-1/2016/Dispense/E02b.pdf>;
- Slide dell'insegnamento - Diagrammi di sequenza: <http://www.math.unipd.it/~tullio/IS-1/2016/Dispense/E03a.pdf>;

Standard di progetto

Architettura dell'applicazione

Descrizione generale

L'architettura scelta per il sistema dell'applicazione è quella a microservizi.

I motivi che hanno spinto il team a questa scelta sono stati i seguenti:

- i servizi utilizzati di AWS permettono l'implementazione di un'architettura serverless, la quale consente una migliore gestione dei microservizi, guadagnandone in facilità d'implementazione e di scalabilità di quest'ultimi;
- api.ai, il software development kit (SDK) scelto per la realizzazione dell'assistente virtuale, accetta in input solo del testo e non dell'audio, avendo così bisogno di un servizio differente e indipendente che, dato un file audio, estrae il testo in esso pronunciato;
- il proponente ha richiesto anche un lato amministrazione, il quale accesso dovrà essere regolamentato da un servizio differente e indipendente di riconoscimento vocale.

Ogni microservizio rende disponibili degli endpoints, tramite i quali è possibile lo scambio di dati. Per poter creare, pubblicare, mantenere e monitorare le API che consentono di usufruire delle funzionalità supportate dal back-end, è necessario realizzare un API Gateway che si interpone tra client e back-end, permettendo al primo di comunicare con il secondo. Esso permette inoltre la creazione di endpoint disponibili all'esterno del back-end, che verranno poi mappati negli endpoints supportati dal back-end, in particolare dai microservizi.

Con lo scopo di semplificare al client l'uso del back-end, è stato deciso di rendere disponibile un solo endpoint esterno per ogni tipo di client. Allo stato attuale un solo tipo di client è supportato, ovvero quello vocale richiesto dal proponente.

I motivi che hanno spinto il team a questa scelta sono stati i seguenti:

- dal punto di vista del client è molto più semplice fare utilizzo di un solo endpoint, demandando all'API Gateway la responsabilità di mappare la richiesta del client negli endpoints supportati dal backend;
- qualora si volesse aggiungere un client di tipo differente, ad esempio uno puramente testuale, sarà sufficiente pubblicare un solo nuovo endpoint esterno, mantenendo quindi semplice l'interfaccia dell'API Gateway;
- in caso ci siano dei cambiamenti nel back-end, la complessità del cambio di mappatura tra gli endpoint interni e l'endpoint esterno sarà contenuta in quest'ultimo;
- in caso più tipi di client vengano supportati, sarà necessario vincolare essi ad utilizzare l'unico endpoint predisposto per il loro tipo, impedendo quindi l'accesso a quelli restanti. È chiaro che meno endpoints esistono, più è facile realizzare quanto appena detto.

In figura 1 si ha una rappresentazione generale, non specifica del progetto AtAVi, del ruolo di un API Gateway in un'architettura a microservizi.

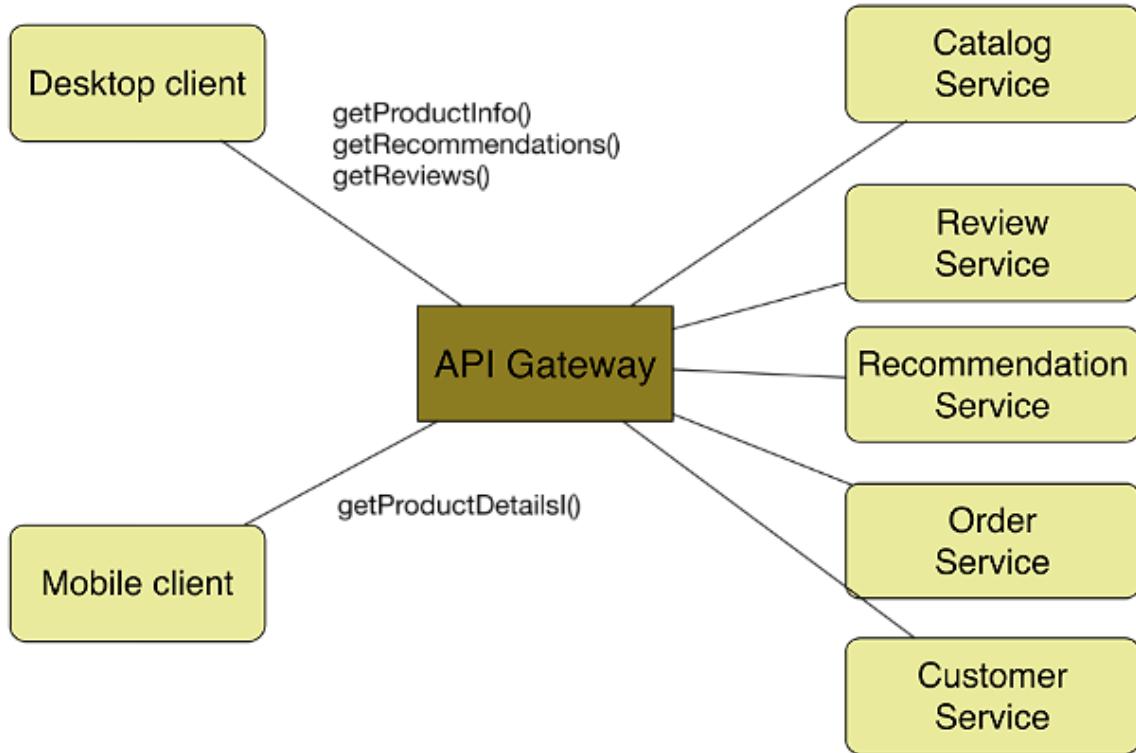


Figura 1: Architettura a microservizi e API Gateway

Nella progettazione dell'architettura, il team ha seguito i vincoli posti dall'approccio architetturale REST, i quali sono consultabili alla pagina https://en.wikipedia.org/wiki/Representational_state_transfer.

Endpoints client

Ora verranno definiti gli endpoints esterni utilizzati dai vari tipi di client.

Per ogni risorsa sono stati specificati i formati per lo scambio dei dati in JSON:

- **Request:** rappresenta l'oggetto JSON che dovrà essere passato alla risorsa REST;
- **Response:** rappresenta l'oggetto JSON che fornirà in risposta la risorsa REST.

L'endpoint utilizzato dal client vocale è:

- /query

- **Method:** POST;
- **Descrizione:** invia all'API Gateway dei dati;
- **Request:** la richiesta deve contenere i campi organizzati come descritto in Back-end::APIGateway::VRequestAPIBody:

```

1 {
2   "app": "String",
3   "audio": "Base64String",
4   "data": "ObjectAssocArray",
5   "session_id": "String"
6 }
```

- **Response:** la risposta deve contenere i campi organizzati come descritto in `Back-end::VirtualAssistant::VAResponse`:

```

1 {
2 "action": "String",
3 "res": {
4   "contexts": "ObjectAssocArray",
5   "data": "Object",
6   "text_request": "String",
7   "text_response": "String"
8 },
9 "session_id": "String"
10 }
```

Microservizi

Di seguito verranno esposti e spiegati i funzionamenti di ogni microservizio da implementare.

Virtual Assistant

Descrizione

Il microservizio Virtual Assistant (VA) fornisce le funzionalità di un assistente virtuale. Fa affidamento ad api.ai, e si occupa di inoltrare le richieste ricevute a tale infrastruttura. Avvalendosi di un database, permette di utilizzare diversi agenti (ciò che trasforma il linguaggio naturale in dati capibili per un'applicazione) durante la stessa interazione, consentendo quindi di definire diverse “applicazioni”. Per ogni applicazione, si dovrà definire un agente.

Le richieste fatte all'unico endpoint di questo microservizio richiedono infatti di comunicare anche il nome dell'applicazione a cui è legata la richiesta. Questo permette di separare in diversi agenti di api.ai dialoghi legati a diverse funzionalità, consentendo lo sviluppo di diverse funzionalità da parte di sviluppatori diversi, e l'integrazione di eventuali funzionalità già esistenti senza dover modificare direttamente gli agenti di api.ai.

Per avere un completo controllo sul flusso della conversazione, si dovrà fare utilizzo di un database contenente gli agenti utilizzabili, in maniera tale che gli agenti utilizzabili siano solo quelli definiti e registrati.

Il microservizio si occupa anche di notificare, tramite l'utilizzo di AWS SNS, dell'avvenuta interazione da parte dell'utente, permettendo così il salvataggio delle conversazioni in un database di supporto, il quale potrebbe essere utilizzato magari per fini di machine learning.

Di seguito vengono esposti i vari passaggi:

- arriva una richiesta;
- interrogo il database, contenente gli agenti, utilizzando il nome dell'applicazione come chiave;
- dall'interrogazione precedente ottengo il token dell'agente relativo all'applicazione;
- invio ad api.ai il token e il testo della richiesta;
- api.ai fornisce la risposta, la quale viene “filtrata” da `Back-end::VirtualAssistant::ApiAiVAAAdapter::query(str: VAQuery): VAResponse`, ottenendo così un formato adatto a `Back-end::VirtualAssistant::VAResponse`;
- pubblico, tramite il servizio Amazon SNS, la risposta filtrata che verrà quindi inviata al Client.

Endpoints

Ora verranno definiti gli Endpoints utilizzati per i passaggi di risorse con il microservizio Virtual Assistant.

Per ogni risorsa sono stati specificati i formati per lo scambio dei dati in JSON:

- **Request:** rappresenta l'oggetto JSON che dovrà essere passato alla risorsa REST;
- **Response:** rappresenta l'oggetto JSON che fornirà in risposta la risorsa REST.

Gli Endpoints sono:

- `/query`

- **Method:** POST;
- **Descrizione:** invia al microservizio una richiesta per interrogare l'assistente virtuale;
- **Request:** la richiesta deve contenere i campi organizzati come descritto in `Back-end::VirtualAssistant::VAQuery`:

```

1 {
2   "data": "Object",
3   "event": { /*Alternativo a "text", oggetto di tipo
4             VAEEventObject*/
5     "data": "Object",
6     "name": "String"
7   },
8   "session_id": "String",
9   "text": "String" /*Alternativo a "event"*/
9 }
```

- **Response:** la risposta deve contenere i campi organizzati come descritto in `Back-end::VirtualAssistant::VAResponse`:

```

1 {
2   "action": "String",
3   "res": { /*oggetto di tipo ResponseBody*/
4     "contexts": "ObjectArray",
5     "data": "Object",
6     "text_request": "String",
7     "text_response": "String"
8   },
9   "session_id": "String"
10 }
```

Notifications

Descrizione

Il microservizio Notifications si occupa di mandare messaggi di notifica nei canali adeguati per notificare gli interessati dell'arrivo di un'ospite in azienda. Fornisce le API per richiedere la lista dei possibili destinatari, e per mandare il messaggio di notifica in un determinato canale. La lista dei canali viene restituita come un'array di stringhe, ognuna delle quali rappresenta un canale.

Il microservizio si occupa di interrogare le diverse liste fornite dalla piattaforma di messaggistica scelta e di combinarle in un'unica lista. Nel nostro caso, la piattaforma di messaggistica è Slack e le diverse liste fornite riguardano utente, canali e gruppi privati. Quando si vuole mandare un messaggio, il campo `Back-end::Notifications::NotificationMessageEvent::send_to` indica

chi è il destinatario di tale messaggio.

`Back-end::Notifications::NotificationMessageEvent::msg` invece contiene il messaggio vero e proprio, nel formato definito dalla piattaforma di messaggistica su cui si appoggia il microservizio. Il formato utilizzato da Slack è consultabile qui <https://api.slack.com/docs/message-buttons>.

Endpoints

Ora verranno definiti gli Endpoints utilizzati per i passaggi di risorse con il microservizio Notifications.

Per ogni risorsa sono stati specificati i formati per lo scambio dei dati in JSON:

- **Request:** rappresenta l'oggetto JSON che dovrà essere passato alla risorsa REST;
- **Response:** rappresenta l'oggetto JSON che fornirà in risposta la risorsa REST.

Gli Endpoints sono:

- `/notifications`

- **Method:** GET;
- **Descrizione:** restituisce la lista dei possibili canali destinatari;
- **Response:** la risposta deve contenere i campi organizzati come descritto in `Back-end::Notifications::NotificationChannel`:

```

1 {
2   "created": "String",
3   "creator": "String",
4   "id": "String",
5   "is_archived": "boolean",
6   "is_member": "boolean",
7   "name": "String"
8   "num_members": "int",
9   "purpose": "Purpose",
10  "topic": "Topic",
11 }
```

- **Method:** POST;
- **Descrizione:** invia la notifica ad una determinata persona;
- **Request:** la richiesta deve contenere i campi organizzati come descritto in `Back-end::Notifications::NotificationMsg`:

```

1 {
2   "msg": [
3     "attachments_array": [
4       {
5         },
6         },
7         {
8           },
9           ],
10      "response_type": "String",
11      "text": "String"
12    ],
```

```

14     "send_to": "String"
15 }
```

Users

Descrizione

Il microservizio Users si occupa della gestione degli amministratori del nostro sistema. Esso fornisce delle API REST per modificare i dati relativi agli amministratori del nostro sistema presenti in un database. Viene integrato col servizio di Speech Recognition dall'API Gateway, per fornire la possibilità di effettuare il login, tramite impronta vocale, nel sistema.

Endpoints

Ora verranno definiti gli Endpoints utilizzati per i passaggi di risorse con il microservizio Users. Per ogni risorsa sono stati specificati i formati per lo scambio dei dati in JSON:

- **Request:** rappresenta l'oggetto JSON che dovrà essere passato alla risorsa REST;
- **Response:** rappresenta l'oggetto JSON che fornirà in risposta la risorsa REST.

Gli Endpoints sono:

- `/auth/users`

- **Method:** GET;
- **Descrizione:** restituisce la lista degli User presenti nel database;
- **Response:** la risposta deve contenere un array contenente i campi organizzati come descritto in `Back-end::Auth::User`:

```

1 {
2   user_array : [
3     {
4       "first_name": "String",
5       "last_name": "String",
6       "password": "String",
7       "slack_channel": "String",
8       "sr_id": "String",
9       "username": "String"
10    },
11    {
12      "first_name": "String",
13      "last_name": "String",
14      "password": "String",
15      "slack_channel": "String",
16      "sr_id": "String",
17      "username": "String"
18    }
19  ]
20 }
```

- **Method:** POST;
- **Descrizione:** vengono inviati i dati necessari all'aggiunta di un nuovo User al database;

- **Request:** la richiesta deve contenere i campi organizzati come descritto in `Back-end::Auth::User`:

```

1 {
2   "first_name": "String",
3   "last_name": "String",
4   "password": "String",
5   "slack_channel": "String",
6   "sr_id": "String",
7   "username": "String"
8 }
```

- **/auth/users/:username**

- **Method:** PUT;
- **Descrizione:** vengono modificati i dati dell’utente tramite sovrascrittura;
- **Request:** la richiesta deve contenere i campi organizzati come descritto in `Back-end::Auth::User`

```

1 {
2   "first_name": "String",
3   "last_name": "String",
4   "password": "String",
5   "slack_channel": "String",
6   "sr_id": "String",
7   "username": "String"
8 }
```

- **Method:** DELETE;

- **Descrizione:** viene eliminato un utente;

- **Method:** GET;
- **Descrizione:** vengono ricevuti i dati relativi ad un utente;
- **Response:** la risposta deve contenere i campi organizzati come descritto in `Back-end::Auth::User`:

```

1 {
2   "first_name": "String",
3   "last_name": "String",
4   "password": "String",
5   "slack_channel": "String",
6   "sr_id": "String",
7   "username": "String"
8 }
```

Rules

Descrizione

Il microservizio Rules si occupa della gestione delle direttive del sistema. Una direttiva è un’istruzione che viene data da un amministratore al sistema, la quale permette di modificare il comportamento del sistema stesso al verificarsi di certe condizioni. Tali condizioni possono essere legate alla persona che interagisce col sistema, la sua azienda di provenienza, oppure alla persona desiderata

che viene richiesta.

Il sistema fornisce una serie di funzioni per modificare il suo comportamento, le quali indicano il modo in cui esso debba essere cambiato.

Una direttiva è costituita da:

- una lista di target, che indica gli obiettivi (persone) ai quali deve essere applicata la direttiva;
- un'istanza di funzione, che indica quale delle funzioni disponibili deve essere applicata e, nel caso in cui tale funzione abbia dei parametri modificabili, con quali valori di quest'ultimi deve essere chiamata;
- un nome, il quale permette agli amministratori di identificare le diverse direttive;
- un id, il quale identifica univocamente la funzione all'interno del sistema;
- una flag di abilitazione, che permette di abilitare e disabilitare l'applicazione della direttiva da parte del sistema.

Endpoints

Ora verranno definiti gli Endpoints utilizzati per i passaggi di risorse con il microservizio Rules. Per ogni risorsa sono stati specificati i formati per lo scambio dei dati in JSON:

- **Request:** rappresenta l'oggetto JSON che dovrà essere passato alla risorsa REST;
- **Response:** rappresenta l'oggetto JSON che fornirà in risposta la risorsa REST.

Gli Endpoints sono:

- **/impostazioni**
 - **Method:** GET;
 - **Descrizione:** viene ricevuta la lista delle direttive;
 - **Response:** la risposta deve contenere un array contenente i campi organizzati come descritto in
Back-end::Rules::Rule:

```

1  {
2   "rule_array" : [
3   {
4     "ac_list": [
5       "adm1": "String",
6       "adm2": "String",
7       ...
8     ],
9     "ac_mode": "int",
10    "enabled": "boolean",
11    "id": "int",
12    "name": "String",
13    "targets": [ /*oggetti di tipo Back-end::Rules::RuleTarget
14      */
15    {
16      "company": "String",
17      "member": "String",
18      "name": "String"
19    },
20    {
21      "company": "String",
22      "member": "String",
23    }
24  ]
25 }
```

```

22     "name": "String"
23   }
24 ]
25 "task": { /* oggetto di tipo Back-end::Rules::RuleTaskInstance*/
26   "params": {
27     "par1": "Object",
28     "par2": "Object",
29     ...
30   }
31   "task": "String"
32 }
33 ]
34 }
35 }
```

- **Method:** POST;
- **Descrizione:** viene creata una nuova direttiva
- **Request:** la richiesta deve contenere i campi organizzati come descritto in Back-end::Rules::Rule:

```

1 {
2   rule : {
3     "ac_list": [
4       "adm1": "String",
5       "adm2": "String",
6       ...
7     ],
8     "ac_mode": "int",
9     "enabled": "boolean",
10    "id": "int",
11    "name": "String",
12    "targets": [ /* oggetti di tipo Back-end::Rules::RuleTarget */
13      {
14        "company": "String",
15        "member": "String",
16        "name": "String"
17      },
18      {
19        "company": "String",
20        "member": "String",
21        "name": "String"
22      }
23    ],
24    "task": { /* oggetto di tipo Back-end::Rules::RuleTaskInstance*/
25      "params": {
26        "par1": "Object",
27        "par2": "Object",
28        ...
29      }
30      "task": "String"
31    }
32 }
```

33 }

- [/impostazioni/:id](#)

- **Method:** PUT;
- **Descrizione:** viene modificata una direttiva tramite sovrascrittura;
- **Request:** la richiesta deve contenere i campi organizzati come descritto in `Back-end::Rules::Rule`:

```

1 {
2   rule : {
3     "ac_list": [
4       "adm1": "String",
5       "adm2": "String",
6       ...
7     ],
8     "ac_mode": "int",
9     "enabled": "boolean",
10    "id": "int",
11    "name": "String",
12    "targets": [ /* oggetti di tipo Back-end::Rules::RuleTarget */
13      {
14        "company": "String",
15        "member": "String",
16        "name": "String"
17      },
18      {
19        "company": "String",
20        "member": "String",
21        "name": "String"
22      }
23    ],
24    "task": { /* oggetto di tipo Back-end::Rules::RuleTaskInstance */
25      "params": {
26        "par1": "Object",
27        "par2": "Object",
28        ...
29      }
30      "task": "String"
31    }
32  }
33 }
```

- **Method:** GET;
- **Descrizione:** vengono richiesti dati relativi ad una specifica direttiva;
- **Response:** la risposta deve contenere i campi organizzati come descritto in `Back-end::Rules::Rule`:

```

1 {
2   rule : {
3     "ac_list": [
4       "adm1": "String",
```

```

5           "adm2": "String",
6           ...
7       ],
8   "ac_mode": "int",
9   "enabled": "boolean",
10  "id": "int",
11  "name": "String",
12  "targets": [ /* oggetti di tipo Back-end::Rules::RuleTarget */
13      {
14          "company": "String",
15          "member": "String",
16          "name": "String"
17      },
18      {
19          "company": "String",
20          "member": "String",
21          "name": "String"
22      }
23  ],
24  "task": { /* oggetto di tipo Back-end::Rules::RuleTaskInstance */
25      "params": {
26          "par1": "Object",
27          "par2": "Object",
28          ...
29      }
30      "task": "String"
31  }
32 }
33 }
```

- **Method:** DELETE;
- **Descrizione:** viene eliminata una direttiva;

/impostazioni/tasks

- **Method:** GET;
- **Descrizione:** viene richiesta la lista dei tipi di funzioni presenti nel sistema;
- **Response:** la risposta deve contenere i campi organizzati come descritto in Back-end::Rules::Task:

```

1  {
2     function_array : [
3     {
4         "function": "String",
5         "id": "int"
6     },
7     {
8         "function": "String",
9         "id": "int"
10    }
11 ]
12 }
```

- **Method:** POST;
- **Descrizione:** viene creato un nuovo tipo di Task;
- **Request:** la richiesta deve contenere i campi organizzati come descritto in Back-end::Rules::Task:

```

1 {
2   function_array : [
3   {
4     "function": "String",
5     "id": "int"
6   },
7   {
8     "function": "String",
9     "id": "int"
10  }
11 ]
12 }
```

/impostazioni/tasks/:type

- **Method:** GET;
- **Descrizione:** viene richiesto un Task in base al tipo di funzione applicata;
- **Response:** la risposta deve contenere i campi organizzati come descritto in Back-end::Rules::Task:

```

1 {
2   function_array : [
3   {
4     "function": "String",
5     "id": "int"
6   },
7   {
8     "function": "String",
9     "id": "int"
10  }
11 ]
12 }
```

- **Method:** PUT;
- **Descrizione:** viene modificata un Task tramite sovrascrittura;
- **Request:** la richiesta deve contenere i campi organizzati come descritto in Back-end::Rules::Task:

```

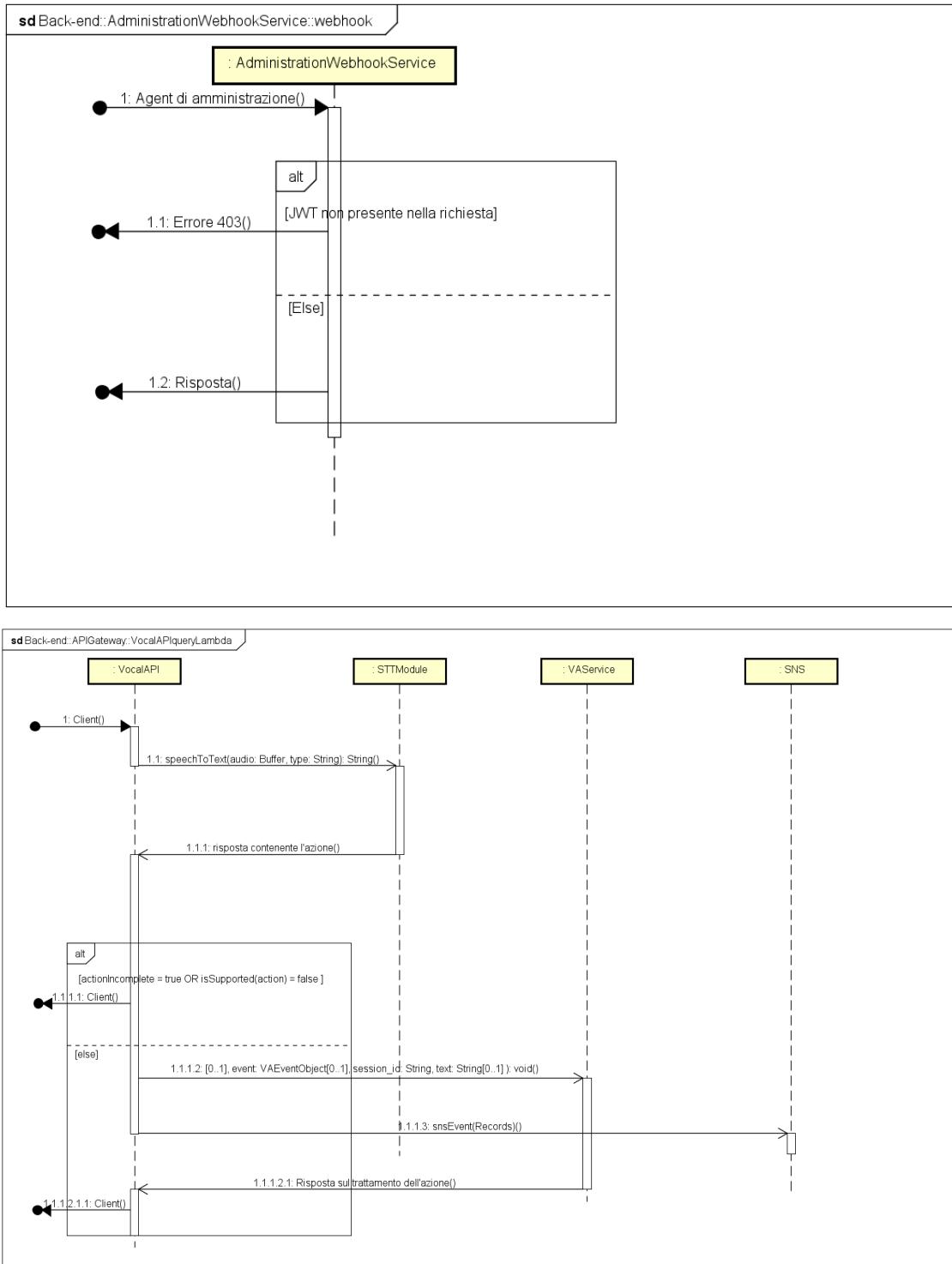
1 {
2   function_array : [
3   {
4     "function": "String",
5     "id": "int"
6   },
7   {
8     "function": "String",
9     "id": "int"
10  }
11 ]
```

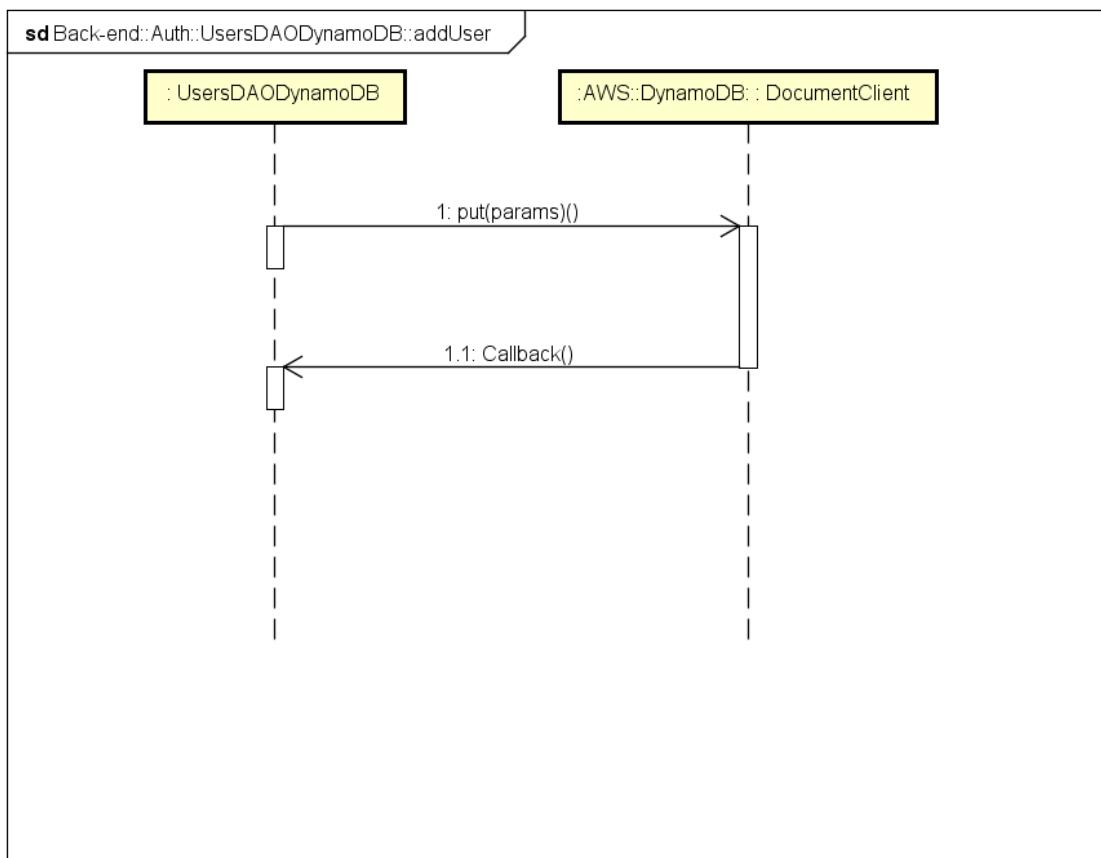
```
11 ]  
12 }
```

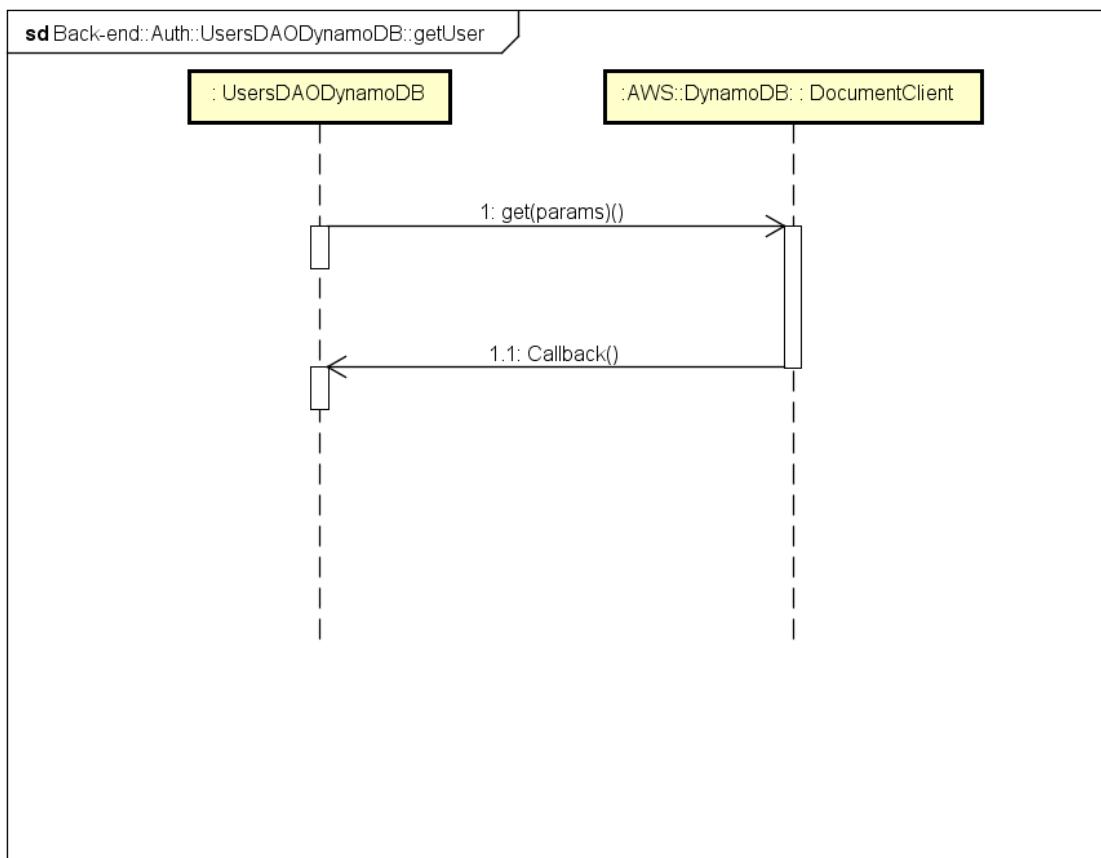
Diagrammi riassuntivi dei package

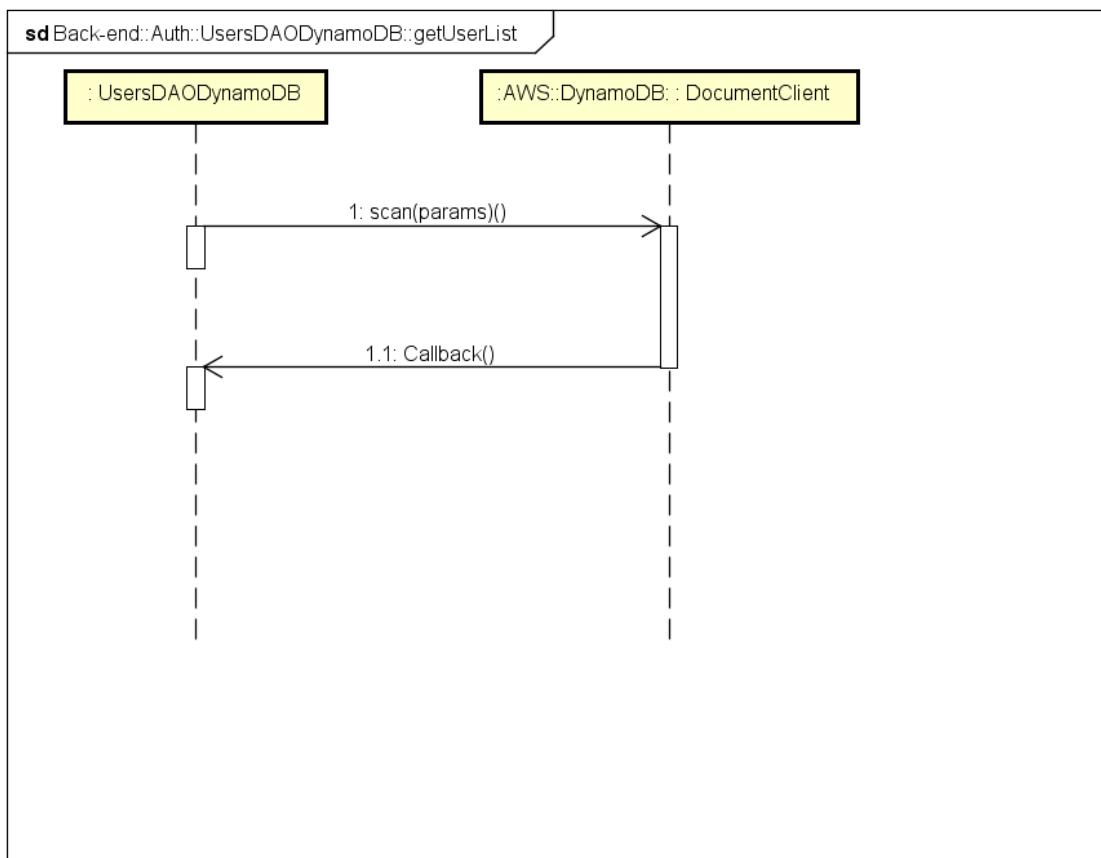
Diagrammi di sequenza

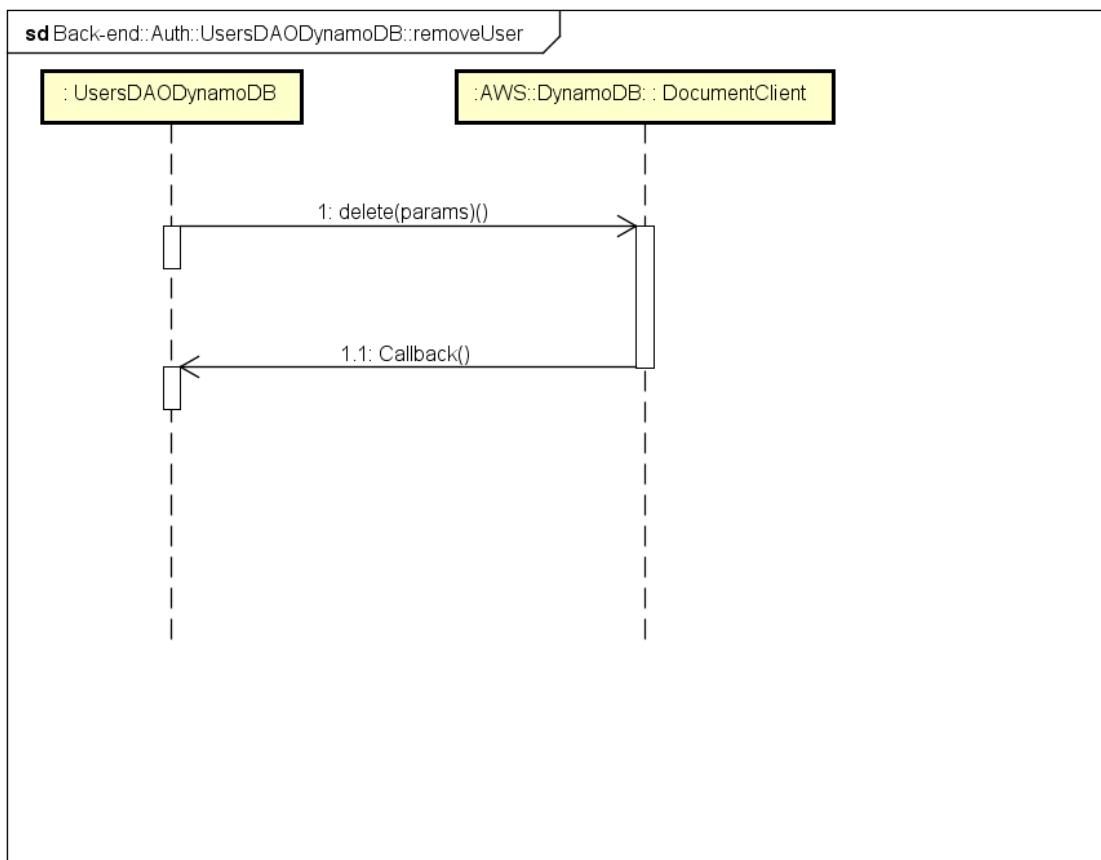
I diagrammi di sequenza.

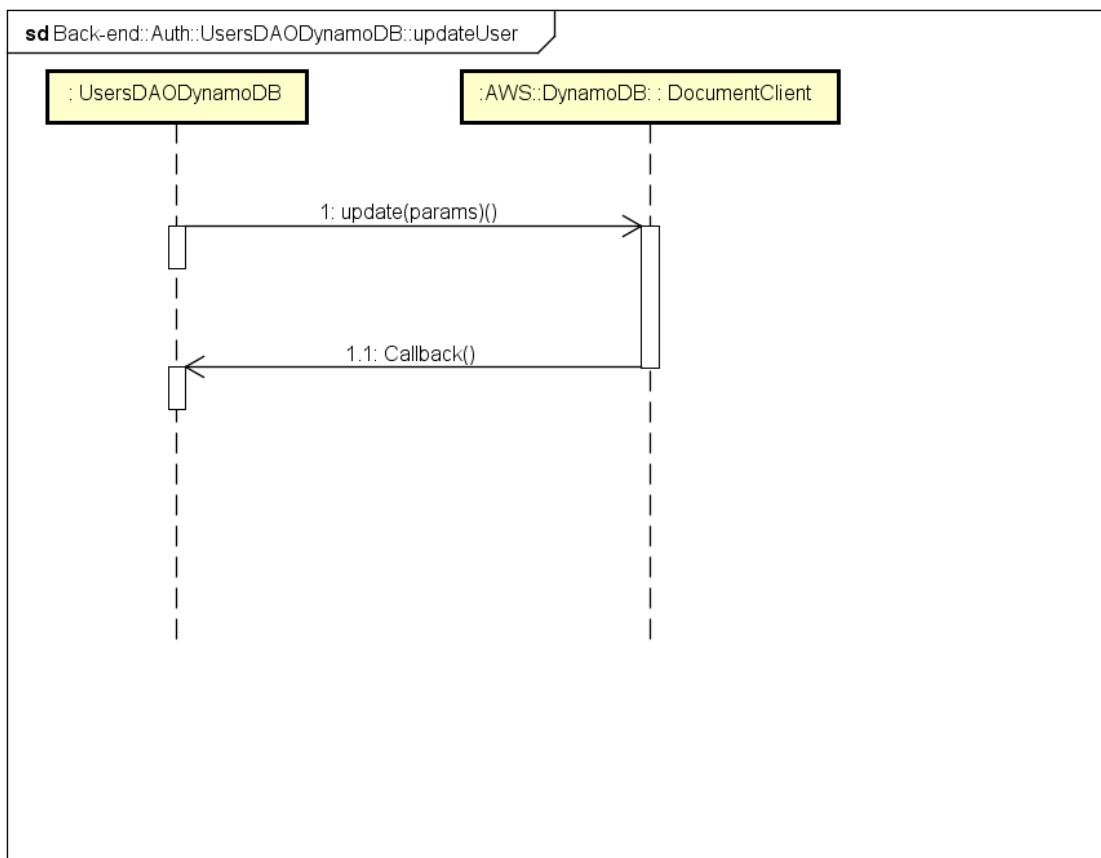


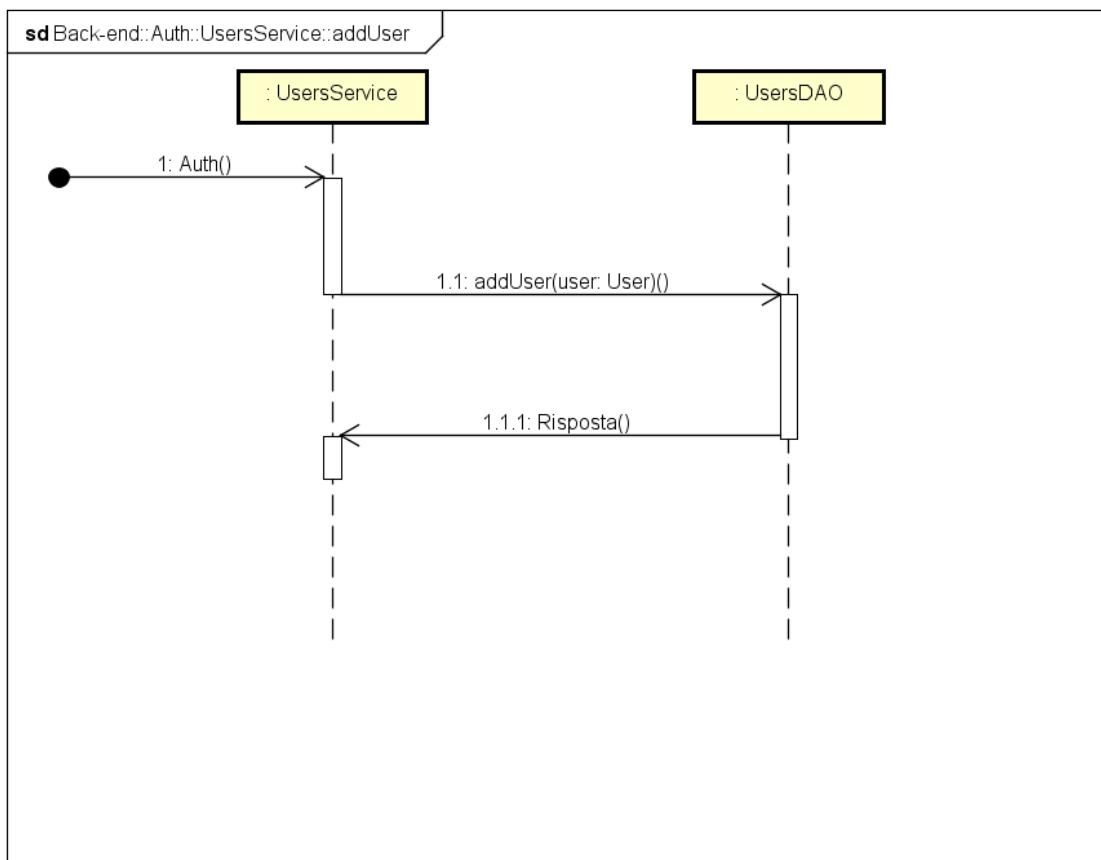


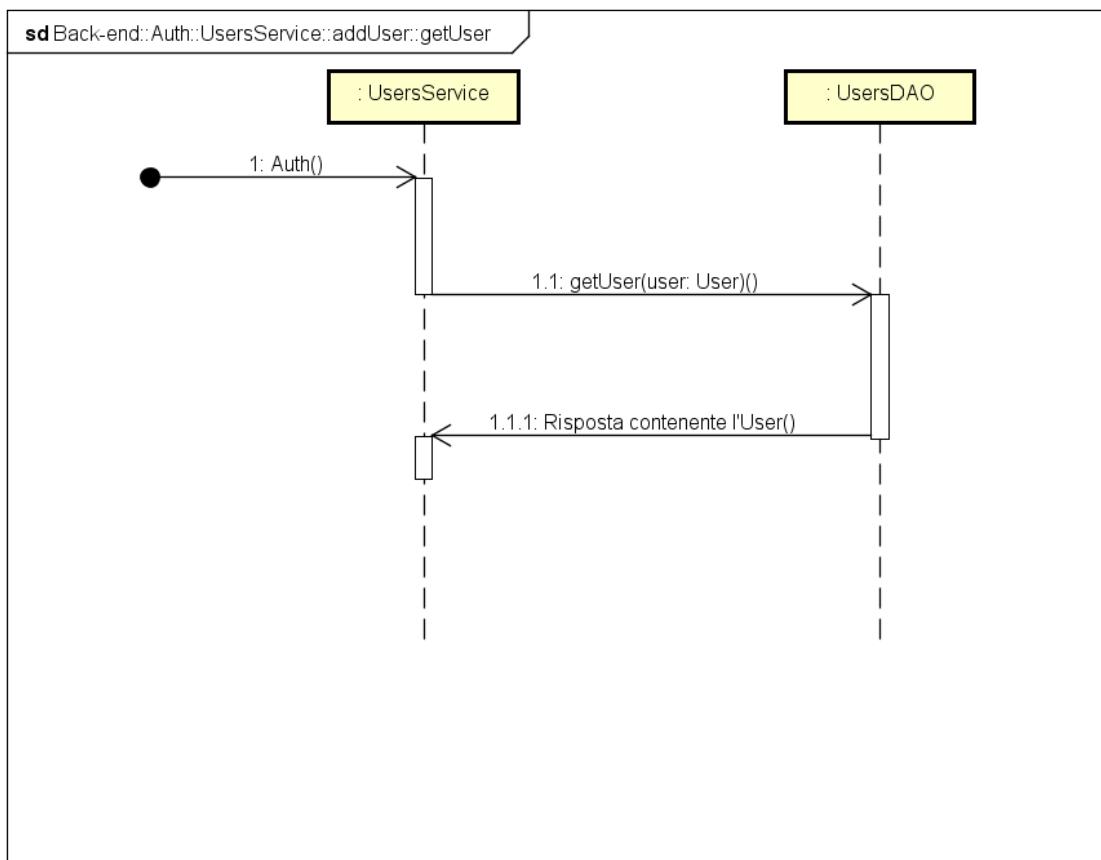


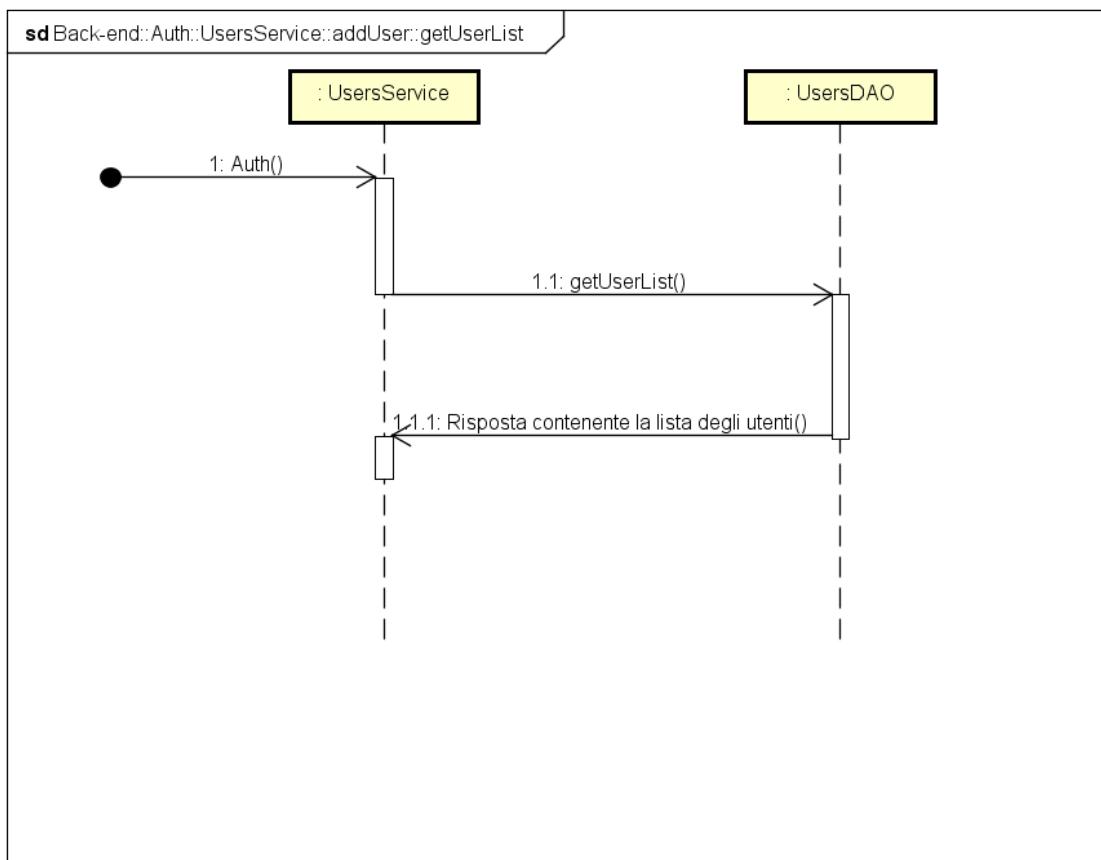


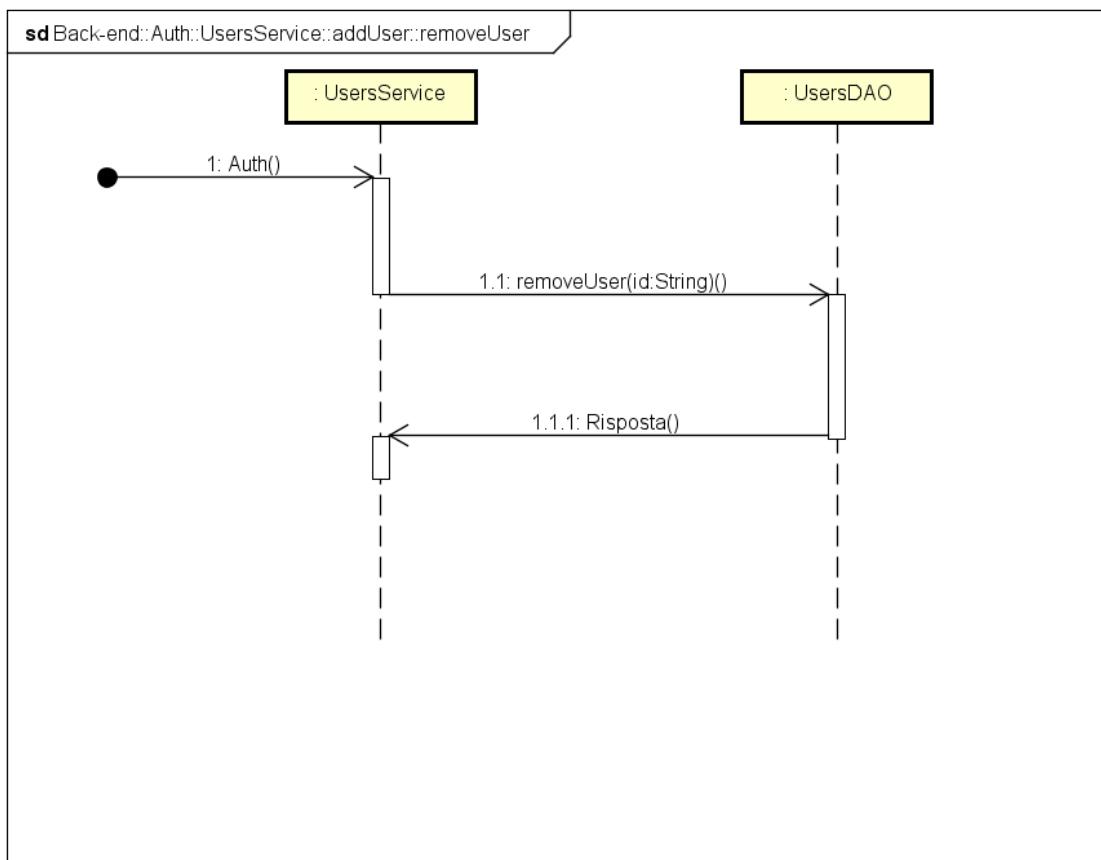


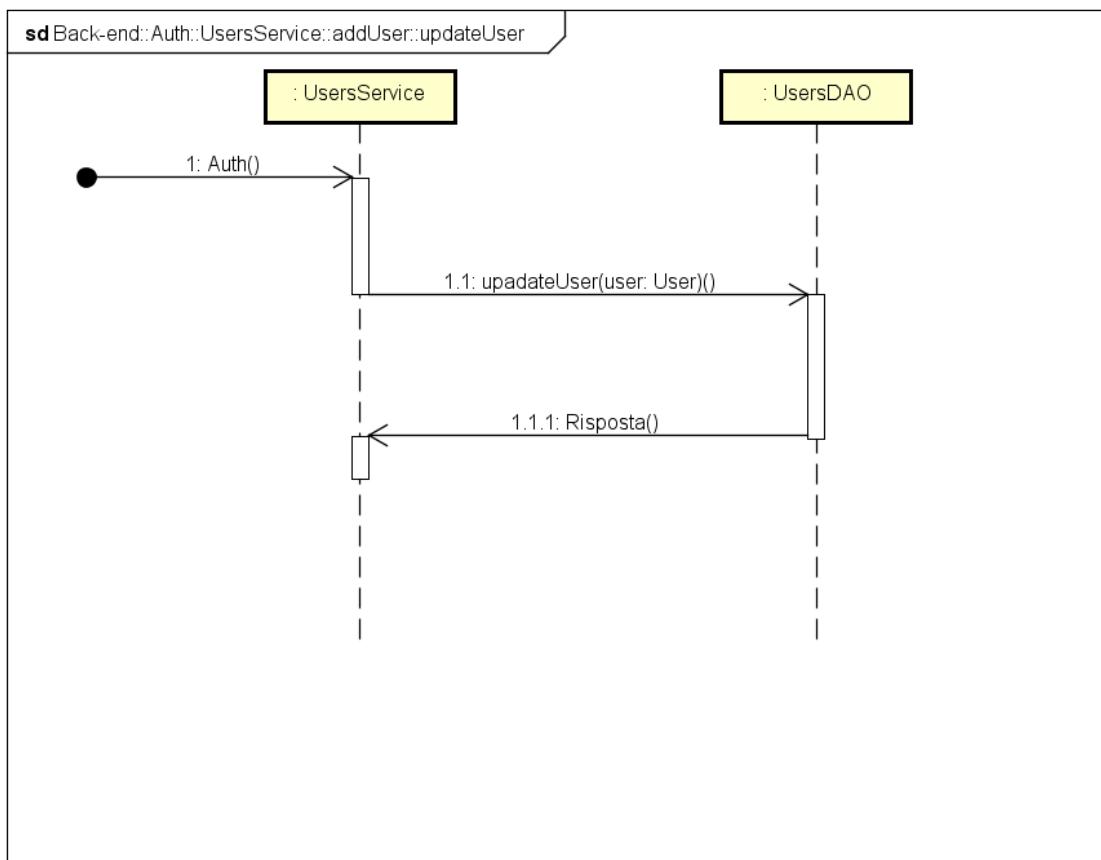


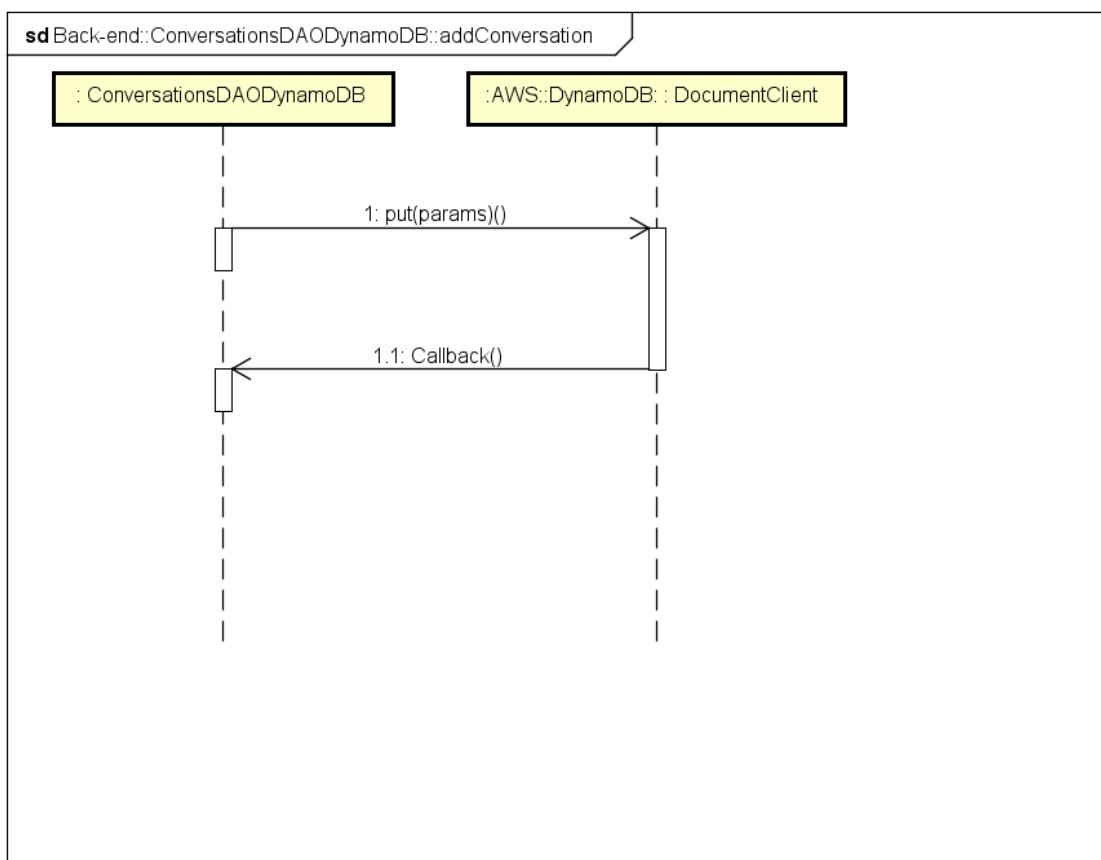


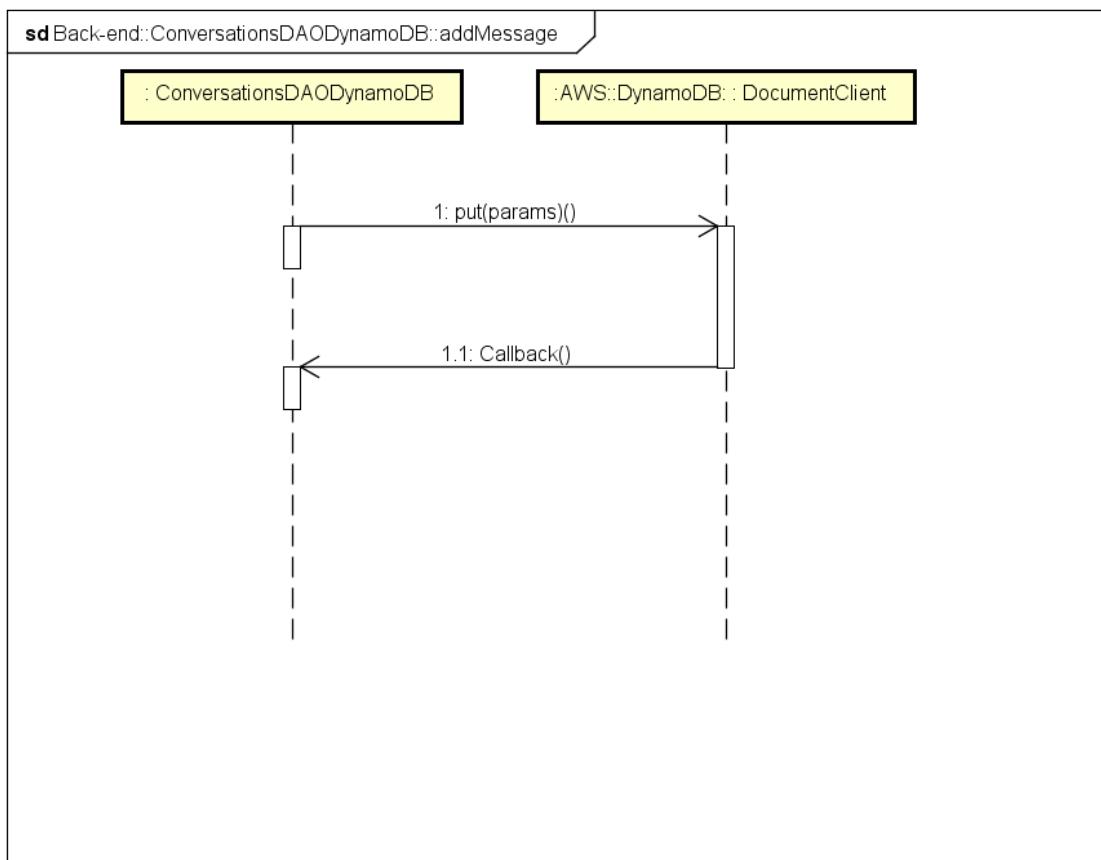


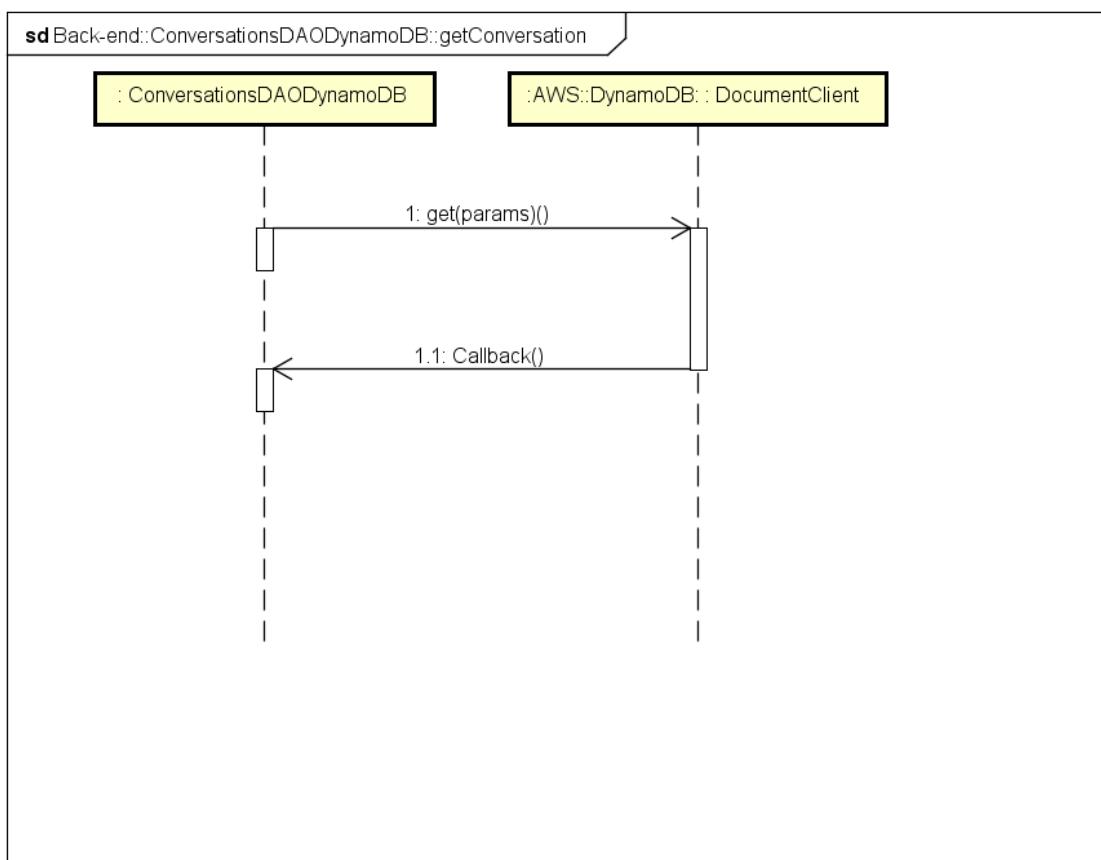


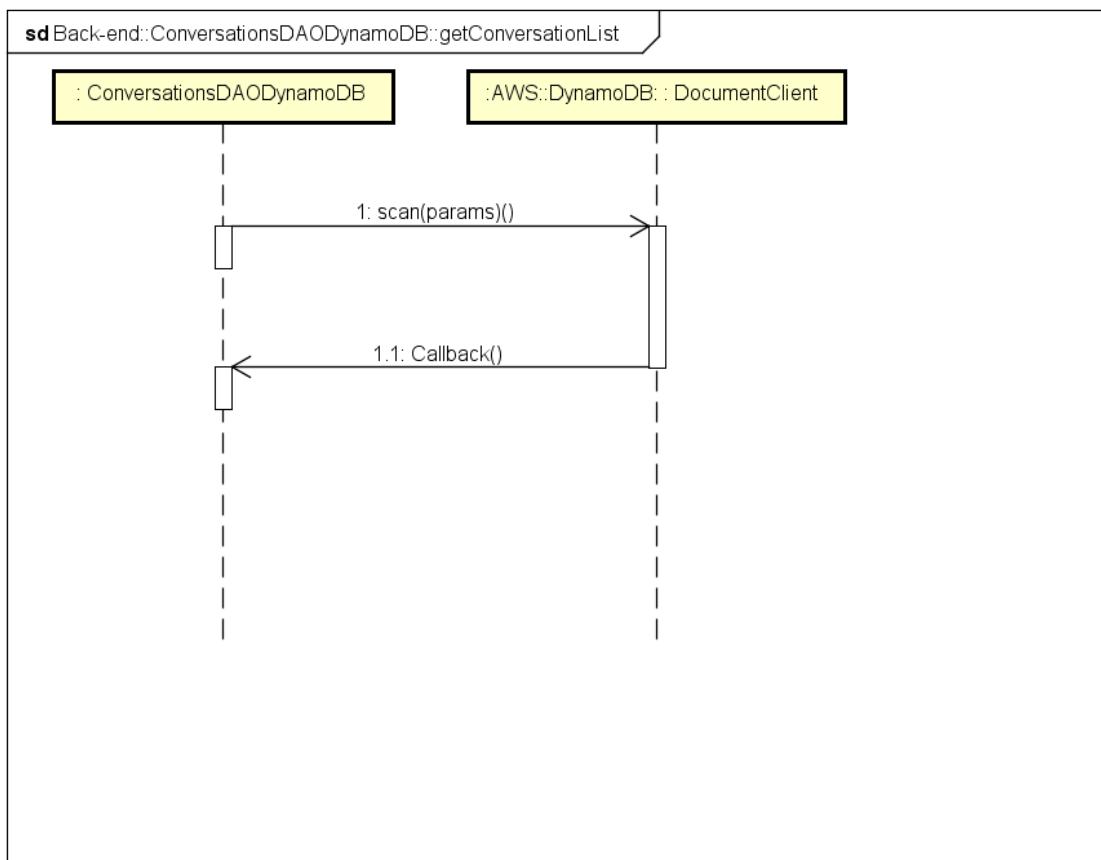


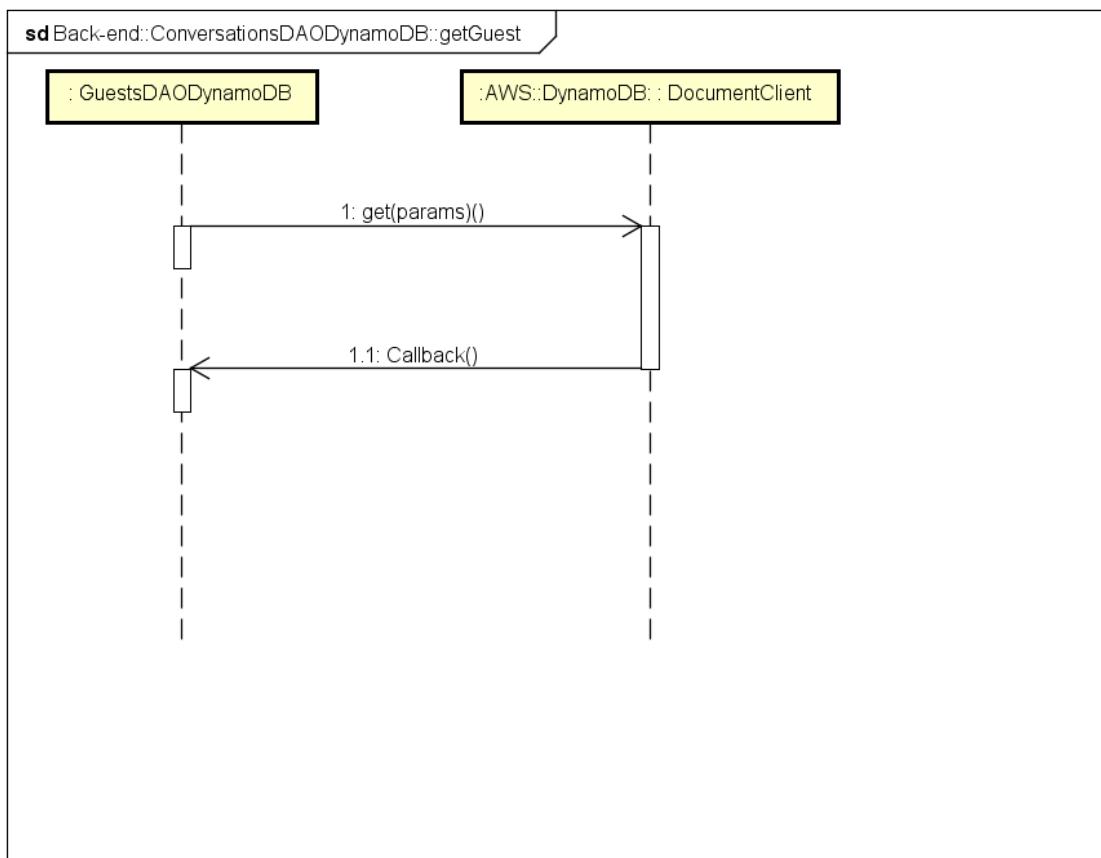


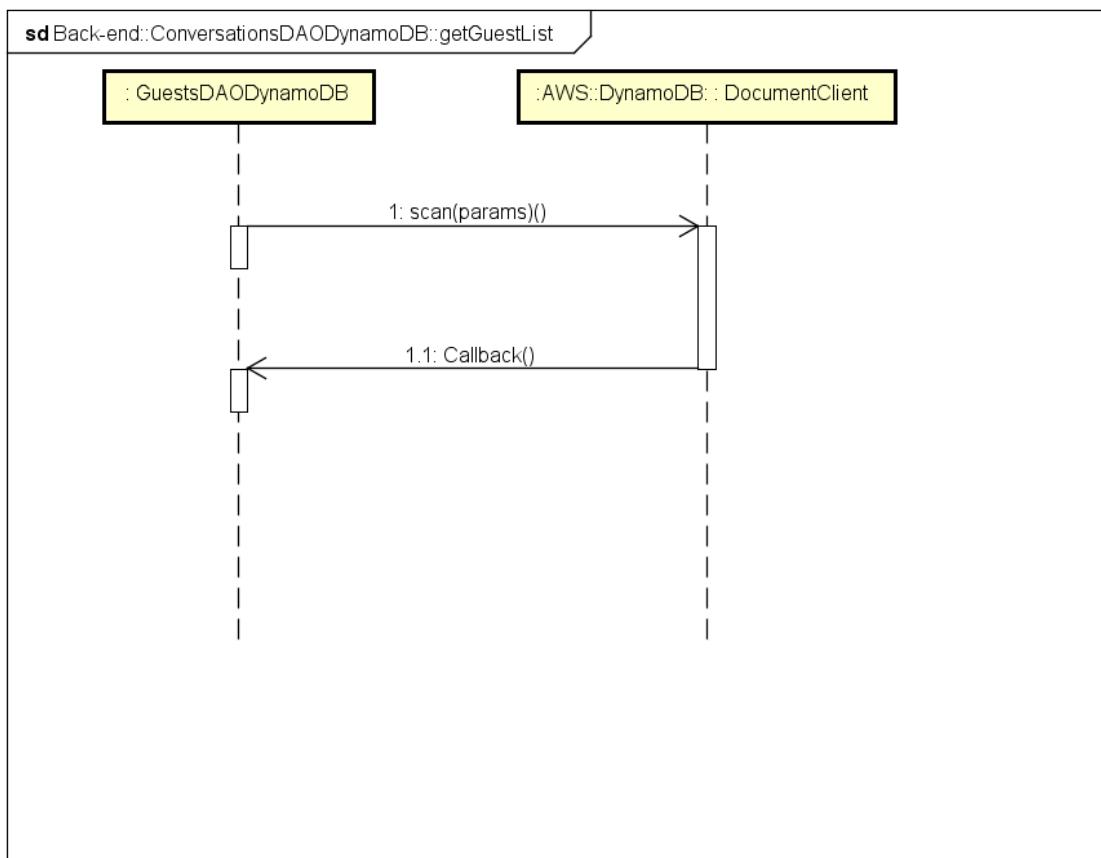


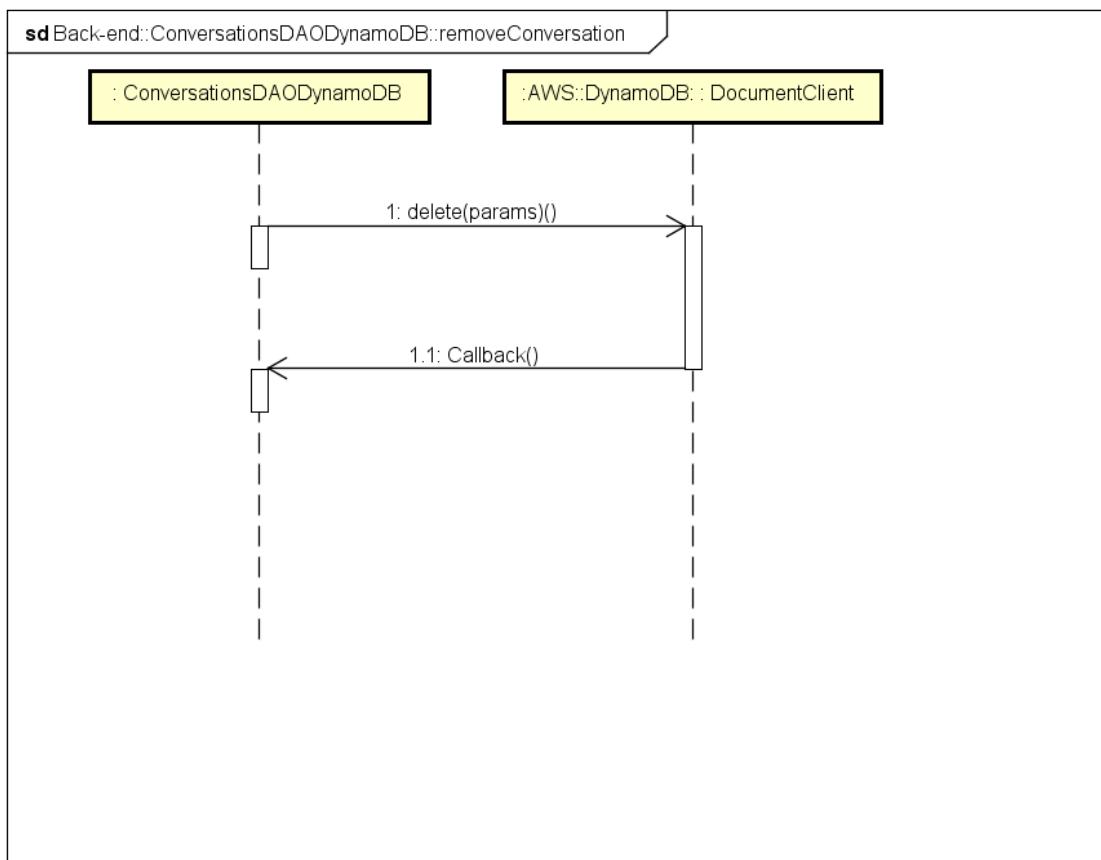


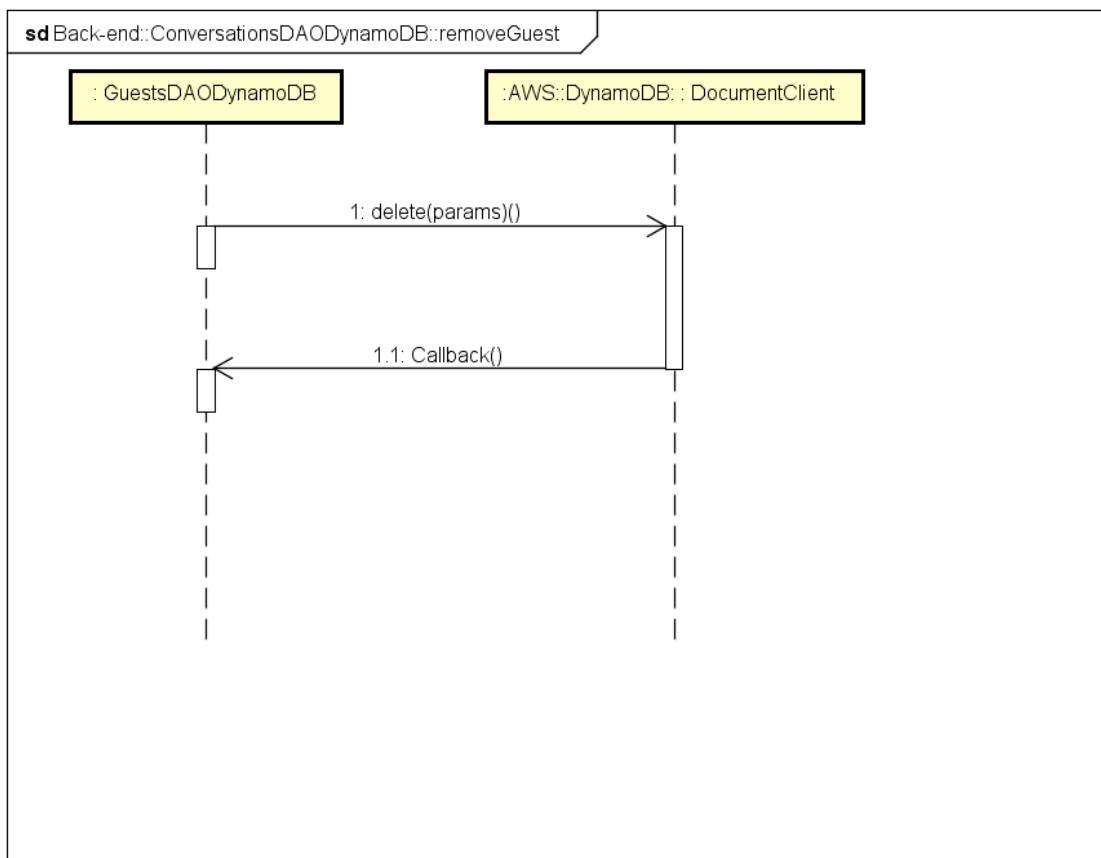


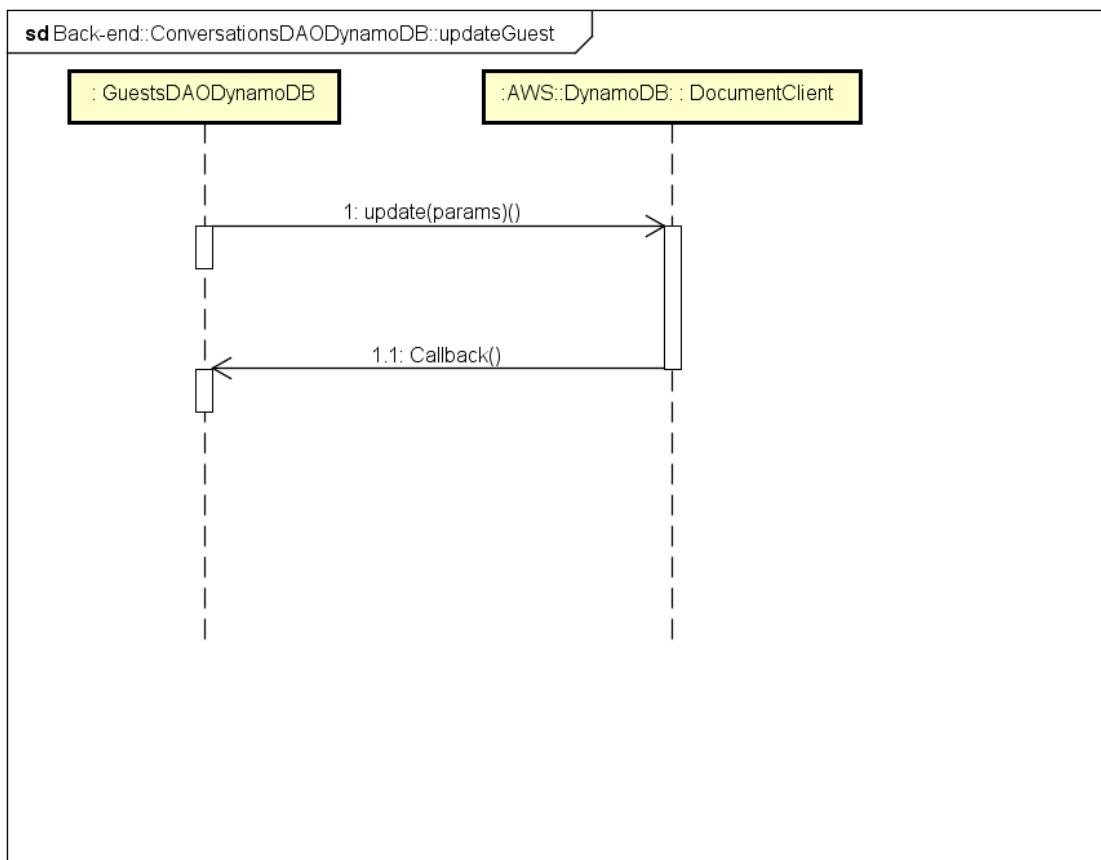


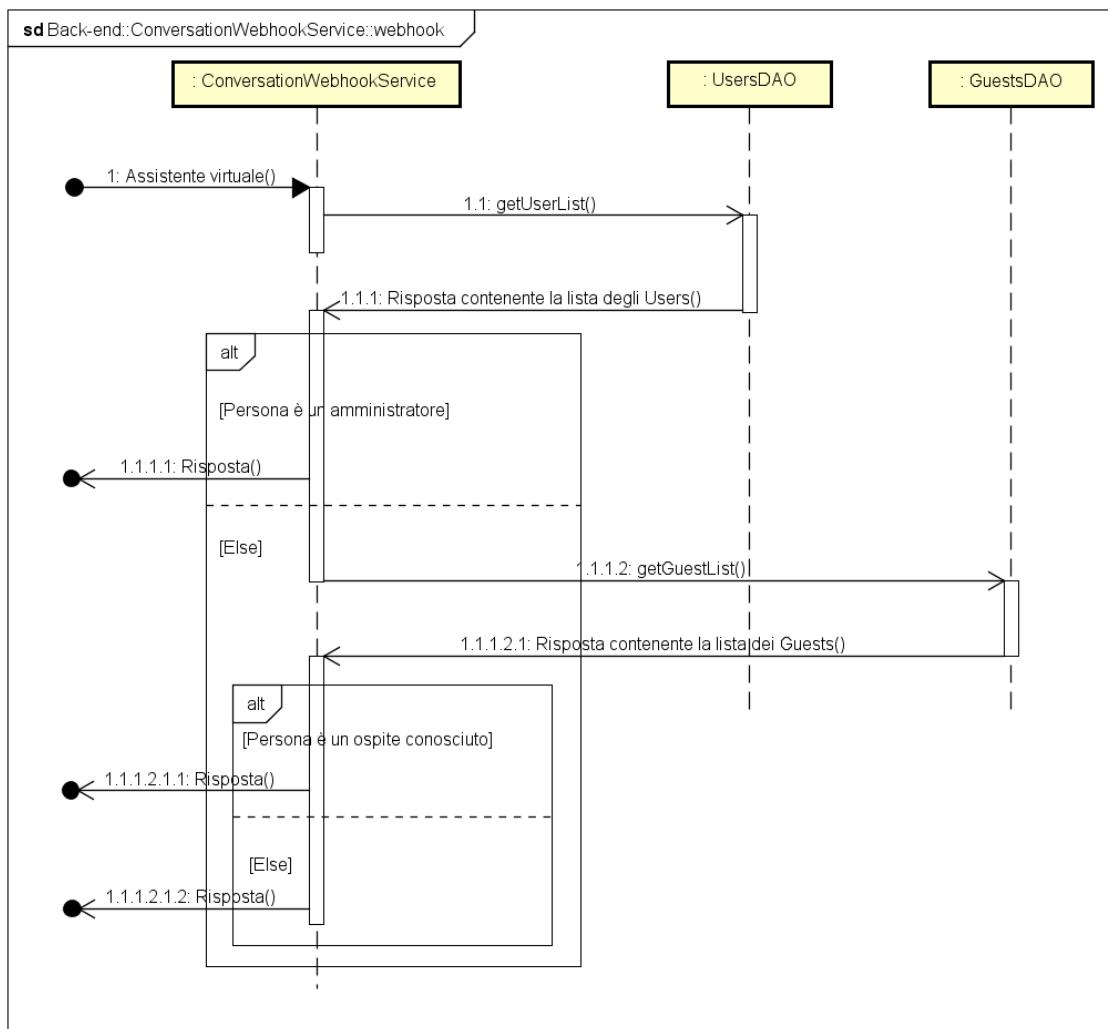


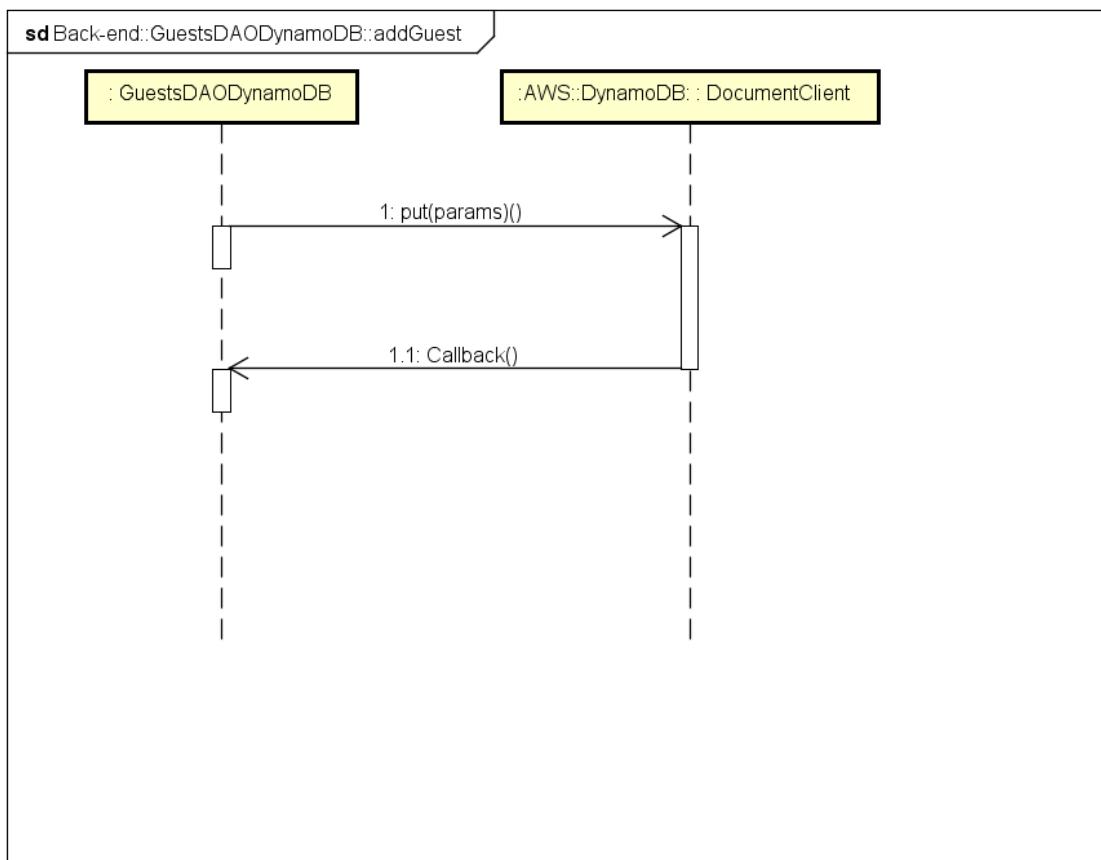


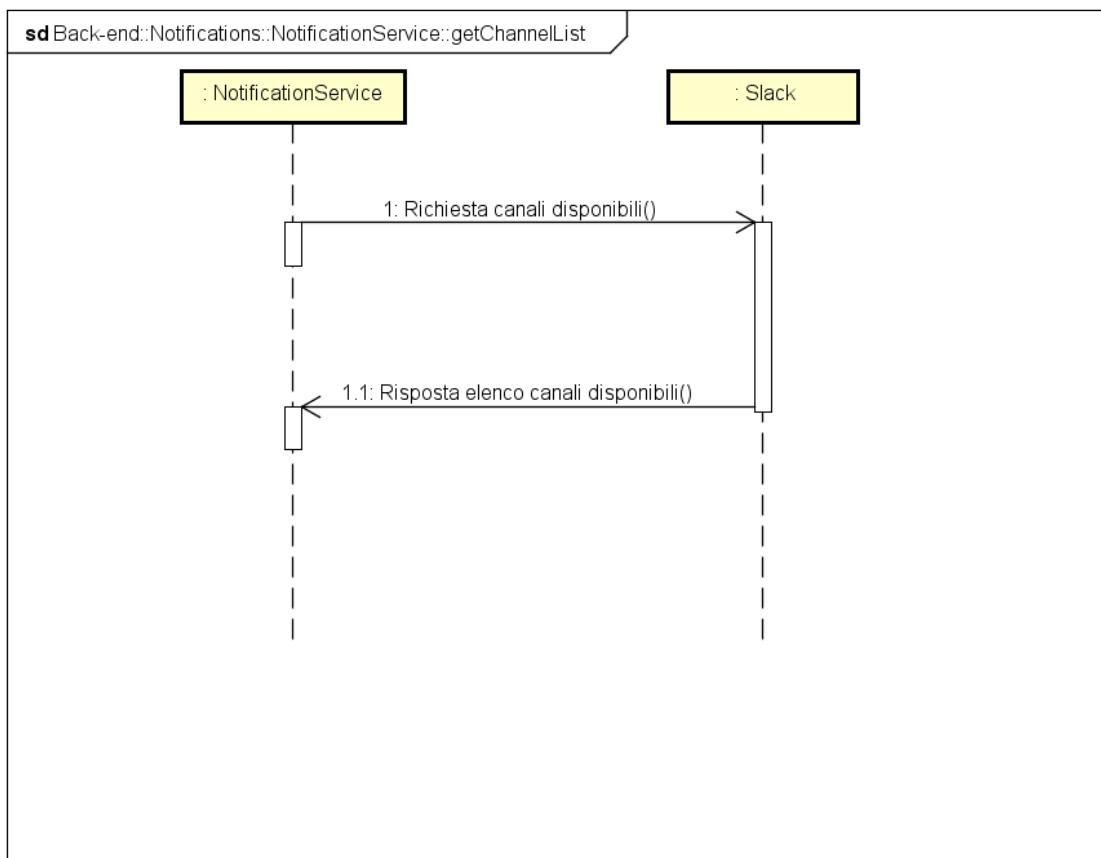


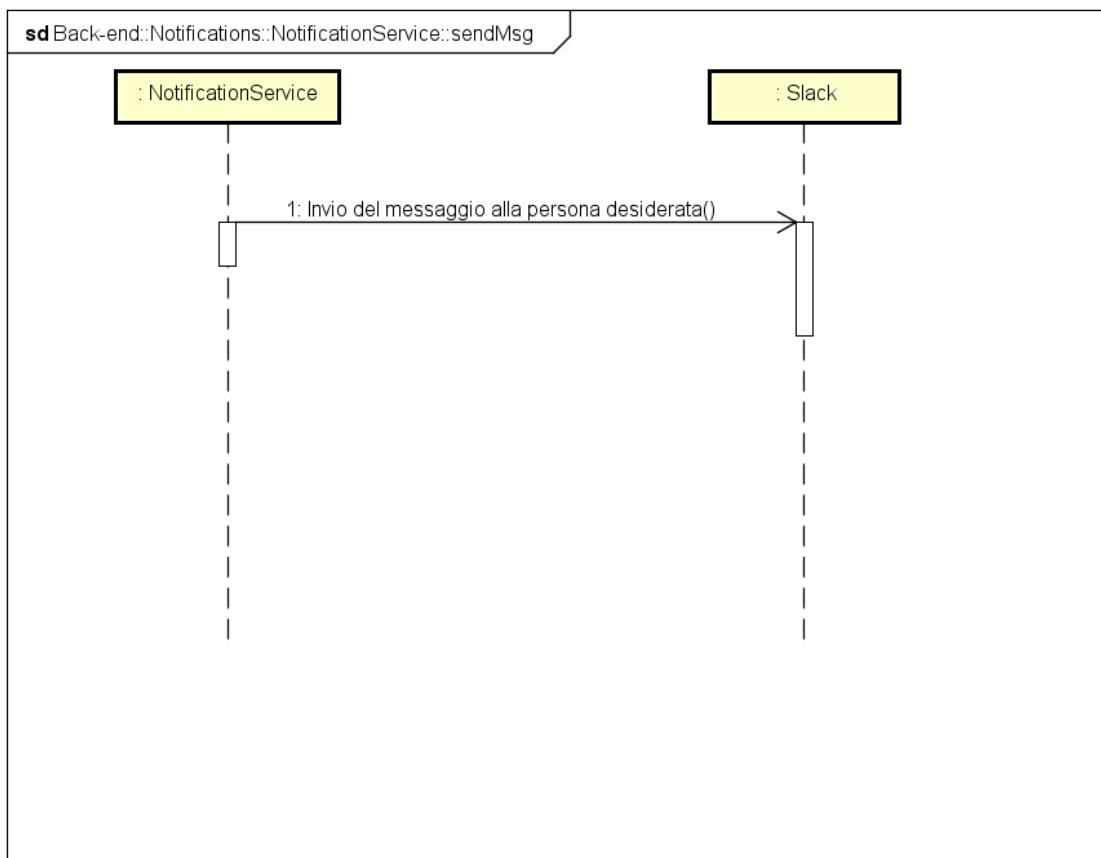


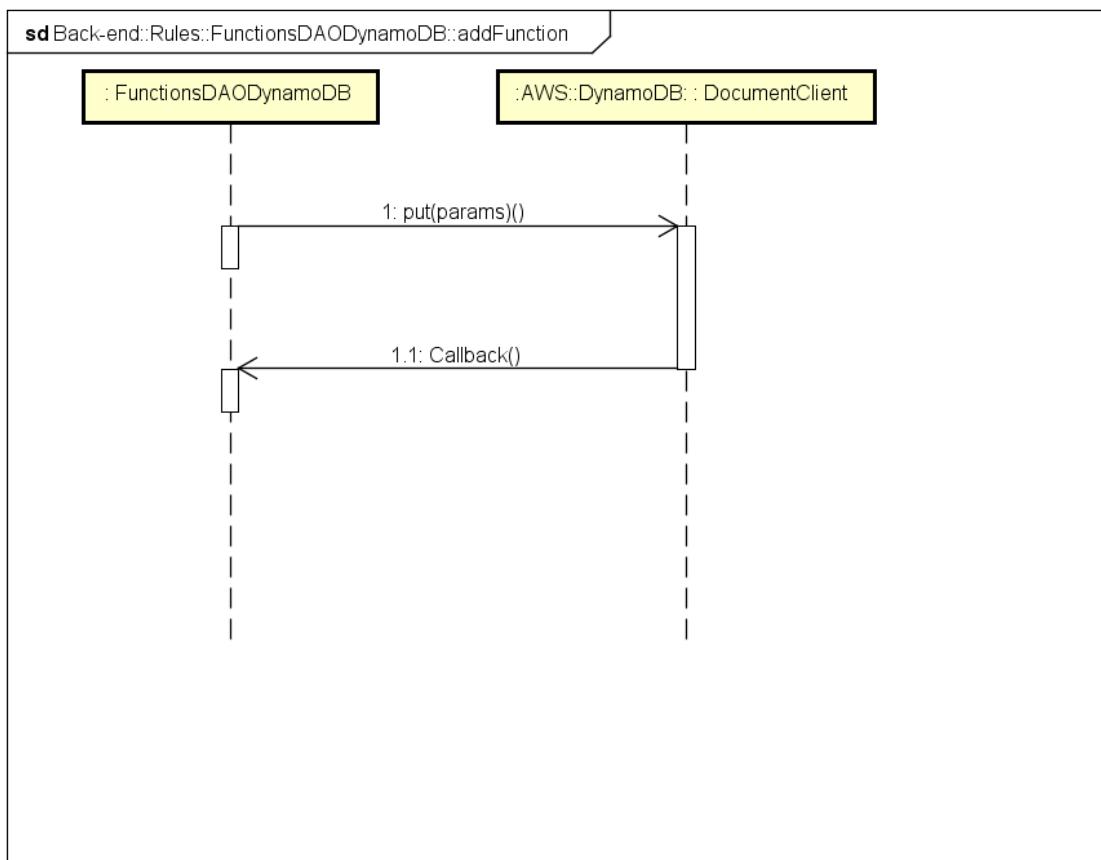


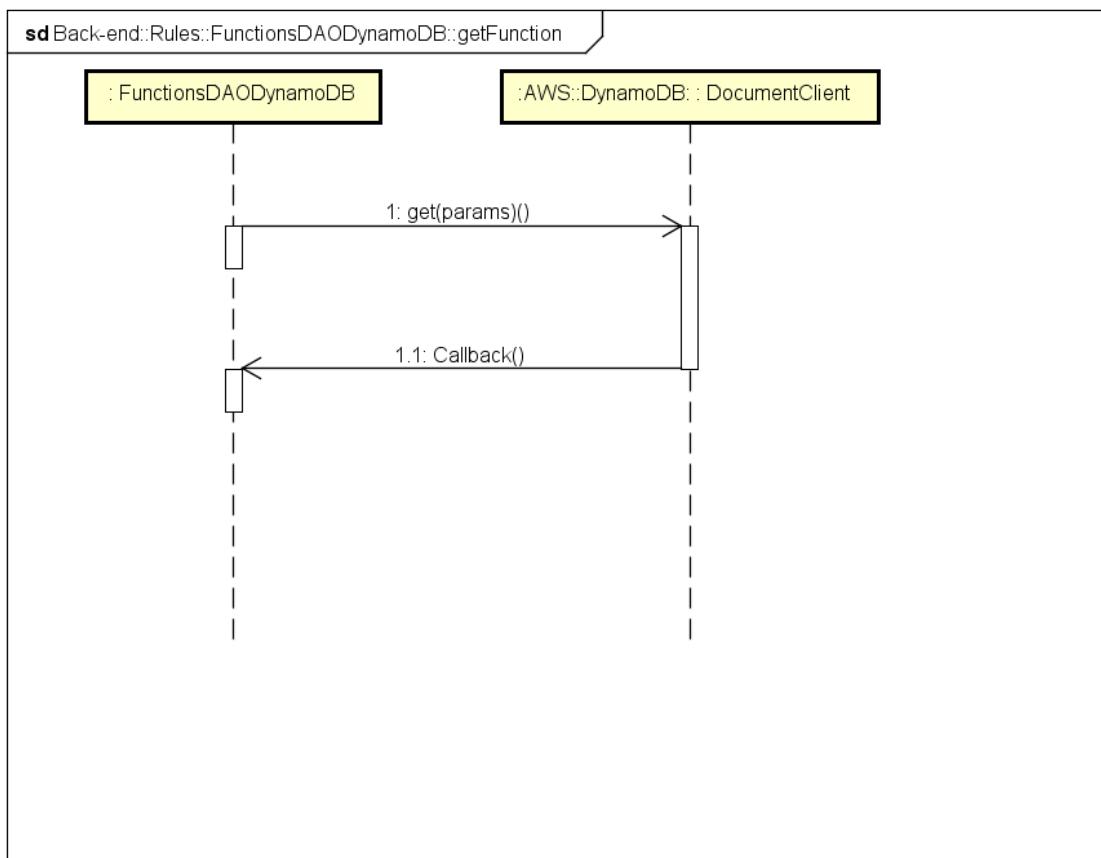


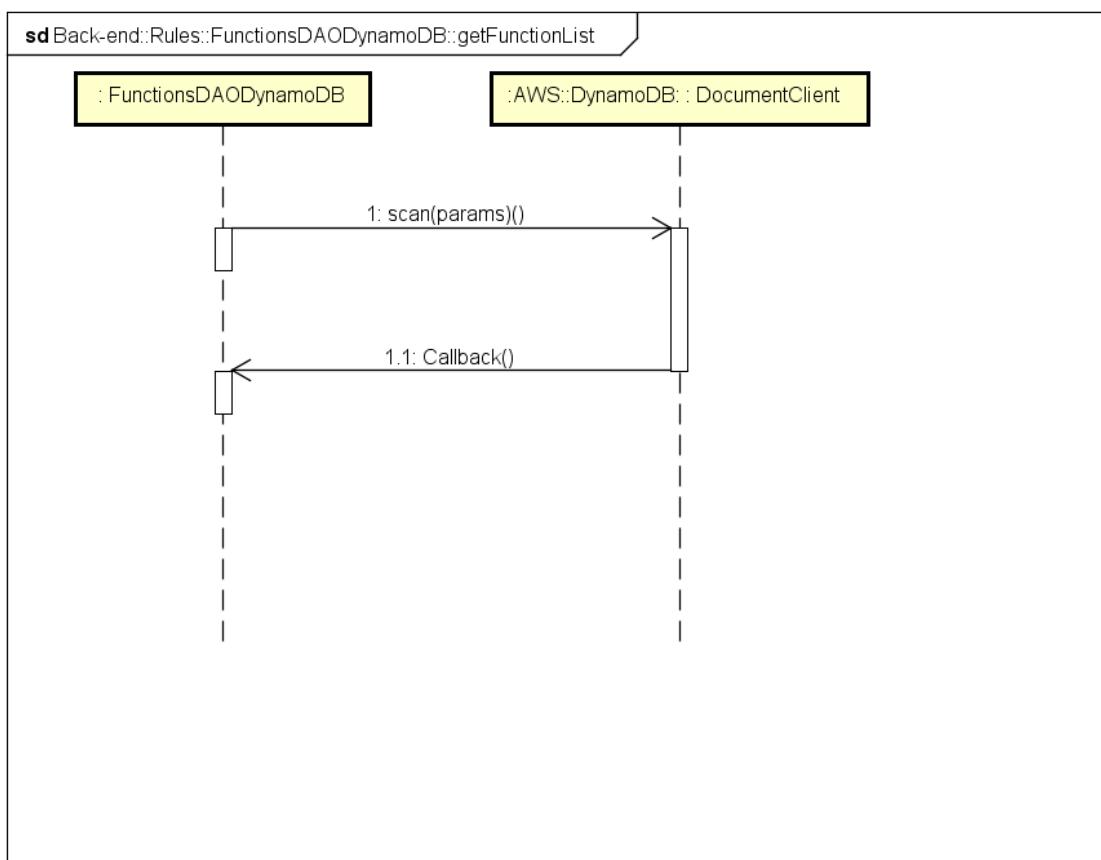


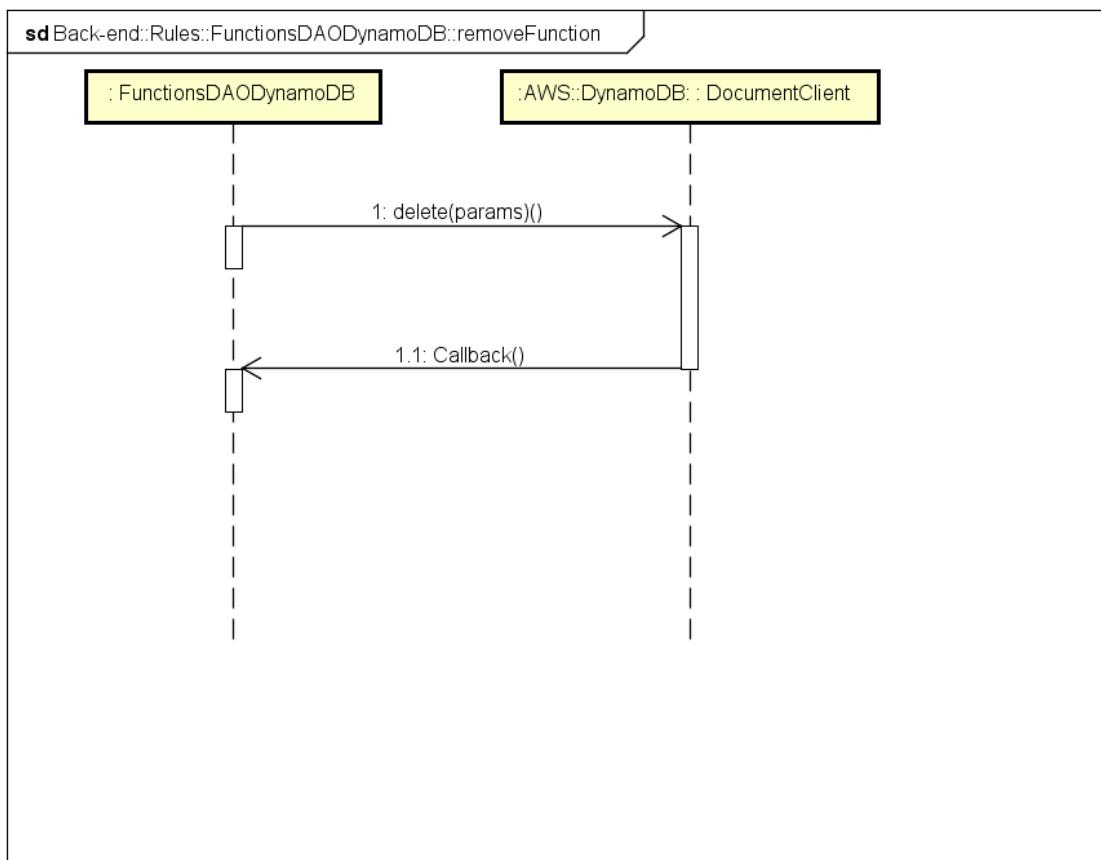


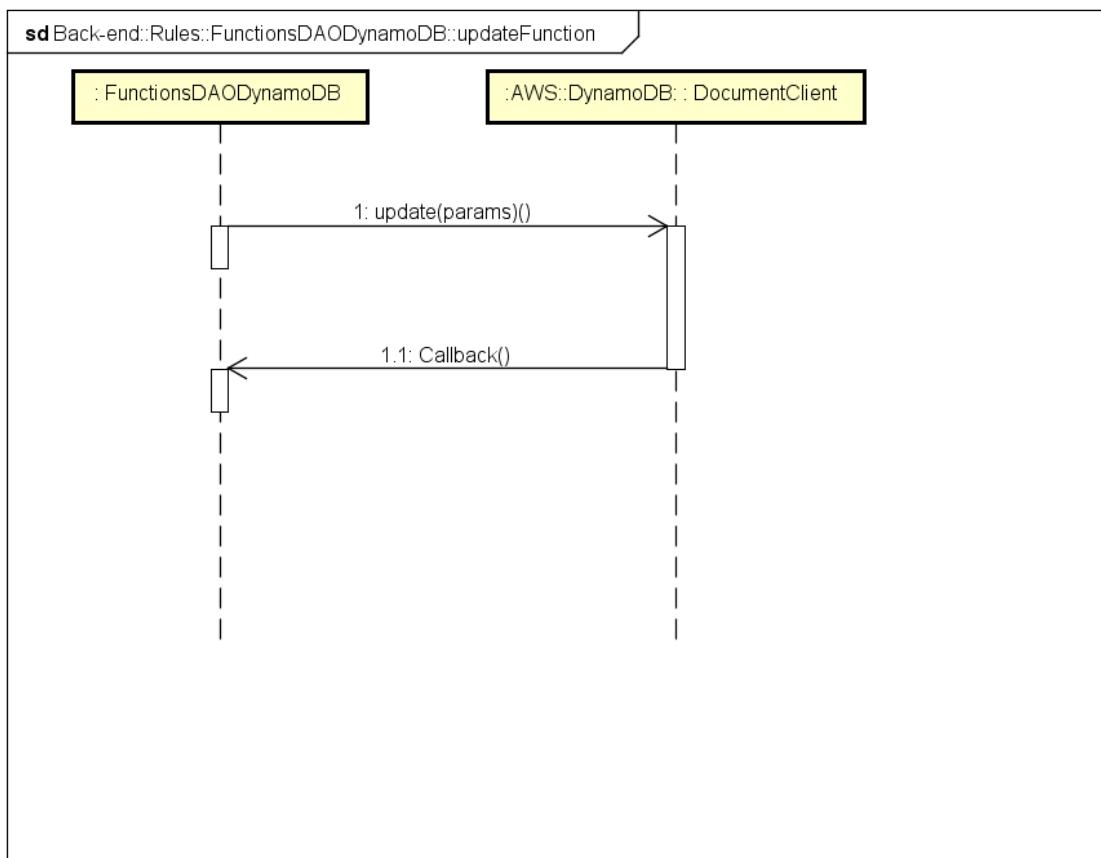


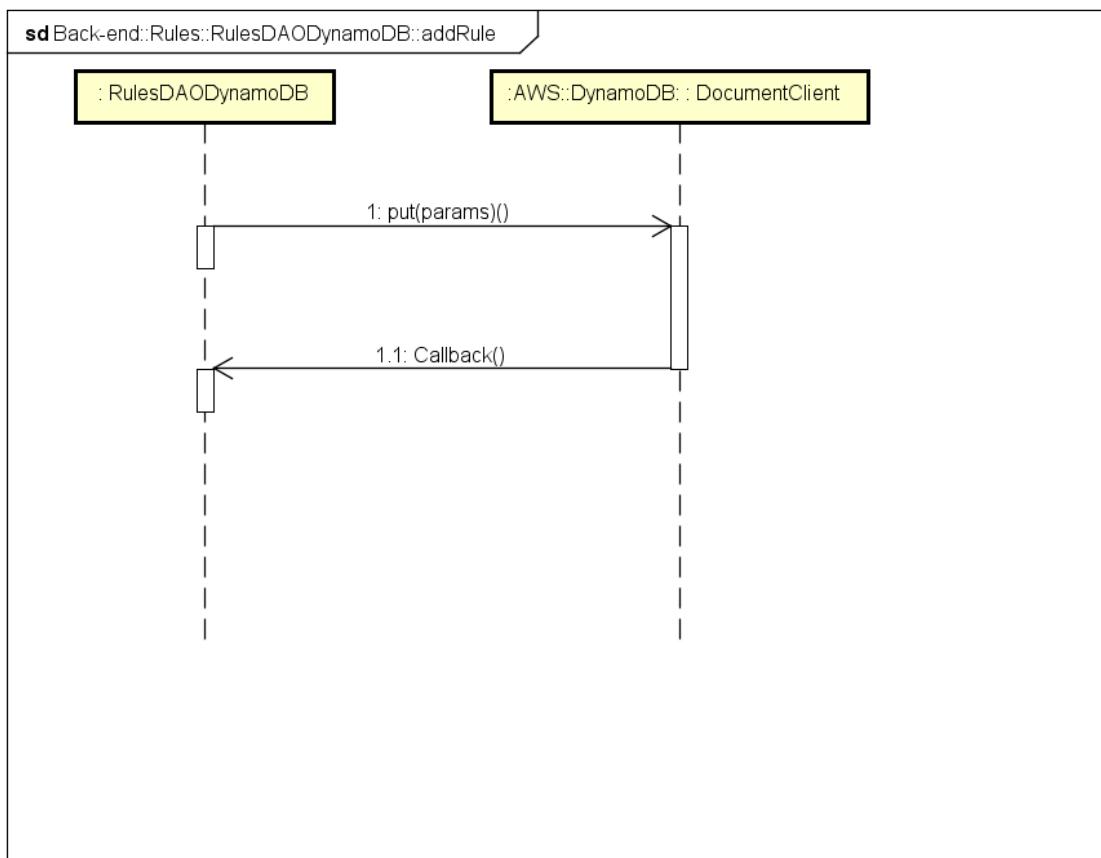


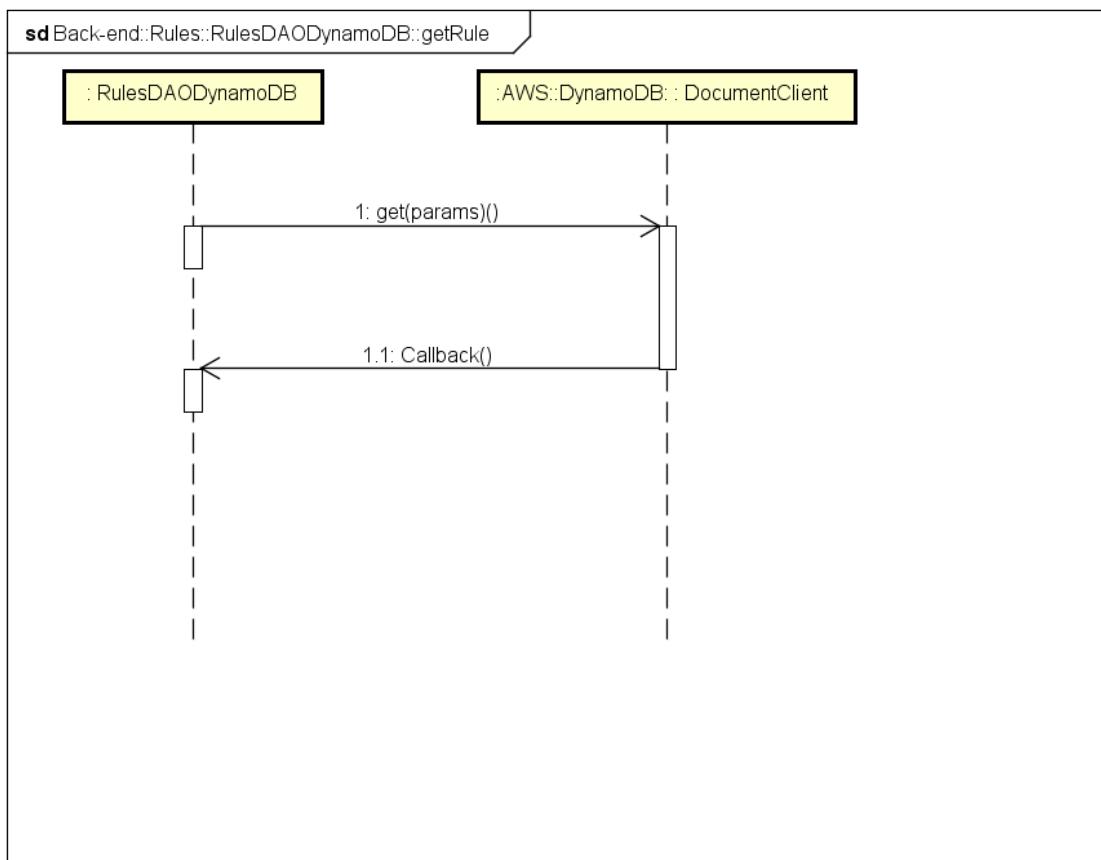


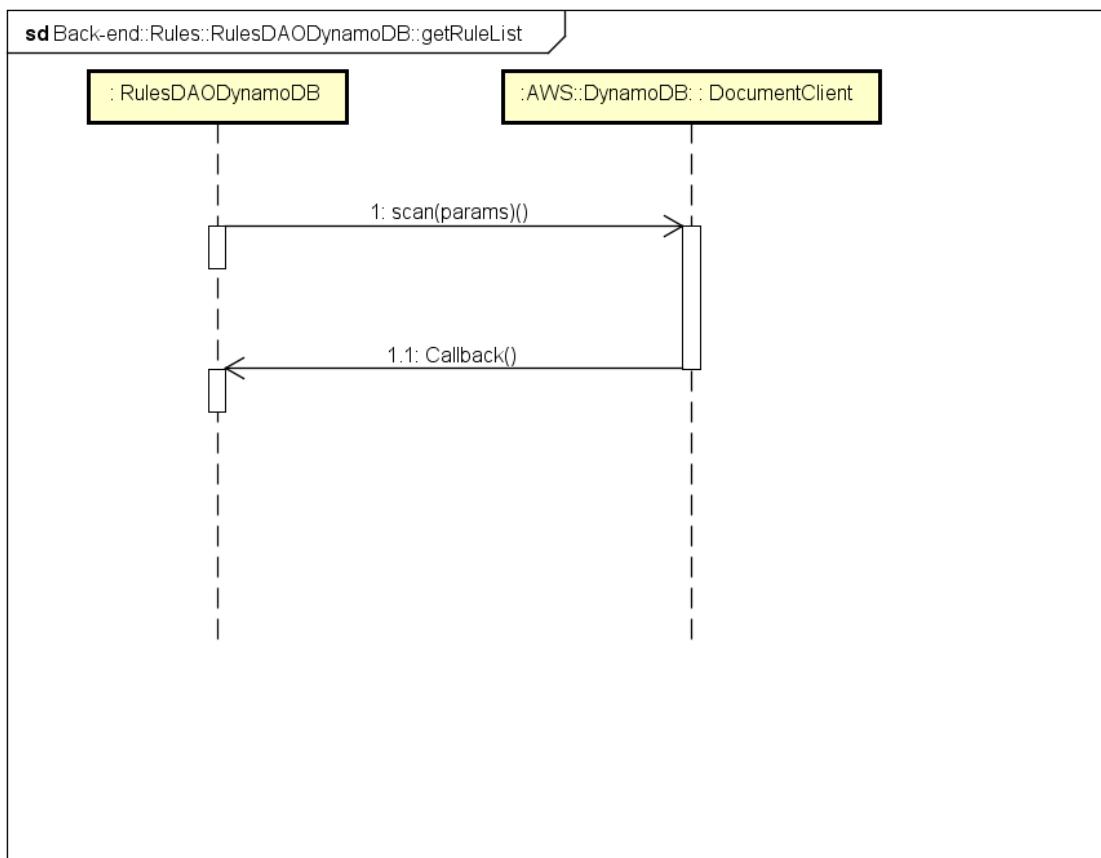


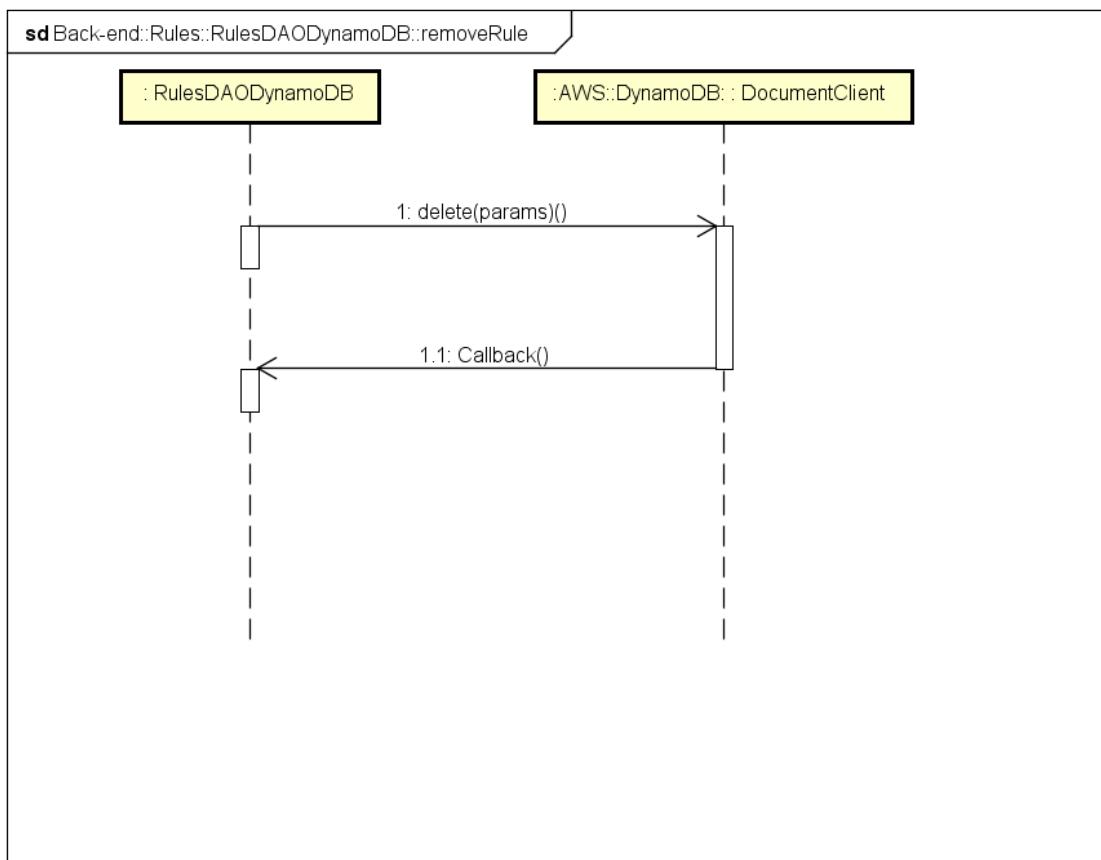


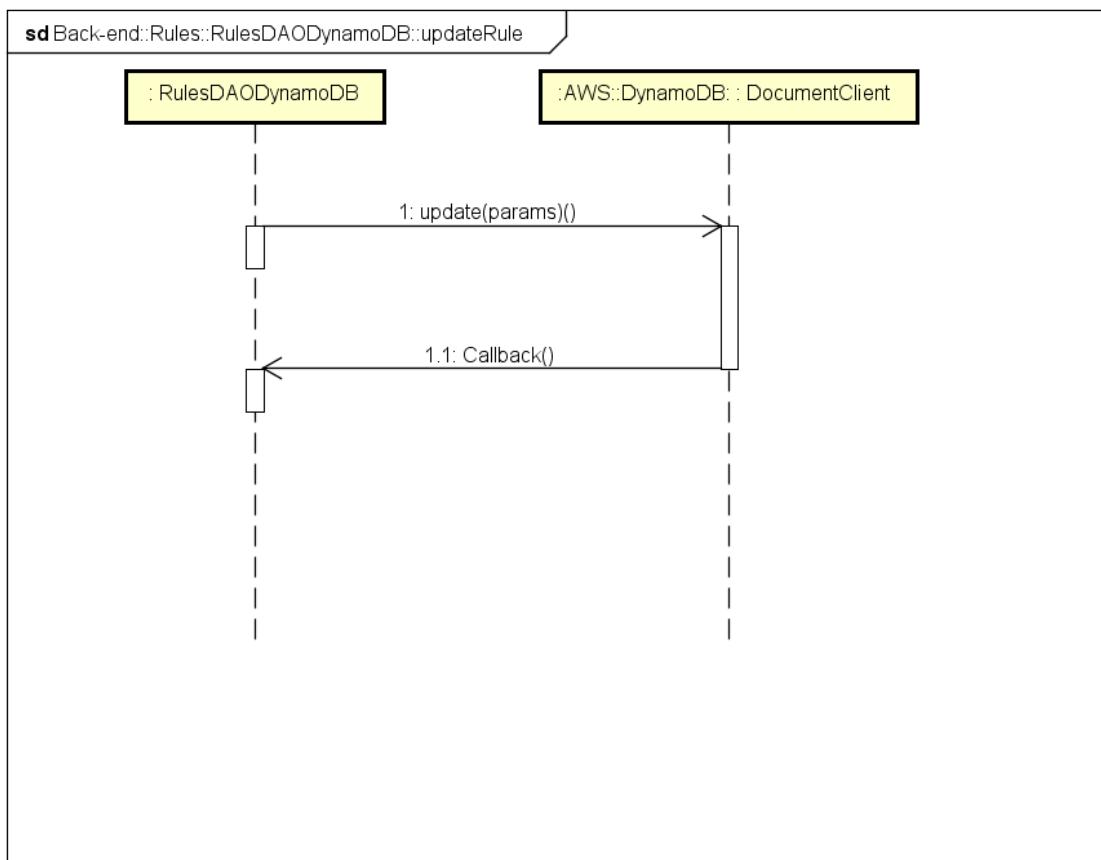


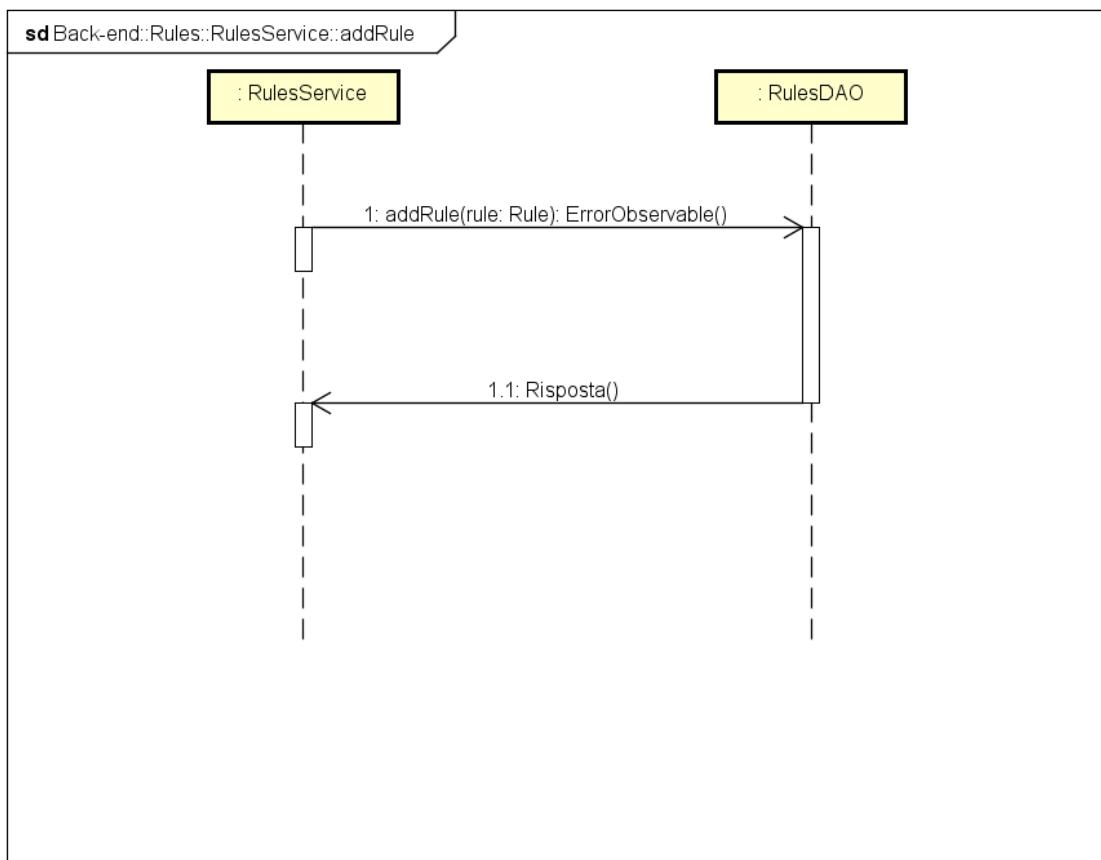


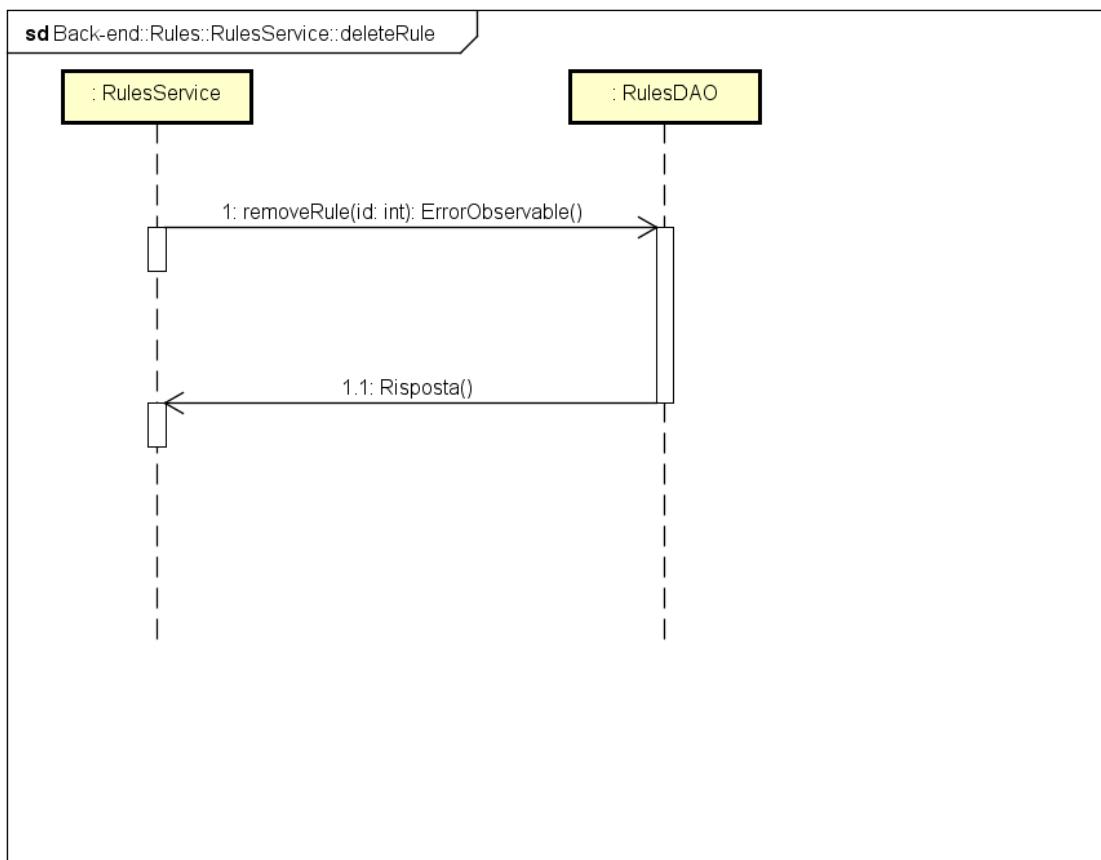


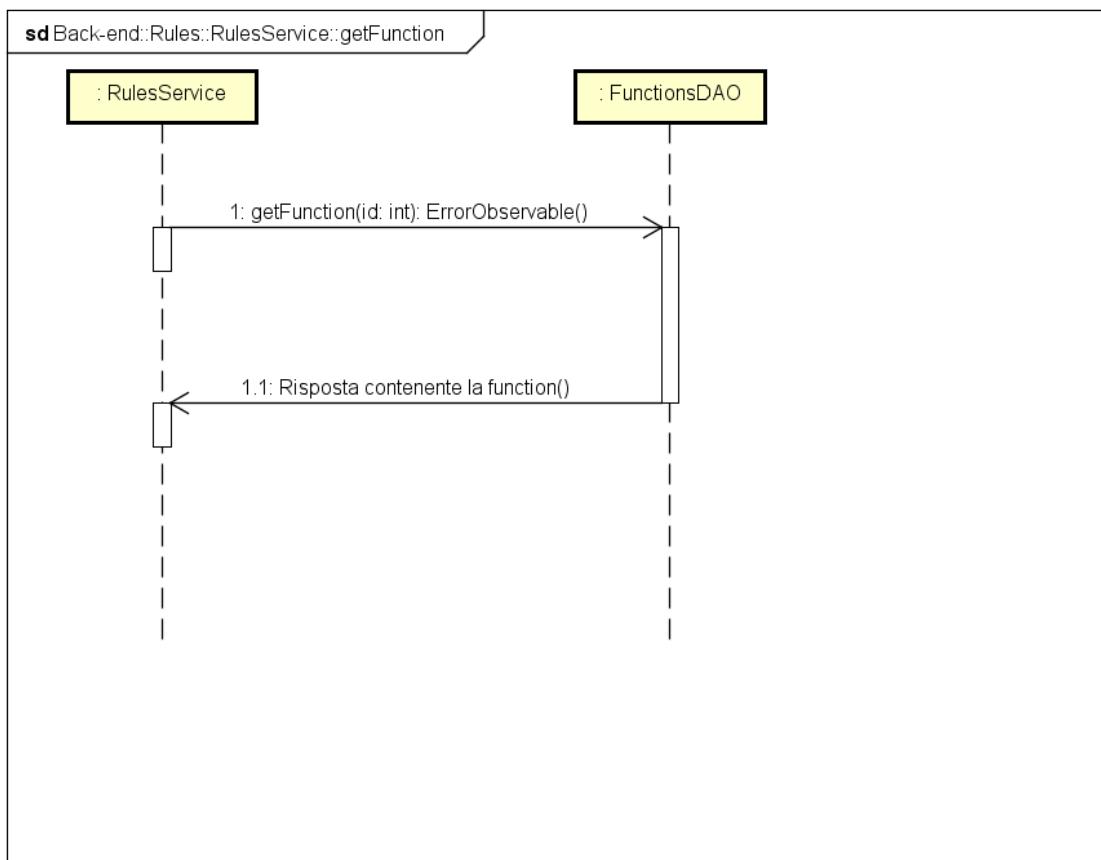


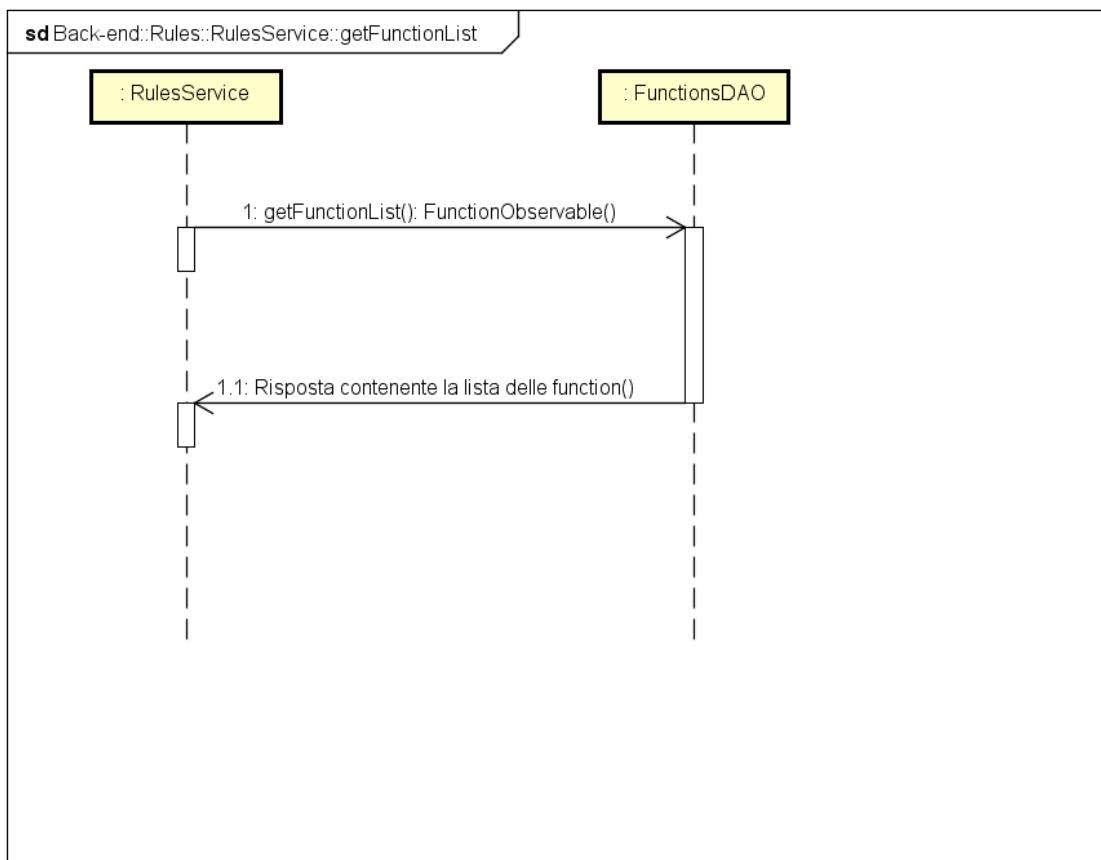


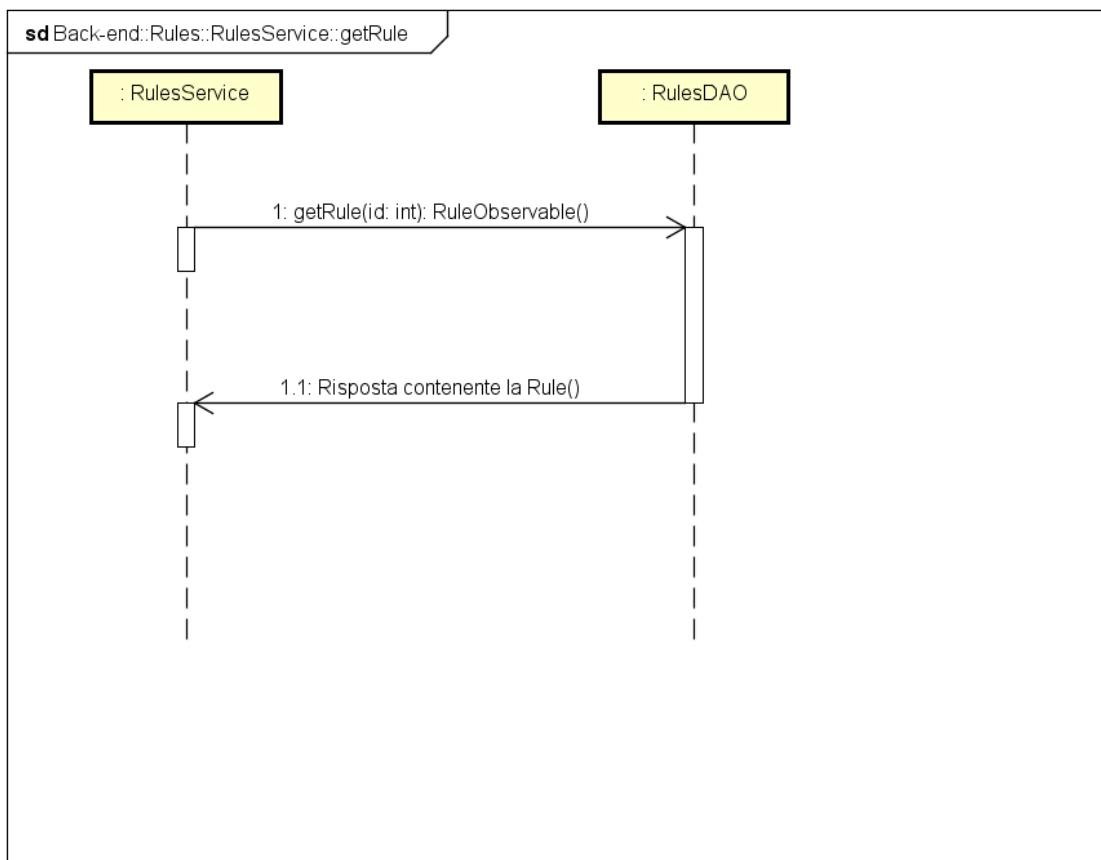


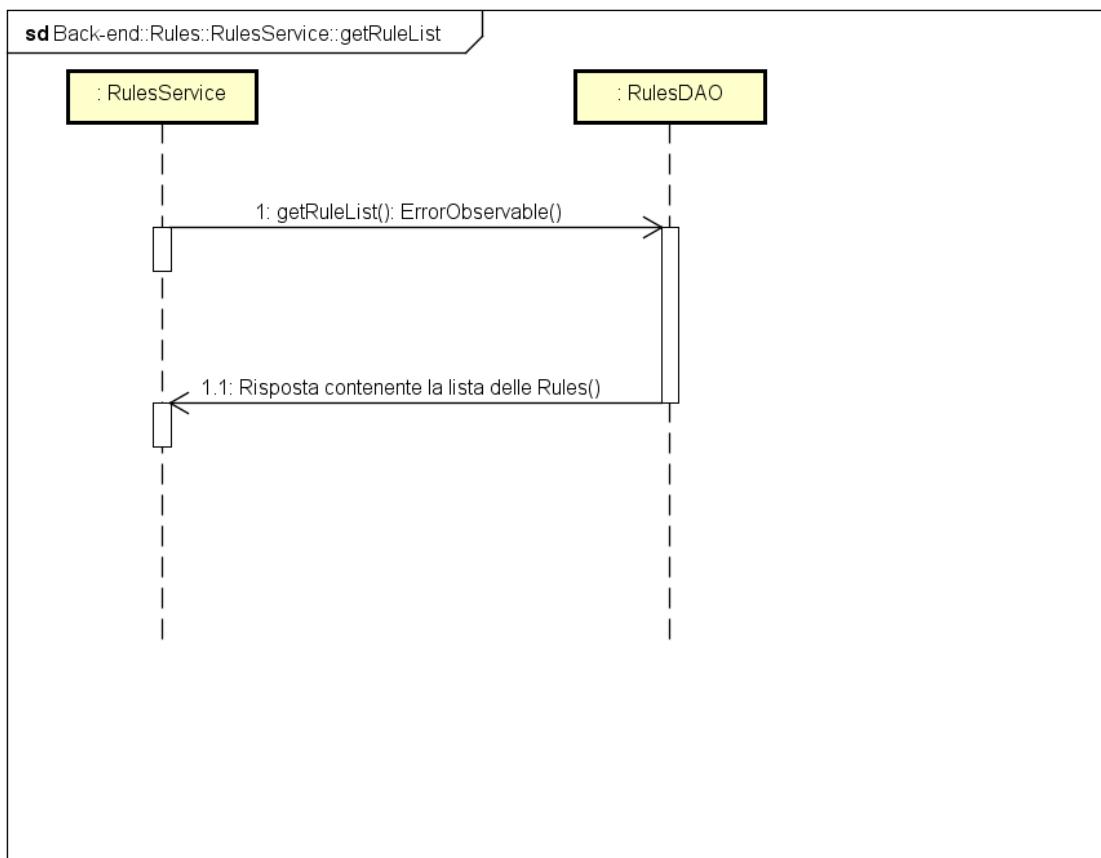


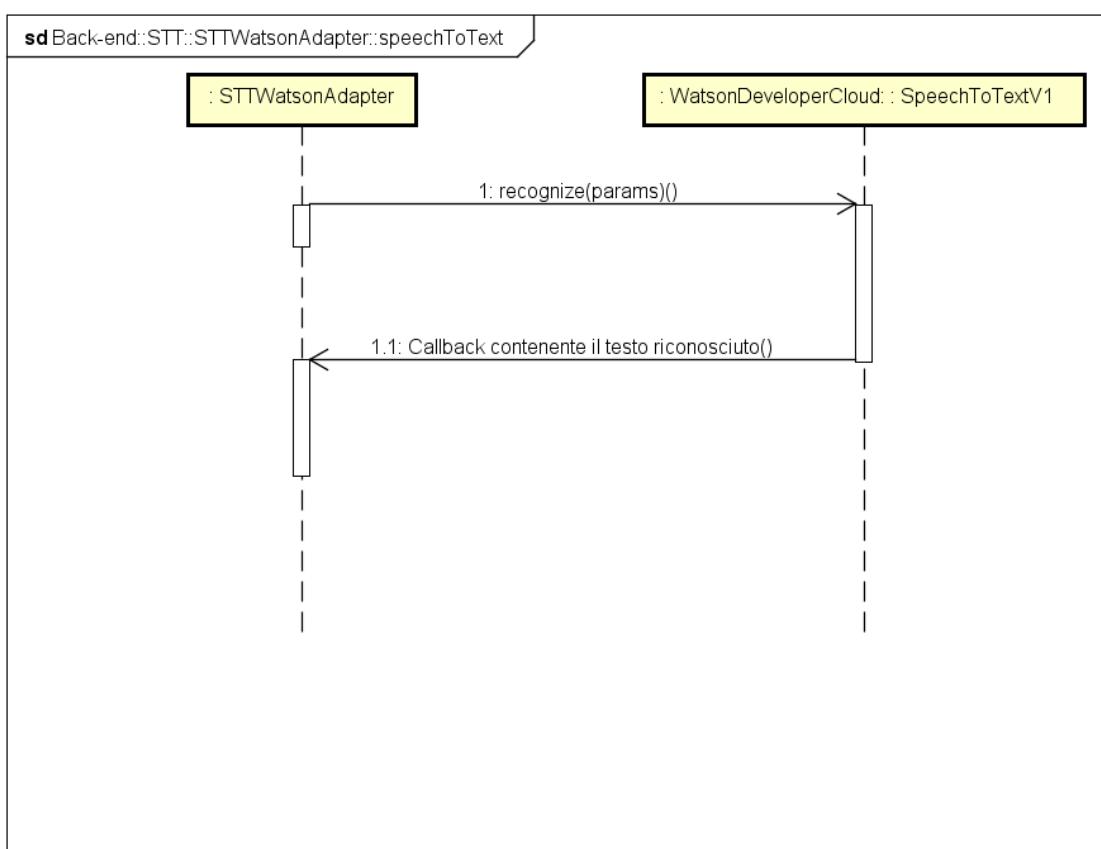
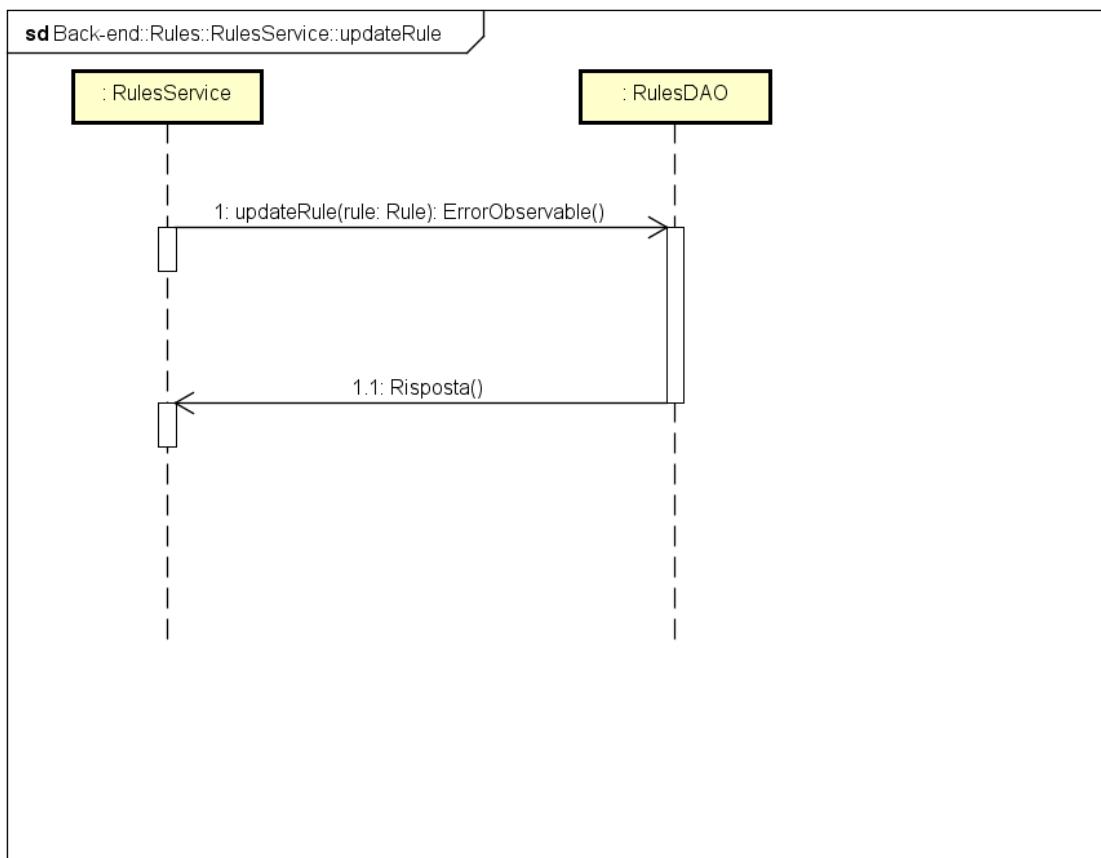












Specifiche delle Componenti

Back-end

Package contenente tutte le componenti che costituiscono il back-end. Le componenti sono organizzate secondo il pattern a microservizi.

Classi

ConversationsDAO

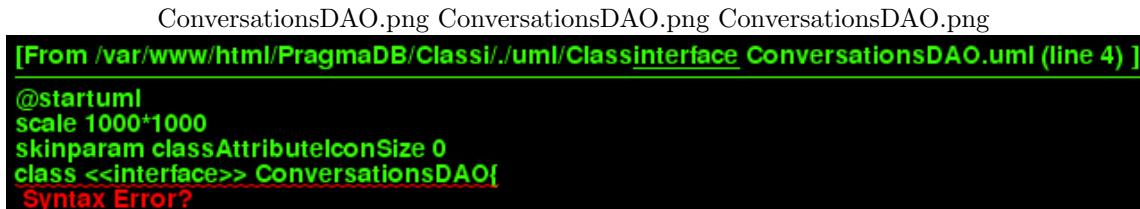


Figura 2: Back-end:: ConversationsDAO

- **Nome:** ConversationsDAO;
- **Tipo:** Interface;
- **Descrizione:** questa classe si occupa di astrarre le modalità d'interazione al database contenente le conversazioni;
- **Utilizzo:** fornisce a ConversationWebhookService un meccanismo per accedere al database contenente le conversazioni, senza conoscerne le modalità di implementazioni e di persistenza di quest'ultimo. Permette operazioni di lettura, scrittura e rimozione di utenti registrati;
- **Attributi:**
 - db: AWS::DynamoDB
Attributo che permette di contattare il database contenente le conversazioni;
 - guest_id: String
Attributo contenente l'id dell'ospite identificato;
- **Metodi:**
 - + getConversationList(): ConversationObservable
L'Observable restituito manderà agli Observer le conversazioni ottenute, uno alla volta, e poi chiama il loro metodo complete. Nel caso in cui si verifichi un errore, gli Observer iscritti verranno notificati tramite la chiamata del loro metodo error con i dati relativi all'errore verificatosi;
 - + getConversation(session_id: String, timestamp: String): ConversationObservable
Metodo che permette di ottenere una conversazione a partire dall'identificativo della sessione e dal suo timestamp.
L'Observable restituito riceverà l'oggetto rappresentante tale Conversation, e verrà completato. Nel caso in cui la conversazione richiesta non sia presente nel database, gli Observer interessati non riceveranno alcun valore, ma verranno notificati tramite la chiamata del loro metodo error();
Parametri:
* session_id: String
Parametro contenente l'id della sessione della conversazione da ricevere;

```

* timestamp: String
    Parametro contenente il timestamp relativo all'inizio della conversazione;

+ addConversation(conversation: Conversation): ErrorObservable
Metodo che permette di aggiungere una conversazione al database. L'Observable restituito non riceverà alcun valore, ma verrà completato in caso di aggiunta della conversazione avvenuta con successo. In caso di errore durante l'aggiunta della conversazione, gli Observer interessati verranno notificati tramite la chiamata del loro metodo error() con i dati relativi all'errore verificatosi;
Parametri:

* conversation: Conversation
    Parametro contenente la conversazione da inserire;

+ addMessage(msg: ConversationMessage, timestamp: String, session_id: String): ErrorObservable
Metodo che permette di aggiungere un messaggio relativo ad una conversazione al database a partire da un messaggio, un timestamp relativo e l'identificativo della relativa conversazione. L'Observable restituito non riceverà alcun valore, ma verrà completato in caso di aggiunta del messaggio avvenuta con successo. In caso di errore durante l'aggiunta del messaggio, gli Observer interessati verranno notificati tramite la chiamata del loro metodo error() con i dati relativi all'errore verificatosi;
Parametri:

* msg: ConversationMessage
    Parametro contenente il messaggio;

* timestamp: String
    Parametro contenente il timestamp relativo all'inizio della conversazione;

* session_id: String
    Parametro contenente l'id della sessione della conversazione alla quale aggiungere il messaggio;

+ removeConversation(session_id: String, timestamp: String): ErrorObservable
Metodo che permette di rimuovere una conversazione a partire dall'identificativo della sessione e dal suo timestamp. L'Observable restituito non riceverà alcun valore, ma verrà completato in caso di aggiunta della conversazione avvenuta con successo. In caso di errore durante l'aggiunta della conversazione, gli Observer interessati verranno notificati tramite la chiamata del loro metodo error() con i dati relativi all'errore verificatosi;
Parametri:

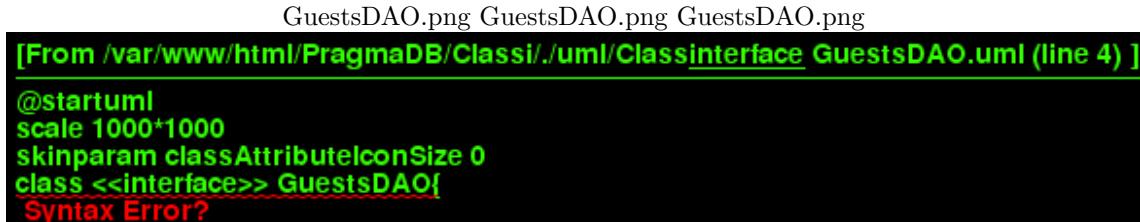
* session_id: String
    Parametro contenente l'id della sessione della conversazione da eliminare;

* timestamp: String
    Parametro contenente il timestamp del messaggio da eliminare;

```

GuestsDAO

- **Nome:** GuestsDAO;
- **Tipo:** Interface;
- **Descrizione:** questa interfaccia si occupa di astrarre le modalità d'interazione al database contenente gli ospiti conosciuti;
- **Utilizzo:** fornisce i metodi per accedere al database contenente i dati relativi agli ospiti conosciuti, senza conoscerne le modalità di implementazioni e di persistenza di quest'ultimo. Permette operazioni di lettura, scrittura e rimozione di ospiti conosciuti;

**Figura 3:** Back-end:: GuestsDAO

- Metodi:

+ `getGuestList(): GuestObservable`

L'`Observable` restituito manderà agli `Observer` gli ospiti ottenuti, uno alla volta, e poi chiama il loro metodo `complete`. Nel caso in cui si verifichi un errore, gli `Observer` iscritti verranno notificati tramite la chiamata del loro metodo `error` con i dati relativi all'errore verificatosi;

+ `getGuest(name: String, company: String): GuestObservable`

Metodo che permette di ottenere un ospite a partire dal nome e l'azienda di provenienza. L'`Observable` restituito riceverà l'oggetto rappresentante tale `Guest`, e verrà completato. Nel caso in cui l'ospite richiesto non sia presente nel database, gli `Observer` interessati non riceveranno alcun valore, ma verranno notificati tramite la chiamata del loro metodo `error()`;

Parametri:

* `name: String`

Parametro contenente il nome dell'ospite;

* `company: String`

Parametro contenente l'azienda di provenienza dell'ospite;

+ `updateGuest(guest: Guest): ErrorObservable`

Metodo che permette di aggiornare un ospite. L'`Observable` restituito non riceverà alcun valore, ma verrà completato in caso di aggiunta dell'ospite avvenuta con successo. In caso di errore durante l'aggiunta dell'ospite, gli `Observer` interessati verranno notificati tramite la chiamata del loro metodo `error()` con i dati relativi all'errore verificatosi;

Parametri:

* `guest: Guest`

Parametro contenente l'ospite da aggiornare;

+ `removeGuest(name: String, company: String): ErrorObservable`

Metodo che permette di eliminare un ospite dal database, a partire dal suo nome e azienda di provenienza. L'`Observable` restituito non riceverà alcun valore, ma verrà completato in caso di eliminazione dell'ospite avvenuta con successo. In caso di errore durante l'eliminazione dell'ospite, gli `Observer` interessati verranno notificati tramite la chiamata del loro metodo `error()` con i dati relativi all'errore verificatosi;

Parametri:

* `name: String`

Parametro contenente il nome dell'ospite;

* `company: String`

Parametro contenente l'azienda di provenienza dell'ospite;

+ `addGuest(guest: Guest): ErrorObservable`

Metodo che permette di aggiungere un ospite al database. L'`Observable` restituito non riceverà alcun valore, ma verrà completato in caso di aggiunta dell'ospite avvenuta con successo. In caso di errore durante l'aggiunta dell'ospite, gli `Observer` interessati

verranno notificati tramite la chiamata del loro metodo `error()` con i dati relativi all'errore verificatosi;

Parametri:

- * `guest: Guest`
Parametro contenente l'ospite da aggiungere;

AdministrationWebhookService

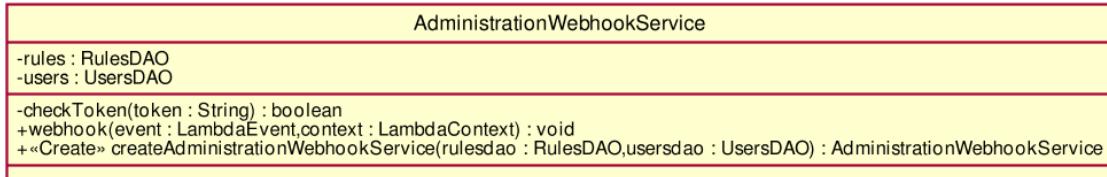


Figura 4: Back-end::AdministrationWebhookService

- **Nome:** `AdministrationWebhookService`;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di implementare l'interfaccia `WebhookService`, realizzando un Webhook che fornisce una risposta all'Agent di amministrazione;
- **Utilizzo:** fornisce il metodo che si occupa di rispondere alle chiamate al microservizio da parte dell'Agent di amministrazione;

- **Padre:** <>interface>> `WebhookService`;
- **Attributi:**

- `rules: RulesDAO`

Attributo che permette di contattare `RulesDAO`, il quale fornisce i meccanismi di accesso ai dati relativi alle direttive;

- `users: UsersDAO`

Attributo che permette di contattare `UsersDAO`, il quale fornisce i meccanismi di accesso ai dati relativi agli utenti registrati;

- `jwt: JSONWebTokenModule`

Attributo contenente un riferimento al modulo di Node.js per la gestione dei JSON Web Tokens, il quale permette di verificare la validità di un determinato token e di estrarre i dati presenti al suo interno. ;

- **Metodi:**

- `checkToken(token: String): boolean`

Metodo che permette di controllare l'autenticità di un JWT. Utilizzato dal metodo `webhook` per controllare che la richiesta sia stata fatta da un amministratore autenticato prima di compiere qualsiasi altra azione;

Parametri:

- * `token: String`

Parametro contenente il JWT per l'autenticazione;

- + `webhook(event: LambdaEvent, context: LambdaContext): void`

Questo metodo soddisfa i requisiti per webhook descritti da api.ai. Viene chiamato ad ogni interazione dell'agente di amministrazione, e per prima cosa si occupa di verificare che nella richiesta sia presente un JSON Web Token (JWT) che confermi un'autenticazione avvenuta con successo. In caso di mancata autenticazione, lo stato della

risposta sarà impostato a 403. Nel caso in cui il token sia presente e valido (la firma sia valida ed il token non sia scaduto), si occupa di eseguire l'operazione richiesta dall'assistente virtuale. Tale operazione è specificata in `fulfillment.messages` (vedi classi `ProcessingResult`, `Fullfillment` e `MsgObject` per eventuali chiarimenti riguardanti il formato della richiesta fatta da `api.ai` al microservizio). In particolare, all'interno di `messages` sarà presente un messaggio con `type=4`, il quale relativo attributo `payload` consisterà in un oggetto di tipo `WebhookCmd`. In base al campo `cmd` di tale oggetto questo metodo interpreterà il formato del campo `params`, ricavando i parametri necessari ad eseguire il comando specificato e poi eseguendo l'azione richiesta. In caso di successo verrà richiesto all'utente se vuole compiere qualche altra azione utilizzando il campo `speech` dell'oggetto `WebhookResponse` utilizzato per rispondere. In caso di mancanza del comando (nessun messaggio con `type=4` in `fulfillment.messages`), il campo `speech` della risposta sarà copiato da `fulfillment.speech` della richiesta, risultando quindi in una risposta uguale a quella fornita da `api.ai` in assenza di webhook;

Parametri:

```
* event: LambdaEvent
  Parametro contenente;
* context: LambdaContext
  Parametro contenente;
+ <<Create>> createAdministrationWebhookService(rulesdao: RulesDAO, usersdao: UsersDAO, jwt: JSONWebTokenModule): AdministrationWebhookService
  Costruttore che si occupa di effettuare la dependency injection di UsersDAO, JSONWebTokenModule, RulesDAO;
Parametri:
* rulesdao: RulesDAO
  Attributo contenente il RulesDAO;
* usersdao: UsersDAO
  Attributo contenente lo UsersDAO;
* jwt: JSONWebTokenModule
  Parametro contenente il riferimento al modulo Node.js per la gestione dei JSON Web Tokens;
```

AgentObservable

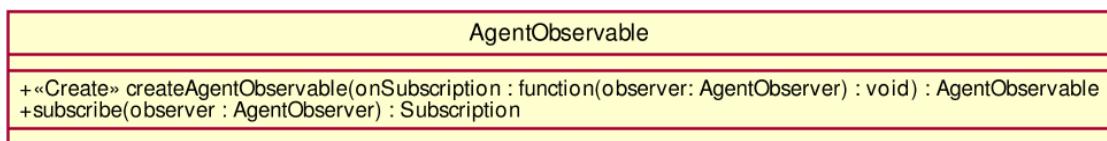


Figura 5: Back-end::AgentObservable

- **Nome:** AgentObservable;
- **Tipo:** Class;
- **Descrizione:** questa classe implementa un `Observable` che permette l'iscrizione di `AgentObserver`;
- **Utilizzo:** fornisce i meccanismi necessari per il passaggio di una serie di `Agent` ad un `Observer` interessato;
- **Padre:** `Observable`;
- **Metodi:**

```
+ <<Create>> createAgentObservable(onSubscription: function(observer: AgentObserver)
: void): AgentObservable
Constructor di AgentObservable;
Parametri:
* onSubscription: function(observer: AgentObserver) : void
Funzione che verrà eseguita quando un Observer si iscrive all'Observable. Si occupa di passare i dati all'Observer, chiamando il metodo next(agent: Agent).
Quando non ci sono più dati da restituire, si occupa di chiamare il metodo complete().
Nel caso in cui si verificasse un errore, si occupa di chiamare il metodo error(err: Object) con i dati relativi all'errore verificatosi;

+ subscribe(observer: AgentObserver): Subscription
Metodo che permette ad uno AgentObserver interessato di iscriversi a questo Observable;
Parametri:
* observer: AgentObserver
Observer che si vuole iscrivere;
```

AgentObserver

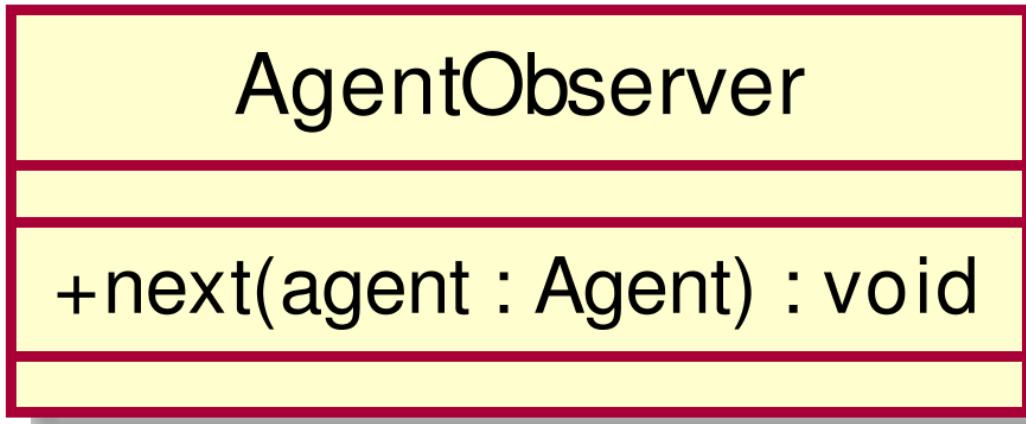


Figura 6: Back-end::AgentObserver

- **Nome:** AgentObserver;
- **Tipo:** Class;
- **Descrizione:** classe che rappresenta un Observer che si aspetta dati di tipo Agent;
- **Utilizzo:** implementa il metodo next() dell'interfaccia, in maniera tale che accetti dati di tipo Agent;
- **Metodi:**
 - + next(agent: Agent): void

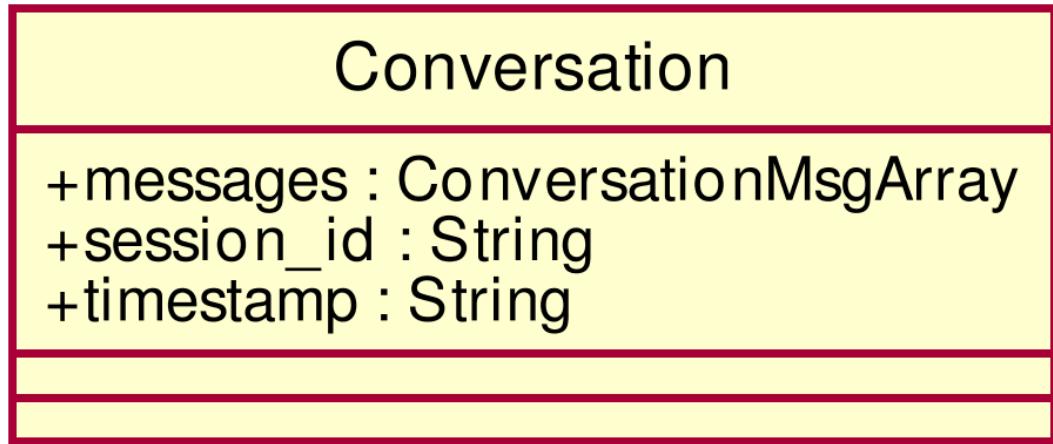
Metodo che permette agli Observable di notificare l'Observer con dati di tipo Agent.

Definisce inoltre le operazioni che l'Observer compierà all'arrivo di tali dati;

Parametri:

 - * agent: Agent

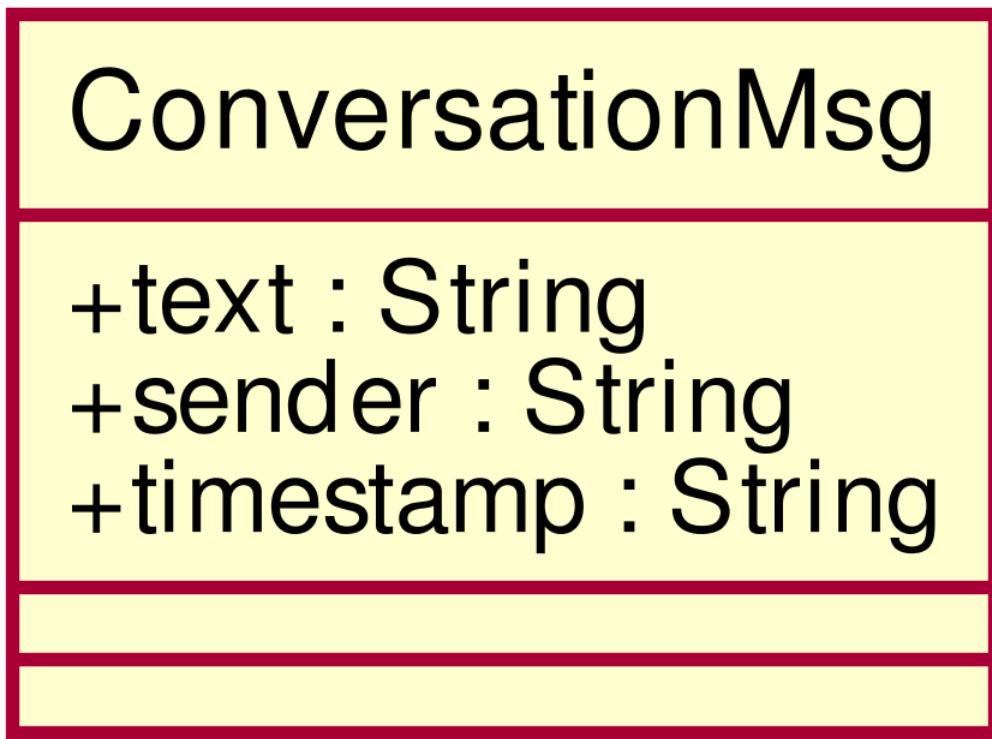
Parametro contenente l'Agent mandato dall'Observable;

Conversation**Figura 7:** Back-end::Conversation

- **Nome:** Conversation;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare e organizzare gli attributi relativi ad una conversazione, la quale dovrà essere salvata nel database;
- **Utilizzo:** fornisce gli attributi relativi ad una conversazione;
- **Attributi:**
 - + messages: ConversationMsgArray
Attributo contenente l'array dei messaggi contenuti in una conversazione;
 - + session_id: String
Attributo contenente l'id della sessione relativa alla conversazione;
 - + timestamp: String
Attributo contenente il timestamp relativo alla conversazione. Tutti i messaggi di una conversazione hanno lo stesso timestamp, il quale indica l'inizio della conversazione;

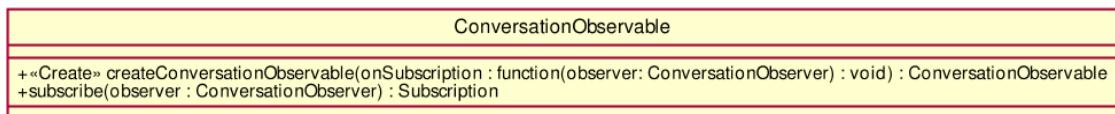
ConversationMsg

- **Nome:** ConversationMsg;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare e organizzare gli attributi relativi ad un messaggio all'interno di una conversazione;
- **Utilizzo:** fornisce gli attributi relativi un messaggio. La classe Conversation contiene un array di essi, in modo da costruire l'intera conversazione;
- **Attributi:**
 - + text: String
Attributo contenente il testo del messaggio;
 - + sender: String
Attributo contenente il mittente del messaggio;

**Figura 8:** Back-end::ConversationMsg

```
+ timestamp: String
Attributo contenente il timestamp del messaggio;
```

ConversationObservable

**Figura 9:** Back-end::ConversationObservable

- **Nome:** ConversationObservable;
- **Tipo:** Class;
- **Descrizione:** questa classe implementa un Observable che permette l'iscrizione di ConversationObserver;
- **Utilizzo:** fornisce i meccanismi necessari per il passaggio di una serie di Conversation ad un Observer interessato;
- **Padre:** Observable;
- **Metodi:**

```
+ <<Create>> createConversationObservable(onSubscription: function(observer: ConversationObserver) : void): ConversationObservable
Constructor di ConversationObservable;
Parametri:
* onSubscription: function(observer: ConversationObserver) : void
Funzione che verrà eseguita quando un Observer si iscrive all'Observable. Si occupa di passare i dati all'Observer, chiamando il metodo next(conversation: Conversation). Quando non ci sono più dati da restituire, si occupa di chiamare il metodo complete(). Nel caso in cui si verificasse un errore, si occupa di chiamare il metodo error(err: Object) con i dati relativi all'errore verificatosi;

+ subscribe(observer: ConversationObserver): Subscription
Metodo che permette ad uno ConversationObserver interessato di iscriversi a questo Observable;
Parametri:
* observer: ConversationObserver
Observer che si vuole iscrivere;
```

ConversationObserver

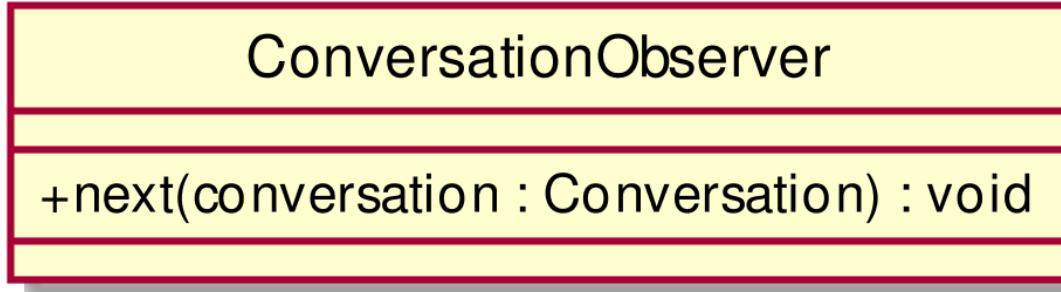


Figura 10: Back-end::ConversationObserver

- **Nome:** ConversationObserver;
- **Tipo:** Class;
- **Descrizione:** classe che rappresenta un Observer che si aspetta dati di tipo Conversation. ;
- **Utilizzo:** implementa il metodo next() dell'interfaccia, in maniera tale che accetti dati di tipo Conversation;
- **Metodi:**
 - + next(conversation: Conversation): void
Metodo che permette agli Observable di notificare l'Observer con dati di tipo Conversation. Definisce inoltre le operazioni che l'Observer compierà all'arrivo di tali dati;
Parametri:
* conversation: Conversation
Parametro contenente la Conversation mandata dall'Observable;

ConversationsDAO^{DynamoDB}

- **Nome:** ConversationsDAO^{DynamoDB};

ConversationsDAODynamoDB
-db : AWS::DynamoDB
+addConversation(conversation : Conversation) : ErrorObservable
+getConversation(session_id : String, timestamp : String) : ConversationObservable
+getConversationList() : ConversationObservable
+removeConversation(session_id : String, timestamp : String) : ErrorObservable
+<> createConversationsDAODynamoDB(guest_id : String, db : AWS::DynamoDB) : ConversationsDAODynamoDB
+addMessage(msg : ConversationMsg, timestamp : String, session_id : String) : ErrorObservable

Figura 11: Back-end::ConversationsDAODynamoDB

- **Tipo:** Class;

- **Descrizione:** classe che si occupa di implementare l’interfaccia `ConversationsDAO`, utilizzando un database DynamoDB come supporto per la memorizzazione dei dati;

- **Utilizzo:** implementa i metodi dell’interfaccia `ConversationsDAO` interrogando un database DynamoDB. Utilizza `AWS::DynamoDB::DocumentClient` per l’accesso al database. La dependency injection dell’oggetto `AWS::DynamoDB` viene fatta utilizzando il costruttore;

- **Attributi:**

- db: `AWS::DynamoDB`

Attributo contenente un riferimento al modulo di Node.js utilizzato per l’accesso al database DynamoDB contenente la tabella degli utenti;

- **Metodi:**

- + `addConversation(conversation : Conversation) : ErrorObservable`

Implementazione del metodo definito nell’interfaccia `ConversationsDAO`. Utilizza il metodo `put` del `DocumentClient` per aggiungere la conversazione al database;
Parametri:

- * `conversation: Conversation`

Parametro contenente la conversazione da inserire;

- + `getConversation(session_id: String, timestamp: String) : ConversationObservable`

Implementazione del metodo definito nell’interfaccia `ConversationsDAO`. Utilizza il metodo `get` del `DocumentClient` per ottenere i dati relativi ad una `Conversation` dal database;
Parametri:

- * `session_id: String`

Parametro contenente l’id della sessione della conversazione da ricevere;

- * `timestamp: String`

Parametro contenente il timestamp relativo all’inizio della conversazione;

- + `getConversationList() : ConversationObservable`

Implementazione del metodo dell’interfaccia `ConversationsDAO`. Utilizza il metodo `scan` del `DocumentClient` per ottenere la lista delle conversazioni dal database;

- + `removeConversation(session_id: String, timestamp: String) : ErrorObservable`

Implementazione del metodo dell’interfaccia `ConversationsDAO`. Utilizza il metodo `delete` del `DocumentClient` per eliminare una conversazione dal database;
Parametri:

- * `session_id: String`

Parametro contenente l’id della sessione della conversazione da eliminare;

```

* timestamp: String
    Parametro contenente il timestamp relativo all'inizio della conversazione;

+ <> createConversationsDAO(DynamoDB(guest_id: String, db: AWS::DynamoDB):
ConversationsDAO)
Constructor della classe ConversationsDAO. Permette di effettuare la dependency injection di AWS::DynamoDB;
Parametri:
* guest_id: String
sdadnoadnaosd;

* db: AWS::DynamoDB
Parametro contenente un riferimento al modulo di Node.js da utilizzare per l'accesso
al database DynamoDB contenente la tabella delle conversazioni;

+ addMessage(msg: ConversationMsg, timestamp: String, session_id: String): ErrorObservable
Implementazione del metodo definito nell'interfaccia ConversationsDAO. Utilizza il metodo put del DocumentClient per aggiungere un messaggio ad una conversazione;
Parametri:
* msg: ConversationMsg
Parametro contenente il messaggio. ;

* timestamp: String
Parametro contenente il timestamp relativo all'inizio della conversazione;

* session_id: String
Parametro contenente l'id della sessione della conversazione dove aggiungere il
messaggio;

```

ConversationWebhookService

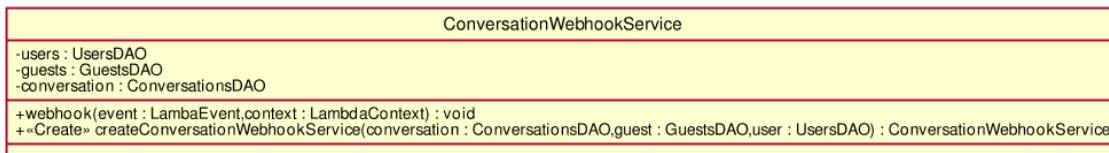


Figura 12: Back-end::ConversationWebhookService

- **Nome:** ConversationWebhookService;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di implementare l'interfaccia WebhookService, implementando un Webhook che fornisce una risposta ad api.ai;
- **Utilizzo:** fornisce il metodo che si occupa di rispondere alle chiamate al microservizio da parte di api.ai;
- **Padre:** <<interface>> WebhookService;
- **Attributi:**
 - users: UsersDAO
Attributo che permette di contattare UsersDAO, il quale fornisce i meccanismi di accesso al database contenente gli utenti registrati;

- **guests:** GuestsDAO

Attributo che permette di contattare GuestDAO, il quale fornisce i meccanismi di accesso al database contenente gli ospiti;

- **conversation:** ConversationsDAO

Attributo che permette di contattare ConversationDAO, il quale fornisce i meccanismi di accesso al database delle conversazioni;

- **Metodi:**

+ **webhook(event: LambdaEvent, context: LambdaContext): void**

Questo metodo soddisfa i requisiti per webhook descritti da api.ai. Si occupa di cercare nei database la presenza di interazioni passate con la persona con cui sta avvenendo l'interazione, e di verificare se la persona in questione può essere un amministratore del sistema. Nel caso di interazioni passate il context accoglienza viene riempito con azienda e dati della persona con cui l'ospite ha avuto il maggior numero di incontri in passato, e viene chiesto di confermare se la persona indicata è quella a cui l'ospite è effettivamente interessato. Nel caso di amministratore viene impostato il context admin, in modo che api.ai riconosca l'utente come un potenziale amministratore e gli chieda se vuole entrare nell'area di amministrazione;

Parametri:

* **event: LambdaEvent**

Parametro contenente, all'interno del campo **body** sotto forma di stringa in formato JSON, un oggetto contenente tutti i dati relativi ad una richiesta di api.ai al **ConversationWebhookService**. Tali dati sono:

```

1 {
2   "id": "String",
3   "lang": "String",
4   "originalRequest": "Object",
5   "result": "ProcessingResult",
6   "sessionId": "String",
7   "status": "StatusObject",
8   "timestamp": "String"
9 }
```

Per la relativa documentazione, consultare la pagina <https://docs.api.ai/docs/query#response>;

* **context: LambdaContext**

Parametro utilizzato dal webhook per inviare la risposta. La risposta, contenuta nel **LambdaResponse** parametro del metodo **LambdaContext::succeed**, possiede un attributo **body**, il quale conterrà il corpo di essa sotto forma di una stringa in formato JSON.

La risposta può essere una tra i seguenti tipi:

- l'utente viene riconosciuto come un potenziale amministratore;
- l'utente viene riconosciuto come un ospite conosciuto.

Nel primo caso, la risposta fornita sarà così organizzata:

```

1 {
2   "contexts": [
3     {
4       "name": "String",
5       "first\_name": "String",
6       "last\_name": "String",
7       "username": "String"
8     }
9 }
```

Dove

- `name` indica il nome del context, che in questo caso sarà "admin";
- `first_name` indica il nome dell'amministratore;
- `last_name` indica il cognome dell'amministratore;
- `username` indica lo username dell'amministratore;

Nel secondo caso, la risposta sarà così organizzata:

```

1 {
2     "contexts": [
3         "name": "String",
4         "name\_guest": "String",
5         "company": "String",
6         "first\_name": "String",
7         "last\_name": "String"
8     ]
9 }
```

Dove

- `name` indica il nome del context, che in questo caso sarà "welcome";

- `name_guest` indica il nome dell'ospite;

- `company` indica l'azienda di provenienza dell'ospite;

- `username` indica lo username dell'amministratore;

- `first_name` indica il nome della persona desiderata;

- `last_name` indica il cognome della persona desiderata;

;

+ <<Create>> createConversationWebhookService(conversation: ConversationsDAO, guest: GuestsDAO, user: UsersDAO): ConversationWebhookService

Metodo che permette la costruzione di un `ConversationWebhookService`. Permette la dependency injection che ha come oggetti un `ConversationsDAO`, `GuestsDAO` e `UsersDAO`;

Parametri:

* `conversation: ConversationsDAO`
Attributo contenente il `ConversationsDAO`;

* `guest: GuestsDAO`
Attributo contenente il `GuestsDAO`;

* `user: UsersDAO`
Attributo contenente lo `UsersDAO`;

ErrorObservable

- **Nome:** ErrorObservable;
- **Tipo:** Class;
- **Descrizione:** ????
- **Utilizzo:** ????
- **Padre:** Observable;

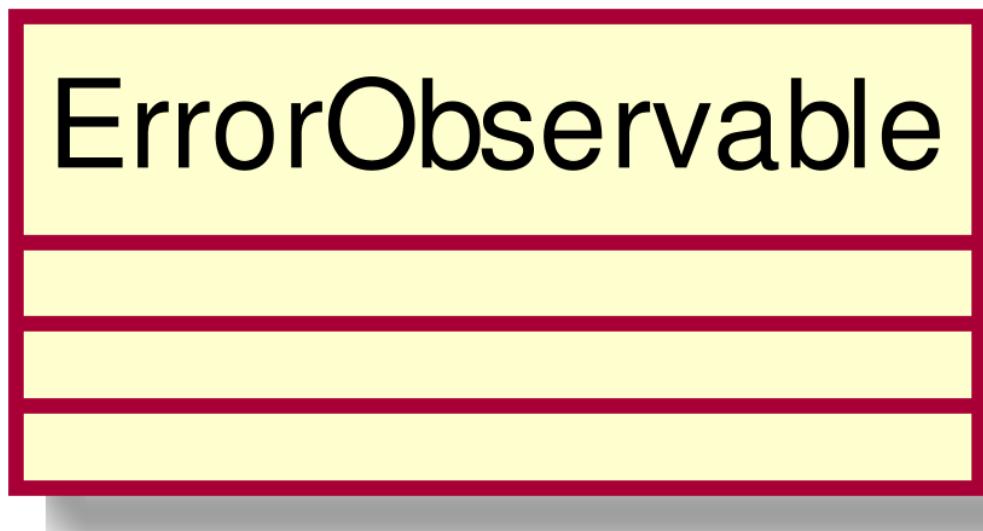


Figura 13: Back-end::ErrorObservable

ErrorObserver

- **Nome:** ErrorObserver;
- **Tipo:** Class;
- **Descrizione:** ????;
- **Utilizzo:** ????;

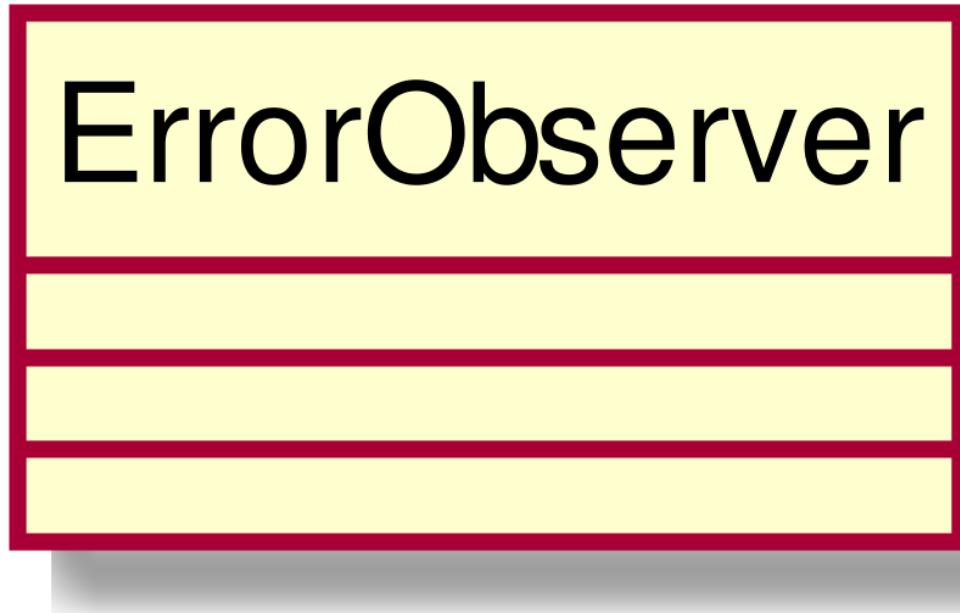
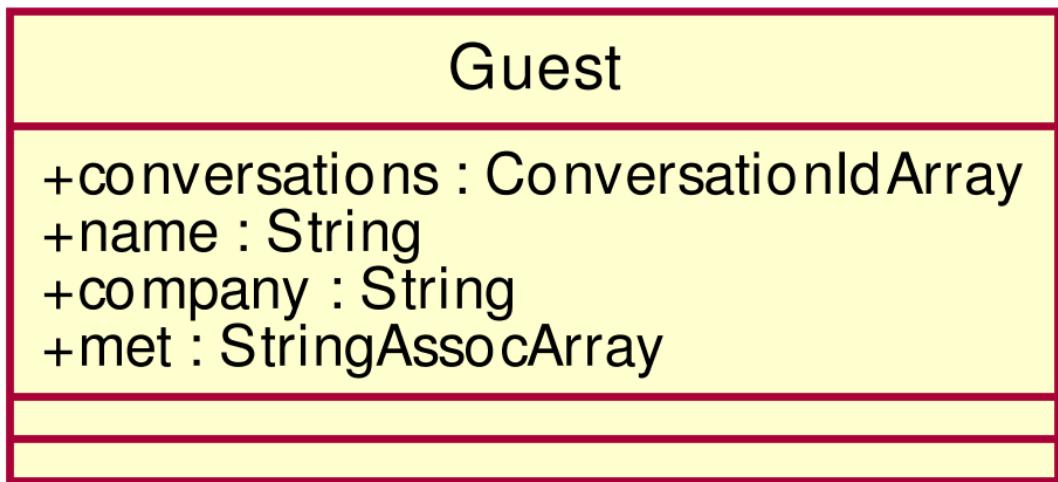
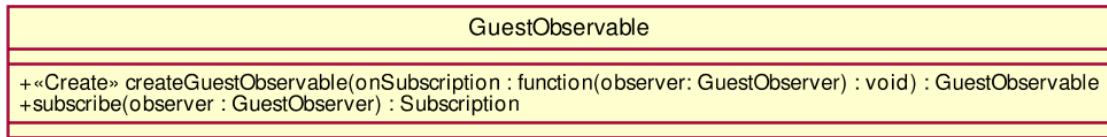


Figura 14: Back-end::ErrorObserver

Guest

- **Nome:** Guest;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare e organizzare gli attributi relativi ad un ospite conosciuto, i quali dovranno essere salvati nel database;
- **Utilizzo:** fornisce gli attributi relativi ad un ospite;
- **Attributi:**
 - + conversations: ConversationIdArray
Attributo contenente l'array delle conversazioni avute con l'ospite;
 - + name: String
Attributo contenente il nome dell'ospite;
 - + company: String
Attributo contenente l'azienda di provenienza dell'ospite;
 - + met: StringAssocArray
Attributo contenente l'array associativo del numero di volte che una persona è stata accolta. La chiave di questo array associativo è il nome dell'ospite;

**Figura 15:** Back-end::Guest**GuestObservable****Figura 16:** Back-end::GuestObservable

- **Nome:** GuestObservable;
- **Tipo:** Class;
- **Descrizione:** questa classe implementa un Observable che permette l'iscrizione di GuestObserver;
- **Utilizzo:** fornisce i meccanismi necessari per il passaggio di una serie di Guest ad un Observer interessato;
- **Padre:** Observable;
- **Metodi:**
 - + <<Create>> createGuestObservable(onSubscription: function(observer: GuestObserver) : void) : GuestObservable
Constructor di GuestObservable;
Parametri:
 - * onSubscription: function(observer: GuestObserver) : void
Funzione che verrà eseguita quando un Observer si iscrive all'Observable. Si occupa di passare i dati all'Observer, chiamando il metodo next(guest: Guest). Quando non ci sono più dati da restituire, si occupa di chiamare il metodo complete(). Nel caso in cui si verificasse un errore, si occupa di chiamare il metodo error(err: Object) con i dati relativi all'errore verificatosi;
 - + subscribe(observer: GuestObserver) : Subscription
Metodo che permette ad un GuestObserver interessato di iscriversi a questo Observable;
Parametri:

* observer: GuestObserver
 Observer che si vuole iscrivere;

GuestObserver

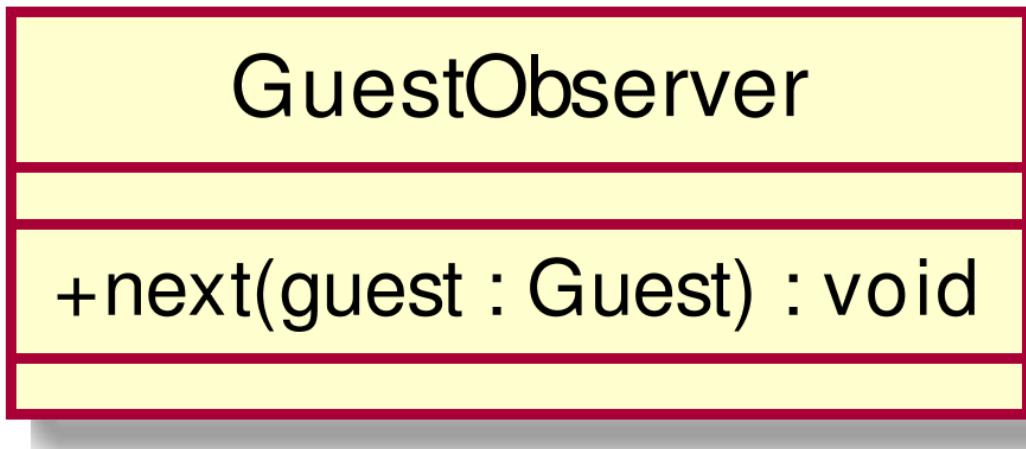


Figura 17: Back-end::GuestObserver

- **Nome:** GuestObserver;
- **Tipo:** Class;
- **Descrizione:** classe che rappresenta un Observer che si aspetta dati di tipo Guest;
- **Utilizzo:** implementa il metodo `next()` dell’interfaccia, in maniera tale che accetti dati di tipo Guest;
- **Metodi:**

+ next(guest: Guest): void
 Metodo che permette agli Observable di notificare l’Observer con dati di tipo Guest.
 Definisce inoltre le operazioni che l’Observer compierà all’arrivo di tali dati;
 Parametri:

* guest: Guest
 Parametro contenente il Guest mandato dall’Observable;

GuestsDAODynamoDB

- **Nome:** GuestsDAODynamoDB;
- **Tipo:** Class;
- **Descrizione:** classe che si occupa di implementare l’interfaccia GuestsDAO, utilizzando un database DynamoDB come supporto per la memorizzazione dei dati;
- **Utilizzo:** implementa i metodi dell’interfaccia GuestsDAO interrogando un database DynamoDB. Utilizza AWS::DynamoDB::DocumentClient per l’accesso al database. La dependency injection dell’oggetto AWS::DynamoDB viene fatta utilizzando il costruttore;
- **Attributi:**

GuestsDAODynamoDB
-db : AWS::DynamoDB
+addGuest(guest : Guest) : ErrorObservable
+getGuest(name : String, company : String) : GuestObservable
+getGuestList() : GuestObservable
+removeGuest(name : String, company : String) : ErrorObservable
+updateGuest(guest : Guest) : ErrorObservable
+<> createGuestsDAODynamoDB(db : AWS::DynamoDB) : GuestsDAODynamoDB

Figura 18: Back-end::GuestsDAODynamoDB

- db: AWS::DynamoDB

Attributo contenente un riferimento al modulo di Node.js utilizzato per l'accesso al database DynamoDB contenente la tabella degli utenti;

- **Metodi:**

+ addGuest(guest: Guest): ErrorObservable

Implementazione del metodo definito nell'interfaccia GuestsDAO. Utilizza il metodo put del DocumentClient per aggiungere l'ospite al database;

Parametri:

* guest: Guest

Parametro contenente l'ospite da aggiungere;

+ getGuest(name: String, company: String): GuestObservable

Implementazione del metodo definito nell'interfaccia GuestsDAO. Utilizza il metodo get del DocumentClient per ottenere i dati relativi ad un Guest dal database;

Parametri:

* name: String

Parametro contenente il nome dell'ospite;

* company: String

Parametro contenente l'azienda di provenienza dell'ospite;

+ getGuestList(): GuestObservable

Implementazione del metodo dell'interfaccia GuestsDAO. Utilizza il metodo scan del DocumentClient per ottenere la lista degli ospiti dal database;

+ removeGuest(name: String, company: String): ErrorObservable

Implementazione del metodo dell'interfaccia GuestsDAO. Utilizza il metodo delete del DocumentClient per eliminare un ospite dal database;

Parametri:

* name: String

Parametro contenente il nome dell'ospite;

* company: String

Parametro contenente l'azienda di provenienza dell'ospite;

+ updateGuest(guest: Guest): ErrorObservable

Implementazione del metodo dell'interfaccia GuestsDAO. Utilizza il metodo update del DocumentClient per aggiornare i dati relativi ad un ospite presenti all'interno del database;

Parametri:

```
* guest: Guest  
Parametro contenente l'ospite da aggiornare;  
  
+ <> createGuestsDAO(DynamoDB db: AWS::DynamoDB): GuestsDAO(DynamoDB)  
Constructor della classe GuestsDAO(DynamoDB). Permette di effettuare la dependency injection di AWS::DynamoDB;  
Parametri:  
* db: AWS::DynamoDB  
Parametro contenente un riferimento al modulo di Node.js da utilizzare per l'accesso al database DynamoDB contenente la tabella degli ospiti;
```

Member

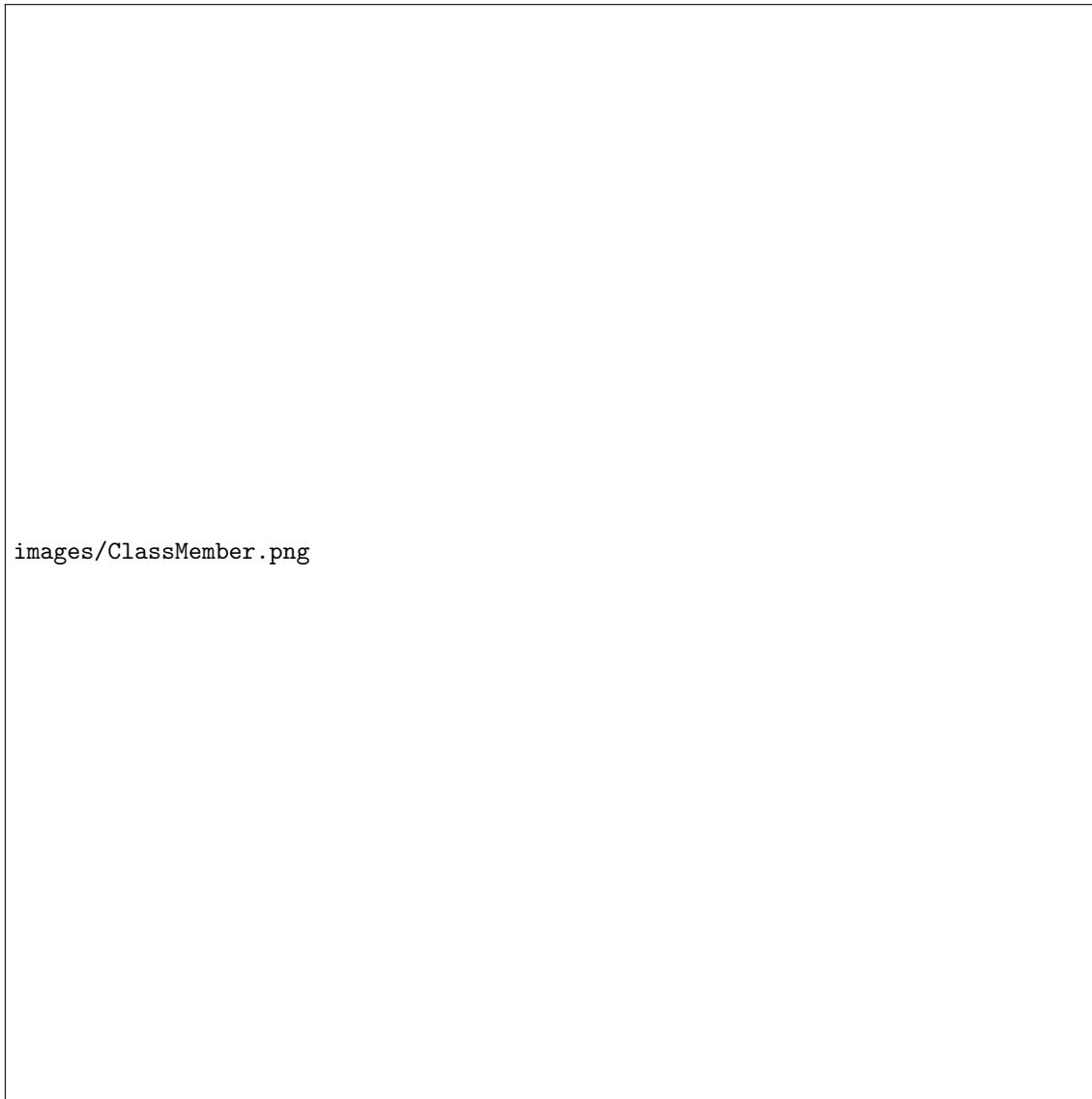


Figura 19: Back-end::Member

- **Nome:** Member;
- **Tipo:** Class;

- **Descrizione:** questa classe si occupa di raggruppare le informazioni relative ad un membro dell'azienda;

- **Utilizzo:** fornisce l'accesso ai dati relativi ad un membro dell'azienda;

- **Attributi:**

- + **id:** String

Stringa contenente l'id del canale Slack del membro dell'azienda;

- + **first_name:** String

Nome del membro dell'azienda;

- + **last_name:** String

Cognome del membro dell'azienda;

MembersSlackDAO

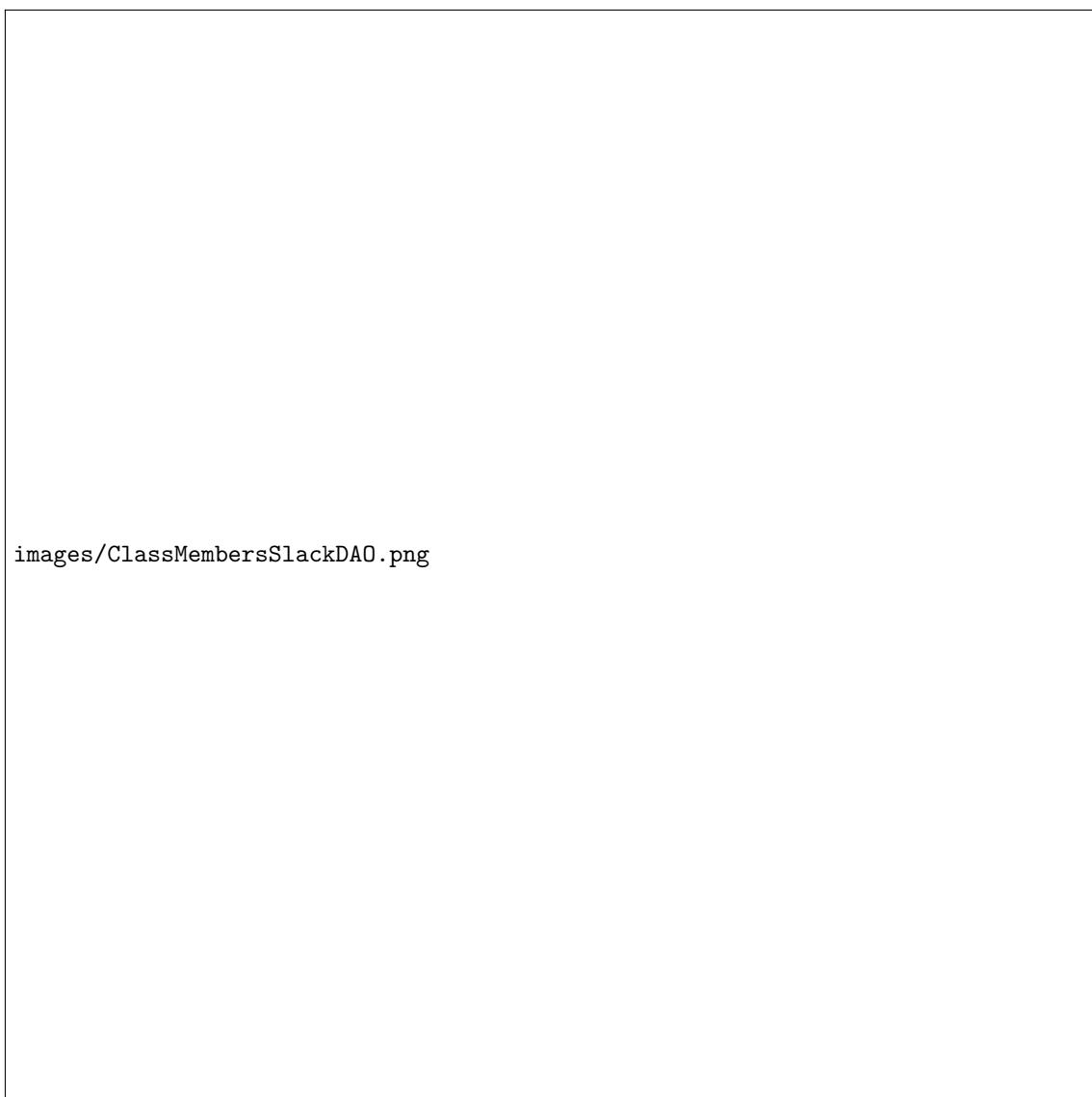


Figura 20: Back-end::MembersSlackDAO

- **Nome:** MembersSlackDAO;

- **Tipo:** Class;
- **Descrizione:** questa classe implementa MembersDAO, ottenendo i dati da Slack;
- **Utilizzo:** implementa i metodi definiti dall'interfaccia, utilizzando Slack::WebClient per recuperare i dati relativi ai membri dell'azienda. ;
- **Attributi:**
 - **client:** Slack::WebClient
Questo attributo è utilizzato per interrogare le API Web di Slack;
- **Metodi:**
 - + **addMember(member: Member): ErrorObservable**
Implementazione del metodo dell'interfaccia. L'Observable restituita genererà sempre un errore, in quanto non è possibile aggiungere membri;
Parametri:
* **member: Member**
Parametro contenente i dati relativi al membro che si vuole aggiungere;
 - + **removeMember(id: String): ErrorObservable**
Implementazione del metodo dell'interfaccia. L'Observable restituita genererà sempre un errore, in quanto non è possibile rimuovere membri;
Parametri:
* **id: String**
Parametro contenente la stringa identificativa del membro che si vuole eliminare;
 - + **updateMember(member: Member): ErrorObservable**
Implementazione del metodo dell'interfaccia. L'Observable restituita genererà sempre un errore, in quanto non è possibile aggiornare i dati dei membri;
Parametri:
* **member: Member**
Parametro contenente i dati relativi al membro che si vuole aggiornare;
 - + **getMember(id: String): MemberObservable**
Implementazione del metodo dell'interfaccia. Utilizza il metodo users.info per ottenere le informazioni necessarie da Slack;
Parametri:
* **id: String**
parametro contenente la stringa identificativa del membro del quale si vogliono ottenere i dati;
 - + **getMemberList(): MemberObservable**
Implementazione del metodo dell'interfaccia. Utilizza il metodo users.list per ottenere le informazioni necessarie da Slack;

RuleObservable

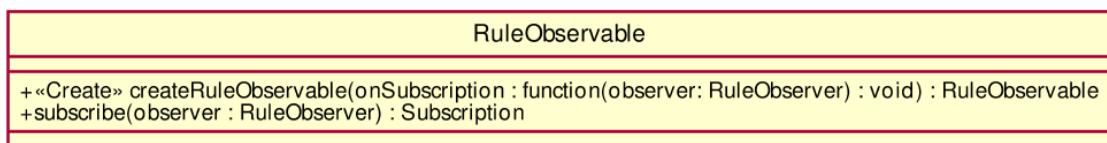


Figura 21: Back-end::RuleObservable

- **Nome:** RuleObservable;
- **Tipo:** Class;
- **Descrizione:** questa classe implementa un **Observable** che permette l'iscrizione di **RuleObserver**;
- **Utilizzo:** fornisce i meccanismi necessari per il passaggio di una serie di **Rule** ad un **Observer** interessato;
- **Padre:** Observable;
- **Metodi:**
 - + <<Create>> createRuleObservable(onSubscription: function(observer: RuleObserver) : void): RuleObservable
Constructor di RuleObservable;
Parametri:

* onSubscription: function(observer: RuleObserver) : void
Funzione che verrà eseguita quando un **Observer** si iscrive all'**Observable**. Si occupa di passare i dati all'**Observer**, chiamando il metodo **next(rule: Rule)**. Quando non ci sono più dati da restituire, si occupa di chiamare il metodo **complete()**. Nel caso in cui si verificasse un errore, si occupa di chiamare il metodo **error(err: Object)** con i dati relativi all'errore verificatosi;
 - + subscribe(observer: RuleObserver): Subscription
Metodo che permette ad un **RuleObserver** interessato di iscriversi a questo **Observable**;
Parametri:

* observer: RuleObserver
Observer che si vuole iscrivere;

RuleObserver

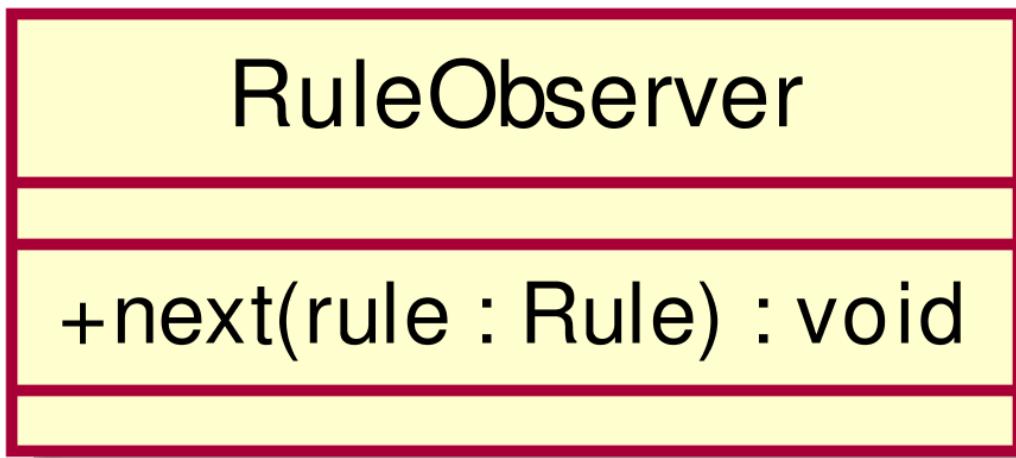


Figura 22: Back-end::RuleObserver

- **Nome:** RuleObserver;
- **Tipo:** Class;
- **Descrizione:** classe che rappresenta un **Observer** che si aspetta dati di tipo **Rule**;

- **Utilizzo:** implementa il metodo `next()` dell'interfaccia, in maniera tale che accetti dati di tipo `Rule`;

- **Metodi:**

```
+ next(rule: Rule): void
```

Metodo che permette agli `Observable` di notificare l'`Observer` con dati di tipo `Rule`.

Definisce inoltre le operazioni che l'`Observer` compierà all'arrivo di tali dati;

Parametri:

```
* rule: Rule
```

Parametro contenente la `Rule` mandata dall'`Observable`;

SNSEvent

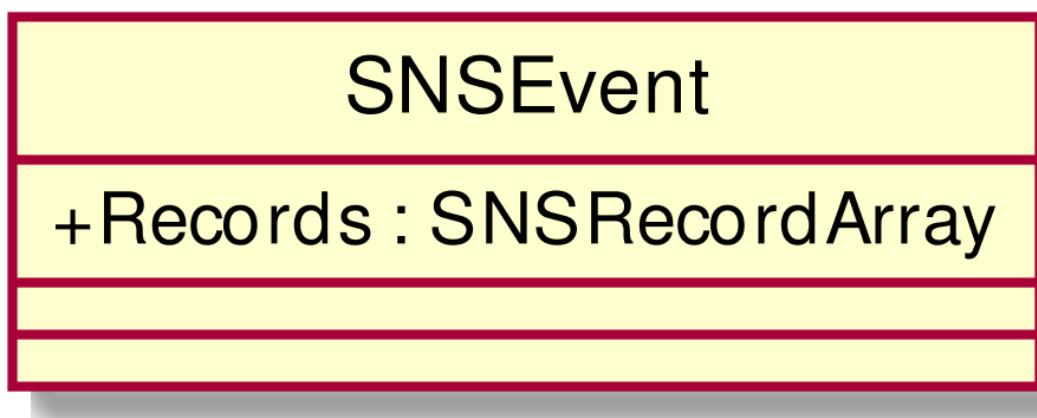


Figura 23: Back-end::SNSEvent

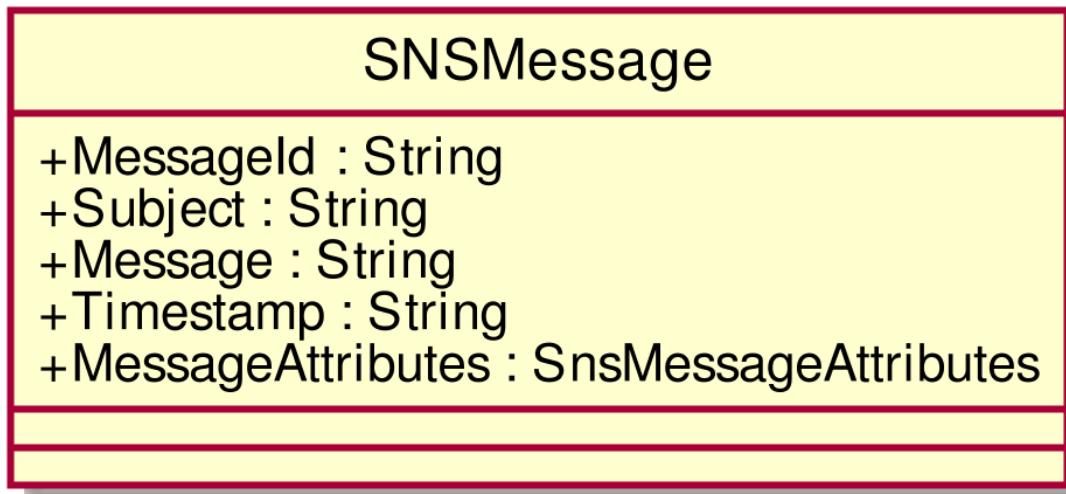
- **Nome:** `SNSEvent`;
- **Tipo:** Class;
- **Descrizione:** questa classe rappresentaa l'oggetto ricevuto da una lambda function in seguito alla pubblicazione di un messaggio su un topic di SNS a cui tale funzione è iscritta;
- **Utilizzo:** fornisce gli attributi relativi ad una notifica mandata da SNS ad una lambda function iscritta ad un topic sul quale sia stato pubblicato un messaggio;
- **Attributi:**

```
+ Records: SNSRecordArray
```

Array contenente i dati della notifica mandata da SNS alla lambda function. Contiene un unico oggetto di tipo Record;

SNSMessage

- **Nome:** `SNSMessage`;
- **Tipo:** Class;
- **Descrizione:** questa classe rappresenta un messaggio mandato da SNS in seguito ad un'interazione con l'assistente virtuale;

**Figura 24:** Back-end::SNSMessage

- **Utilizzo:** fornisce un meccanismo event-driven per la gestione dei dati relativi alle interazioni col sistema.

Per la relativa documentazione, consultare la pagina <http://docs.aws.amazon.com/sns/latest/dg/json-formats.html#http-subscription-confirmation-json>;

- **Attributi:**

- + **MessageId:** String
Attributo contenente l'identificativo del messaggio;
- + **Subject:** String
Attributo contenente il Subject che dev'essere notificato dal sistema SNS;
- + **Message:** String
Attributo contenente il messaggio da pubblicare;
- + **Timestamp:** String
Attributo contenente il timestamp relativo al momento della pubblicazione del messaggio;
- + **MessageAttributes:** Object
Attributo contenente le coppie chiave-valore degli attributi del messaggio che è stato pubblicato su SNS;

SNSRecord

- **Nome:** SNSRecord;
- **Tipo:** Class;
- **Descrizione:** questa classe rappresenta uno dei records mandati da SNS ad una lambda function. ;
- **Utilizzo:** fornisce gli attributi di un record mandato da SNS ad una lambda function. ;
- **Attributi:**
 - + **Sns:** SNSMessage
;

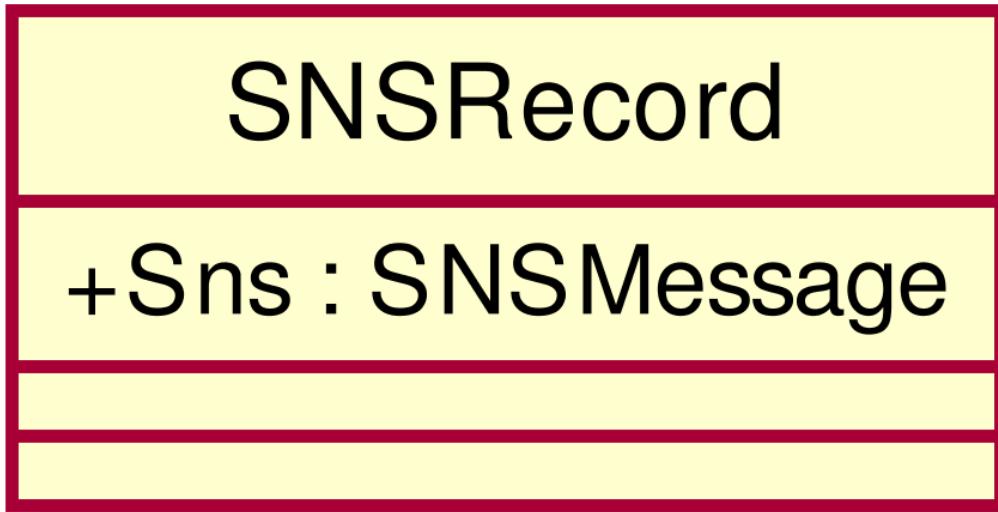
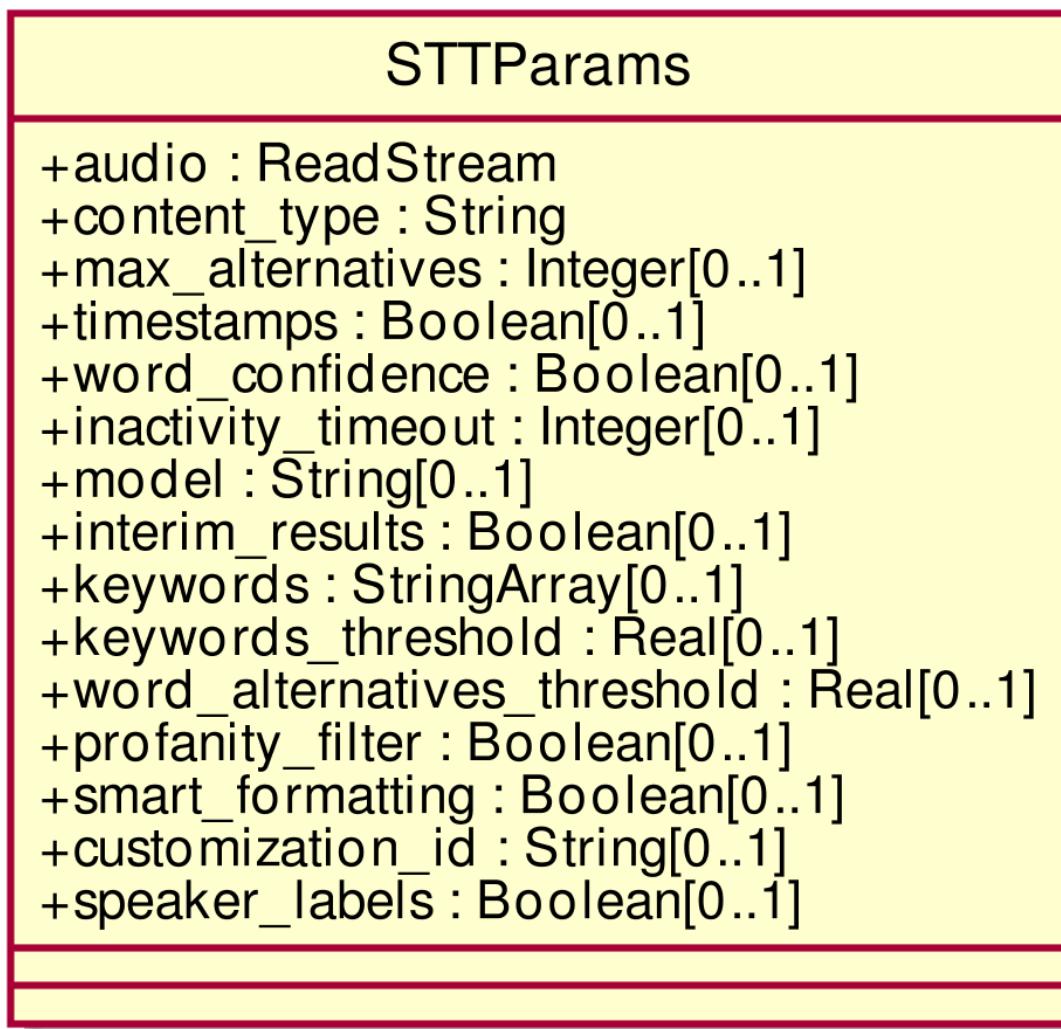


Figura 25: Back-end::SNSRecord

STTParams

- **Nome:** STTParams;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare ed organizzare i parametri necessari a richiamare le API Watson Speech to Text;
- **Utilizzo:** fornisce l'insieme dei parametri necessari ad identificare un audio, nel formato richiesto dall'API Watson Speech to Text, che verrà poi convertito in testo. Per la relativa documentazione, consultare la pagina https://www.ibm.com/watson/developercloud/speech-to-text/api/v1/#recognize_audio_websockets;
- **Attributi:**
 - + **audio:** ReadStream
Attributo contenente lo stream per la lettura del file audio;
 - + **content_type:** String
Attributo contenente il formato dell'audio;
 - + **max_alternatives:** Integer[0..1]
Attributo contenente il numero massimo di alternative testuali che devono essere fornite per il relativo audio. Il valore di default di questo attributo è uguale a 1;
 - + **timestamps:** Boolean[0..1]
Attributo contenente un valore booleano che indica se ottenere un timestamp per ogni parola. Il valore di default per questo attributo è false;
 - + **word_confidence:** Boolean[0..1]
Attributo contenente un valore booleano che indica se ottenere il grado di confidenza, contenuto nell'intervallo [0,1], per ogni parola. Il valore di default per questo attributo è false;
 - + **inactivity_timeout:** Integer[0..1]
Attributo contenente il tempo in secondi dopo il quale, se nell'audio viene rilevato solo

**Figura 26:** Back-end::STTParams

del silenzio, la connessione deve essere chiusa. Il valore di default di questo attributo è 30 secondi.

Per non chiudere mai la connessione, si può impostare questo attributo a -1. ;

`+ model: String[0..1]`

Attributo contenente il nome del model relativo all'audio da trascrivere.

Un model è un parametro che indica la lingua e il tipo di sampling rates usato per essa. Il tipo del sampling rates supportato può assumere uno tra i seguenti valori:

* broadband;

* narrowband.

Si rimanda alla relativa documentazione (https://www.ibm.com/watson/developercloud/speech-to-text/api/v1/?curl#get_models) per ulteriori chiarimenti;

`+ interim_results: Boolean[0..1]`

Attributo contenente un valore booleano che indica se posso essere ritornati risultati parziali o meno. Se impostato a true, i risultati parziali sono ritornati come uno stream di oggetti JSON ed ognuno di essi rappresenta un singolo SpeechRecognitionEvent.

Se impostato a false, verrà ritornato un unico SpeechRecognitionEvent contenente il risultato finale.

Questo attributo ha valore di default uguale a false;

+ **keywords:** `StringArray[0..1]`

Attributo contenente l'array delle parole da ricercare nell'audio. Ogni cella di questo array può contenere una o più parole da cercare. ;

+ **keywords_threshold:** `Real[0..1]`

Attributo contenente un valore di confidenza, contenuto nell'intervallo [0,1], che è il limite inferiore per una keyword trovata. Una parola fa match in una keyword se la sua confidenza è maggiore o uguale al valore di questo attributo.

Se questo parametro è omesso, nessuna keyword sarà trovata, altrimenti deve essere fornita almeno una keyword;

+ **word_alternatives_threshold:** `Real[0..1]`

Attributo contenente un valore di confidenza, contenuto nell'intervallo [0,1], che è il limite inferiore per identificare un'ipotetica parola come possibile alternativa.

Una parola alternativa è considerata tale se la sua confidenza è maggiore o uguale al valore di questo attributo. Se questo parametro è omesso, nessuna parola alternativa verrà fornita;

+ **profanity_filter:** `Boolean[0..1]`

Attributo contenente un valore booleano che indica se il profanity filter verrà applicato al testo trascritto.

Il profanity filter è un meccanismo che sostituisce parole inappropriate con degli asterischi. Questo filtro può essere applicato solo a trascrizioni in lingua US English.

Il valore di default per questo attributo è uguale a true;

+ **smart_formatting:** `Boolean[0..1]`

Attributo contenente un valore booleano che indica se date, orari, serie di numeri e cifre, numeri di telefono, valori monetari e indirizzi Internet devono essere convertiti, nella trascrizione finale, in un formato più leggibile. Questo meccanismo può essere applicato solo a trascrizioni in lingua US English.

Il valore di default per questo attributo è uguale a false;

+ **customization_id:** `String[0..1]`

Attributo contenente il GUID relativo al model personalizzato utilizzato. Per default, nessun modello personalizzato è utilizzato;

+ **speaker_labels:** `Boolean[0..1]`

Indicates whether labels that identify which words were spoken by which participants in a multi-person exchange are to be included in the response. If true, speaker labels are returned; if false (the default), they are not. Speaker labels can be returned only for the following language models: en-US_NarrowbandModel es-ES_NarrowbandModel ja-JP_NarrowbandModel Setting speaker_labels to true forces the continuous and time-stamps parameters to be true, as well, regardless of whether the user specifies false for the parameters. For more information, see Speaker labels;

TaskObservable

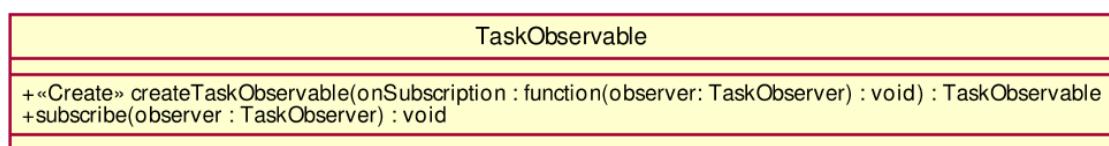


Figura 27: Back-end::TaskObservable

- **Nome:** TaskObservable;
- **Tipo:** Class;
- **Descrizione:** questa classe implementa un Observable che permette l'iscrizione di TaskObserver;
- **Utilizzo:** fornisce i meccanismi necessari per il passaggio di una serie di Task ad un Observer interessato;
- **Padre:** Observable;
- **Metodi:**
 - + <<Create>> createTaskObservable(onSubscription: function(observer: TaskObserver) : void): TaskObservable
Constructor di TaskObservable;
Parametri:

* onSubscription: function(observer: TaskObserver) : void
Funzione che verrà eseguita quando un Observer si iscrive all'Observable. Si occupa di passare i dati all'Observer, chiamando il metodo next(function: Task). Quando non ci sono più dati da restituire, si occupa di chiamare il metodo complete(). Nel caso in cui si verificasse un errore, si occupa di chiamare il metodo error(err: Object) con i dati relativi all'errore verificatosi;
 - + subscribe(observer: TaskObserver): void
Metodo che permette ad un TaskObserver interessato di iscriversi a questo Observable;
Parametri:

* observer: TaskObserver
Observer che si vuole iscrivere;

TaskObserver

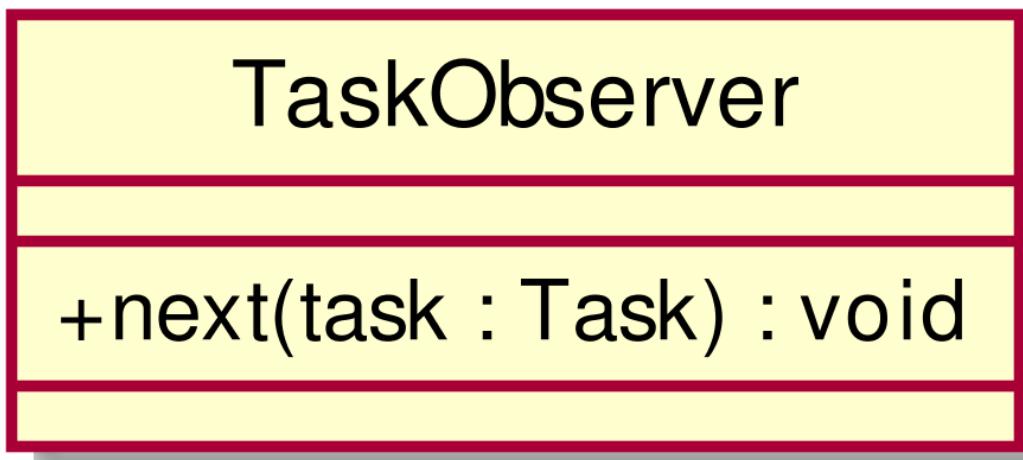


Figura 28: Back-end::TaskObserver

- **Nome:** TaskObserver;
- **Tipo:** Class;
- **Descrizione:** classe che rappresenta un Observer che si aspetta dati di tipo Task;

- **Utilizzo:** implementa il metodo `next()` dell'interfaccia, in maniera tale che accetti dati di tipo `Task`;

- **Metodi:**

```
+ next(task: Task): void
```

Metodo che permette agli `Observable` di notificare l'`Observer` con dati di tipo `Task`.

Definisce inoltre le operazioni che l'`Observer` compierà all'arrivo di tali dati;

Parametri:

```
* task: Task
```

Parametro contenente la `Task` mandata dall'`Observable`;

UserObservable

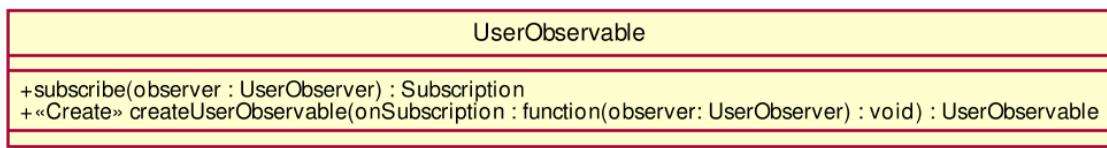


Figura 29: Back-end::UserObservable

- **Nome:** `UserObservable`;

- **Tipo:** Class;

- **Descrizione:** questa classe implementa un `Observable` che permette l'iscrizione di `UserObserver`;

- **Utilizzo:** fornisce i meccanismi necessari per il passaggio di una serie di `User` ad un `Observer` interessato;

- **Padre:** `Observable`;

- **Metodi:**

```
+ subscribe(observer: UserObserver): Subscription
```

Metodo che permette ad uno `UserObserver` interessato di iscriversi a questo `Observable`;

Parametri:

```
* observer: UserObserver
```

Observer che si vuole iscrivere;

```
+ <<Create>> createUserObservable(onSubscription: function(observer: UserObserver) : void): UserObservable
```

Constructor di `UserObservable`;

Parametri:

```
* onSubscription: function(observer: UserObserver) : void
```

Funzione che verrà eseguita quando un `Observer` si iscrive all'`Observable`. Si occupa di passare i dati all'`Observer`, chiamando il metodo `next(user: User)`. Quando non ci sono più dati da restituire, si occupa di chiamare il metodo `complete()`. Nel caso in cui si verificasse un errore, si occupa di chiamare il metodo `error(err: Object)` con i dati relativi all'errore verificatosi;

UserObserver

- **Nome:** `UserObserver`;

- **Tipo:** Class;

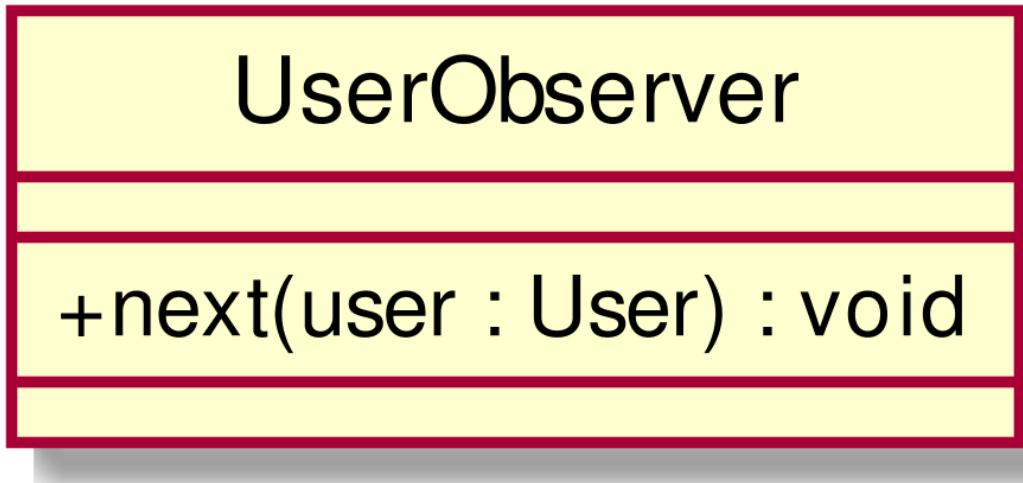


Figura 30: Back-end::UserObserver

- **Descrizione:** classe che rappresenta un `Observer` che si aspetta dati di tipo `User` ;
- **Utilizzo:** implementa il metodo `next()` dell’interfaccia, in maniera tale che accetti dati di tipo `User`;
- **Metodi:**

`+ next(user: User): void`

Metodo che permette agli `Observable` di notificare l’`Observer` con dati di tipo `User`. Definisce inoltre le operazioni che l’`Observer` compierà all’arrivo di tali dati;
Parametri:

`* user: User`

Parametro contenente lo `User` mandato dall’`Observable`;

VAMessageListener

- **Nome:** `VAMessageListener`;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di registrare i dati relativi alle interazioni degli ospiti col nostro sistema;
- **Utilizzo:** fornisce un meccanismo per registrare i dialoghi che gli ospiti hanno con il sistema. Fornisce una lambda function che quando viene generato un evento `SNSMessage` in seguito all’arrivo di una risposta da parte dell’assistente virtuale, si occupa di registrare i dati della relativa interazione tramite `ConversationsDAO`, ed eventualmente di aggiornare i dati relativi all’ospite utilizzando `GuestsDAO`.
SNS chiama tale lambda function con un oggetto del tipo `SNSEvent`, il quale contiene al suo interno un array di `SNSRecord`. Questo array in realtà ha un unico oggetto, il quale al suo interno contiene l’`SNSMessage` inviato.
Di seguito viene riportato un esempio dell’oggetto utilizzato.

```

1 {
2   "Records" :
3   [
4     {

```

**Figura 31:** Back-end::VAMessageListener

```

5     "Sns" :
6     {
7         "Message": "Corpo del messaggio pubblicato sul topic di
8             SNS",
9         "MessageAttributes": {"key": "value", "key2": "value2"},
10        "MessageId": "stringa-contenente-id-del-messaggio",
11        "Subject": "oggetto del messaggio",
12        "Timestamp": "2017-03-26T20:39:48.599Z"
13    }
14 }
15
;
```

- **Attributi:**

- guests: GuestsDAO
Attributo che permette di contattare il GuestsDAO;
- conversations: ConversationDAO
Attributo che permette di contattare il ConversationsDAO;

- **Eventi gestiti:**

- SNSEvent
Messaggio mandato ad sns quando arriva la risposta dall'assistente virtuale. All'arrivo del messaggio si occupa di salvare i dati dell'interazione nel DAO.

Back-end::APIGateway

Package contenente le componenti necessarie a gestire le richieste ricevute dal backend.

Classi

Enrollment

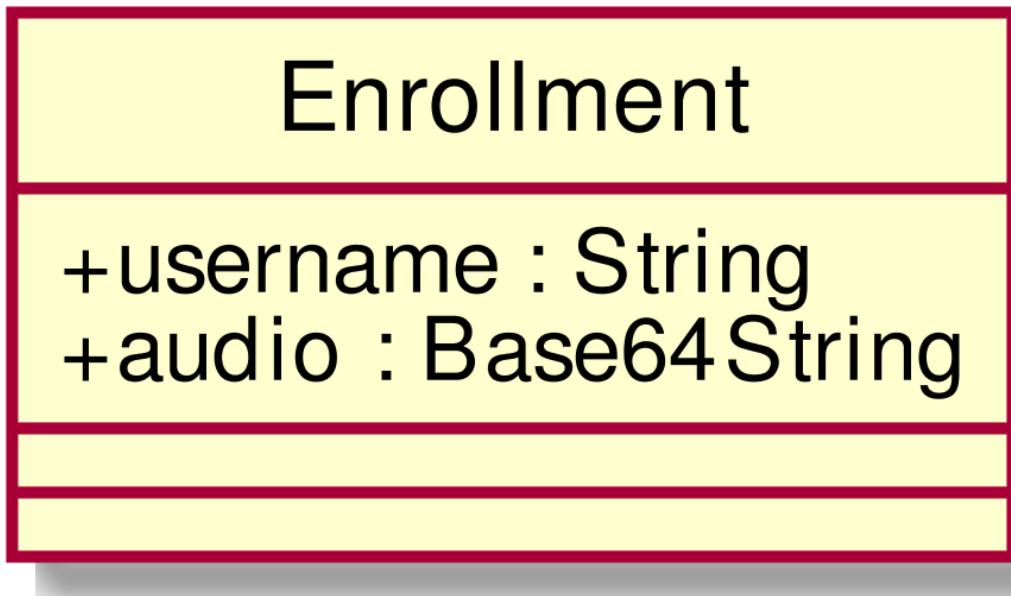


Figura 32: Back-end::APIGateway::Enrollment

- **Nome:** Enrollment;
- **Tipo:** Class;
- **Descrizione:** questa classe fornisce gli attributi necessari al passaggio di un Enrollment alle lambda function;
- **Utilizzo:** fornisce gli attributi relativi ad un Enrollment;
- **Attributi:**
 - + username: String
Attributo contenente l'username dell'utente associato all'Enrollment;
 - + audio: Base64String
Attributo contenente la traccia audio che sarà oggetto dell'Enrollment, codificata in Base64;

VAResponse APIBody DA MODIFICARE DA MODIFICARE DA MODIFICARE DA MODIFICARE

- **Nome:** VAResponse APIBody DA MODIFICARE DA MODIFICARE DA MODIFICARE DA MODIFICARE;
- **Tipo:** Class;
- **Descrizione:** questa classe fornisce gli attributi necessari al client per effettuare una richiesta all'API REST del back-end;
- **Utilizzo:** fornisce gli attributi relativi ad una richiesta all'API REST del back-end e viene utilizzata dalla classe VAResponse APIEvent. DA MODIFICARE DA MODIFICARE DA MODIFICARE DA MODIFICARE;

```
DA MODIFICARE DA MODIFICARE DA MODIFICARE DA MODIFICARE.png DA
MODIFICARE DA MODIFICARE DA MODIFICARE DA MODIFICARE.png DA
MODIFICARE DA MODIFICARE DA MODIFICARE DA MODIFICARE.png
[From /var/www/html/PragmaDB/Classi/.uml/ClassVARequestAPIBody DA MODIFICARE DA MODIFICARE DA MODIFICARE DA MODIFICARE.uml (line 4)]
@startuml
scale 1000*1000
skinparam classAttributeIconSize 0
class VARequestAPIBody DA MODIFICARE DA MODIFICARE DA MODIFICARE!
Syntax Error?
```

Figura 33: Back-end::APIGateway::VARequestAPIBody DA MODIFICARE DA MODIFICARE DA MODIFICARE DA MODIFICARE

MODIFICARE DA MODIFICARE DA MODIFICARE DA MODIFICARE (manca la classe VARequestAPIEvent);

- **Attributi:**

+ app: String

Attributo contenente il nome dell'applicazione da cui arriva la richiesta;

+ session_id: String

Attributo contenente l'id della sessione corrente, creato dal Client;

+ audio: Base64String

Attributo contenente l'audio della richiesta codificata in Base64;

+ data: ObjectAssocArray

Attributo contenente un array associativo di Object ricevuti dall'Assistente Virtuale;

VocalAPI

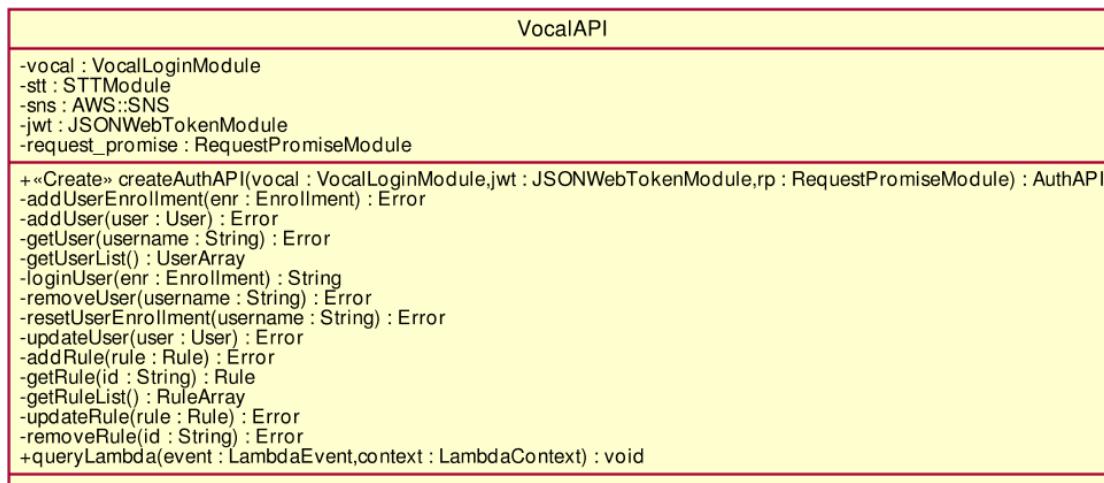


Figura 34: Back-end::APIGateway::VocalAPI

- **Nome:** VocalAPI;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di implementare l'endpoint dell'API Gateway utilizzato dal client vocale;
- **Utilizzo:** grazie al metodo pubblico di questa classe, fornisce un meccanismo che:
 - permette di dedurre, tramite i microservizi di STT e di assistente virtuale, il servizio necessario al client vocale, utilizzando il metodo `queryLambda`;

- permette al client vocale di usufruire delle funzionalità, supportate dal Backend, tramite i metodi privati che questa classe fornisce.

;

- **Attributi:**

- **vocal:** VocalLoginModule

Attributo contenente il **VocalLoginModule** di cui è stata eseguita la dependency injection nel costruttore. Viene utilizzato per effettuare il login nel servizio di Speaker Recognition;

- **stt:** STTModule

Attributo contenente il modulo utilizzato per contattare le API per il servizio di Watson Speech to Text di IBM;

- **sns:** AWS::SNS

Attributo che permette di contattare il servizio SNS;

- **jwt:** JSONWebTokenModule

Attributo contenente il **JSONWebTokenModule** di cui è stata eseguita la dependency injection nel costruttore. Viene utilizzato per creare un JSONWebToken in caso di autenticazione al sistema avvenuta con successo;

- **request_promise:** RequestPromiseModule

Attributo contenente il **RequestPromiseModule** di cui è stata eseguita la dependency injection nel costruttore. Viene utilizzato per effettuare richieste HTTP sostituendo callbacks con promises;

- **Metodi:**

- + <<Create>> **createAuthAPI(vocal: VocalLoginModule, jwt: JSONWebTokenModule, rp: RequestPromiseModule): AuthAPI**

Costruttore della classe **AuthAPI** che permette la dependency injection di **VocalLoginModule**; Parametri:

- * **vocal:** VocalLoginModule

Parametro che permette di effettuare la dependency injection di **VocalLoginModule**;

- * **jwt:** JSONWebTokenModule

Parametro che permette di effettuare la dependency injection di **JSONWebTokenModule**;

- * **rp:** RequestPromiseModule

Parametro che permette di effettuare la dependency injection di **RequestPromiseModule**;

- **addUserEnrollment(enr: Enrollment): Error**

Metodo che permette di aggiungere un enrollment ad un utente del sistema. Restituisce un oggetto di tipo Error, con code impostato a 1 nel caso in cui l'utente non esista; Parametri:

- * **enr:** Enrollment

Parametro contenente l'enrollment da aggiungere a un utente;

- **addUser(user: User): Error**

Metodo che permette di aggiungere un utente al sistema. Restituisce un oggetto di tipo Error, con code impostato a 1 in caso di username già esistente, 2 in caso di username non valido (troppo lungo o troppo corto);

Parametri:

- * **user:** User

Parametro contenente l'user che si vuole aggiungere al sistema;

- **getUser(username: String): Error**

Metodo che permette di ottenere i dati relativi ad un utente del sistema. Restituisce

l'oggetto User relativo all'utente con lo username indicato. In caso tale utente non esista, restituisce un oggetto vuoto;

Parametri:

- * `username: String`

Parametro contenente l'username dell'utente del quale si vogliono ottenere i dati;

- `getUserList(): UserArray`

Metodo che permette di ottenere una lista degli utenti del sistema;

- `loginUser(enr: Enrollment): String`

Metodo che si occupa di gestire il login vocale degli utenti. Restituisce una stringa contenente il JWT in caso di autenticazione avvenuta con successo, altrimenti restituisce una stringa vuota;

Parametri:

- * `enr: Enrollment`

Attributo contenente l'`Enrollment (audio + username)` con il quale tentare il login;

- `removeUser(username: String): Error`

Metodo che permette di eliminare i dati relativi ad un utente dal sistema. Restituisce un oggetto di tipo Error, con code impostato a 1 nel caso in cui l'utente non esista;

Parametri:

- * `username: String`

Parametro contenente l'username dell'user da eliminare dal sistema;

- `resetUserEnrollment(username: String): Error`

Metodo che permette di eliminare tutti gli enrollments di un utente del sistema. Restituisce un oggetto di tipo Error, con code impostato a 1 nel caso in cui l'utente non esista;

Parametri:

- * `username: String`

Parametro contenente l'username dell'utente a cui si vogliono eliminare tutti gli enrollments;

- `updateUser(user: User): Error`

Metodo che permette di modificare i dati relativi ad un utente del sistema. Restituisce un oggetto di tipo Error, con code impostato a 1 nel caso in cui l'utente non esista;

Parametri:

- * `user: User`

Parametro contenente l'user da modificare;

- `addRule(rule: Rule): Error`

Metodo che permette di aggiungere una direttiva al sistema. Restituisce un oggetto di tipo Error;

Parametri:

- * `rule: Rule`

Parametro contenente la Rule;

- `getRule(id: String): Rule`

Metodo che permette di ottenere i dati relativi ad una direttiva del sistema a partire dal suo id. Restituisce la direttiva in questione, oppure un oggetto vuoto nel caso in cui tale direttiva non esista;

Parametri:

- * `id: String`

Parametro contenente l'identificativo della Rule;

- `getRuleList(): RuleArray`

Metodo che permette di ottenere la lista delle direttive del sistema. ;

- `updateRule(rule: Rule): Error`

Metodo che permette di aggiornare una direttiva presente nel sistema. Restituisce un oggetto di tipo Error;

Parametri:

* `rule: Rule`

Parametro contenente la Rule aggiornata;

- `removeRule(id: String): Error`

Metodo che permette di rimuovere una direttiva presente nel sistema. Restituisce un oggetto di tipo Error;

Parametri:

* `id: String`

Parametro contenente l'identificativo della Rule;

+ `queryLambda(event: LambdaEvent, context: LambdaContext): void`

Metodo che si occupa di chiamare prima il servizio di Speech To Text e, una volta ottenuta risposta da esso, di interrogare l'assistente virtuale. Quando viene ricevuta una risposta dall'assistente virtuale, il metodo controlla il valore del campo `action` di tale risposta e, nel caso in cui corrisponda ad una delle action supportate, si occupa di eseguire le azioni necessarie utilizzando i metodi privati di questa classe. Nel caso in cui invece `action` non corrisponda ad una delle azioni supportate (ovvero `action` è un'azione che non richiede operazione da parte del back-end, oppure `actionIncomplete` è impostato a true), tale risposta viene rielaborata ed inoltrata. Le azioni supportate ed i relativi compiti da svolgere sono disponibili alla sezione [hyperef a sezione].

Ad ogni interazione viene inoltre pubblicato un messaggio su un topic di sns, utilizzando il metodo `sns.publish()`, in modo che sia possibile registrare i dati relativi a tali interazioni;

Parametri:

* `event: LambdaEvent`

Parametro contenente, all'interno del campo body sotto forma di stringa in formato JSON, un oggetto contenente tutti i dati relativi ad un messaggio da inviare. Tali dati sono:

```

1 {
2   "app": "String",
3   "audio": "Base64String",
4   "data": "ObjectAssocArray",
5   "session\_id": "String"
6 }
```

;

* `context: LambdaContext`

Parametro utilizzato dalle lambda function per inviare la risposta. La risposta, contenuta nel `LambdaResponse` parametro del metodo `LambdaContext::succeed`, possiede un attributo `body`, il quale conterrà il corpo di essa sotto forma di una stringa in formato JSON, organizzando i dati nel seguente modo:

```

1 {
2   "action": "String",
3   "res": {
4     "contexts": "ObjectAssocArray",
5     "data": "Object",
```

```

6   "text\_request": "String",
7   "text\_response": "String"
8 },
9 "session\_id": "String"
10 }
```

Per la relativa documentazione, consultare la pagina https://www.ibm.com/watson/developercloud/speech-to-text/api/v1/#recognize_sessionless_nonmp12. ;

VocalLoginModuleConfig

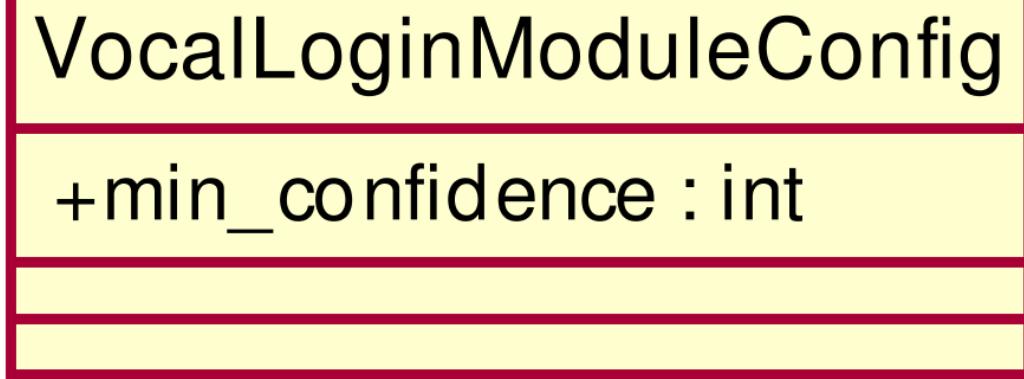


Figura 35: Back-end::APIGateway::VocalLoginModuleConfig

- **Nome:** VocalLoginModuleConfig;
- **Tipo:** Class;
- **Descrizione:** questa classe viene utilizzata per la configurazione di VocalLoginModule;
- **Utilizzo:** fornisce gli attributi necessari alla configurazione di VocalLoginModule;
- **Attributi:**
 - + min_confidence: int
Questo attributo indica la confidence minima richiesta perchè il login avvenga con successo;

Back-end::Auth

Package contenente le componenti del microservizio necessario all'autenticazione.

Classi

UsersDAO

- **Nome:** UsersDAO;
- **Tipo:** Interface;



Figura 36: Back-end::Auth:: UsersDAO

- **Descrizione:** questa classe si occupa di astrarre le modalità d'interazione al database per questo microservizio. ;
- **Utilizzo:** fornisce a `UserService` un meccanismo per accedere al database contenente gli utenti registrati, senza conoscerne le modalità di implementazione e di persistenza di quest'ultimo. A partire da un identificativo, permette operazioni di lettura, scrittura e rimozione di utenti registrati;
- **Figlio:** `UsersDAODynamoDB`;
- **Metodi:**

+ `addUser(user: User): ErrorObservable`
Metodo che permette di aggiungere un utente.

L'`Observable` restituito non riceverà alcun valore, ma verrà completato in caso di aggiunta dell'utente avvenuta con successo. In caso di errore durante l'aggiunta dell'utente, gli `Observer` interessati verranno notificati tramite la chiamata del loro metodo `error()` con i dati relativi all'errore verificatosi;

Parametri:

* `user: User`
Parametro contenente un utente registrato;

+ `removeUser(id: String): ErrorObservable`

Metodo che permette di rimuovere un utente registrato. L'`Observable` restituito non riceverà alcun valore, ma verrà completato in caso di aggiunta dell'utente avvenuta con successo. In caso di errore durante l'aggiunta dell'utente, gli `Observer` interessati verranno notificati tramite la chiamata del loro metodo `error()` con i dati relativi all'errore verificatosi;

Parametri:

* `id: String`
Parametro contenente l'id dello `User` che si vuole eliminare;

+ `getUser(username: String): UserObservable`

Metodo che permette di ottenere i dati relativi ad un utente.

L'`Observable` restituito riceverà l'oggetto rappresentante tale `User`, e verrà completato. Nel caso in cui l'utente richiesto non sia presente nel database, gli `Observer` interessati non riceveranno alcun valore, ma verranno notificati tramite la chiamata del loro metodo `error()`;

Parametri:

* `username: String`
Parametro contenente lo `username` dello `User` che si vuole ottenere;

+ `getUserList(): UserObservable`

L'`Observable` restituito manderà agli `Observer` gli utenti ottenuti, uno alla volta, e poi chiama il loro metodo `complete`. Nel caso in cui si verifichi un errore, gli `Observer` iscritti verranno notificati tramite la chiamate del loro metodo `error` con i dati relativi

all'errore verificatosi;

+ updateUser(user: User): ErrorObservable

Metodo che permette di aggiornare un utente registrato. L'Observable restituito non riceverà alcun valore, ma verrà completato in caso di aggiunta dell'utente avvenuta con successo. In caso di errore durante l'aggiunta dell'utente, gli Observer interessati verranno notificati tramite la chiamata del loro metodo error() con i dati relativi all'errore verificatosi;

Parametri:

* user: User

Parametro contenente lo User aggiornato;

SRUser

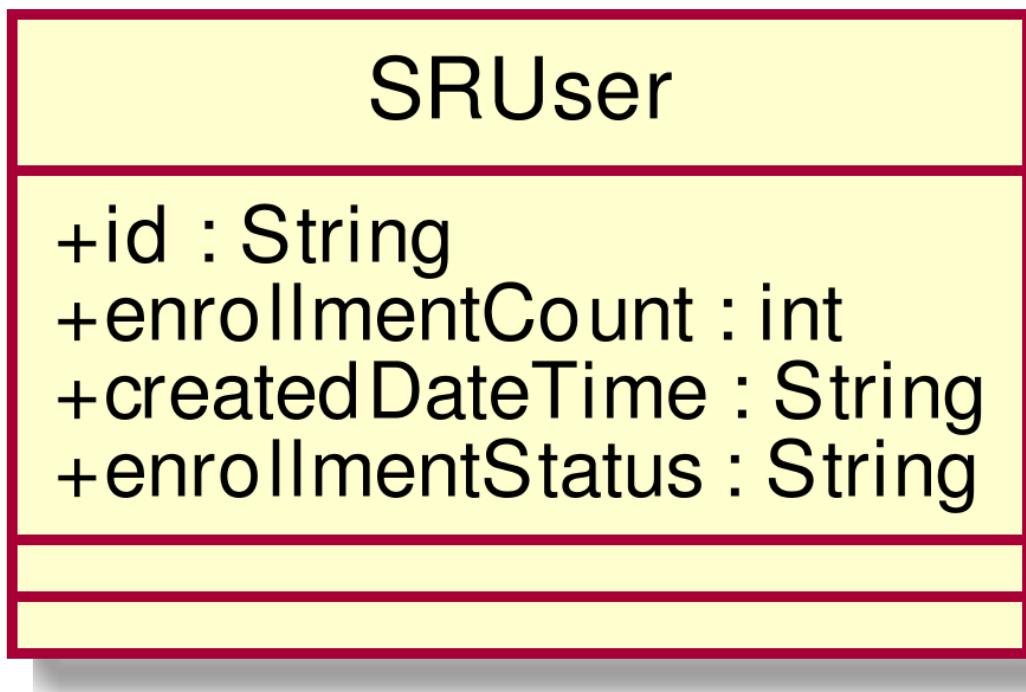


Figura 37: Back-end::Auth::SRUser

- **Nome:** SRUser;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare ed organizzare i parametri dell'autenticazione tramite Speaker Recognition (SR);
- **Utilizzo:** fornisce l'insieme dei parametri necessari ad identificare, tramite SR, un utente sottoposto alla procedura di Enrollment del servizio di Speaker Recognition. Per la relativa documentazione consultare la pagina <https://www.microsoft.com/cognitive-services/en-us/speaker-recognition-api/documentation.> ;
- **Attributi:**

```
+ id: String
Attributo contenente l'identificativo del profilo utente del microservizio di Speaker Recognition;

+ enrollmentCount: int
Attributo contenente il numero di Enrollment dello User;

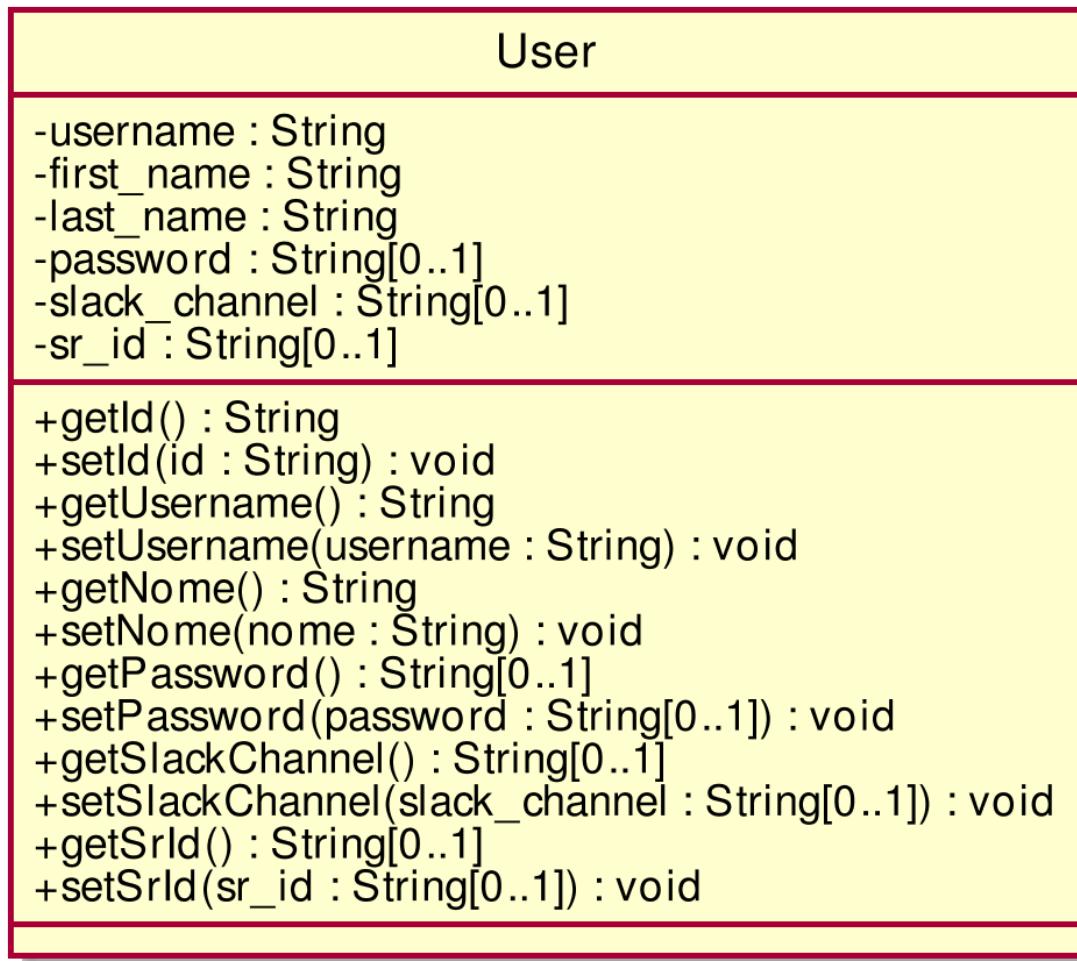
+ createdDateTime: String
Attributo contenente la data di creazione del profilo utente nel microservizio esterno di Speaker Recognition;

+ enrollmentStatus: String
Attributo contenente lo stato dell'Enrollment.
Lo stato può essere uno tra i seguenti valori:
    * Enrolling: indica che la fase di enrollment è in corso;
    * Training: indica che il microservizio di Speaker Recognition sta elaborando e organizzando i dati ricevuti (analizza le frasi comunicate e ne costruisce un'impronta vocale);
    * Enrolled, ovvero che le due fasi precedenti sono state completate.
```

;

User

- **Nome:** User;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare e organizzare i dati relativi ad un utente registrato;
- **Utilizzo:** fornisce i metodi getter e setter per i parametri relativi ad un utente registrato, i quali dovranno essere memorizzati nel database per questo microservizio.
È utilizzata dalla classe UsersDAO e dalle classi che utilizzano quest'ultima;
- **Attributi:**
 - username: String
Attributo contenente l'username dell'utente registrato;
 - first_name: String
Attributo contenente il nome dell'utente registrato;
 - last_name: String
Attributo contenente il cognome dell'utente registrato;
 - password: String[0..1]
Attributo contenente la password dell'utente registrato;
 - slack_channel: String[0..1]
Attributo contenente il canale Slack dell'utente registrato;
 - sr_id: String[0..1]
Attributo contenente l'id del profilo utente nel microservizio esterno di Speaker Recognition;
- **Metodi:**
 - + getId(): String
Metodo che permette di ottenere l'id dell'utente registrato;

**Figura 38:** Back-end::Auth::User

+ setId(id: String): void

Metodo che permette di impostare l'id dell'utente registrato;
Parametri:

* id: String

Parametro contenente l'id;

+ getUsername(): String

Metodo che permette di ottenere lo username dell'utente registrato;

+ setUsername(username: String): void

Metodo che permette di impostare lo username dell'utente registrato;
Parametri:

* username: String

Parametro relativo all'username da settare;

+ getNome(): String

Metodo che permette di ottenere il nome dell'utente registrato;

```
+ setNome(nome: String): void
Metodo che permette di impostare il nome dell'utente registrato;
Parametri:
    * nome: String
        Parametro contenente il nome;

+ getPassword(): String[0..1]
Metodo che permette di ottenere la password dell'utente registrato;

+ setPassword(password: String[0..1]): void
Metodo che permette di impostare la password dell'utente registrato;
Parametri:
    * password: String[0..1]
        Parametro relativo alla password da settare;

+ getSlackChannel(): String[0..1]
Metodo che permette di ottenere il canale Slack dell'utente registrato;

+ setSlackChannel(slack_channel: String[0..1]): void
Metodo che permette di impostare il canale Slack dell'utente registrato necessario per contattarlo;
Parametri:
    * slack_channel: String[0..1]
        Parametro relativo al canale Slack da settare;

+ getSrId(): String[0..1]
Metodo che permette di ottenere l'id del profilo utente nel microservizio esterno di Speaker Recognition;

+ setSrId(sr_id: String[0..1]): void
Metodo che permette di impostare l'id dello Speaker Recognition associato all'utente registrato;
Parametri:
    * sr_id: String[0..1]
        Parametro relativo all'id dello Speaker Recognition da settare;
```

UsersDAODynamoDB

UsersDAODynamoDB
-db : AWS::DynamoDB
+addUser(user : User) : ErrorObservable
+getUser(username : String) : UserObservable
+getUserList() : UserObservable
+removeUser(username : String) : ErrorObservable
+updateUser(user : User) : ErrorObservable
+«Create» createUsersDAODynamoDB(db : AWS::DynamoDB) : UsersDAODynamoDB

Figura 39: Back-end::Auth::UsersDAODynamoDB

- **Nome:** UsersDAODynamoDB;

- **Tipo:** Class;

- **Descrizione:** classe che si occupa di implementare l'interfaccia `UsersDAO`, utilizzando un database DynamoDB come supporto per la memorizzazione dei dati;

- **Utilizzo:** implementa i metodi dell'interfaccia `UsersDAO` interrogando un database DynamoDB. Utilizza `AWS::DynamoDB::DocumentClient` per l'accesso al database. La dependency injection dell'oggetto `AWS::DynamoDB` viene fatta utilizzando il costruttore;

- **Padre:** <>interface>> `UsersDAO`;

- **Attributi:**

- `db: AWS::DynamoDB`

Attributo contenente un riferimento al modulo di Node.js utilizzato per l'accesso al database DynamoDB contenente la tabella degli utenti;

- **Metodi:**

- + `addUser(user: User): ErrorObservable`

Implementazione del metodo definito nell'interfaccia `UsersDAO`. Utilizza il metodo `put` del `DocumentClient` per aggiungere l'utente al database;

Parametri:

- * `user: User`

Utente che si vuole aggiungere al sistema;

- + `getUser(username: String): UserObservable`

Implementazione del metodo definito nell'interfaccia `UsersDAO`. Utilizza il metodo `get` del `DocumentClient` per ottenere i dati relativi ad uno `User` dal database;

Parametri:

- * `username: String`

Parametro contenente lo `username` dello `User` che si vuole ottenere;

- + `getUserList(): UserObservable`

Implementazione del metodo dell'interfaccia `UsersDAO`. Utilizza il metodo `scan` del `DocumentClient` per ottenere la lista degli utenti dal database;

- + `removeUser(username: String): ErrorObservable`

Implementazione del metodo dell'interfaccia `UsersDAO`. Utilizza il metodo `delete` del `DocumentClient` per eliminare un utente dal database;

Parametri:

- * `username: String`

Username dell'utente che si vuole rimuovere dal sistema;

- + `updateUser(user: User): ErrorObservable`

Implementazione del metodo dell'interfaccia `UsersDAO`. Utilizza il metodo `update` del `DocumentClient` per aggiornare i dati relativi ad un utente presente all'interno del database;

Parametri:

- * `user: User`

Parametro contenente i dati relativi all'utente che si vuole modificare;

- + <>Create>> `createUsersDAODynamoDB(db : AWS::DynamoDB): UsersDAODynamoDB`

Constructor della classe `UsersDAODynamoDB`. Permette di effettuare la dependency injection di `AWS::DynamoDB`;

Parametri:

* db : AWS::DynamoDB
 Parametro contenente un riferimento al modulo di Node.js da utilizzare per l'accesso al database DynamoDB contenente la tabella degli utenti;

UsersService

UsersService
-users : UsersDAO
+getUserList(event : LambdaEvent,context : LambdaContext) : void
+addUser(event : LambdaEvent,context : LambdaContext) : void
+removeUser(event : LambdaEvent,context : LambdaContext) : void
+updateUser(event : LambdaEvent,context : LambdaContext) : void
+getUser(event : LambdaEvent,context : LambdaContext) : void
+«create» createUsersService(user : UserDAO) : UsersService

Figura 40: Back-end::Auth::UsersService

- **Nome:** UsersService;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di realizzare il microservizio Auth e, tramite UsersDAO, di interagire con il database degli utenti registrati;
- **Utilizzo:** fornisce i metodi che implementano le lambda function necessarie alla gestione degli utenti. Questa classe non interagisce direttamente con il database, ma fa utilizzo di UsersDAO, il quale nasconde i meccanismi di accesso e persistenza dei dati nel database;
- **Attributi:**
 - users: UsersDAO
 Attributo che permette di contattare UsersDAO, il quale fornisce i meccanismi d'accesso al database degli utenti registrati;
- **Metodi:**
 - + getUserList(event: LambdaEvent, context: LambdaContext): void
 Metodo che implementa la lambda function che si occupa di restituire l'array degli utenti registrati;
 Parametri:
 - * event: LambdaEvent
 Parametro che rappresenta la richiesta ricevuta dal VocalAPI. Il campo body di questo attributo conterrà una stringa vuota;
 - * context: LambdaContext
 Parametro utilizzato dalle lambda function per inviare la risposta. Il body del LambdaResponse, ottenuto dal metodo LambdaContext::succeed, conterrà un Array di oggetti di tipo User;
 - + addUser(event: LambdaEvent, context: LambdaContext): void
 Metodo che implementa la lambda function che si occupa di aggiungere un utente registrato;
 Parametri:

* event: LambdaEvent
 Parametro contenente, all'interno del campo body sotto forma di stringa in formato JSON, un oggetto User contenente tutti i dati relativi ad un utente da inserire;

* context: LambdaContext
 Parametro utilizzato dalle lambda function per inviare la risposta. La risposta, contenuta nel LambdaResponse parametro del metodo LambdaContext::succeed, possiede un attributo body, il quale conterrà una stringa vuota. Il risultato delle operazioni di questo metodo sarà deducibile tramite il valore dell'attributo LambdaResponse::statusCode;

+ removeUser(event: LambdaIdEvent, context: LambdaContext): void
 Metodo che implementa la lambda function che si occupa di rimuovere un utente registrato;
 Parametri:
 * event: LambdaIdEvent
 Parametro contenente, all'interno del campo pathParameters, lo username dell'utente registrato che si vuole eliminare;

* context: LambdaContext
 Parametro utilizzato dalle lambda function per inviare la risposta. Il body del LambdaResponse, parametro del metodo LambdaContext::succeed, conterrà una stringa vuota e il risultato di questa operazione sarà deducibile dal valore dell'attributo LambdaResponse::statusCode;

+ updateUser(event: LambdaIdEvent, context: LambdaContext): void
 Metodo che implementa la lambda function che si occupa di aggiornare i dati di un utente registrato;
 Parametri:
 * event: LambdaIdEvent
 Parametro contenente all'interno del campo body, sotto forma di stringa in formato JSON, un oggetto di tipo User contenente i dati da aggiornare e, all'interno del campo pathParameters, lo username dell'utente da modificare;

* context: LambdaContext
 Parametro utilizzato dalle lambda function per inviare la risposta. Il body del LambdaResponse, parametro del metodo LambdaContext::succeed, conterrà una stringa vuota e il risultato di questa operazione sarà deducibile dal valore dell'attributo LambdaResponse::statusCode;

+ getUser(event: LambdaIdEvent, context: LambdaContext): void
 Metodo che implementa la lambda function che si occupa di restituire i dati relativi ad un utente a partire dal suo username;
 Parametri:
 * event: LambdaIdEvent
 Parametro contenente, all'interno del campo pathParameters, lo username dell'utente registrato del quale si vogliono ottenere i dati;

* context: LambdaContext
 Parametro utilizzato dalle lambda function per inviare la risposta. Il body del LambdaResponse, parametro del metodo LambdaContext::succeed, conterrà un oggetto , sotto forma di stringa in formato JSON, di tipo User, contenente i dati relativi all'utente ritornato;

+ <<create>> createUserService(user: UserDAO): UsersService
 Metodo che permette di creare uno UsersService. Permette la dependency injection avente come oggetto uno UsersDAO;
 Parametri:

```
* user: UserDAO
Attributo contenente lo UsersDAO;
```

VocalLoginModule

VocalLoginModule
-min_confidence : int
+createUser() : String
+«Create» createVocalLoginModule(conf : VocalLoginModuleConfig) : VocalLoginModule
+addEnrollment(id : String, audio : Blob) : Error
+deleteUser(id : String) : Error
+getList() : SRUserArray
+getUser(id : String) : SRUser
+resetEnrollments(id : String) : Error
+doLogin(id : String, audio : Blob) : Error

Figura 41: Back-end::Auth::VocalLoginModule

- **Nome:** VocalLoginModule;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di realizzare e raggruppare tutte le operazioni necessarie all’identificazione di un utente registrato, tramite il microservizio di Speaker Recognition (SR). ;
- **Utilizzo:** fornisce un meccanismo per creare, eliminare ed ottenere un utente registrato associato ad un Enrollment, il quale è necessario all’identificazione di un utente tramite il microservizio di Speak Recognition (SR). Permette quindi di associare i parametri relativi ad un Enrollment ad un utente registrato.
È soggetta ad una constructor-based dependency injection, la quale ha come oggetto un VocalLoginModulConfig;

- **Attributi:**

- min_confidence: int

Attributo contenente il grado di confidenza minimo accettabile nel confronto tra ciò che l’utente comunica, al fine di effettuare l’accesso come utente registrato, e quella che dovrebbe essere la sua impronta vocale precedentemente costruita tramite il meccanismo di Enrollment;

- **Metodi:**

+ createUser(): String

Metodo che permette di creare un User;

+ <<Create>> createVocalLoginModule(conf: VocalLoginModuleConfig): VocalLoginModule

Metodo che permette di costruire un VocalLoginModule. Permette la dependency injection che ha come oggetto un VocalLoginModuleConfig;

Parametri:

* conf: VocalLoginModuleConfig

Parametro attraverso il quale viene passata la configurazione di VocalLoginModule;

+ addEnrollment(id: String, audio: Blob): Error

Metodo che permette di aggiungere un Enrollment;

Parametri:

```

* id: String
    Parametro contenente la stringa identificativa dell'utente al quale si vuole aggiungere un enrollment;

* audio: Blob
    Parametro contenente l'audio relativo alla frase di riconoscimento pronunciata;

+ deleteUser(id: String): Error
Metodo che permette di eliminare un User a partire da un id;
Parametri:
* id: String
    Parametro contenente l'identificativo dell'utente che si vuole eliminare;

+ getList(): SRUserArray
Metodo che ritorna una lista di SRUser;

+ getUser(id: String): SRUser
Metodo che ritorna un SRUser a partire da un id;
Parametri:
* id: String
    Parametro contenente l'identificativo dello User;

+ resetEnrollments(id: String): Error
Metodo che permette di resettare un enrollment a partire da un id;
Parametri:
* id: String
    Parametro contenente l'identificativo dell'utente dei quali relativi enrollment verrà effettuato il reset;

+ doLogin(id: String, audio: Blob): Error
Metodo che permette di effettuare il login a partire da un id e da un audio di identificazione;
Parametri:
* id: String
    Parametro contenente l'identificativo dell'utente che vuole effettuare il login;
* audio: Blob
    Parametro contenente l'audio relativo alla frase di riconoscimento pronunciata;

```

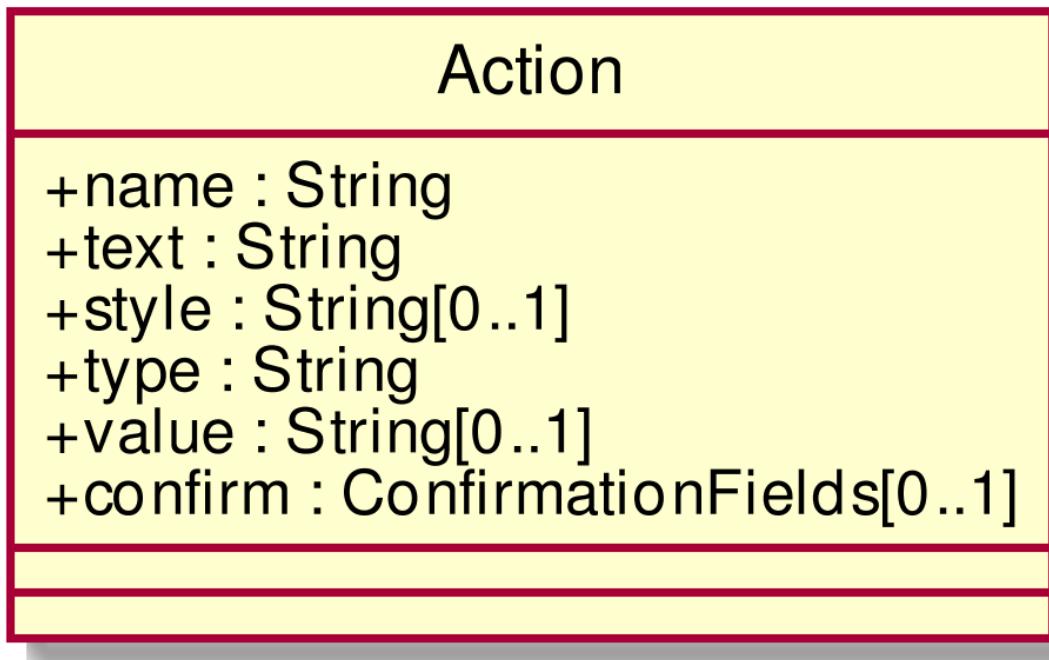
Back-end::Notifications

Package che contiene le componenti relative al microservizio che si occupa delle notifiche.

Classi

Action

- **Nome:** Action;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare e organizzare gli attributi relativi ad una Action come descritto nelle API di Slack. Rappresenta un button in un messaggio Slack;

**Figura 42:** Back-end::Notifications::Action

- **Utilizzo:** fornisce gli attributi di una Action. La classe Attachment ne contiene un array. Per la relativa documentazione, consultare la pagina <https://api.slack.com/docs/message-buttons>;

- **Attributi:**

+ name: String

Attributo contenente il nome dell'azione. Se ci sono più azioni con lo stesso nome, solo una di esse può essere in uno stato attivato;

+ text: String

Attributo contenente il testo del bottone dell'azione;

+ style: String[0..1]

Attributo che definisce lo stile del bottone. Per una lista degli stili disponibili e una loro descrizione fare riferimento alla documentazione di Slack (https://api.slack.com/docs/message-buttons#action_fields);

+ type: String

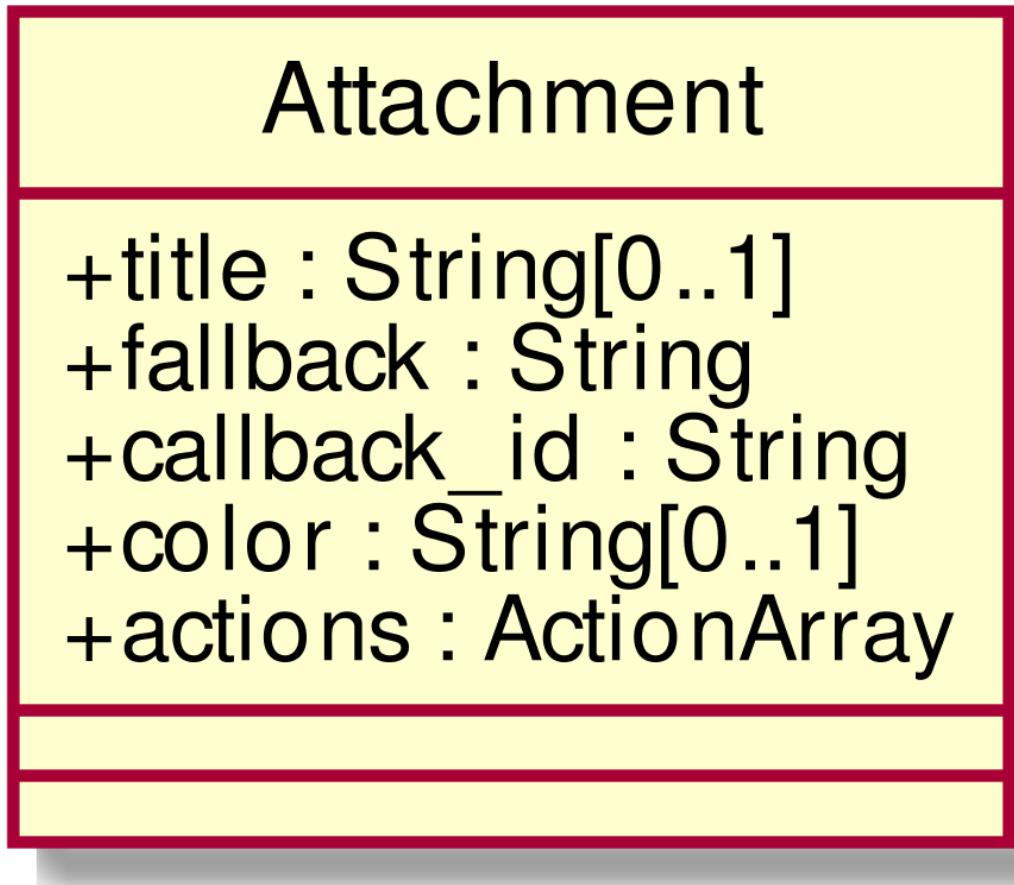
Attributo contenente il tipo dell'azione. Al momento l'unico valore accettato è "button". Fare riferimento alle API di Slack per informazioni aggiornate (https://api.slack.com/docs/message-buttons#action_fields). ;

+ value: String[0..1]

Attributo contenente il valore dell'azione. Se sono presenti diverse azioni con lo stesso nome, può essere utilizzato per distinguere diversi intenti;

+ confirm: ConfirmationFields[0..1]

Attributo contenete i dati relativi al dialogo di conferma, che nel caso di bottoni con azioni che possono avere effetti particolarmente "distruttivi" permette di chiedere un'ulteriore conferma prima di compiere effettivamente tali azioni;

Attachment**Figura 43:** Back-end::Notifications::Attachment

- **Nome:** Attachment;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare e organizzare i dati relativi ad un Attachment. ;
- **Utilizzo:** fornisce gli attributi degli Attachment che dovranno essere aggiunti ad un NotificationMessage. Un Attachment, contenuto in un NotificationsMessage, permette di aggiungere del significato ad un NotificationMessage ed arricchirlo tramite immagini, colori ed altro. Per la relativa documentazione, consultare questa pagina <https://api.slack.com/docs/message-buttons>;
- **Attributi:**
 - + title: String[0..1]
Attributo contenente il titolo dell'attachment;
 - + fallback: String
Attributo contenente un messaggio mostrato agli utenti che utilizzano un'interfaccia che non supporta gli attachments;

```
+ callback_id: String
```

Attributo contenente l'id della collezione di bottoni all'interno dell'attachment;

```
+ color: String[0..1]
```

Attributo contenente il colore dell'attachment;

```
+ actions: ActionArray
```

Attributo contenente una array di Action da includere nell'attachment. Questo array può contenere al massimo cinque Action;

ConfirmationFields

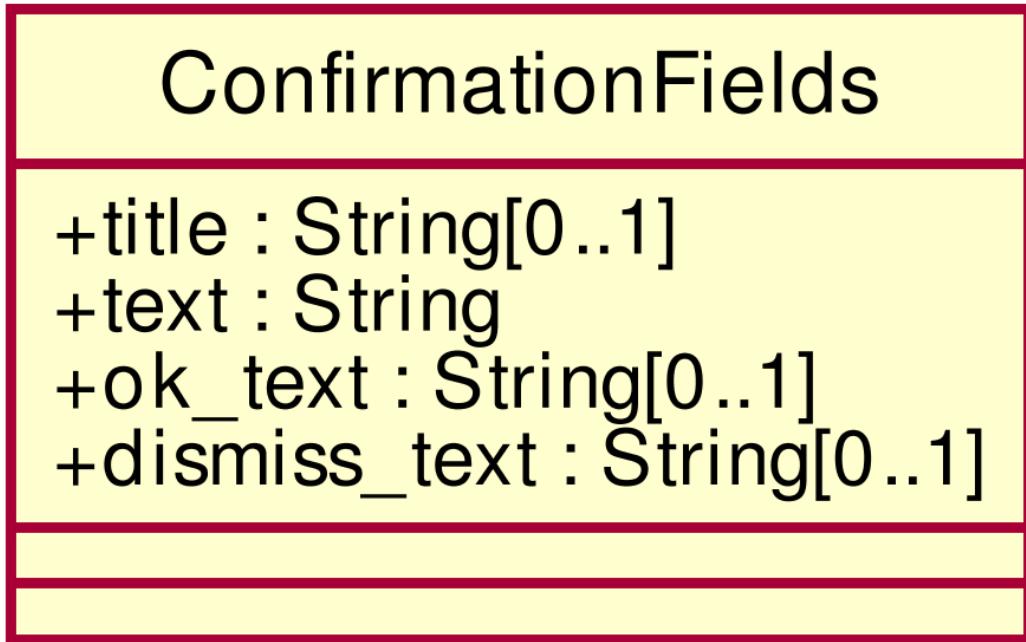


Figura 44: Back-end::Notifications::ConfirmationFields

- **Nome:** ConfirmationFields;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare e organizzare gli attributi relativi ad un messaggio di conferma di Slack;
- **Utilizzo:** fornisce gli attributi per rappresentare un messaggio di conferma di Slack.
Per la relativa documentazione, consultare la pagina <https://api.slack.com/docs/message-buttons>;
- **Attributi:**
 - + title: String[0..1]
Attributo contenente il titolo della finestra di pop up;
 - + text: String
Attributo contenente la descrizione dettagliata delle conseguenze della relativa Action e contestualizza le scelte fornite dal button;

```
+ ok_text: String[0..1]
```

Attributo contenente il testo del bottone che serve per confermare la relativa Action.
Il valore di default per questo attributo è "Okay". ;

```
+ dismiss_text: String[0..1]
```

Attributo contenente il testo del bottone che serve per cancellare la relativa Action. Il
valore di default per questo attributo è "Cancel". ;

NotificationChannel

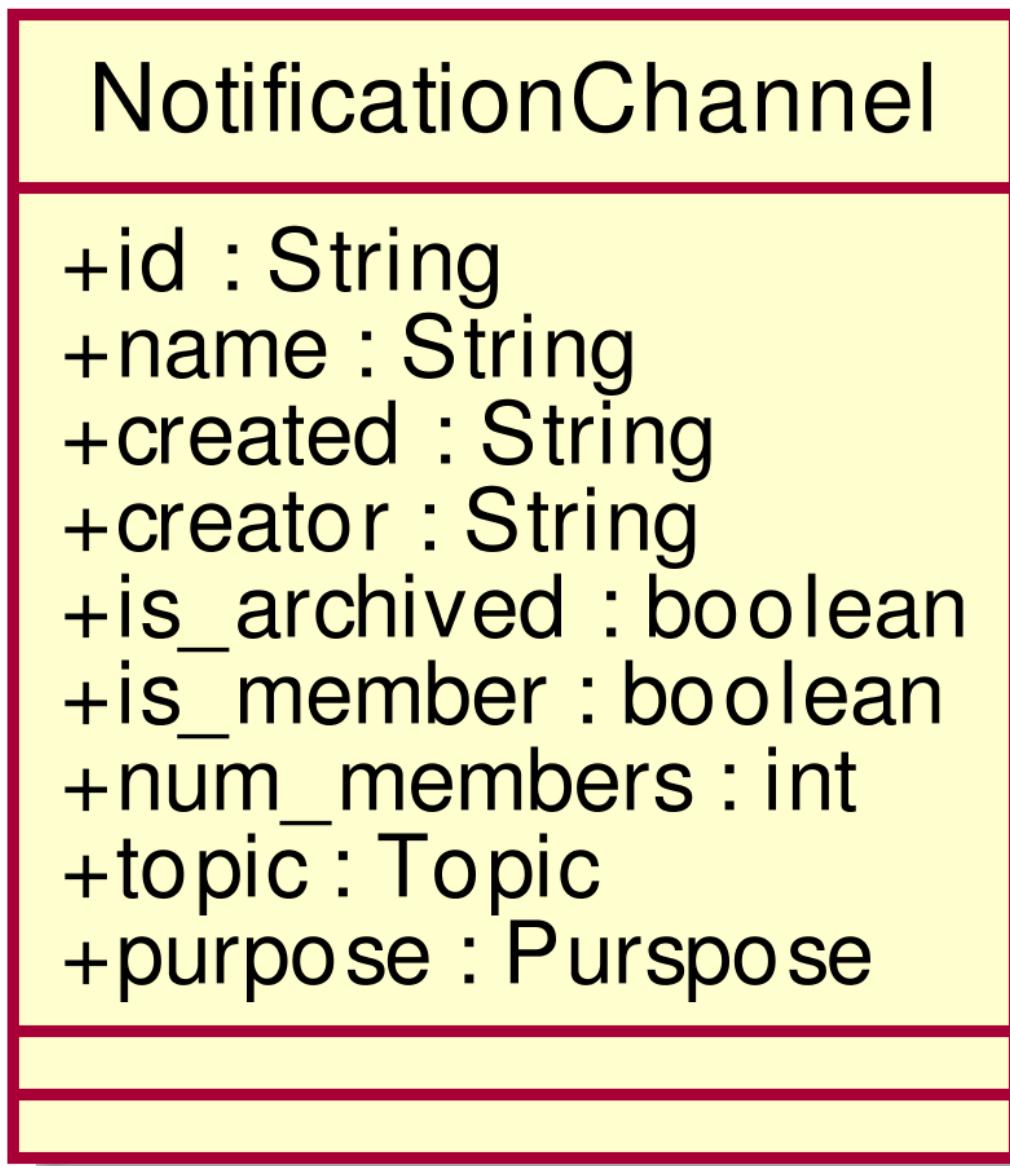


Figura 45: Back-end::Notifications::NotificationChannel

- **Nome:** `NotificationChannel`;

- **Tipo:** Class;

- **Descrizione:** questa classe si occupa di rappresentare e organizzare i parametri relativi al canale Slack nel quale spedire il NotificationMessage;

- **Utilizzo:** fornisce gli attributi di un NotificationChannel.

Per consultare la relativa documentazione, consultare questa pagina <https://api.slack.com/types/channel>;

- **Attributi:**

`+ id: String`

Attributo contenente l'id del canale Slack;

`+ name: String`

Attributo contenente il nome del canale Slack;

`+ created: String`

Attributo contenente l'unix timestamp relativo a creator;

`+ creator: String`

Attributo contenente l'id dell'utente Slack che ha creato il canale Slack;

`+ is_archived: boolean`

Attributo contenente un valore che permette di capire se un canale è stato archiviato o meno. Un canale è archiviato se non è più parte delle conversazioni attive;

`+ is_member: boolean`

Attributo contenente un valore booleano che permette di capire se il membro chiamante fa parte del canale Slack;

`+ num_members: int`

Attributo contenente il numero dei partecipanti al canale Slack;

`+ topic: Topic`

Attributo contenente l'argomento di discussione del canale Slack;

`+ purpose: Purpose`

Attributo contenente lo scopo per il quale il canale Slack è stato creato;

NotificationMessage

NotificationMessage

`+text : String`

`+attachments : AttachmentArray[0..1]`

`+response_type : String[0..1]`

Figura 46: Back-end::Notifications::NotificationMessage

- **Nome:** NotificationMessage;

- **Tipo:** Class;

- **Descrizione:** questa classe si occupa di rappresentare e organizzare i parametri relativi al messaggio da spedire nel NotificationChannel;

- **Utilizzo:** fornisce gli attributi di un NotificationChannel.

Per consultare la relativa documentazione, consultare questa pagina <https://api.slack.com/docs/message-formatting>;

- **Attributi:**

`+ text: String`

Attributo contenente il testo del NotificationMessage;

`+ attachments: AttachmentArray[0..1]`

Attributo contenente l'array degli attachments del NotificationMessage;

`+ response_type: String[0..1]`

Attributo contenente il modo con il quale notificare. Questo attributo può assumere uno tra i seguenti valori:

- * `in_channel`, che mostra il NotificationMessage ai membri del canale con il message button già cliccato;

- * `ephemeral`, che mostra il NotificationMessage ai membri del canale che cliccano sul message button.

Per la relativa documentazione, consultare questa pagina <https://api.slack.com/docs/message-buttons>;

NotificationService



Figura 47: Back-end::Notifications::NotificationService

- **Nome:** NotificationService;

- **Tipo:** Class;

- **Descrizione:** questa classe si occupa di realizzare il microservizio Notifications;

- **Utilizzo:** fornisce i metodi che implementano le lambda function necessarie per notificare la persona desiderata sul relativo canale Slack;

- **Attributi:**

- `- client: Slack::WebClient`

Questo attributo è utilizzato per interagire con le API Web di Slack;

- **Metodi:**

- `+ getChannelList(event : LambdaEvent, context : LambdaContext) : void`

Metodo che implementa la lambda function che si occupa di restituire l'array dei canali

Slack disponibili. ;

Parametri:

* `event: LambdaEvent`

Parametro che rappresenta la richiesta ricevuta dal VocalAPI. Il campo `body` di questo attributo conterrà una stringa vuota;

* `context: LambdaContext`

Parametro utilizzato dalle lambda function per inviare la risposta. Il `body` del `LambdaResponse`, parametro del metodo `LambdaContext::succeed`, conterrà un Array di oggetti di tipo `NotificationChannel`;

+ `sendMsg(event: LambdaEvent, context: LambdaContext): Msg`

Metodo che implementa la lambda function che si occupa di inviare il messaggio alla persona desiderata;

Parametri:

* `event: LambdaEvent`

Parametro contenente, all'interno del campo `body` sotto forma di stringa in formato JSON, un oggetto contenente tutti i dati relativi ad un messaggio da inviare. Tali dati sono:

```

1 {
2     "msg": "NotificationMessage",
3     "send\_to": "String"
4 }
```

Dove `msg` è un oggetto di tipo `NotificationMessage`, mentre `send_to` è una stringa contenente il mittente del messaggio;

* `context: LambdaContext`

Parametro utilizzato dalle lambda function per inviare la risposta. La risposta, contenuta nel `LambdaResponse` parametro del metodo `LambdaContext::succeed`, possiede un attributo `body`, il quale conterrà una stringa vuota. Il risultato delle operazioni di questo metodo sarà deducibile tramite il valore dell'attributo `LambdaResponse::statusCode`;

+ <<Create>> `createNotificationService(client: Slack::WebClient): NotificationService`

Costruttore, si occupa di gestire la dependency injection di `Slack::WebClient`;

Parametri:

* `client: Slack::WebClient`

Parametro contenente l'istanza di `Slack::WebClient` di cui viene effettuata la dependency injection;

Purpose

- **Nome:** Purpose;

- **Tipo:** Class;

- **Descrizione:** questa classe si occupa di rappresentare e organizzare i dati relativi al Purpose di un canale Slack. ;

- **Utilizzo:** fornisce un meccanismo per impostare e ottenere i valori dei parametri relativi ad un Purpose. Il purpose di un canale Slack è lo scopo per il quale è stato creato. Per la relativa documentazione, consultare la seguente pagina <https://api.slack.com/methods/channels.setPurpose>;

- **Attributi:**

- + `value: String`

Attributo contenente il valore del Purpose;

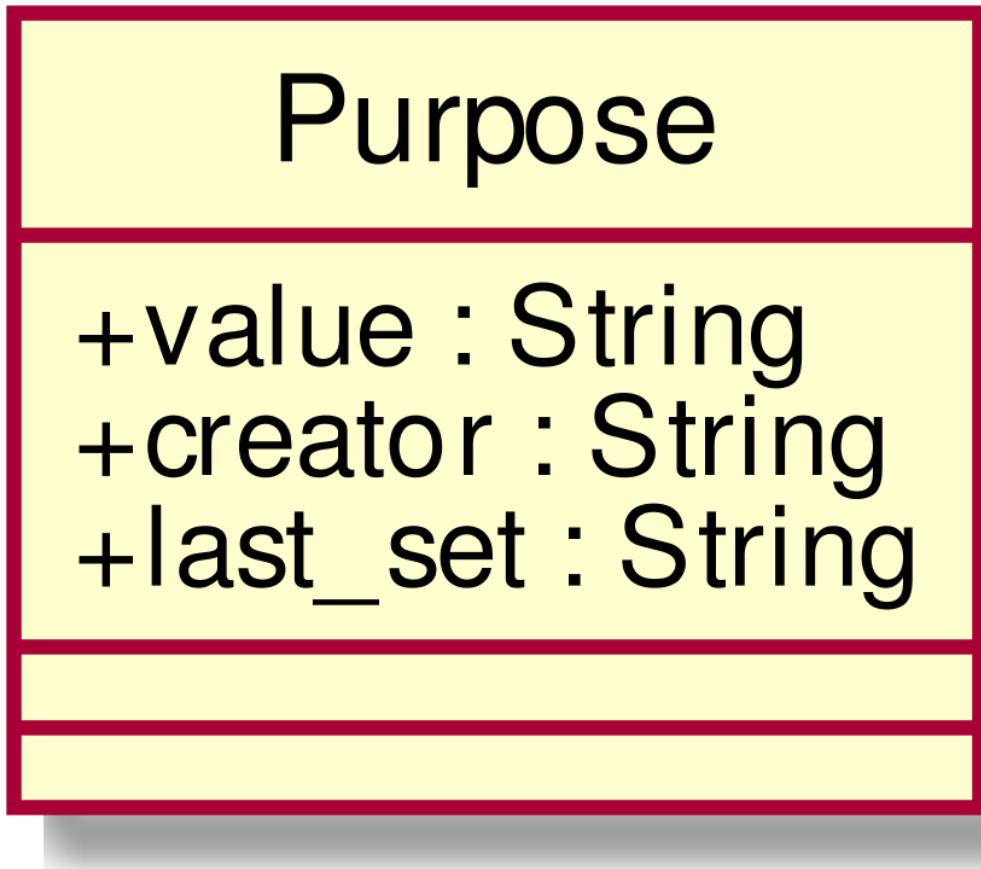


Figura 48: Back-end::Notifications::Purpose

+ creator: String

Attributo contenente l'id del creatore dello scopo del canale Slack;

+ last_set: String

Attributo contenente un unix timestamp relativo all'ultima modifica;

Topic

- **Nome:** Topic;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare e organizzare i dati relativi al Topic di un canale Slack;
- **Utilizzo:** fornisce gli attributi del topic di un canale Slack. Il Topic di un canale Slack è l'oggetto o tema delle discussioni in esso. Per la relativa documentazione, consultare questa pagina <https://api.slack.com/methods/channels.setTopic>;
- **Attributi:**
 - + value: String
Attributo contenente il valore del Topic;

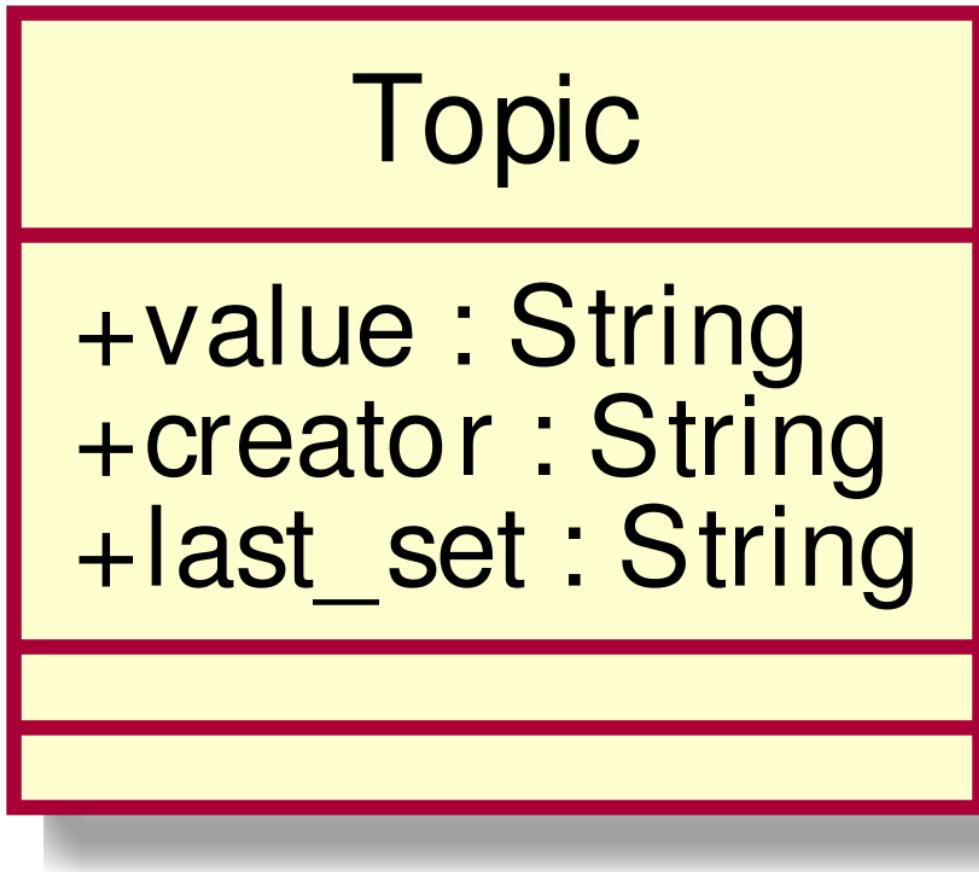


Figura 49: Back-end::Notifications::Topic

+ creator: String

Attributo contenente l'id del creatore del topic del canale Slack;

+ last_set: String

Attributo contenente un unix timestamp relativo all'ultima modifica;

Back-end::Rules

Package che contiene le componenti necessarie alla gestione delle impostazioni (direttive) dell'assistente virtuale.

Classi

RulesDAO

- **Nome:** RulesDAO;
- **Tipo:** Interface;
- **Descrizione:** questa classe si occupa di astrarre le modalità di accesso al database per questo microservizio, contenente le Rule;

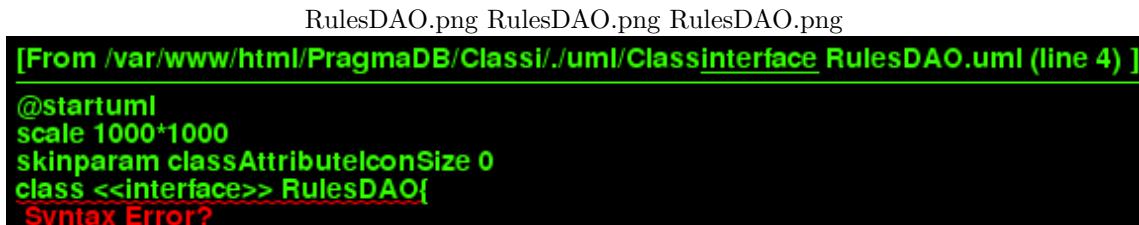


Figura 50: Back-end::Rules:: RulesDAO

- **Utilizzo:** fornisce a `RuleService` un meccanismo per accedere al database, contenente le `Rule`, senza conoscerne le modalità di implementazioni e di persistenza del database. A partire da un identificativo, permette operazioni di lettura, scrittura e rimozione delle `Rule`;

- **Attributi:**

- DB: `AWS::DynamoDB`
Attributo che permette di contattare il database contenente le direttive;

- **Metodi:**

+ `addRule(rule: Rule): ErrorObservable`

Metodo che permette di aggiungere una `Rule` al database. L'`Observable` restituito non riceverà alcun valore, ma verrà completato in caso di aggiunta della `Rule` avvenuta con successo. In caso di errore durante l'aggiunta della `Rule`, gli `Observer` interessati verranno notificati tramite la chiamata del loro metodo `error()` con i dati relativi all'errore verificatosi;

Parametri:

* `rule: Rule`

Parametro contenente la `Rule` da aggiungere;

+ `getRulesList(): RuleObservable`

L'`Observable` restituito manderà agli `Observer` le direttive ottenute, una alla volta, e poi chiama il loro metodo `complete`. Nel caso in cui si verifichi un errore, gli `Observer` iscritti verranno notificati tramite la chiamata del loro metodo `error` con i dati relativi all'errore verificatosi;

+ `removeRule(id: int): ErrorObservable`

Metodo che permette di rimuovere una `Rule` dal database. L'`Observable` restituito non riceverà alcun valore, ma verrà completato in caso di rimozione della `Rule` avvenuta con successo. In caso di errore durante la rimozione della `Rule`, gli `Observer` interessati verranno notificati tramite la chiamata del loro metodo `error()` con i dati relativi all'errore verificatosi;

Parametri:

* `id: int`

Parametro contenente l'id della `Rule`;

+ `getRule(id: int): RuleObservable`

Metodo che permette di ottenere una `Rule` a partire dal suo Id. L'`Observable` restituito riceverà l'oggetto rappresentante tale `Rule`, e verrà completato. Nel caso in cui la `Rule` richiesta non sia presente nel database, gli `Observer` interessati non riceveranno alcun valore, ma verranno notificati tramite la chiamata del loro metodo `error()`;

Parametri:

* `id: int`

Parametro contenente l'id della `Rule` da recuperare;

```
+ updateRule(rule: Rule): ErrorObservable
```

Metodo che permette di aggiornare una Rule. L'Observable restituito non riceverà alcun valore, ma verrà completato in caso di aggiornamento della Rule avvenuta con successo. In caso di errore durante l'aggiornamento della Rule, gli Observer interessati verranno notificati tramite la chiamata del loro metodo `error()` con i dati relativi all'errore verificatosi;

Parametri:

```
* rule: Rule
```

Parametro contenente la Rule;

TasksDAO

TasksDAO.png TasksDAO.png TasksDAO.png

```
[From /var/www/html/PragmaDB/Classi/.uml/Classinterface TasksDAO.uml (line 4)]
@startuml
scale 1000*1000
skinparam classAttributeIconSize 0
class <<interface>> TasksDAO{
    Syntax Error?
```

Figura 51: Back-end::Rules:: TasksDAO

- **Nome:** TasksDAO;
- **Tipo:** Interface;
- **Descrizione:** questa classe si occupa di astrarre le modalità d'accesso al database per questo microservizio, contenente i Task;
- **Utilizzo:** fornisce a RuleService un meccanismo per accedere al database, contenente i compiti da applicare a certeRule, senza conoscerne le modalità di implementazioni e di persistenza del database. A partire da un identificativo, permette operazioni di lettura, scrittura e rimozione dei Task;
- **Metodi:**

```
+ getTaskList(): TaskObservable
```

Metodo che permette di ottenere la lista dei compiti. L'Observable restituito manderà agli Observer i compiti ottenuti, una alla volta, e poi chiama il loro metodo `complete`. Nel caso in cui si verifichi un errore, gli Observer iscritti verranno notificati tramite la chiamata del loro metodo `error` con i dati relativi all'errore verificatosi;

```
+ addTask(task: Task): ErrorObservable
```

Metodo che permette di aggiungere un Task al database.

L'Observable restituito non riceverà alcun valore, ma verrà completato in caso di aggiunta della funzione avvenuta con successo. In caso di errore durante l'aggiunta del Task, gli Observer interessati verranno notificati tramite la chiamata del loro metodo `error()` con i dati relativi all'errore verificatosi;

Parametri:

```
* task: Task
```

Parametro contenente il compito;

```
+ removeTask(id: int): ErrorObservable
```

Metodo che permette di rimuovere un compito dal DB a partire dal suo id.

L'Observable restituito non riceverà alcun valore, ma verrà completato in caso di rimozione del Task avvenuta con successo. In caso di errore durante la rimozione del Task, gli Observer interessati verranno notificati tramite la chiamata del loro metodo

error() con i dati relativi all'errore verificatosi;

Parametri:

* id: int

Parametro contenente l'id della funzione;

+ getTask(id: int): ErrorObservable

Metodo che permette di ottenere una funzione a partire dal suo id.

L'Observable restituito riceverà l'oggetto rappresentante tale Function, e verrà completato. Nel caso in cui la funzione richiesta non sia presente nel database, gli Observer interessati non riceveranno alcun valore, ma verranno notificati tramite la chiamata del loro metodo **error()**;

Parametri:

* id: int

Parametro contenente l'id della funzione;

+ updateTask(task: Task): ErrorObservable

Metodo che permette di aggiornare un compito. L'Observable restituito non riceverà alcun valore, ma verrà completato in caso di aggiunta del Task con successo. In caso di errore durante l'aggiunta del Task, gli Observer interessati verranno notificati tramite la chiamata del loro metodo **error()** con i dati relativi all'errore verificatosi;

Parametri:

* task: Task

Parametro contenente il compito;

Rule

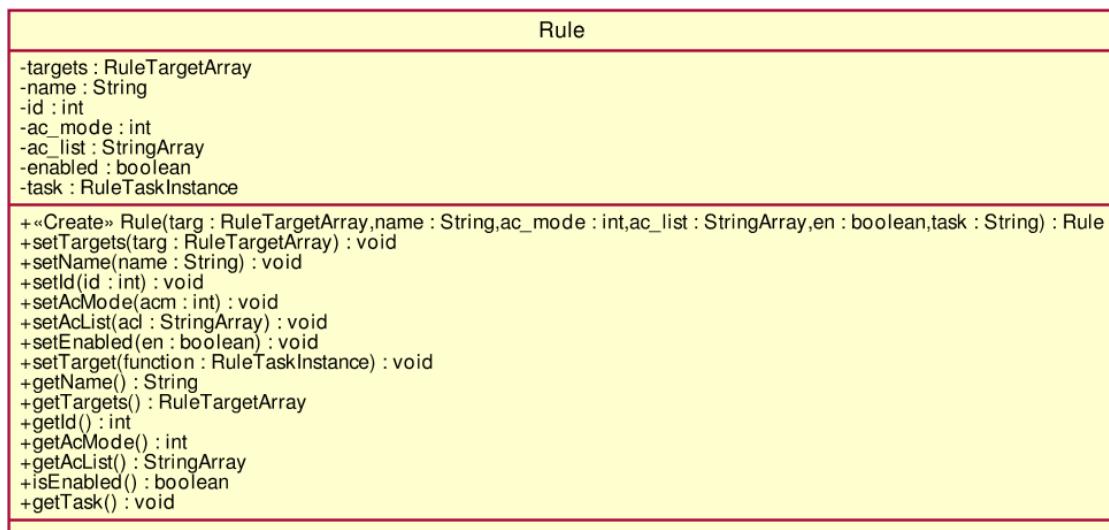


Figura 52: Back-end::Rules::Rule

- **Nome:** Rule;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare e organizzare i dati relativi ad una Rule, ovvero una direttiva definita da un amministratore;
- **Utilizzo:** fornisce i metodi getter e setter per i parametri relativi ad una direttiva, i quali dovranno essere memorizzati nel database per questo microservizio. Tramite il metodo **setTask**, è soggetta ad una setter-based dependency injection che ha come oggetto una

`RuleTaskInstance.`

È utilizzata dalla classe `RulesDAO` e dalle classi che utilizzano quest'ultima;

- **Attributi:**

- `targets: RuleTargetArray`

Attributo contenente i targets della Rule;

- `name: String`

Attributo contenente il nome della Rule;

- `id: int`

Attributo contenente l'id della Rule;

- `ac_mode: int`

Attributo contenente 0 per positivo, 1 per negativo;

- `ac_list: StringArray`

Attributo contenente l'array di stringhe di id amministratori abilitati/disabilitati in base a valore di `ac_mode`;

- `enabled: boolean`

Attributo contenente un valore che dice se la Rule è abilitata o meno;

- `task: RuleTaskInstance`

Attributo contenente il compito della Rule;

- **Metodi:**

+ <<Create>> `Rule(targ: RuleTargetArray, name: String, ac_mode: int, ac_list: StringArray, en: boolean, task: String): Rule`

Metodo che permette di instanziare un oggetto `Rule` a partire da un nome, una lista dei targets un `ac_mode`, un `ac_list`, un compito da applicare e un valore booleano per abilitarla o meno;

Parametri:

* `targ: RuleTargetArray`

Parametro contenente l'array dei targets da assegnare alla Rule;

* `name: String`

Parametro contenente il nome da assegnare alla Rule;

* `ac_mode: int`

Parametro contenente l'`ac_mode` da assegnare alla Rule;

* `ac_list: StringArray`

Parametro contenente l'array di id degli amministratori abilitati/disabilitati da assegnare alla Rule;

* `en: boolean`

Parametro contenente il valore booleano da assegnare alla Rule per abilitarla o meno;

* `task: String`

Parametro contenente il compito da assegnare alla Rule;

+ `setTargets(targ: RuleTargetArray): void`

Metodo che permette di passare un Array contenente i targets per la `Rule`;

Parametri:

* `targ: RuleTargetArray`

Parametro contenente l'array dei targets da passare;

+ **setName(name: String): void**

Metodo che permette di passare il nome per la Rule;
Parametri:

* name: String

Parametro contenente il nome della Rule da passare;

+ **setId(id: int): void**

Metodo che permette di passare l'id per la Rule;
Parametri:

* id: int

Parametro contenente l'id da passare;

+ **setAcMode(acm: int): void**

Metodo che permette di passare l'ac_mode per la Rule;
Parametri:

* acm: int

Parametro contenente l'ac_mode da passare;

+ **setAcList(acl: StringArray): void**

Metodo che permette di passare gli id degli amministratori abilitati/disabilitati per la Rule;

Parametri:

* acl: StringArray

Parametro contenente l'ac_list da passare;

+ **setEnabled(en: boolean): void**

Metodo che permette di passare un valore da impostare ad enabled per la Rule;
Parametri:

* en: boolean

Parametro contenente il valore booleano da passare;

+ **setTarget(function: RuleTaskInstance): void**

Metodo che permette di passare il compito da applicare per la Rule;
Parametri:

* function: RuleTaskInstance

Parametro contenente la function da passare;

+ **getName(): String**

Metodo che permette di ottenere il nome della Rule;

+ **getTargets(): RuleTargetArray**

Metodo che permette di ottenere l'array contenente i targets della Rule;

+ **getId(): int**

Metodo che permette di ottenere l'id della Rule;

+ **getAcMode(): int**

Metodo che permette di ottenere l'ac_mode della Rule;

+ **getAcList(): StringArray**

Metodo che permette di ottenere la lista degli id degli amministratori abilitati/disabilitati della Rule;

```
+ isEnabled(): boolean
```

Metodo che permette di capire se la Rule è abilitata o meno;

```
+ getTask(): void
```

Metodo che permette di ottenere la Task applicata dalla Rule;

RulesDAODynamoDB

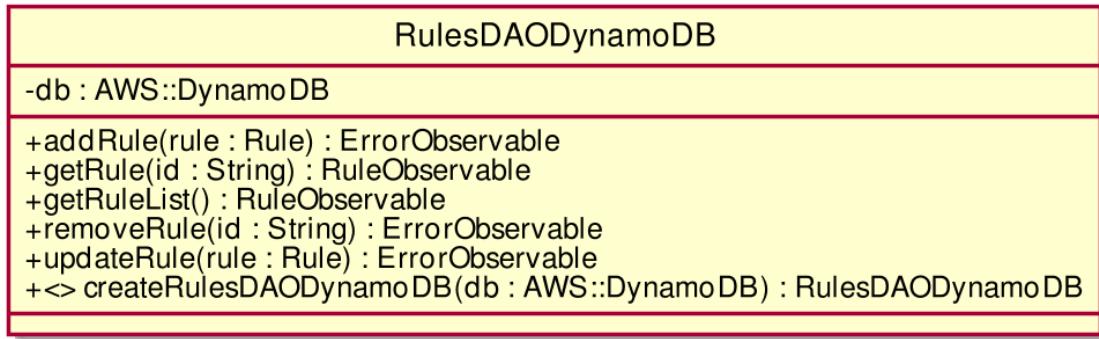


Figura 53: Back-end::Rules::RulesDAODynamoDB

- **Nome:** RulesDAODynamoDB;
- **Tipo:** Class;
- **Descrizione:** classe che si occupa di implementare l’interfaccia RulesDAO, utilizzando un database DynamoDB come supporto per la memorizzazione dei dati;
- **Utilizzo:** implementa i metodi dell’interfaccia RulesDAO interrogando un database DynamoDB. Utilizza AWS::DynamoDB::DocumentClient per l’accesso al database. La dependency injection dell’oggetto AWS::DynamoDB viene fatta utilizzando il costruttore;
- **Attributi:**
 - db: AWS::DynamoDB
Attributo contenente un riferimento al modulo di Node.js utilizzato per l’accesso al database DynamoDB contenente la tabella degli utenti;
- **Metodi:**
 - + addRule(rule: Rule): ErrorObservable
Implementazione del metodo definito nell’interfaccia RulesDAO. Utilizza il metodo put del DocumentClient per aggiungere la Rule al database;
Parametri:
* rule: Rule
Parametro contenente la Rule da aggiungere;
 - + getRule(id: String): RuleObservable
Implementazione del metodo definito nell’interfaccia RulesDAO. Utilizza il metodo get del DocumentClient per ottenere i dati relativi ad uno Rule dal database;
Parametri:
* id: String
Parametro contenente l’id della Rule da recuperare;

```
+ getRuleList(): RuleObservable
```

Implementazione del metodo dell'interfaccia RulesDAO. Utilizza il metodo scan del DocumentClient per ottenere la lista delle Rule dal database;

```
+ removeRule(id: String): ErrorObservable
```

Implementazione del metodo dell'interfaccia RulesDAO. Utilizza il metodo delete del DocumentClient per eliminare una Rule dal database;

Parametri:

```
* id: String
```

Parametro contenente l'id della Rule;

```
+ updateRule(rule: Rule): ErrorObservable
```

Implementazione del metodo dell'interfaccia RulesDAO. Utilizza il metodo update del DocumentClient per aggiornare i dati relativi ad una Rule presente all'interno del database;

Parametri:

```
* rule: Rule
```

Parametro contenente la Rule da aggiornare;

```
+ <> createRulesDAODynamoDB(db: AWS::DynamoDB): RulesDAODynamoDB
```

Constructor della classe RulesDAODynamoDB. Permette di effettuare la dependency injection di AWS::DynamoDB;

Parametri:

```
* db: AWS::DynamoDB
```

Parametro contenente un riferimento al modulo di Node.js da utilizzare per l'accesso al database DynamoDB contenente la tabella delle rule;

RulesService

RulesService
<pre>-rules : RulesDAO -task : TasksDAO</pre>
<pre>+addRule(event : LambdaEvent,context : LambdaContext) : void +deleteRule(event : LambdaEvent,context : LambdaContext) : void +updateRule(event : LambdaEvent,context : LambdaContext) : void +getRule(event : LambdaEvent,context : LambdaContext) : void +getRuleList(event : LambdaEvent,context : LambdaContext) : void +getTaskList(event : LambdaEvent,context : LambdaContext) : void +getTask(event : LambdaEvent,context : LambdaContext) : void +«Create» createRulesService(task : TasksDAO,rules : RulesDAO) : RulesService</pre>

Figura 54: Back-end::Rules::RulesService

- **Nome:** RulesService;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di realizzare il microservizio Rules e di interagire con il database delle direttive;
- **Utilizzo:** fornisce i metodi che implementano le lambda function necessarie alla gestione delle Rule e relative Task. Questa classe non interagisce direttamente con il database, ma fa

utilizzo di `TasksDAO` e `RulesDAO`, le quali nascondono i meccanismi di accesso e persistenza dei dati nel database;

- **Attributi:**

- `rules: RulesDAO`

Attributo che permette di contattare il `RulesDAO`, il quale permette l'accesso al database delle `Rule`;

- `task: TasksDAO`

Attributo che permette di contattare il `TasksDAO`, il quale permette di accedere al database delle `Task`;

- **Metodi:**

- + `addRule(event: LambdaEvent, context: LambdaContext): void`

Metodo che implementa la lambda function che si occupa di aggiungere una `Rule`;

Parametri:

- * `event: LambdaEvent`

Parametro contenente, all'interno del campo `body` sotto forma di stringa in formato JSON, un oggetto `Rule` contenente tutti i dati relativi ad una `Rule` da inserire;

- * `context: LambdaContext`

Parametro utilizzato dalle lambda function per inviare la risposta. Il `body` del `LambdaResponse`, parametro del metodo `LambdaContext::succeed`, conterrà una stringa vuota e il risultato di questa operazione sarà deducibile dal valore dell'attributo di `LambdaResponse::statusCode`;

- + `deleteRule(event: LambdaIdEvent, context: LambdaContext): void`

Metodo che implementa la lambda function che si occupa di eliminare una `Rule`;

Parametri:

- * `event: LambdaIdEvent`

Parametro contenente, all'interno del campo `pathParameters`, l'identificativo della `Rule` che si vuole eliminare;

- * `context: LambdaContext`

Parametro utilizzato dalle lambda function per inviare la risposta. Il `body` del `LambdaResponse`, ottenuto dal metodo `LambdaContext::succeed`, conterrà una stringa vuota e il risultato di questa operazione sarà deducibile dal valore dell'attributo `LambdaResponse::statusCode`;

- + `updateRule(event: LambdaIdEvent, context: LambdaContext): void`

Metodo che implementa la lambda function che si occupa di aggiornare una `Rule`;

Parametri:

- * `event: LambdaIdEvent`

Parametro contenente all'interno del campo `body`, sotto forma di stringa in formato JSON, un oggetto di tipo `Rule` contenente i dati da aggiornare e, all'interno del campo `pathParameters`, l'identificativo della `Rule` da modificare;

- * `context: LambdaContext`

Parametro utilizzato dalle lambda function per inviare la risposta. Il `body` del `LambdaResponse`, ottenuto dal metodo `LambdaContext::succeed`, conterrà una stringa vuota e il risultato di questa operazione sarà deducibile dal valore dell'attributo `LambdaResponse::statusCode`;

- + `getRule(event: LambdaIdEvent, context: LambdaContext): void`

Metodo che implementa la lambda function che si occupa di ottenere una `Rule`;

Parametri:

```

* event: LambdaIdEvent
Parametro contenente, all'interno del campo pathParameters, l'identificativo della Rule della quale si vogliono ottenere i dati;

* context: LambdaContext
Parametro utilizzato dalle lambda function per inviare la risposta. Il body del LambdaResponse, ottenuto dal metodo LambdaContext::succeed, conterrà, sotto forma di stringa in formato JSON, un oggetto di tipo Rule, contenente i dati relativi alla Rule ritornata;

+ getRuleList(event: LambdaEvent, context: LambdaContext): void
Metodo che implementa la lambda function che si occupa di ottenere l'array delle Rule;
Parametri:

* event: LambdaEvent
Parametro contenente, all'interno del campo body, una stringa vuota;

* context: LambdaContext
Parametro utilizzato dalle lambda function per inviare la risposta. Il body del LambdaResponse, parametro del metodo LambdaContext::succeed, conterrà, sotto forma di una stringa in formato JSON, l'array delle Rule disponibili;

+ getTaskList(event: LambdaEvent, context: LambdaContext): void
Metodo che implementa la lambda function che si occupa ottenere l'array delle Task disponibili;
Parametri:

* event: LambdaEvent
Parametro contenente, all'interno del campo body, una stringa vuota;

* context: LambdaContext
Parametro utilizzato dalle lambda function per inviare la risposta. Il body del LambdaResponse, ottenuto dal metodo LambdaContext::succeed, conterrà, sotto forma di una stringa in formato JSON, un Array di oggetti di tipo Function;

+ getTask(event: LambdaIdEvent, context: LambdaContext): void
Metodo che implementa la lambda function che si occupa di ottenere una Task;
Parametri:

* event: LambdaIdEvent
Parametro contenente, all'interno del campo pathParameters, l'identificativo della Rule della quale si vuole ottenere la Task;

* context: LambdaContext
Parametro utilizzato dalle lambda function per inviare la risposta. Il body del LambdaResponse, parametro del metodo LambdaContext::succeed, conterrà, sotto forma di una stringa in formato JSON, un oggetto di tipo RuleTaskInstance, contenente i dati relativi alla Task ritornata;

+ <<Create>> createRulesService(task: TasksDAO, rules: RulesDAO): RulesService
Metodo che permette di creare un RulesService. Permette la dependency injection avente come oggetti un RulesDAO e TasksDAO;
Parametri:

* task: TasksDAO
Attributo contenente il TasksDAO;

* rules: RulesDAO
Attributo contenente il RulesDAO;

```

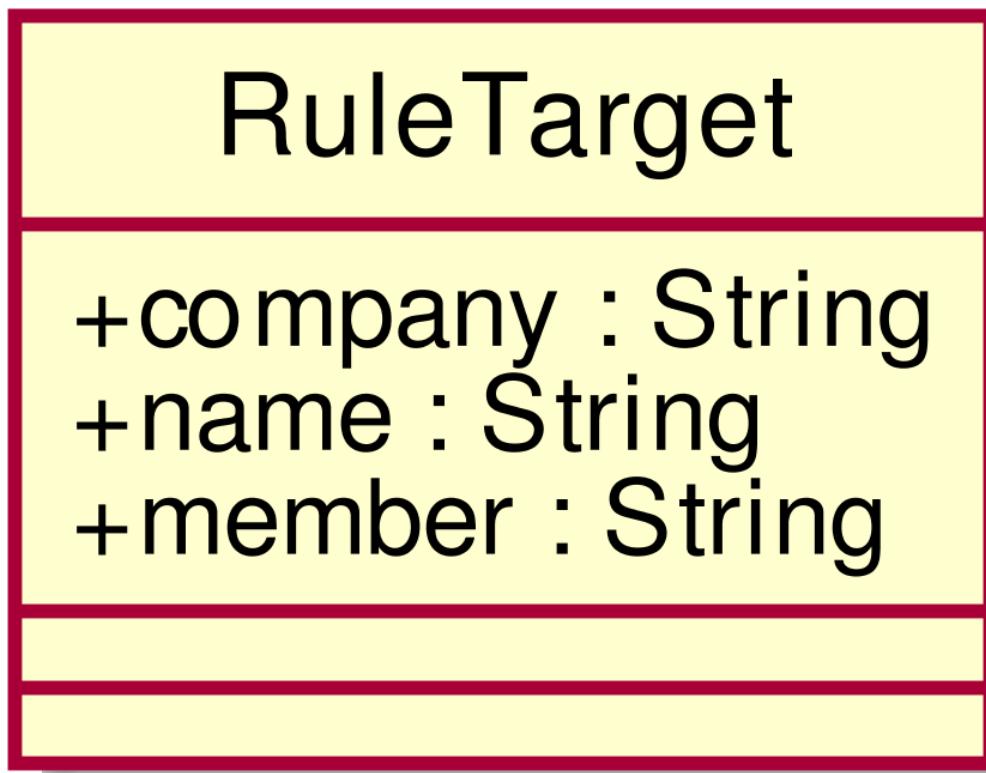


Figura 55: Back-end::Rules::RuleTarget

RuleTarget

- **Nome:** RuleTarget;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare e organizzare i dati relativi a un target di una Rule, ovvero la persona alla quale è indirizzata quest'ultima;
- **Utilizzo:** fornisce gli attributi di un target di una Rule che dovranno essere memorizzati nel database per questo microservizio. Viene utilizzata dalla classe Rule per definire un Array di Target ai quali applicare la sua funzione;
- **Attributi:**
 - + company: String
Attributo contenente il nome dell'azienda;
 - + name: String
Attributo contenente il nome del target;
 - + member: String
Attributo contenente il nome del membro dell'azienda interessato dalla direttiva;

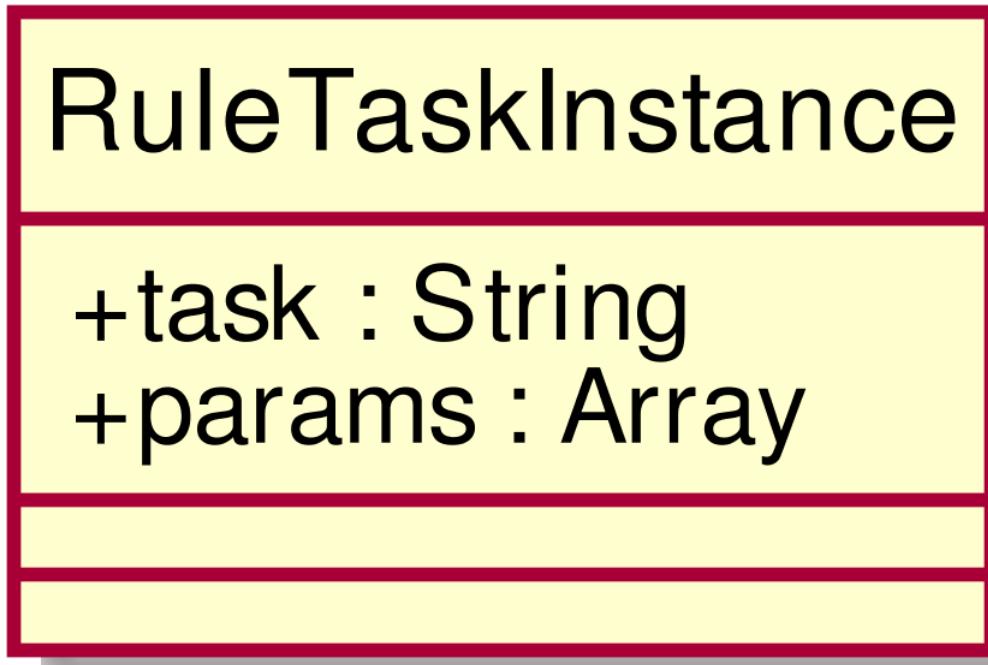


Figura 56: Back-end::Rules::RuleTaskInstance

RuleTaskInstance

- **Nome:** RuleTaskInstance;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare i dati relativi al compito di una direttiva;
- **Utilizzo:** fornisce un meccanismo per specificare la modifica di comportamento del sistema in seguito all'applicazione di una determinata direttiva. ;
- **Attributi:**
 - + task: String
Attributo contenente il task, ovvero una funzione, da applicare;
 - + params: Array
Attributo contenente l'array dei parametri relativi alla funzione;

Task

- **Nome:** Task;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare la funzione che dovrà essere applicata ad una Rule;

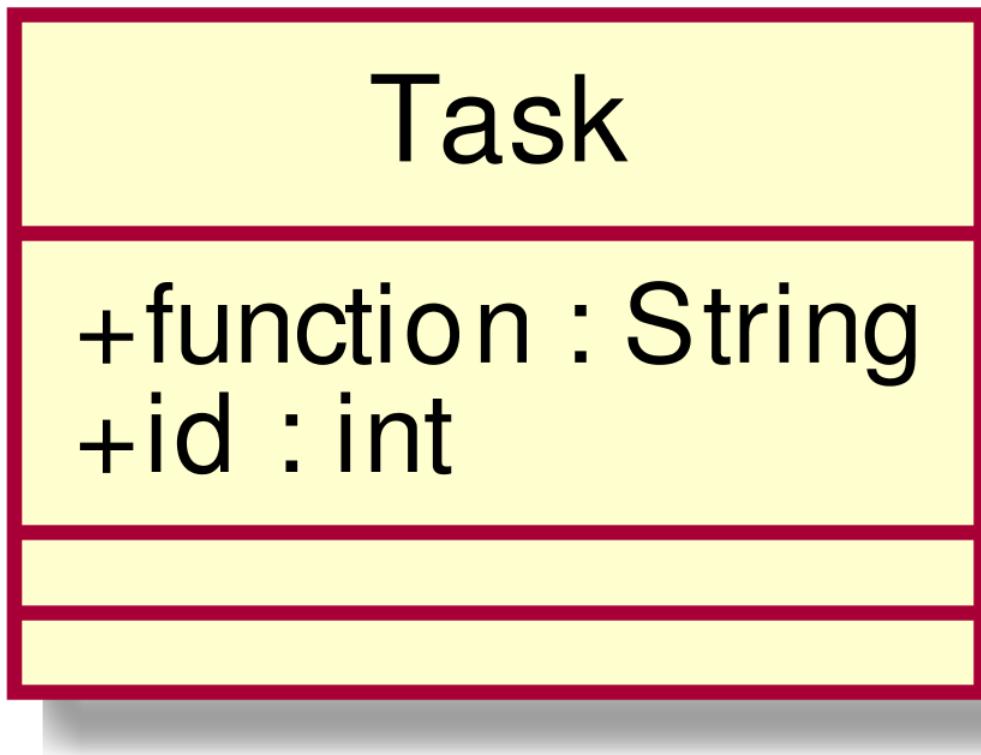


Figura 57: Back-end::Rules::Task

- **Utilizzo:** fornisce gli attributi delle funzioni (ovvero dei compiti che una certa Rule svolge) che dovranno essere applicate a certe Rule, i quali dovranno essere memorizzati nel database per questo microservizio;

- **Attributi:**

+ function: String

Attributo contenente la funzione da applicare;

+ type: String

Attributo contenente il tipo della funzione;

TasksDAODynamoDB

- **Nome:** TasksDAODynamoDB;
- **Tipo:** Class;
- **Descrizione:** classe che si occupa di implementare l’interfaccia TasksDAO, utilizzando un database DynamoDB come supporto per la memorizzazione dei dati;
- **Utilizzo:** implementa i metodi dell’interfaccia TasksDAO interrogando un database DynamoDB. Utilizza AWS::DynamoDB::DocumentClient per l’accesso al database. La dependency injection dell’oggetto AWS::DynamoDB viene fatta utilizzando il costruttore;
- **Attributi:**

TasksDAO ^{DynamoDB}
-db : AWS::DynamoDB
+addFunction(fun : Function) : ErrorObservable
+getFunction(id : String) : TaskObservable
+getFunctionList() : TaskObservable
+removeFunction(id : String) : ErrorObservable
+updateFunction(fun : Function) : ErrorObservable
+<> createFunctionsDAO ^{DynamoDB} (db : AWS::DynamoDB) : TasksDAO ^{DynamoDB}

Figura 58: Back-end::Rules::TasksDAO^{DynamoDB}

- db: AWS::DynamoDB

Attributo contenente un riferimento al modulo di Node.js utilizzato per l'accesso al database DynamoDB contenente la tabella degli utenti;

- **Metodi:**

+ addFunction(fun: Function): ErrorObservable

Implementazione del metodo definito nell'interfaccia FunctionsDAO. Utilizza il metodo put del DocumentClient per aggiungere la funzione al database;
Parametri:

* fun: Function

Parametro contenente la funzione;

+ getFunction(id: String): TaskObservable

Implementazione del metodo definito nell'interfaccia FunctionsDAO. Utilizza il metodo get del DocumentClient per ottenere i dati relativi ad una Function dal database;
Parametri:

* id: String

Parametro contenente l'id della funzione;

+ getFunctionList(): TaskObservable

Implementazione del metodo dell'interfaccia FunctionsDAO. Utilizza il metodo scan del DocumentClient per ottenere la lista delle funzioni dal database;

+ removeFunction(id: String): ErrorObservable

Implementazione del metodo dell'interfaccia FunctionsDAO. Utilizza il metodo delete del DocumentClient per eliminare una funzione dal database;
Parametri:

* id: String

Parametro contenente l'id della funzione;

+ updateFunction(fun: Function): ErrorObservable

Implementazione del metodo dell'interfaccia FunctionsDAO. Utilizza il metodo update del DocumentClient per aggiornare i dati relativi ad una funzione presente all'interno del database;

Parametri:

* fun: Function

Parametro contenente la funzione;

+ <> createFunctionsDAO^{DynamoDB}(db: AWS::DynamoDB) : TasksDAO^{DynamoDB}

Constructor della classe FunctionsDAO^{DynamoDB}. Permette di effettuare la dependency

injection di AWS::DynamoDB;
 Parametri:

* db: AWS::DynamoDB

Parametro contenente un riferimento al modulo di Node.js da utilizzare per l'accesso al database DynamoDB contenente la tabella delle funzioni;

Back-end::STT

Package che include le classi che si occupano di fornire le funzionalità di Speech to text.

Classi

STTModule

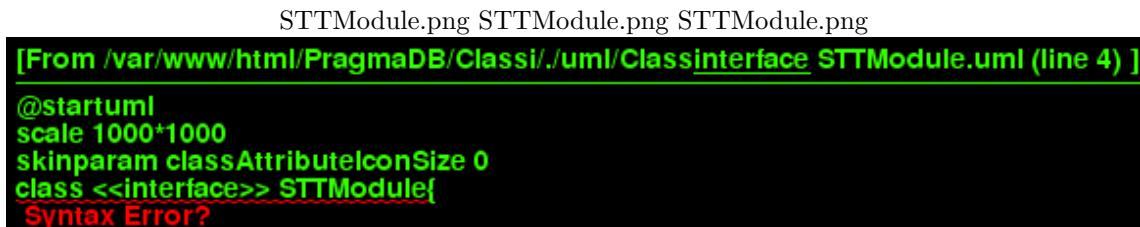


Figura 59: Back-end::STT:: STTModule

- **Nome:** STTModule;
- **Tipo:** Interface;
- **Descrizione:** classe che definisce l'interfaccia dei moduli utilizzati per le operazioni di Speech-To-Text (STT);
- **Utilizzo:** fornisce l'interfaccia che deve essere implementata dai moduli che permettono l'accesso alle funzionalità di STT;
- **Metodi:**

+ speechToText(audio: Buffer, type: String): StringPromise

Questo metodo permette di ricavare in modo asincrono il testo da un file audio. Viene utilizzato da VirtualAssistantAPI, il quale invierà il testo ottenuto dall'invocazione di questo metodo all'assistente virtuale. Restituisce una promise che verrà soddisfatta con una stringa contenente il testo estratto;

Parametri:

* audio: Buffer

Buffer contenente l'audio da cui si vuole estrarre il testo;

* type: String

Parametro contenete la descrizione del formato in cui i dati sono memorizzati in audio;

STTWatsonAdapter

- **Nome:** STTWatsonAdapter;
- **Tipo:** Class;

STTWatsonAdapter
-stt : SpeechToTextV1 -stream_buffer : StreamBufferModule
+«Create» createSTTWatsonAdapter(sb : StreamBufferModule,stt : SpeechToTextV1) : STTWatsonAdapter +speechToText(audio : Buffer,type : String) : StringPromise

Figura 60: Back-end::STT::STTWatsonAdapter

- **Descrizione:** questa classe si occupa di convertire l’interfaccia fornita dal servizio di Speech to Text di IBM in una più adatta alle esigenze dell’applicazione, definita da **STTModule**. Facendo da Adapter tra le API del servizio di Speech to Text di IBM (adaptee) e l’interfaccia **STTModule** (target) utilizzata da **APIGateway::VocalAPI**, permette l’interoperabilità tra queste due interfacce;
- **Utilizzo:** fornisce a **STTModule** un meccanismo che permette di interrogare le API del servizio Watson Speech to Text di IBM, in modo da consentire a **APIGateway::VocalAPI** l’utilizzo di quest’ultime utilizzando un’interfaccia distinta e più consona alle proprie esigenze. Per ulteriori informazioni, consultare le documentazioni presenti alle seguenti pagine:
 - <https://www.ibm.com/watson/developercloud/doc/speech-to-text/>;
 - <https://github.com/watson-developer-cloud/node-sdk#speech-to-text>.
;

- **Attributi:**

- **stt:** `SpeechToTextV1`

Attributo contenente il **SpeechToTextV1Module**, creato a partire dai parametri **name** e **password** forniti al costruttore. Viene utilizzato per estrarre il contenuto testuale di un file audio utilizzando il servizio Watson Speech To Text di IBM;

- **stream_buffer:** `StreamBufferModule`

Attributo contenente il **StreamBufferModule** di cui è stata effettuata la dependency injection nel costruttore. Viene utilizzata per creare un **ReadableStream** di node a partire da un buffer;

- **Metodi:**

- + «Create» `createSTTWatsonAdapter(sb: StreamBufferModule, stt: SpeechToTextV1): STTWatsonAdapter`

Costruttore di **STTWatsonAdapter**, che permette di effettuare la dependency injection di **StreamBufferModule** e di **SpeechToTextV1**;

Parametri:

- * **sb:** `StreamBufferModule`

Parametro tramite i quale si effettua la dependency injection di **StreamBufferModule**;

- * **stt:** `SpeechToTextV1`

Parametro tramite il quale viene effettuata la dependency injection di **SpeechToTextV1**.

;

- + `speechToText(audio: Buffer, type: String): StringPromise`

Implementa il metodo **speechToText** contenuto nell’interfaccia;

Parametri:

- * **audio:** `Buffer`

Parametro contenente l’audio dal quale si vuole estrarre il testo;

* type: String

Parametro contenente il formato in cui sono memorizzati i dati all'interno di audio;

Back-end::Utility

Package contenente classi e interfacce, dallo scopo generico, utili ad altri package del back-end.

Classi

MembersDAO

MembersDAO.png MembersDAO.png MembersDAO.png



Figura 61: Back-end::Utility:: MembersDAO

- **Nome:** MembersDAO;
- **Tipo:** Interface;

- **Descrizione:** questa classe si occupa di astrarre le modalità d'accesso ai dati relativi ai membri dell'azienda;
- **Utilizzo:** fornisce i meccanismi necessari per inserire, modificare, rimuovere ed ottenere i dati relativi ai membri dell'azienda;
- **Metodi:**

+ `addMember(member: Member): ErrorObservable`

Metodo che permette di aggiungere un membro;

Parametri:

* `member: Member`

Membro da aggiungere;

+ `removeMember(id: String): ErrorObservable`

Metodo che permette di rimuovere un membro;

Parametri:

* `id: String`

Stringa identificativa del membro che si vuole rimuovere;

+ `updateMember(member: Member): ErrorObservable`

Metodo che permette di modificare un utente inserito in precedenza;

Parametri:

* `member: Member`

Parametro contenente i dati relativi al membro che si vuole modificare;

+ `getMemberList(): MemberObservable`

Metodo che permette di ottenere la lista dei membri inseriti;

+ `getMember(id: String): MemberObservable`

Metodo che permette di ottenere i dati relativi ad un determinato membro;

Parametri:

* `id: String`

Parametro contenente la stringa identificativa del membro del quale si vogliono ottenere i dati;

Error

- **Nome:** Error;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare e organizzare i dati relativi ad un errore che può avvenire;
- **Utilizzo:** fornisce gli attributi necessari a descrivere gli errori che si possono verificare. L'attributo `code` contiene un valore uguale a 0 nel caso non si sia verificato nessun errore, diverso da 0 altrimenti. Nel caso di `code` diverso da 0, l'attributo `msg` contiene una stringa che descrive l'errore verificatosi;

- **Attributi:**

+ `code: String`

Attributo contenente il codice dell'errore;

+ `msg: String[0..1]`

Attributo contenente il messaggio dell'errore;

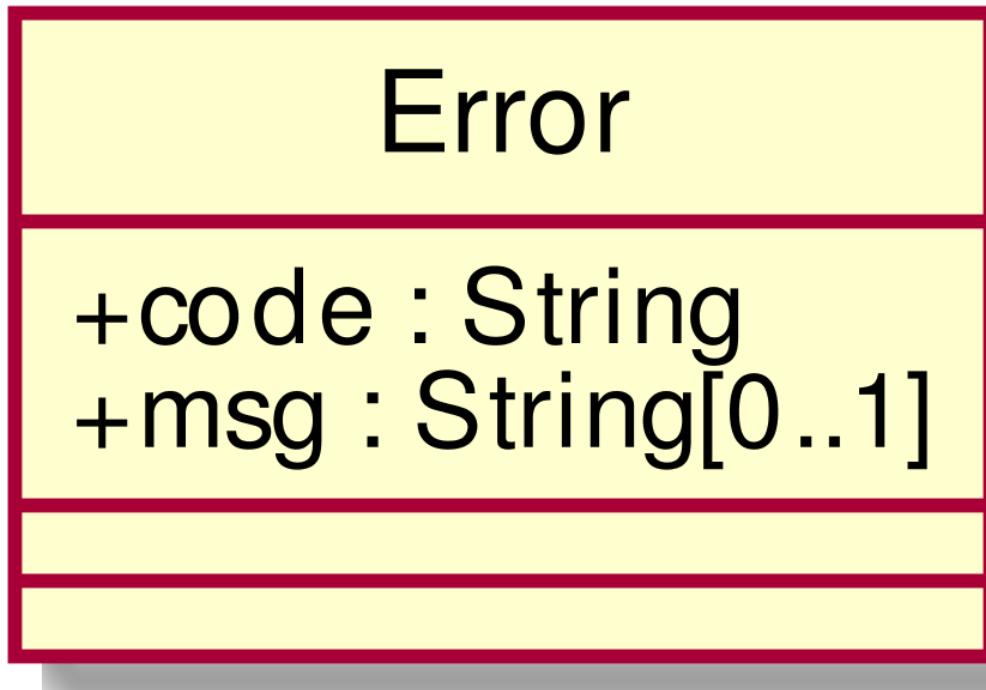


Figura 62: Back-end::Utility::Error

LambdaContext

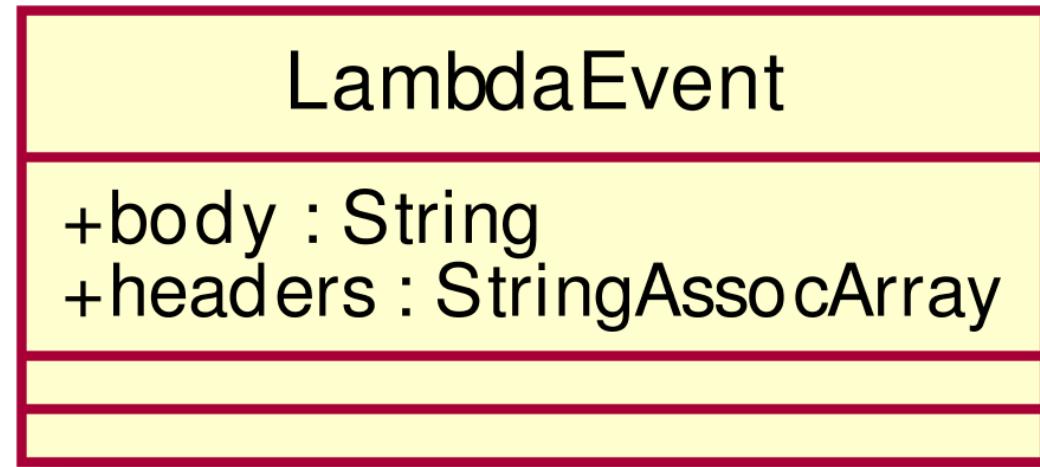
- **Nome:** LambdaContext;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare un oggetto context passato alle lambda function;
- **Utilizzo:** fornisce un metodo che permette di inviare una risposta all'API Gateway. ;
- **Metodi:**

```
+ succeed(res: LambdaResponse): void
Metodo che permette di inviare una risposta all'API Gateway;
Parametri:
    * res: LambdaResponse
        Attributo contenente la risposta da inviare;

+ getRemainingTimeInMillis(): int
DA TOGLIERE DA TOGLIERE DA TOGLIERE DA TOGLIEREDA TOGLIERE
DA TOGLIEREDA TOGLIERE DA TOGLIEREDA TOGLIERE DA TOGLIERE-
DA TOGLIERE DA TOGLIEREDA TOGLIERE DA TOGLIEREDA TOGLIERE DA
TOGLIEREDA TOGLIERE DA TOGLIEREDA TOGLIERE DA TOGLIEREDA TO-
GLIERE DA TOGLIEREDA TOGLIERE DA TOGLIEREDA TOGLIERE DA TO-
GLIERE;
```

**Figura 63:** Back-end::Utility::LambdaContext

LambdaEvent

**Figura 64:** Back-end::Utility::LambdaEvent

- **Nome:** LambdaEvent;
- **Tipo:** Class;
- **Descrizione:** classe che rappresenta l'oggetto event che viene passato alle LambdaFunction dall'API Gateway con l'integrazione Lambda Proxy;
- **Utilizzo:** fornisce i parametri necessari alla lambda function per gestire le richieste che arrivano all'API Gateway;
- **Figlio:** LambdaIdEvent;
- **Attributi:**
 - + body: String
Stringa contenente i dati ricevuti dall'API Gateway. Le singole Lambda Function si dovranno occupare dell'interpretazione di tale stringa nel formato adeguato (ad esempio JSON, testo, ecc.);

+ headers: StringAssocArray

Array associativo di stringhe contenente gli headers HTTP della richiesta ricevuta dall'API Gateway;

LambdaIdEvent

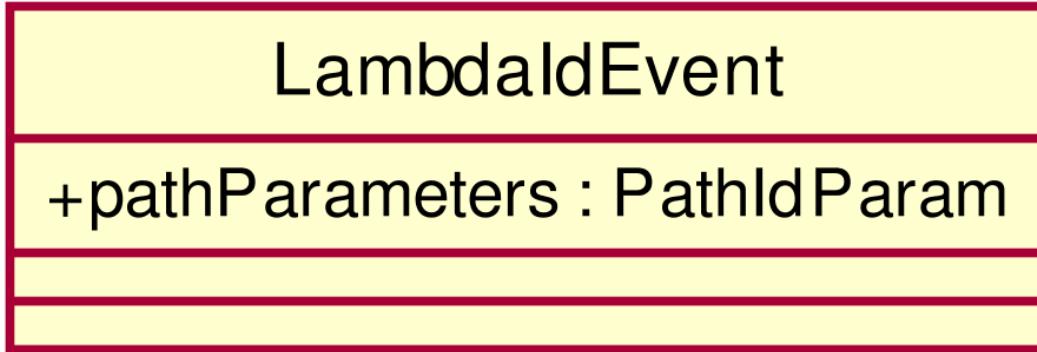


Figura 65: Back-end::Utility::LambdaIdEvent

- **Nome:** LambdaIdEvent;
- **Tipo:** Class;
- **Descrizione:** classe che rappresenta un oggetto event che viene passato alle Lambda function dall'API Gateway con l'integrazione Lambda Proxy;
- **Utilizzo:** eredita da LambdaEvent ed aggiunge l'attributo pathParameters;
- **Padre:** LambdaEvent;
- **Attributi:**
 - + pathParameters: PathIdParam
Parametro contenente l'id dell'utente del quale si vogliono ottenere i dati;

LambdaResponse

- **Nome:** LambdaResponse;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare la risposta di una lambda function;
- **Utilizzo:** fornisce alle lambda function gli attributi necessari per inviare una risposta ad API Gateway;
- **Attributi:**
 - + statusCode: int
Attributo contenente il codice di stato HTTP che dovrà avere la risposta;
 - + headers: StringAssocArray
Attributo contenente l'array associativo nel quale la chiave indica il nome di un header HTTP da mandare nella risposta ed il valore è una stringa contenente il valore di tale header;

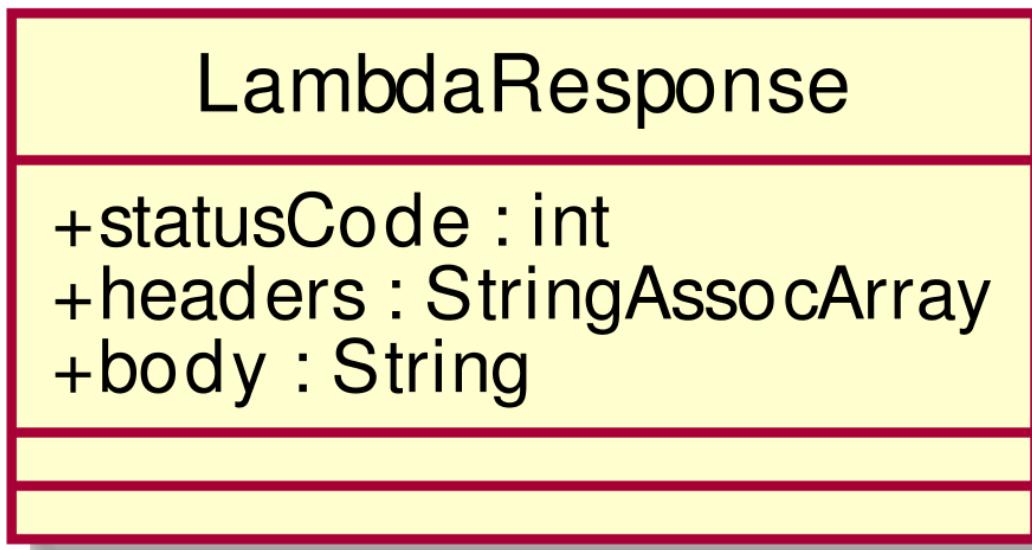


Figura 66: Back-end::Utility::LambdaResponse

+ body: String
Attributo contenente il corpo della risposta;

PathParam

- **Nome:** PathIdParam;
- **Tipo:** Class;
- **Descrizione:** questa classe rappresenta i parametri path di una richiesta caratterizzata da un unico parametro che è un identificativo;
- **Utilizzo:** fornisce l'attributo che contiene l'identificativo della risorsa richiesta;
- **Attributi:**
 - + id: String
Attributo contenente l'id della risorsa richiesta;

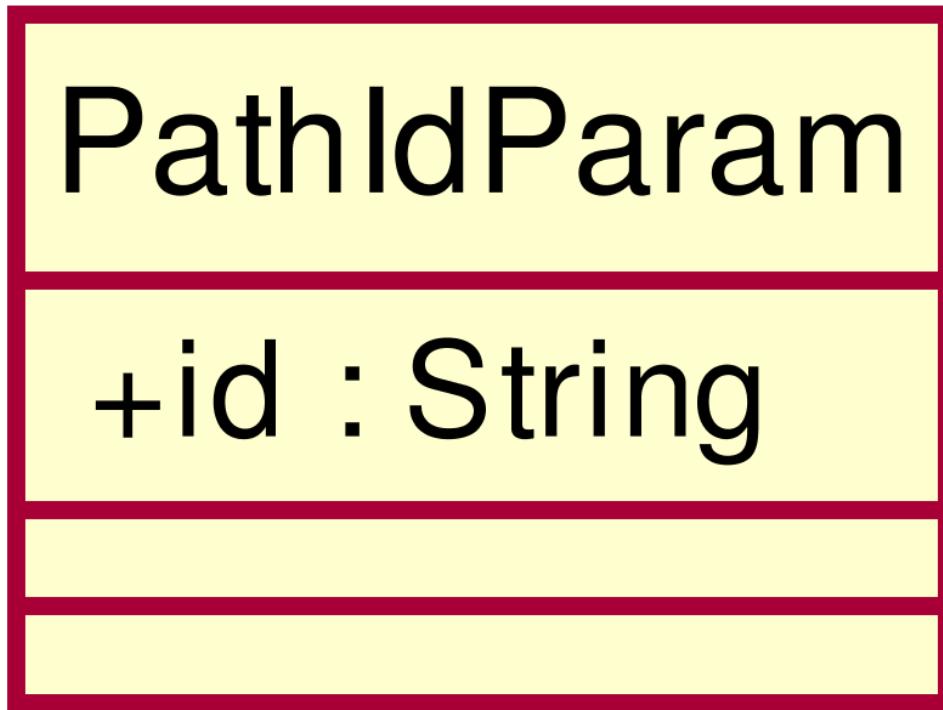


Figura 67: Back-end::Utility::PathIdParam

ProcessingResult

- **Nome:** ProcessingResult;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare il campo result della risposta fornita da api.ai;
- **Utilizzo:** fornisce gli attributi per rappresentare l'oggetto result relativo ad una risposta di api.ai.

Per la relativa documentazione, consultare la pagina <https://docs.api.ai/docs/query#response;>

- **Attributi:**

+ source: String

Attributo contenente la sorgente dalla quale è stata ricavata la risposta. ;

+ resolvedQuery: String

Attributo contenente il testo dell'intents utilizzato per interrogare api.ai;

+ action: String

Attributo contenente l'azione da eseguire;

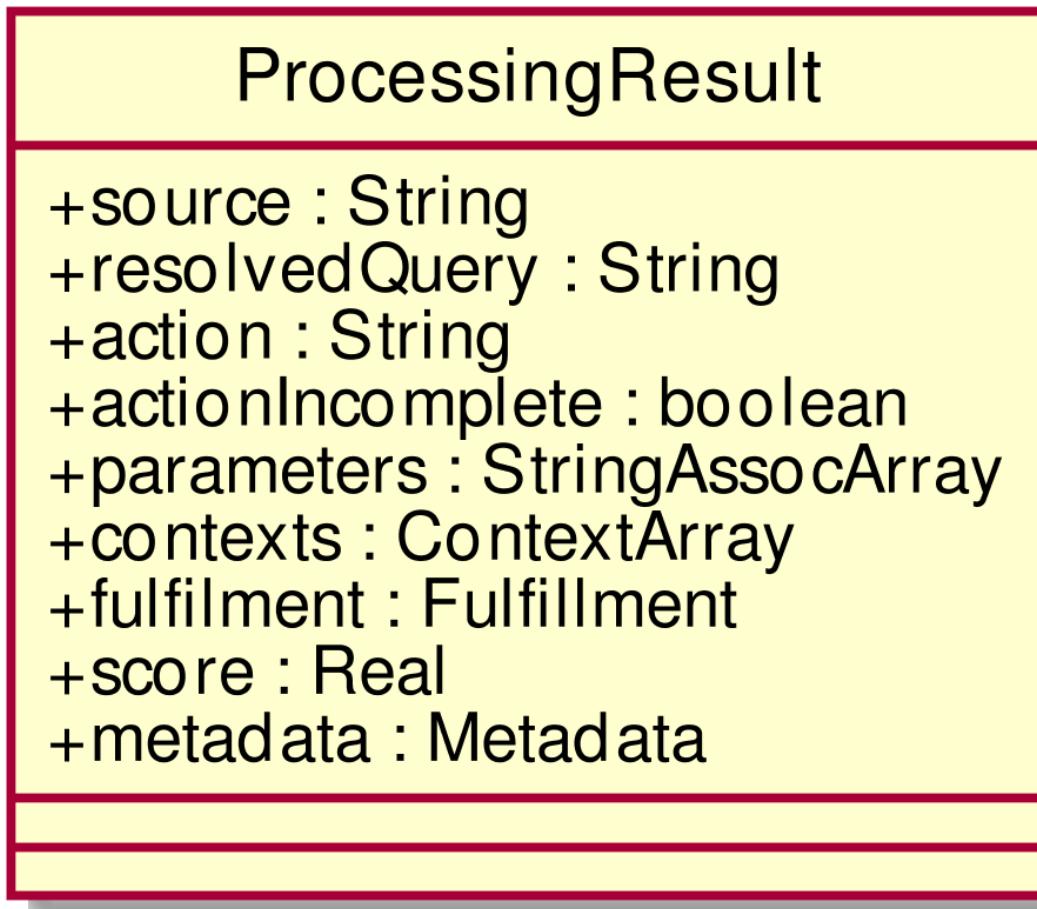


Figura 68: Back-end::Utility::ProcessingResult

+ actionIncomplete: boolean

Attributo contenente un valore booleano che indica se i parametri necessari a far eseguire l'azione sono stati forniti tutti o meno. ;

+ parameters: StringAssocArray

Attributo contenente un oggetto costituito dai parametri necessari per portare a termine l'azione;

+ contexts: ContextArray

Attributo contenente l'array dei context attivi;

+ fulfilment: Fulfillment

Attributo contenente i dati ricevuti dal webhook;

+ score: Real

Attributo contenente un numero, contenuto nell'intervallo [0,1], che indica la sicurezza con la quale è stato trovato il relativo intent;

+ metadata: Metadata

Attributo contenente dati relativi a intents e contexts;

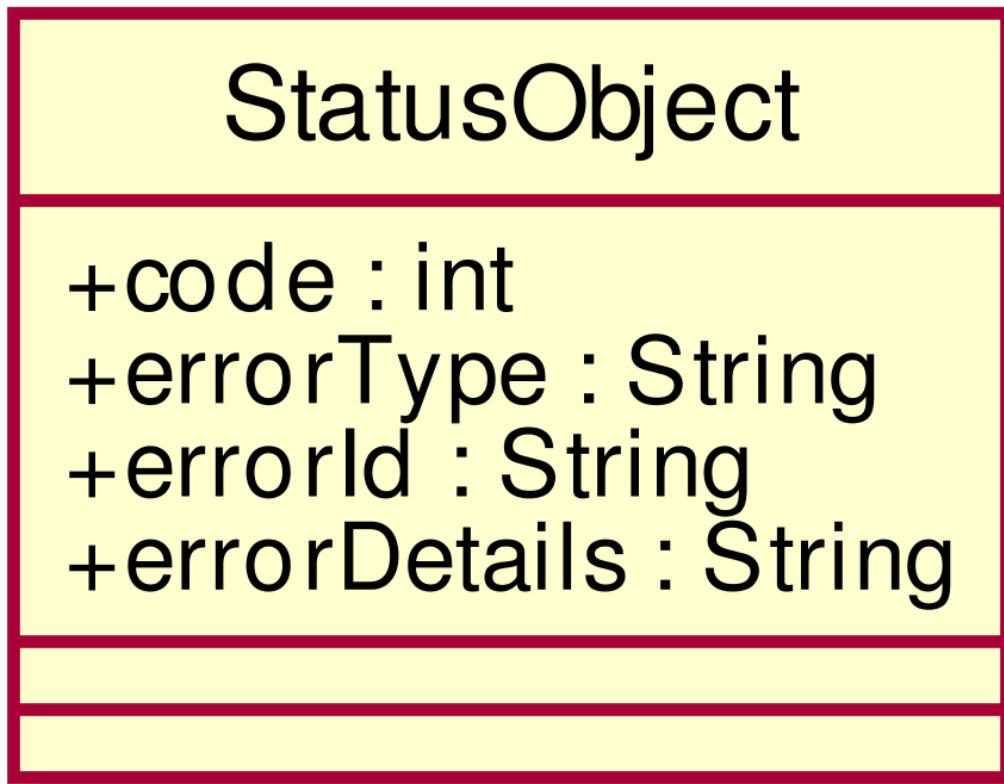


Figura 69: Back-end::Utility::StatusObject

StatusObject

- **Nome:** StatusObject;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare lo status object della richiesta mandata ad api.ai, che indica se quest'ultima ha avuto successo o meno. ;
- **Utilizzo:** fornisce gli attributi per rappresentare l'oggetto status object relativo ad una richiesta mandata ad api.ai.
Per la relativa documentazione, consultare la pagina <https://docs.api.ai/docs/status-object>;
- **Attributi:**
 - `+ code: int`
Attributo contenente il codice di stato HTTP;
 - `+ errorType: String`
Attributo contenente una breve descrizione dell'errore verificatosi. In caso non se ne verifichino, questo attributo avrà valore pari a "success";
 - `+ errorId: String`
Attributo contenente l'id dell'errore verificatosi;

```
+ errorDetails: String
```

Attributo contenente la descrizione dettagliata dell'errore verificatosi. In caso non se ne verifichino, questo attributo non sarà ritornato;

Back-end::VirtualAssistant

Package contenente le componenti del microservizio dell'assistente virtuale.

Classi

AgentsDAO

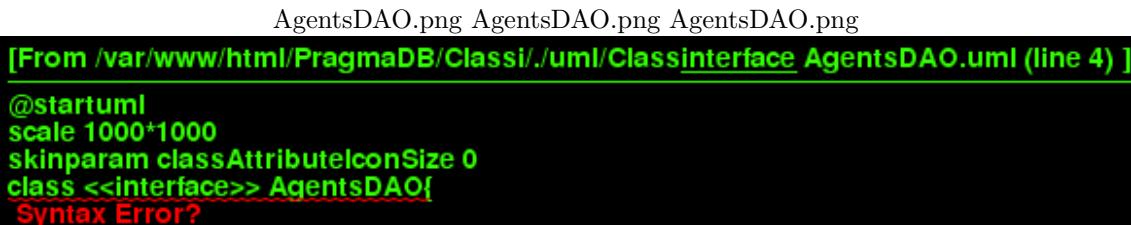


Figura 70: Back-end::VirtualAssistant:: AgentsDAO

- **Nome:** AgentsDAO;
- **Tipo:** Interface;
- **Descrizione:** questa classe si occupa di astrarre le modalità di accesso al database contenente gli Agent disponibili;
- **Utilizzo:** fornisce a `WebhookService` un meccanismo per accedere ai dati relativi agli Agent, senza conoscerne le modalità di implementazioni e di persistenza del database. A partire da un identificativo, permette operazioni di lettura, scrittura e rimozione degli Agent;
- **Metodi:**

```
+ getAgentsList(): AgentObservable
```

L'`Observable` restituito manderà agli `Observer` gli agents ottenuti, uno alla volta, e poi chiama il loro metodo `complete`. Nel caso in cui si verifichi un errore, gli `Observer` iscritti verranno notificati tramite la chiamata del loro metodo `error` con i dati relativi all'errore verificatosi;

```
+ updateAgent(agent: Agent): ErrorObservable
```

Metodo che permette di aggiornare un `Agent`. L'`Observable` restituito non riceverà alcun valore, ma verrà completato in caso di aggiornamento dell'`Agent` avvenuta con successo. In caso di errore durante l'aggiornamento dell'`Agent`, gli `Observer` interessati verranno notificati tramite la chiamata del loro metodo `error()` con i dati relativi all'errore verificatosi;

Parametri:

```
* agent: Agent
```

Parametro contenente l'agente da aggiornare;

```
+ removeAgent(name: String): ErrorObservable
```

Metodo che permette di rimuovere un `Agent` a partire da un `name`. L'`Observable` restituito non riceverà alcun valore, ma verrà completato in caso di rimozione dell'`Agent` avvenuta con successo. In caso di errore durante la rimozione dell'`Agent`, gli `Observer` interessati verranno notificati tramite la chiamata del loro metodo `error()` con i dati

relativi all'errore verificatosi;

Parametri:

* name: String

Parametro contenente il name dell'agente da rimuovere;

+ addAgent(agent: Agent): ErrorObservable

Metodo che permette di aggiungere un Agent. L'Observable restituito non riceverà alcun valore, ma verrà completato in caso di aggiunta dell'Agent avvenuta con successo. In caso di errore durante l'aggiunta dell'Agent, gli Observer interessati verranno notificati tramite la chiamata del loro metodo error() con i dati relativi all'errore verificatosi;

Parametri:

* agent: Agent

Parametro contenente l'agente da aggiungere al database;

+ getAgent(name: String): AgentObservable

Metodo che ritorna un Agent a partire da un name. L'Observable restituito riceverà l'oggetto rappresentante tale Agent, e verrà completato. Nel caso in cui l'Agent richiesto non sia presente nel database, gli Observer interessati non riceveranno alcun valore, ma verranno notificati tramite la chiamata del loro metodo error();

Parametri:

* name: String

Parametro contenente il name dell'agente da ricevere;

VAModule

VAModule.png VAModule.png VAModule.png

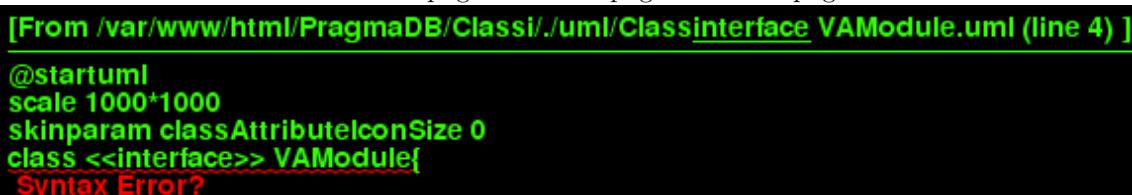


Figura 71: Back-end::VirtualAssistant:: VAModule

- **Nome:** VAModule;
- **Tipo:** Interface;
- **Descrizione:** questa classe definisce l'interfaccia dei moduli che si occupano di interrogare le API di un assistente virtuale;
- **Utilizzo:** fornisce la definizione dei metodi che dovranno essere implementati, in maniera tale da permettere ad un modulo che si interfacci con api.ai di essere utilizzato da VAService;
- **Figlio:** ApiAiVAAdapter;
- **Metodi:**

+ query(str: VAQuery): VAResponse

Metodo che permette di interrogare l'assistente virtuale;

Parametri:

* str: VAQuery

Parametro contenente i dati necessari all'interrogazione dell'assistente virtuale;

WebhookService

```
WebhookService.png WebhookService.png WebhookService.png
[From /var/www/html/PragmaDB/Classi/.uml/Classinterface WebhookService.uml (line 4)]
@startuml
scale 1000*1000
skinparam classAttributeIconSize 0
class <>interface> WebhookService{
    Syntax Error?
```

Figura 72: Back-end::VirtualAssistant:: WebhookService

- **Nome:** WebhookService;
- **Tipo:** Interface;
- **Descrizione:** questa classe definisce l'interfaccia dei moduli che implementano dei webhook conformi alle specifiche di api.ai;
- **Utilizzo:** fornisce una serie di metodi necessari per definire un servizio che soddisfi i requisiti per webhook di api.ai. Per la relativa documentazione, consultare la pagina <https://docs.api.ai/docs/webhook>;
- **Figli:** ConversationWebhookService, AdministrationWebhookService;
- **Metodi:**

+ webhook(event: LambdaEvent, context: LambdaContext): void

Metodo che fornisce l'interfaccia di una lambda function compatibile con i requisiti per webhook di api.ai;
Parametri:

* event: LambdaEvent

Parametro contenente i dati mandati da api.ai al WebhookService;

* context: LambdaContext

Parametro utilizzato dalle lambda function per inviare la risposta dal WebhookService ad api.ai. Il body del LambdaResponse, parametro del metodo LambdaContext::succeed, conterrà un oggetto , sotto forma di stringa in formato JSON, contenente i dati relativi all'utente ritornato.

Tali dati sono così organizzati:

```
1 {
2     "contextOut": "ContextArray",
3     "data": "Object",
4     "displayText": "String",
5     "followupEvent": "Object",
6     "source": "String",
7     "speech": "String"
8 }
```

Dove:

- **contextOut:** attributo contenente l'array dei context che la richiesta ha attivato;
- **data:** attributo contenente i dati che saranno inviati al client. Tali dati non sono processati da api.ai e quindi saranno nella forma originale;
- **displayText:** attributo contenente il testo che verrà mostrato sullo schermo dell'utente;

- **followupEvent**: attributo contenente parametri opzionali che il servizio web utilizzato vuole inviare ad api.ai;
- **source**: attributo contenente la risorse dei dati forniti come risposta;
- **speech**: attributo contenente la risposta testuale alla richiesta.

Per la relativa documentazione, consultare la pagina Per una descrizione dettagliata si rimanda alla pagina <https://docs.api.ai/docs/webhook#section-format-of-response-from-the-agent>;

Agent

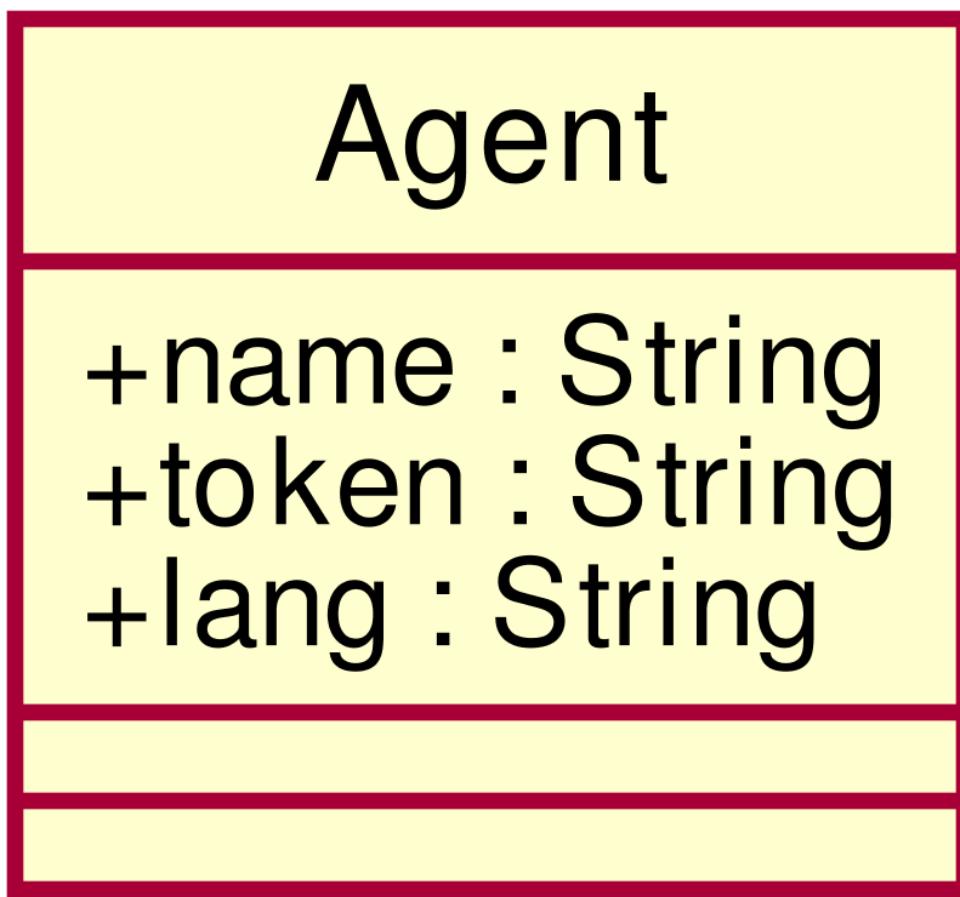


Figura 73: Back-end::VirtualAssistant::Agent

- **Nome:** Agent;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare e organizzare i dati relativi ad un Agent di api.ai;

- **Utilizzo:** fornisce gli attributi di un Agent che dovranno essere memorizzati nel database per questo microservizio. In api.ai, un Agent ha lo scopo di trasformare il linguaggio naturale, ricevuto in input, in dati capibili per le applicazioni. Per la relativa documentazione, consultare questa pagina <https://docs.api.ai/docs/concept-agents>;

- **Attributi:**

+ name: String

Nome dell'applicazione a cui è collegato l'agent. Per ogni applicazione, abbiamo un Agent;

+ token: String

Attributo contenente il valore del token associato. Un agent è identificabile tramite esso;

+ lang: String

Attributo contenente la lingua, la quale dovrà essere sempre inglese (en);

AgentsDAODynamoDB

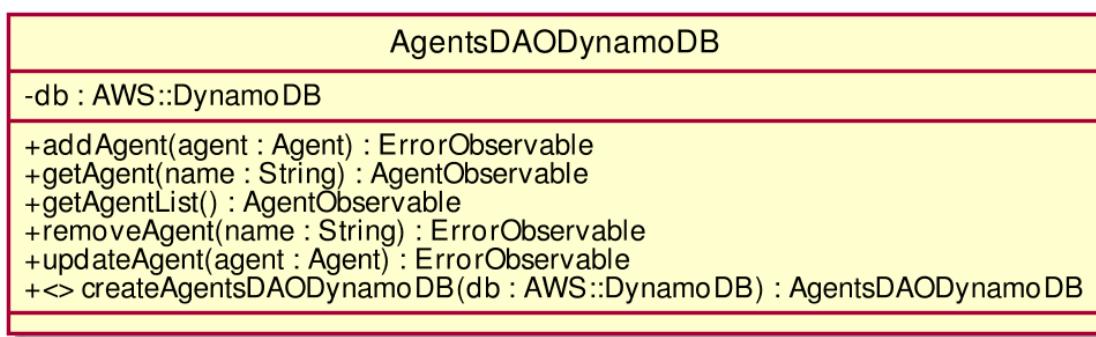


Figura 74: Back-end::VirtualAssistant::AgentsDAODynamoDB

- **Nome:** AgentsDAODynamoDB;
- **Tipo:** Class;
- **Descrizione:** classe che si occupa di implementare l'interfaccia AgentsDAO, utilizzando un database DynamoDB come supporto per la memorizzazione dei dati;
- **Utilizzo:** implementa i metodi dell'interfaccia AgentsDAO interrogando un database DynamoDB. Utilizza AWS::DynamoDB::DocumentClient per l'accesso al database. La dependency injection dell'oggetto AWS::DynamoDB viene fatta utilizzando il costruttore;
- **Attributi:**

- db: AWS::DynamoDB

Attributo contenente un riferimento al modulo di Node.js utilizzato per l'accesso al database DynamoDB contenente la tabella degli utenti;

- **Metodi:**

+ addAgent(agent: Agent): ErrorObservable

Implementazione del metodo definito nell'interfaccia AgentsDAO. Utilizza il metodo put del DocumentClient per aggiungere l'utente al database;

Parametri:

* agent: Agent

Parametro contenente l'agente da aggiungere al database;

+ `getAgent(name: String): AgentObservable`
 Implementazione del metodo definito nell'interfaccia AgentsDAO. Utilizza il metodo get del DocumentClient per ottenere i dati relativi ad uno User dal database;
 Parametri:
`* name: String`
 Parametro contenente il nome dell'agente da ricevere;

+ `getAgentList(): AgentObservable`
 Implementazione del metodo dell'interfaccia AgentsDAO. Utilizza il metodo scan del DocumentClient per ottenere la lista degli utenti dal database;

+ `removeAgent(name: String): ErrorObservable`
 Implementazione del metodo dell'interfaccia AgentsDAO. Utilizza il metodo delete del DocumentClient per eliminare un utente dal database;
 Parametri:
`* name: String`
 Parametro contenente il nome dell'agente da rimuovere;

+ `updateAgent(agent: Agent): ErrorObservable`
 Implementazione del metodo dell'interfaccia AgentsDAO. Utilizza il metodo update del DocumentClient per aggiornare i dati relativi ad un utente presente all'interno del database;
 Parametri:
`* agent: Agent`
 Parametro contenente l'agente da aggiornare;

+ <> `createAgentsDAODynamoDB(db: AWS::DynamoDB): AgentsDAODynamoDB`
 Constructor della classe AgentsDAODynamoDB. Permette di effettuare la dependency injection di AWS::DynamoDB;
 Parametri:
`* db: AWS::DynamoDB`
 Parametro contenente un riferimento al modulo di Node.js da utilizzare per l'accesso al database DynamoDB contenente la tabella degli agenti;

ApiAiVAAdapter

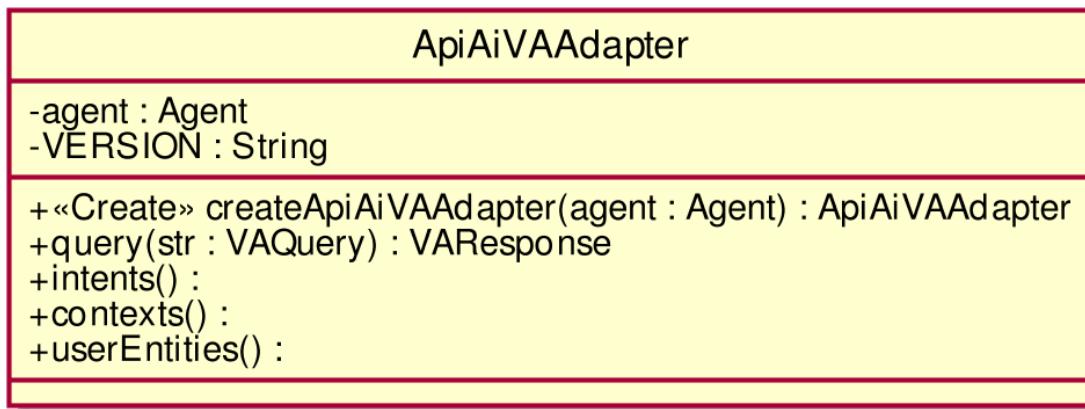


Figura 75: Back-end::VirtualAssistant::ApiAiVAAdapter

- **Nome:** ApiAiVAAdapter;

- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di convertire l'interfaccia fornita da api.ai in una più adatta alle esigenze dell'applicazione, definita da **VAModule**.
Facendo da Adapter tra le API di api.ai (adaptee) e l'interfaccia **VAModule** (target) utilizzata da **VAService**, permette l'interoperabilità tra queste due interfacce;
- **Utilizzo:** fornisce a **VAModule** un meccanismo che permette di interrogare le API di api.ai, in modo da consentire a **VAService** l'utilizzo di quest'ultime utilizzando un'interfaccia distinta e più consona alle proprie esigenze.
Per fare le richieste HTTP alle API, il metodo `query` utilizza il modulo request-promise. È soggetta ad una constructor-based dependency injection, la quale ha come oggetto un **Agent** relativo alle richieste da inviare. ;
- **Padre:** <<interface>> **VAModule**;
- **Attributi:**
 - `agent: Agent`
Attributo contenente lo **Agent** al quale inviare le richieste;
 - `VERSION: String`
Attributo contenente il campo **versioning** da inviare in una richiesta ad api.ai. Per chiarire quanto appena detto, consultare la pagina <https://docs.api.ai/docs/versioning>;
- **Metodi:**
 - + <<Create>> `createApiAiVAAdapter(agent: Agent): ApiAiVAAdapter`
Costruttore che realizza una dependency injection avente come oggetto un **Agent**;
Parametri:
 - * `agent: Agent`
Parametro contenente l'**Agent**;
 - + `query(str: VAQuery): VAResponse`
Metodo che permette di interrogare lo **Agent** in api.ai;
Parametri:
 - * `str: VAQuery`
Attributo contenente i dati relativi all'interrogazione da porre allo **Agent** in api.ai;
 - + `intents():`
Metodo che permette;
 - + `contexts():`
Metodo che permette;
 - + `userEntities():`
Metodo che permette;

ButtonObject

- **Nome:** `ButtonObject`;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare l'oggetto buttons di api.ai;
- **Utilizzo:** fornisce gli attributi per rappresentare l'oggetto buttons relativo ad una risposta.
Per la relativa documentazione, consultare la pagina <https://docs.api.ai/docs/rich-messages>;

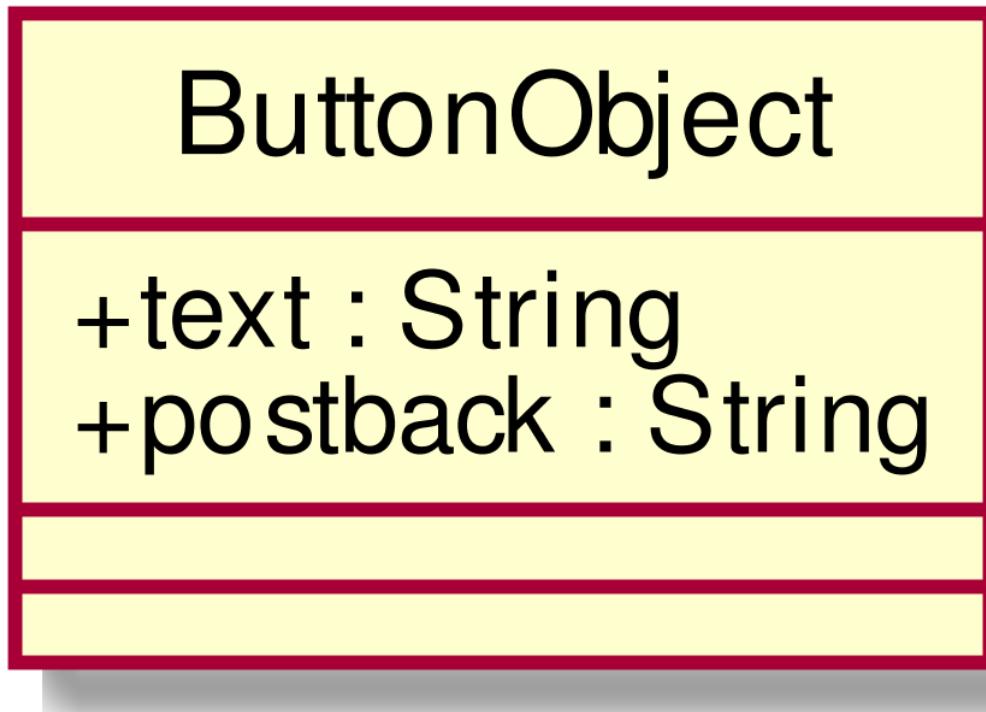


Figura 76: Back-end::VirtualAssistant::ButtonObject

- **Attributi:**

+ text: String

Attributo contenente il testo del button;

+ postback: String

Attributo contenente il testo da inviare, ad api.ai o un URL, dopo che il button è stato selezionato;

Context

- **Nome:** Context;

- **Tipo:** Class;

- **Descrizione:** questa classe si occupa di rappresentare e organizzare gli attributi relativi ad un Context in api.ai;

- **Utilizzo:** fornisce gli attributi di un Context. In api.ai, un Context è una stringa che rappresenta il contesto corrente nel quale un utente somministra una richiesta, in maniera tale da disambiguare il più possibile quest'ultime.

Per la relativa documentazione, consultare questa pagina <https://docs.api.ai/docs/concept-contexts>;

- **Attributi:**

+ name: String

Attributo contenente il nome del Context;

+ parameters: StringAssocArray

Attributo contenente l'array dei parametri che compongono il Context;

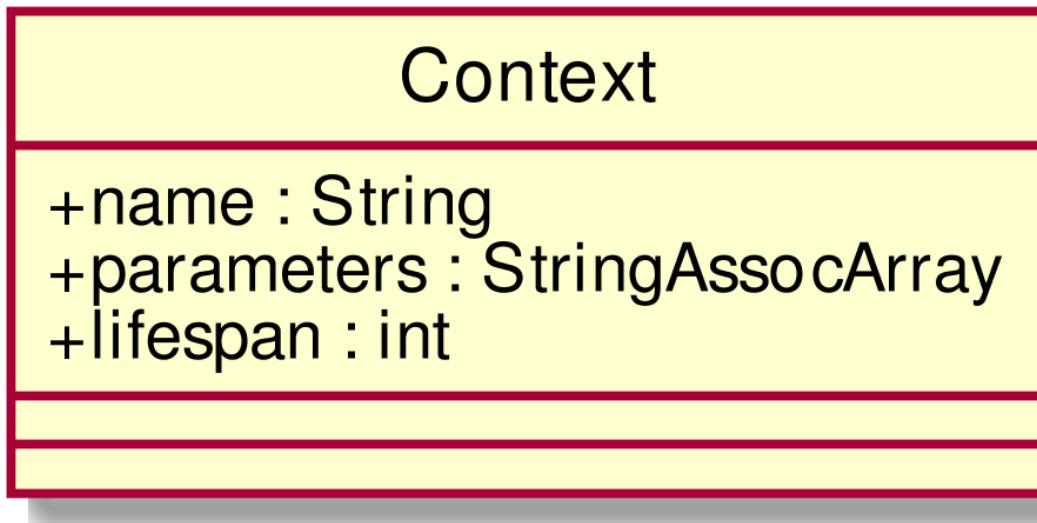


Figura 77: Back-end::VirtualAssistant::Context

+ lifespan: int

Attributo contenente il lifespan del Context. Questo valore indica per quanti altri matched intents bisogna mantenere il Context. Per chiarire quanto appena detto, consultare la relativa documentazione in questa pagina <https://api.ai/blog/2015/11/23/Contexts/>;

Fulfillment

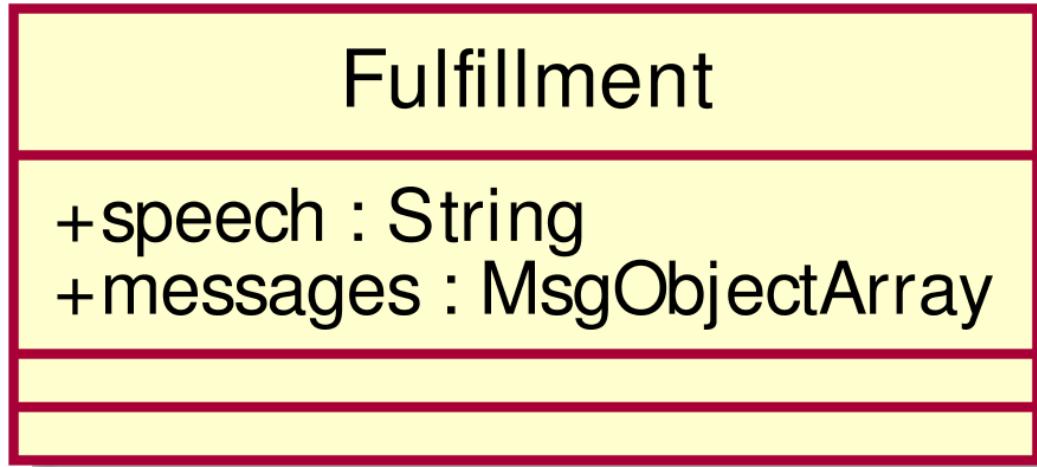


Figura 78: Back-end::VirtualAssistant::Fulfillment

- **Nome:** Fulfillment;
- **Tipo:** Class;

- **Descrizione:** questa classe si occupa di rappresentare e organizzare gli attributi relativi ad un fulfillment di api.ai;
- **Utilizzo:** fornisce gli attributi di un fulfillment. In api.ai un fulfillment è tutto ciò che viene ritornato dalla richiesta di utente. In particolare, è l'adempimento ad una richiesta e può comprendere diversi campi dati. Per chiarire quanto appena detto, consultare la relativa documentazione <https://docs.api.ai/docs/webhook> ;
- **Attributi:**

+ speech: String

Attributo contenente il testo relativo alla risposta fornita;

+ messages: MsgObjectArray

Attributo contenente l'array di tutti i campi dati presenti nel messages. Il tipo è un MsgObject in quanto i tipi degli elementi di messages sono eterogenei fra loro;

Metadata

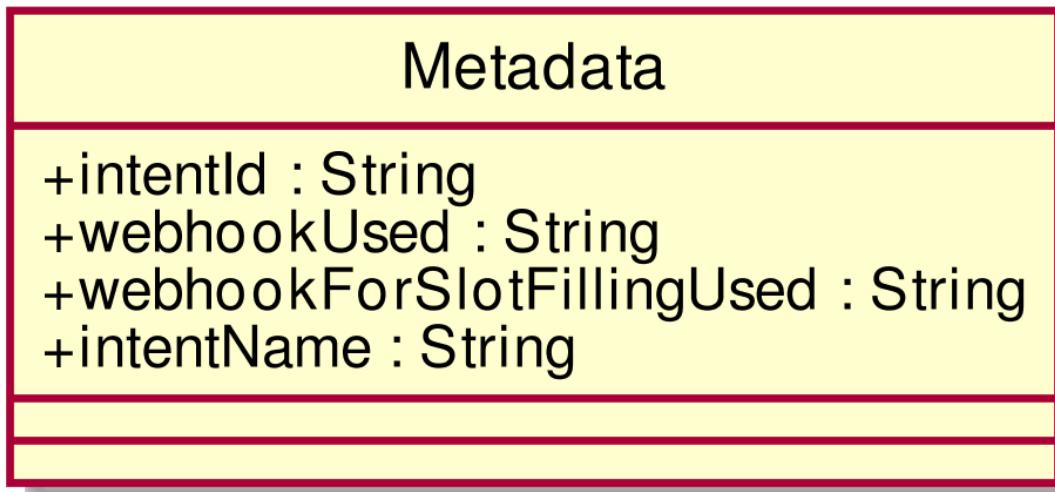


Figura 79: Back-end::VirtualAssistant::Metadata

- **Nome:** Metadata;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare e organizzare i dati relativi a intents e contexts;
- **Utilizzo:** fornisce gli attributi relativi ai dati di intents e contexts e viene utilizzato come attributo della classe ProcessingResult. Per la relativa documentazione, consultare la pagina <https://docs.api.ai/docs/query#response>;
- **Attributi:**
 - + intentId: String
Attributo contenente l'identificativo per un intent;
 - + webhookUsed: String
Attributo contenente un valore booleano che indica se è stato usato un webhook. Per chiarire quanto detto, consultare la relativa documentazione <https://docs.api.ai/docs/webhook>;

```
+ webhookForSlotFillingUsed: String
Attributo contenente un valore booleano che indica se è stato usato un webhook for slot filling. Per chiarire quanto detto, consultare la relativa documentazione https://docs.api.ai/docs/webhook#webhook-for-slot-filling;
+ intentName: String
Attributo contenente il nome dell'intent;
```

MsgObject

MsgObject

```
+type : int
+speech : String[0..1]
+imageUrl : String[0..1]
+title : String[0..1]
+subtitle : String[0..1]
+buttons : ButtonObjectArray[0..1]
+replies : StringArray[0..1]
+payload : Object[0..1]
```

Figura 80: Back-end::VirtualAssistant::MsgObject

- **Nome:** MsgObject;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare e organizzare gli attributi relativi ad un MsgObject, il quale definisce un tipo unico tra tutti i messaggi (dal contenuto eterogeneo) contenuti in un **Fulfillment**;
- **Utilizzo:** fornisce gli attributi di un MsgObject, il quale sarà contenuto in un **Fulfillment**. La classe **Fulfillment** fa utilizzo di un array di essi per adempire alla richiesta di un utente. Per la relativa documentazione, consultare le seguenti pagine:
<https://docs.api.ai/docs/rich-messages>
<https://docs.api.ai/docs/query#section-message-objects>;
- **Attributi:**

```
+ type: int
Attributo il quale valore indica il tipo del messaggio. I possibili valori ed i relativi messaggi sono:
* 0: messaggio testuale, prevede la presenza del campo speech;
* 3: immagine, prevede la presenza del campo imgUrl;
* 1: card, rappresenta un messaggio più articolato, con possibile presenza di bottoni. Prevede la presenza dei campi title e subtitle. Può essere presente anche il campo buttons;
* 2: quick replies, rappresenta una serie di risposte rapide. Prevede la presenza dei campi title e replies;
* 4: payload, prevede la presenza del campo payload.

;

+ speech: String[0..1]
Attributo contenente il testo di un messaggio di tipo messaggio testuale;

+ imageUrl: String[0..1]
Attributo contenente l'url dell'immagine di un messaggio di tipo immagine;

+ title: String[0..1]
Attributo contenente il titolo di un messaggio di tipo card o quick replies;

+ subtitle: String[0..1]
Attributo contenente il sottotitolo di un messaggio di tipo card;

+ buttons: ButtonObjectArray[0..1]
Attributo contenente l'array di button di un messaggio di tipo card;

+ replies: StringArray[0..1]
Attributo contenente l'array delle risposte rapide di un messaggio di tipo quick replies;

+ payload: Object[0..1]
Attributo contenente il payload di un messaggio di tipo payload;
```

ResponseBody

- **Nome:** ResponseBody;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare il corpo della risposta, fornita dall'assistente virtuale, in seguito ad un'interrogazione;
- **Utilizzo:** fornisce gli attributi necessari per rappresentare il corpo della risposta dell'assistente virtuale.

Viene passata come parametro al metodo runCmd delle varie applicazioni, il quale si occupa di estrarre i parametri necessari ad eseguire il comando ricevuto;

- **Attributi:**

```
+ text_request: String
Attributo contenente il testo della richiesta dell'utente;

+ text_response: String
Attributo che contiene il testo della risposta dell'assistente virtuale;

+ contexts: ObjectAssocArray[0..1]
Array contenente i context ricevuti dall'assistente virtuale;
```

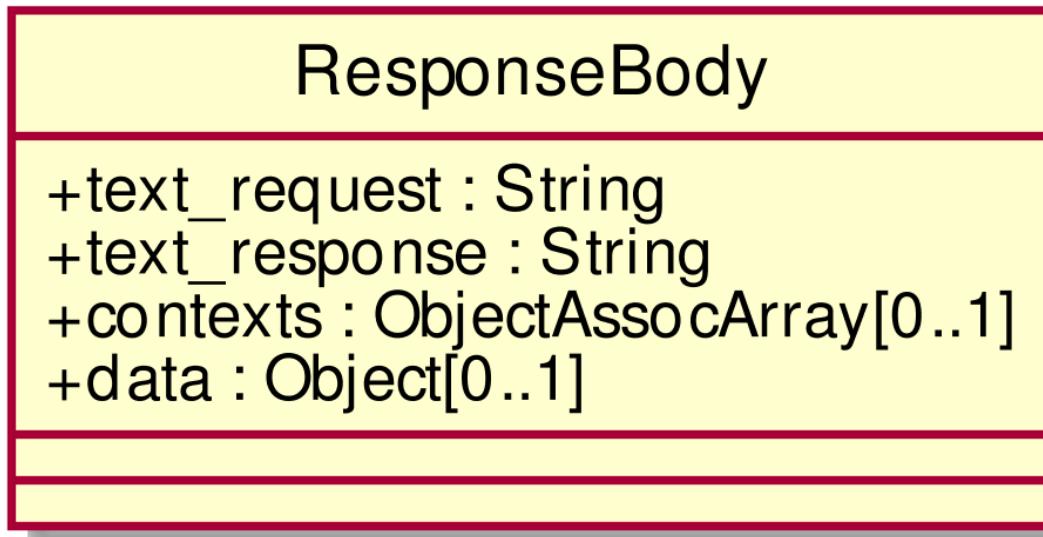


Figura 81: Back-end::VirtualAssistant::ResponseBody

+ data: Object[0..1]

Attributo che può essere utilizzato per scambiare dati tra client e l'eventuale webhook. Il suo contenuto dipende dal servizio webhook utilizzato. Il client si deve occupare di reinviare i dati presenti in questo campo di una determinata risposta nella richiesta successiva. In particolare, questo attributo viene utilizzato per lo scambio del token dello agent in api.ai;

VAEventObject

- **Nome:** VAEventObject;
- **Tipo:** Class;
- **Descrizione:** questa classe rappresenta il campo event di una richiesta all'assistente virtuale;
- **Utilizzo:** fornisce i campi necessari all'interrogazione di un assistente virtuale, utilizzando il nome di un evento e dei parametri al posto della stringa di query VAQuery::text. Per la relativa documentazione, consultare la pagina <https://docs.api.ai/docs/concept-events>;
- **Attributi:**

+ name: String

Nome dell'evento che si vuole far eseguire all'assistente virtuale;

+ data: Object[0..1]

Oggetto contenente i parametri necessari all'assistente virtuale per l'esecuzione dell'evento specificato. Tali parametri saranno inseriti come coppie chiave-valore, dove la chiave indica il nome del parametro;

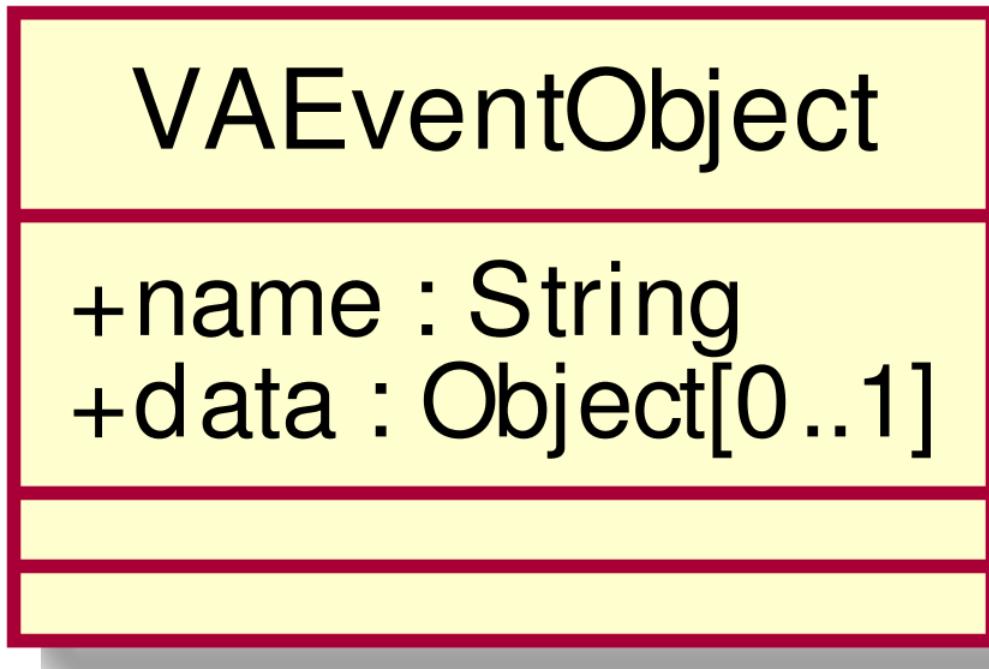


Figura 82: Back-end::VirtualAssistant::VAEventObject

VAQuery

- **Nome:** VAQuery;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare una richiesta da porre all'assistente virtuale;
- **Utilizzo:** fornisce gli attributi necessari che compongono una richiesta all'assistente virtuale. Viene utilizzata dal metodo `query` delle classi che implementano l'interfaccia `VAModule` per interrogare i rispettivi assistenti virtuali;
- **Attributi:**

+ `text: String[0..1]`

Attributo la cui presenza è obbligatoria a meno che non sia presente l'attributo `event`. Contiene il testo della frase che si vuole comunicare all'assistente;

+ `event: VAEVENTOBJECT[0..1]`

Attributo la cui presenza è obbligatoria, a meno che non sia presente l'attributo `text`. Indica l'evento di cui si richiede l'esecuzione.

Per una definizione di evento fare riferimento alla documentazione di api.ai alla pagina <https://docs.api.ai/docs/concept-events>;

+ `session_id: String`

Attributo contenente l'id della sessione dell'assistente virtuale. Deve essere generato dal client;

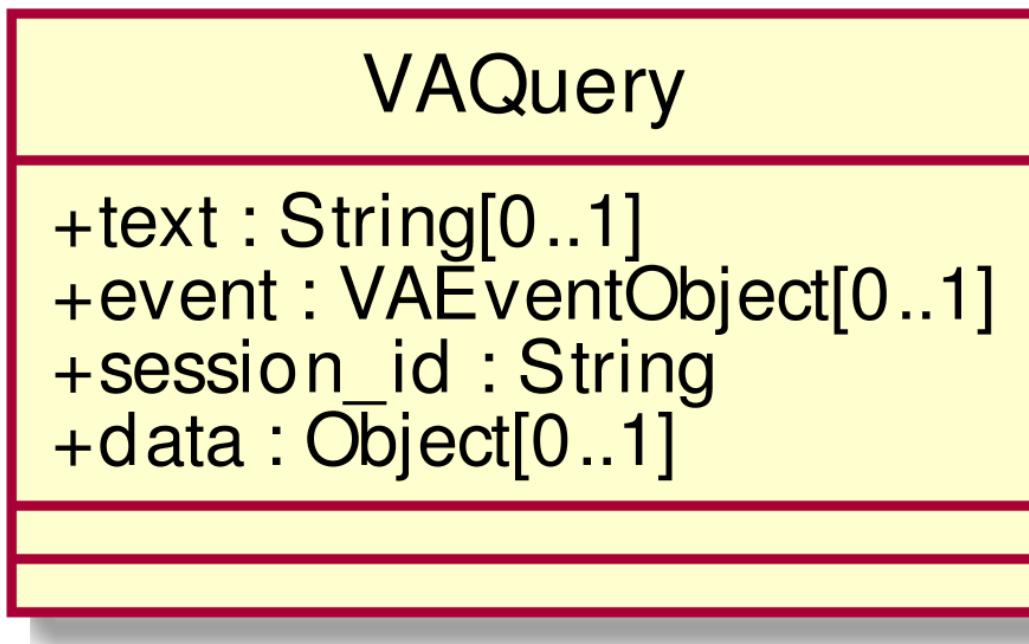


Figura 83: Back-end::VirtualAssistant::VAQuery

+ data: Object [0..1]

Oggetto che permette l'invio dei dati necessari all'eventuale webhook per svolgere la sua funzione;

VAResponse

- **Nome:** VAResponse;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare l'intera risposta dell'assistente virtuale, fornita in seguito ad un'interrogazione ;
- **Utilizzo:** fornisce gli attributi relativi ad una risposta dell'assistente virtuale. Viene utilizzata dalle classi che implementano l'interfaccia VAModule, in maniera tale da fornire la risposta dell'assistente virtuale;
- **Attributi:**

+ session_id: String

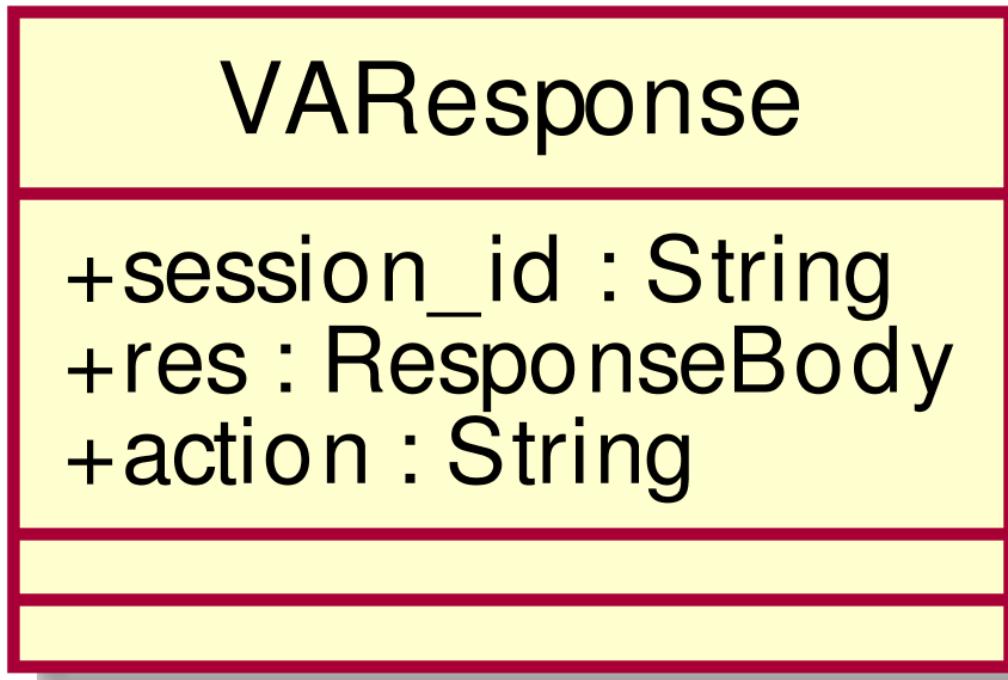
Rappresenta l'id univoco della sessione corrente dell'assistente virtuale;

+ res: ResponseBody

Attributo contenente il corpo della risposta ricevuta dall'assistente virtuale. Il contenuto di questo attributo sarà passato senza modifiche al metodo Client::ApplicationManager::Application::dell'applicazione;

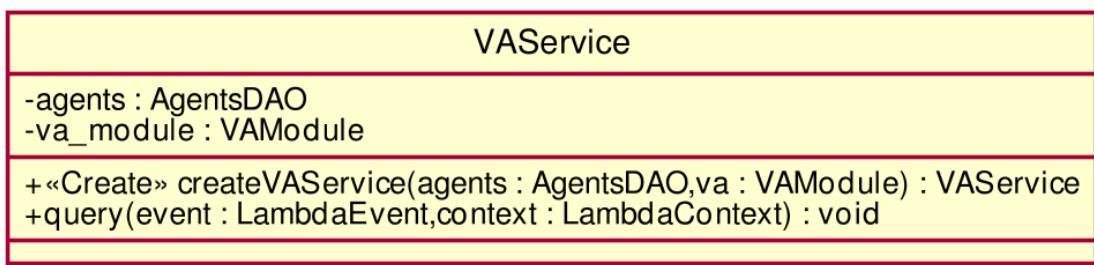
+ action: String

Attributo contenente l'azione che l'assistente virtuale richiede di compiere al client. La stringa è nel formato [nome_applicazione.comando], dove nome_applicazione indica l'applicazione che deve essere eseguita, mentre comando indica il comando che il client deve far eseguire all'applicazione.

**Figura 84:** Back-end::VirtualAssistant::VAResponse

Ad esempio, `conversation.displayMsgs` comunica al client di far eseguire il comando `displayMsgs` all'applicazione che si occupa di mostrare la conversazione. ;

VAService

**Figura 85:** Back-end::VirtualAssistant::VAService

- **Nome:** VAService;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare il microservizio Virtual Assistant. ;
- **Utilizzo:** fornisce il metodo necessario all'interrogazione dell'assistente virtuale;
- **Attributi:**

- **agents:** AgentsDAO

Attributo che permette di contattare AgentsDAO, il quale fornisce i meccanismi d'accesso al database degli Agent disponibili;

- **va_module:** VAModule

Attributo contenente il VAModule.

VAModule è un'interfaccia, implementata da ApiAiVAAAdapter. I metodi invocati saranno quelli appartenenti a quest'ultima classe;

- **Metodi:**

```
+ <<Create>> createVAService(agents: AgentsDAO, va: VAModule): VAService
Costruttore che realizza una dependency injection, avente come oggetti un AgentsDAO e un VAModule;
```

Parametri:

* **agents:** AgentsDAO

Parametro contenente l'AgentsDAO;

* **va:** VAModule

Parametro contenente il VAModule;

```
+ query(event: LambdaEvent, context: LambdaContext): void
```

Questo metodo implementa la lambda function che si occupa dell'interrogazione dell'assistente virtuale. A partire da una richiesta contenente il testo della richiesta dell'utente, questo metodo esegue una chiamata all'endpoint /query di api.ai e, a partire dalla risposta ricevuta, manda la risposta utilizzando il context;

Parametri:

* **event:** LambdaEvent

Parametro contenente l'evento con i dati relativi alla richiesta di API Gateway. Il campo body di questo evento conterrà una stringa in formato JSON nel formato seguente:

```
1 {
2     "app": "nome\_applicazione",
3     "query": "VAQuery"
4 }
```

con **app** stringa che corrisponde al nome dell'applicazione che manda la richiesta, e **query** oggetto del tipo VAQuery contenente i dati relativi alla query da mandare all'assistente virtuale;

* **context:** LambdaContext

Parametro utilizzato dalle lambda function per inviare la risposta. La risposta, contenuta nel LambdaResponse parametro del metodo LambdaContext::succeed, possiede un attributo **body**, il quale conterrà il corpo di essa sotto forma di una stringa in formato JSON, organizzando i dati nel seguente modo:

```
1 {
2     "action": "String",
3     "res": {
4         "contexts": "ObjectAssocArray",
5         "data": "Object",
6         "text\_request": "String",
7         "text\_response": "String"
8     },
9     "session\_id": "String"
10 }
11 }
```

Dove:

- **action**: attributo contenente l'azione che l'assistente virtuale richiede di compiere al client. La stringa è nel formato [nome_applicazione.comando], dove nome_applicazione indica l'applicazione che deve essere eseguita, mentre comando indica il comando che il client deve far eseguire all'applicazione.
Ad esempio, `conversation.displayMsgs` comunica al client di far eseguire il comando `displayMsgs` all'applicazione che si occupa di mostrare la conversazione;
- **res**: attributo contenente il corpo della risposta ricevuta dall'assistente virtuale. Il contenuto di questo attributo sarà passato senza modifiche al metodo `Client::ApplicationManager::Application::runCmd` dell'applicazione.
In questo oggetto abbiamo i campi `contexts` (array contenente i context ricevuti dall'assistente virtuale), `data` (attributo che può essere utilizzato per scambiare dati tra client e l'eventuale webhook). Il suo contenuto dipende dal servizio webhook utilizzato. Il client si deve occupare di reinviare i dati presenti in questo campo di una determinata risposta nella richiesta successiva. In particolare, questo attributo viene utilizzato per lo scambio, tra client e back-end, del token dello agent in `api.ai`), `text_request` (attributo contenente il testo della richiesta dell'utente) e `text_response`(attributo che contiene il testo della risposta dell'assistente virtuale).
- **session_id**: attributo contenente l'id univoco della sessione corrente dell'assistente virtuale.

;

Client

Package che racchiude tutte le componenti del client. Il pattern utilizzato per organizzare le componenti è quello di un'architettura event-driven.

Classi

`ApplicationLocalRegistry`

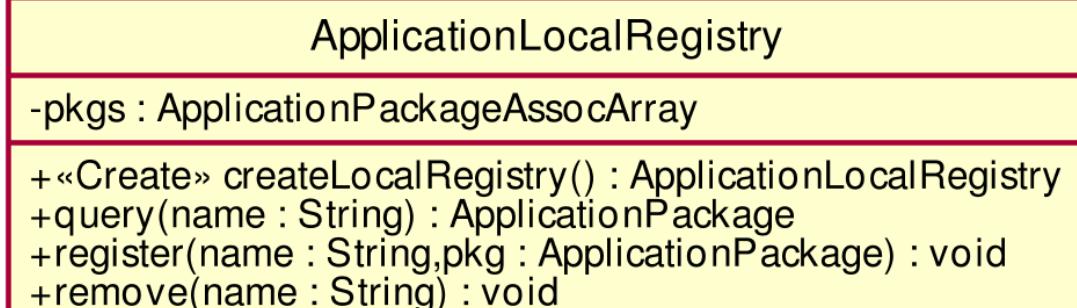


Figura 86: Client::ApplicationLocalRegistry

- **Nome:** `ApplicationLocalRegistry`;
- **Tipo:** Class;

- **Descrizione:** questa classe si occupa di mantenere una lista degli `ApplicationPackage` disponibili. ;

- **Utilizzo:** fornisce a `LocalApplicationRegistryClient` i metodi necessari ad inserire, rimuovere ed ottenere degli `ApplicationPackage`. Implementa un Registry all'interno del client, semplificando il compito del `RegistryClient` ed eliminando la necessità di effettuare richieste HTTP per recuperare un `ApplicationPackage`. ;

- **Attributi:**

- `pkgs: ApplicationPackageAssocArray`

Attributo contenente l'Array dei nomi delle applicazioni contenute nel registry;

- **Metodi:**

- + <<Create>> `createLocalRegistry(): ApplicationLocalRegistry`

Metodo che permette di istanziare un `ApplicationLocalRegistry`;

- + `query(name: String): ApplicationPackage`

Metodo che permette di interrogare il `LocalRegistry` a partire dal nome dell'applicazione, ottenendo l' `ApplicationPackage` relativo ad essa;

Parametri:

- * `name: String`

Parametro contenente il nome dell'applicazione da recuperare;

- + `register(name: String, pkg: ApplicationPackage): void`

Metodo che permette la registrazione di una applicazione nel `LocalRegistry`;

Parametri:

- * `name: String`

Parametro contenente il nome dell'applicazione da registrare;

- * `pkg: ApplicationPackage`

Parametro contenente il package relativo all'applicazione da registrare;

- + `remove(name: String): void`

Metodo che permette di rimuovere un'applicazione a partire dal suo nome;

Parametri:

- * `name: String`

Parametro contenente il nome dell'applicazione da rimuovere;

- **Relazioni con le altre classi:**

- IN `ApplicationRegistryLocalClient`

- OUT `ApplicationPackage`

ConversationApp

- **Nome:** `ConversationApp`;

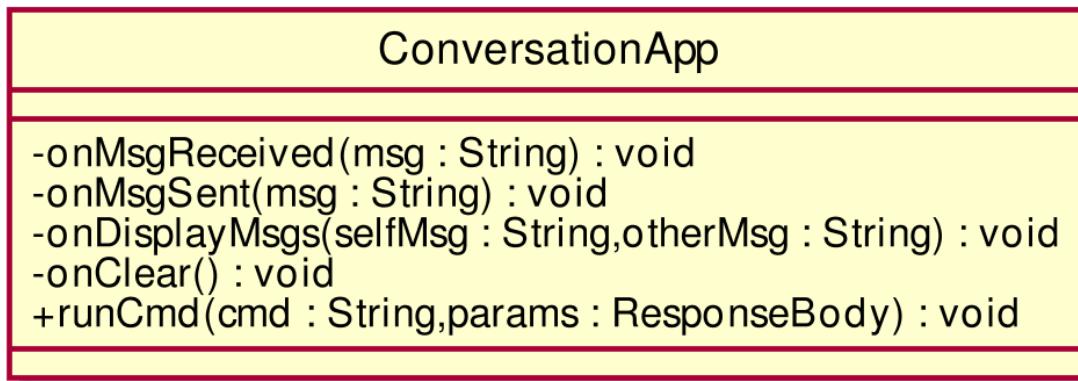
- **Tipo:** Class;

- **Descrizione:** questa classe si occupa di rappresentare l'applicazione di conversazione tra un utente e l'assistente virtuale;

- **Utilizzo:** fornisce un meccanismo, tramite i metodi e gli attributi forniti, per rappresentare e gestire la conversazione tra un utente e l'assistente virtuale;

- **Padre:** `Application`;

- **Metodi:**

**Figura 87:** Client::ConversationApp

- `onMsgReceived(msg: String): void`

Metodo che permette di creare l'HTMLElement del messaggio di risposta e di appenderlo all'interfaccia;

Parametri:

* `msg: String`

Parametro contenente il messaggio di risposta;

- `onMsgSent(msg: String): void`

Metodo che permette di creare l'HTMLElement del messaggio di richiesta e di appenderlo all'interfaccia;

Parametri:

* `msg: String`

Parametro contenente il messaggio di richiesta;

- `onDisplayMsgs(selfMsg: String, otherMsg: String): void`

Metodo che permette di appendere e visualizzare nell'interfaccia il messaggio di richiesta e il messaggio di risposta;

Parametri:

* `selfMsg: String`

Parametro contenente il messaggio di richiesta;

* `otherMsg: String`

Parametro contenente il messaggio di risposta;

- `onClear(): void`

Metodo che permette di ripulire l'interfaccia dai messaggi;

+ `runCmd(cmd: String, params: ResponseBody): void`

Metodo che permette di chiamare uno dei metodi privati, in base al comando ricevuto.

I comandi supportati sono:

* `clear`: causa la chiamata del metodo `onClear`;

* `msg`: causa la chiamata del metodo `onDisplayMsgs`, al quale vengono passati `params.text_request` e `params.text_response`.

;

Parametri:

* `cmd: String`

Parametro contenente il comando da somministrare all'applicazione;

- * params: ResponseBody
Parametro contenente l'array dei parametri del comando da somministrare all'applicazione;
- Relazioni con le altre classi:
 - OUT Application

IndexView

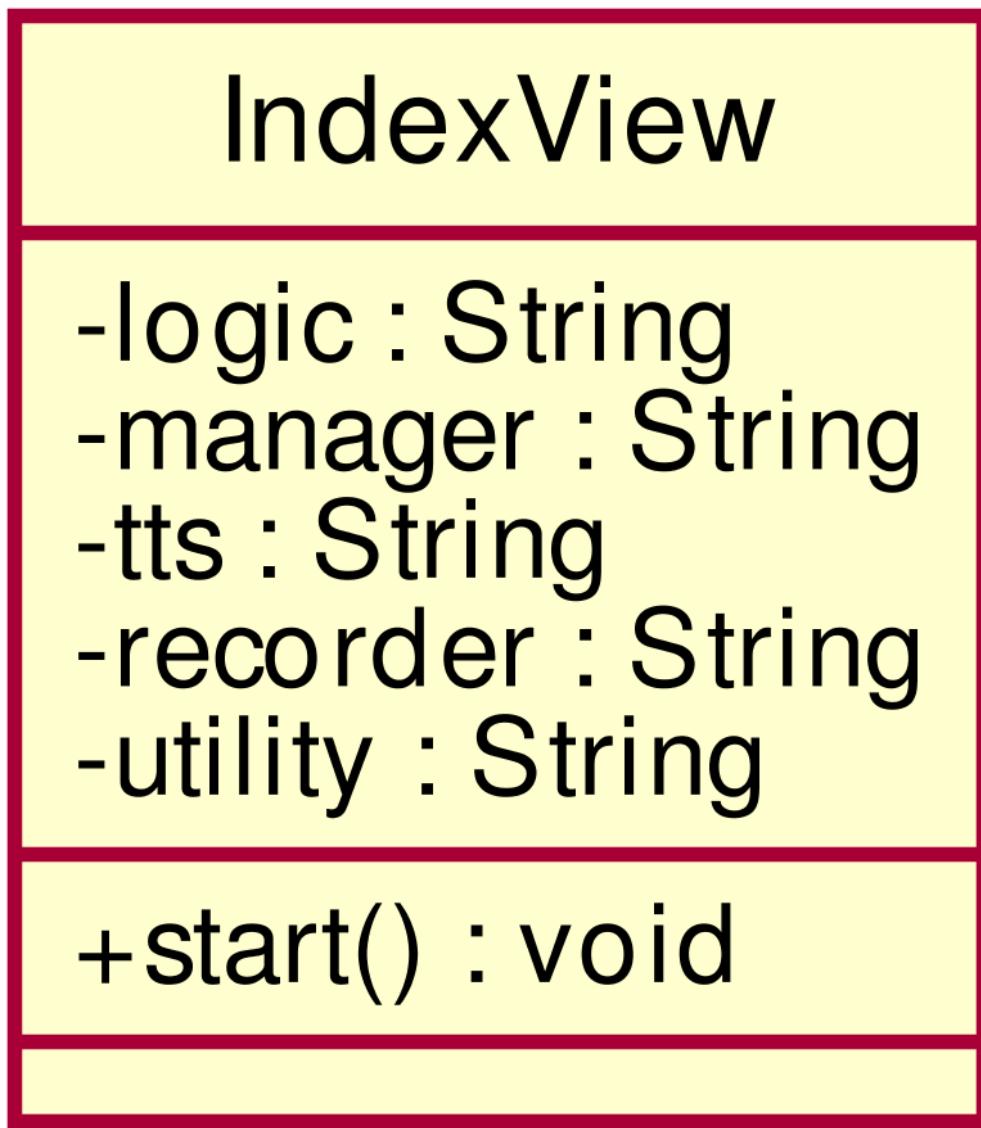


Figura 88: Client::IndexView

- Nome: IndexView;
- Tipo: Class;

- **Descrizione:** classe che rappresenta la pagina web all'interno della quale verranno mostrate le applicazioni dell'assistente virtuale. ;

- **Utilizzo:** questa classe è composta dal codice HTML della pagina web che costituisce l'interfaccia dell'assistente virtuale;

- **Attributi:**

- logic: String

Stringa contenente l'URL del file relativo al package Logic;

- manager: String

Stringa contenente l'URL del file relativo al package ApplicationManager;

- tts: String

Stringa contenente l'URL del file relativo al package TTS;

- recorder: String

Stringa contenente l'URL del file relativo al package Recorder;

- utility: String

Stringa contenente l'URL del file relativo al package Utility;

- **Metodi:**

+ start(): void

Metodo che permette di avviare le funzionalità dell'assistente virtuale;

ObserverAdapter

ObserverAdapter

```

-paused : boolean
-next_cb : function(data : Object) : void
-error_cb : function(err: Error): void
-complete_cb : function(): void

+next(data : Object) : void
+pause() : void
+resume() : void
+«Create» createPausableObserver() : PausableObserver
+isPaused() : boolean
+complete() : void
+error(err : Error) : void
+onNext(cb : function(data: Object): void) : void
+onComplete(cb : function(): void) : void
+onError(cb : function(err: Error) : void) : void

```

Figura 89: Client::ObserverAdapter

- **Nome:** ObserverAdapter;

- **Tipo:** Abstract Class;
- **Descrizione:** questa classe astratta fornisce i metodi che permettono di mettere in pausa un Observer. Questa classe implementa l'interfaccia della libreria esterna RxJS::Observer;
- **Utilizzo:** implementa l'interfaccia `Observer`, definendo i metodi `next`, `complete`, `error` in modo che chiamino le rispettive funzioni di callback. Fornisce inoltre un meccanismo per mettere in pausa un `Observer`, in modo che ignori gli eventi che gli sono notificati. Nel caso in cui l'`Observer` sia messo in pausa, le funzioni di callback impostate non verranno chiamate;
- **Figli:** `PlayerObserver`, `ApplicationManagerObserver`, `LogicObserver`;
- **Attributi:**
 - `paused: boolean`
Attributo che indica se l'Observer è in pausa;
 - `next_cb: function(data : Object) : void`
Funzione di callback da chiamare quando viene chiamato il metodo `next`;
 - `error_cb: function(err: Error): void`
Funzione di callback che viene chiamata quando viene chiamato il metodo `error`;
 - `complete_cb: function(): void`
Funzione di callback che viene chiamata quando viene chiamato il metodo `complete`;
- **Metodi:**
 - + `next(data: Object): void`
Implementazione del metodo dell'interfaccia. Si occupa di chiamare la funzione di callback impostata dalla chiamata al metodo `onNext`, passandole come parametro `data`;
Parametri:
 - * `data: Object`
Parametro contenente i dati da passare a `next_cb`;
 - + `pause(): void`
Metodo che permette di mettere in pausa l'Observer;
 - + `resume(): void`
Metodo che permette di riavviare l'Observer;
 - + <<Create>> `createPausableObserver(): PausableObserver`
Costruttore che impone il valore di `paused` a false;
 - + `isPaused(): boolean`
Metodo che permette di ottenere il valore di `paused`, in modo da capire se l'Observer è in pausa o meno;
 - + `complete(): void`
Implementazione del metodo dell'interfaccia. Si occupa di chiamare la funzione `complete_cb`;
 - + `error(err: Error): void`
Implementazione del metodo dell'interfaccia. Si occupa di chiamare la funzione di callback `error_cb`, passandole come parametro `err`;
Parametri:

```

* err: Error
    Parametro che contiene i dati relativi all'errore verificatosi;

+ onNext(cb: function(data: Object): void): void
    Metodo che permette di impostare la funzione di callback da chiamare quando viene
    chiamato il metodo next;
    Parametri:
        * cb: function(data: Object): void
            Funzione di callback;

+ onComplete(cb: function(): void): void
    Metodo che permette di impostare la funzione di callback da chiamare quando viene
    chiamato il metodo complete;
    Parametri:
        * cb: function(): void
            Funzione di callback;

+ onError(cb: function(err: Error) : void): void
    Metodo che permette di impostare la funzione di callback da chiamare quando viene
    chiamato il metodo error;
    Parametri:
        * cb: function(err: Error) : void
            Funzione di callback;

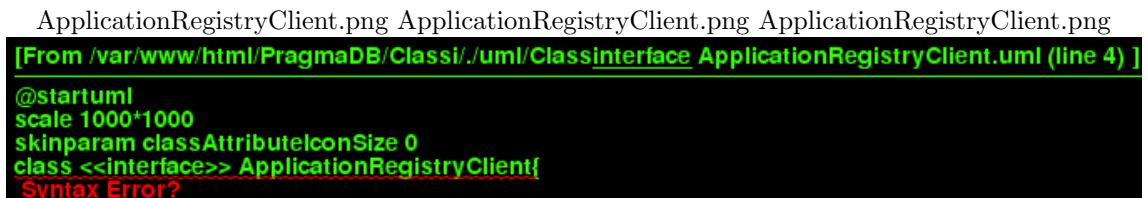
```

Client::ApplicationManager

Package contenente le classi che si occupano della gestione delle applicazioni con le quali l'utente può interagire.

Classi

ApplicationRegistryClient



```

ApplicationRegistryClient.png ApplicationRegistryClient.png ApplicationRegistryClient.png
[From /var/www/html/PragmaDB/Classi//uml/ClassInterface ApplicationRegistryClient.uml (line 4)]
@startuml
scale 1000*1000
skinparam classAttributeIconSize 0
class <<interface>> ApplicationRegistryClient{
    Syntax Error?

```

The screenshot shows a UML class diagram for 'ApplicationRegistryClient'. The class is defined as an interface ('<<interface>>'). There is a red box highlighting the closing brace '}' with the text 'Syntax Error?' below it. The background of the code area is black, and the text is in white and green.

Figura 90: Client::ApplicationManager:: ApplicationRegistryClient

- **Nome:** ApplicationRegistryClient;
- **Tipo:** Interface;
- **Descrizione:** interfaccia che fornisce i metodi che devono essere implementati dalle classi che si occupano di interrogare un registry delle applicazioni;
- **Utilizzo:** fornisce al Manager l'interfaccia delle classi che si occuperanno di ottenere gli ApplicationPackage;
- **Metodi:**

+ register(name: String, pkg: ApplicationPackage): boolean

Metodo che permette la registrazione di una applicazione nell'IAplicationRegistryClient. Questo metodo deve però impedire l'inserimento di package "Parziali" di RegistryPackage, ovvero un package di tipo ApplicationPackage. ;
Parametri:

* name: String

Parametro contenente il nome sotto cui registrare il Package dell'applicazione;

* pkg: ApplicationPackage

Parametro contenente il Package che si desidera registrare nel registry;

+ query(name: String): ApplicationPackage

Metodo che permette di ottenere il package di un'applicazione dal registry. Restituisce null in caso l'applicazione non sia disponibile;
Parametri:

* name: String

Parametro contenente il nome dell'applicazione che si vuole recuperare, come restituito dalle API;

- **Relazioni con le altre classi:**

- IN Manager

- IN ApplicationRegistryLocalClient

- OUT ApplicationPackage

Application

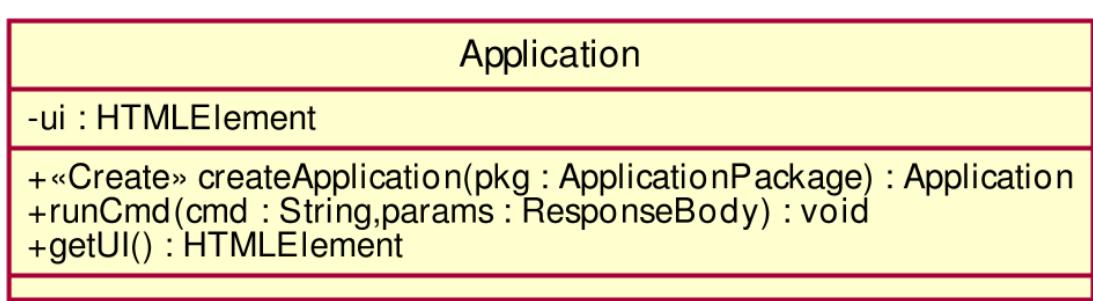


Figura 91: Client::ApplicationManager::Application

- **Nome:** Application;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa della gestione dell'applicazione in esecuzione. È una classe astratta;
- **Utilizzo:** fornisce al client funzionalità per l'istanziazione dell'applicazione necessaria. ;
- **Figlio:** ConversationApp;
- **Attributi:**
 - ui: HTMLElement

Attributo contenente l'HTMLElement dell'interfaccia dell'applicazione. Per informazioni sul tipo HTMLElement, fare riferimento alla pagina seguente:<https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement>;

- **Metodi:**

+ <<Create>> **createApplication(pkg: ApplicationPackage): Application**
 Metodo che permette di costruire un Application a partire da un ApplicationPackage.

Esegue il codice presente all'interno del campo **setup** di **pkg**, eseguendo il binding di **this** per far sì che tale codice abbia accesso all'oggetto che sta venendo creato, quindi crea il metodo **runCmd** a partire dal codice presente all'interno di **pkg.cmdHandler**, eseguendo nuovamente il binding di **this**;

Parametri:

* **pkg: ApplicationPackage**

Package contenente i dati relativi all'applicazione da istanziare;

+ **runCmd(cmd: String, params: ResponseBody): void**

Metodo che permette di eseguire un comando sull'applicazione. Viene creato a partire da **pkg**;

Parametri:

* **cmd: String**

Parametro contenente il comando da somministrare all'applicazione;

* **params: ResponseBody**

Parametro contenente l'array dei parametri del comando da somministrare all'applicazione;

+ **getUI(): HTMLElement**

Metodo che permette di restituire l'interfaccia dell'applicazione da eseguire, sotto forma di un oggetto di tipo **Element** (<https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement>). Questo **HTMLElement** verrà poi appeso al proprio frame da Manager;

- **Relazioni con le altre classi:**

- IN **State**
- IN **ConversationApp**
- OUT **ApplicationPackage**

ApplicationManagerObserver

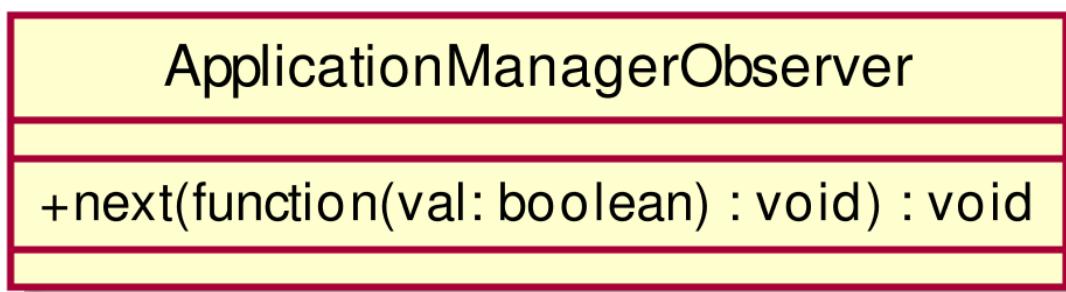


Figura 92: Client::ApplicationManager::ApplicationManagerObserver

- **Nome:** ApplicationManagerObserver;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di inviare il testo contenente la risposta fornita dal sistema al Manager;

- **Utilizzo:** fornisce un meccanismo per eseguire una funzione all'arrivo di una notifica dal DataArrivedObservable;

- **Padre:** ObserverAdapter;

- **Metodi:**

```
+ next(function(val: boolean): void): void
```

Questo metodo, all'arrivo della notifica dal DataArrivedSubject, permette di passare all'ApplicationManagerObserver una funzione da eseguire. ;

Parametri:

```
* function(val: boolean): void
```

Parametro contenente la funzione da eseguire;

- **Relazioni con le altre classi:**

- IN Manager

- IN DataArrivedObservable

ApplicationPackage

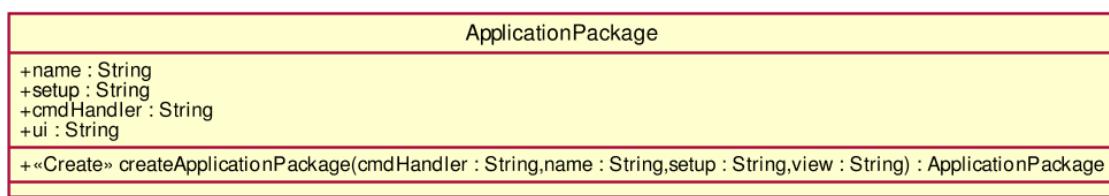


Figura 93: Client::ApplicationManager::ApplicationPackage

- **Nome:** ApplicationPackage;

- **Tipo:** Class;

- **Descrizione:** questa classe si occupa di rappresentare e definire il tipo di package che viene restituito da una interrogazione di un registry delle applicazioni;

- **Utilizzo:** fornisce gli attributi caratterizzanti una Application, necessari alla sua istanziazione. ;

- **Attributi:**

```
+ name: String
```

Attributo contenente il nome dell'applicazione recuperata;

```
+ setup: String
```

Attributo contenente il codice Javascript che deve essere eseguito nel costruttore dell'Application. Tale codice ha accesso al riferimento this dell'Application, e può utilizzarlo per definire i metodi e gli attributi dell'Application implementata;

```
+ cmdHandler: String
```

Attributo contenente il codice Javascript del metodo runCmd dell'Application implementata. Tale codice ha accesso al riferimento this dell'Application;

```
+ ui: String
```

Attributo contenente l'Element che racchiude l'interfaccia dell'applicazione;

- **Relazioni con le altre classi:**

- IN Application

- IN <> ApplicationRegistryClient
- IN ApplicationLocalRegistry

ApplicationRegistryLocalClient

ApplicationRegistryLocalClient
-registry : ApplicationLocalRegistry
+query(app_name : String) : ApplicationPackage
+register(name : String, pkg : ApplicationPackage) : boolean
+«Create» createApplicationRegistryLocalClient(client : ApplicationLocalRegistry) : ApplicationRegistryLocalClient

Figura 94: Client::ApplicationManager::ApplicationRegistryLocalClient

- **Nome:** ApplicationRegistryLocalClient;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di implementare l’interfaccia fornita da ApplicationRegistryClient. Interroga un LocalRegistry;
- **Utilizzo:** fornisce i meccanismi necessari ad interrogare il LocalRegistry e ad aggiungere un ApplicationPackage al suo interno;
- **Attributi:**
 - registry: ApplicationLocalRegistry
Attributo che permette l’accesso all’ApplicationLocalRegistry;
- **Metodi:**
 - + query(app_name: String): ApplicationPackage
Metodo che permette di interrogare ApplicationRegistryClient a partire dal nome dell’applicazione, ottenendo l’ApplicationPackage relativa ad essa. ;
Parametri:
 - * app_name: String
Parametro contenente il nome dell’applicazione che si vuole recuperare;
 - + register(name: String, pkg: ApplicationPackage): boolean
Metodo relativo alla registrazione di una applicazione nell’IRegistryClient. Questo metodo deve però impedire l’inserimento di package ”Parziali” di RegistryPackage, ovvero un package di tipo ApplicationPackage. ;
Parametri:
 - * name: String
Parametro contenente il nome dell’applicazione che si vuole registrare;
 - * pkg: ApplicationPackage
Parametro contenente l’ApplicationPackage relativo all’applicazione da registrare;
 - + <> createApplicationRegistryLocalClient(client: ApplicationLocalRegistry): ApplicationRegistryLocalClient
Costruttore che permette di effettuare una dependency injection di ApplicationLocalRegistry;
Parametri:
 - * client: ApplicationLocalRegistry
Parametro relativo all’ApplicationLocalRegistry di cui viene effettuata la dependency injection;

- Relazioni con le altre classi:

- OUT <> ApplicationRegistryClient
- OUT ApplicationLocalRegistry

Manager

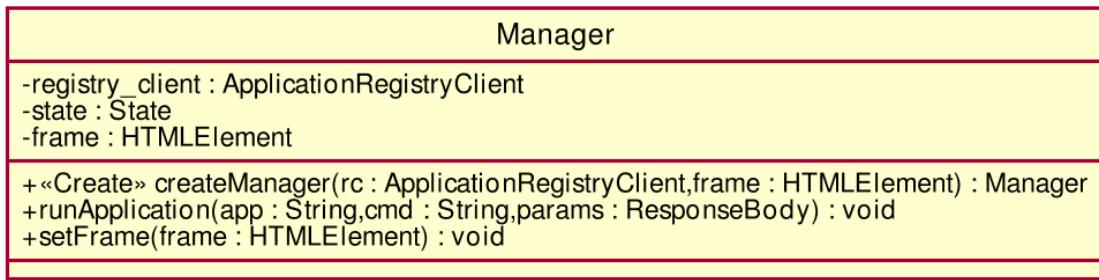


Figura 95: Client::ApplicationManager::Manager

- **Nome:** Manager;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di gestire il cambio delle applicazioni nel client;
- **Utilizzo:** fornisce all'ApplicationManager un meccanismo per cambiare l'applicazione in esecuzione. Questo avviene salvando lo stato dell'applicazione corrente nello State e recuperando la nuova applicazione da State (se presente) o dal ApplicationRegistryClient;
- **Attributi:**

– **registry_client:** ApplicationRegistryClient
 Attributo utilizzato per ottenere i dati relativi ad un'applicazione da istanziare;
 – **state:** State
 Attributo che tiene traccia delle applicazione già eseguite e del loro stato;
 – **frame:** HTMLElement
 Attributo che contiene l'elemento del DOM al quale verrà appesa la view dell'applicazione. Per informazioni sul tipo HTMLElement fare riferimento alla pagina <https://developer.mozilla.org/en/docs/Web/API/HTMLElement>;

- **Metodi:**

+ <> **createManager(rc: ApplicationRegistryClient, frame: HTMLElement) : Manager**
 Metodo che permette di costruire un Manager. Permette di effettuare la dependency injection di un ApplicationRegistryClient e di un Element al manager, e si occupa di creare lo state iniziale;
 Parametri:
 * **rc:** ApplicationRegistryClient
 Parametro che permette la dependency injection di un ApplicationRegistryClient all'interno di Manager;
 * **frame:** HTMLElement
 Parametro che permette la dependency injection di HTMLElement all'interno di Manager. Tale oggetto rappresenta l'elemento del DOM al quale verrà appesa l'interfaccia dell'applicazione;

+ runApplication(app: String, cmd: String, params: ResponseBody): void
 Metodo che permette di passare a Manager il nome dell'applicazione e del comando da eseguire;
 Parametri:

- * app: String
 Parametro contenente il nome dell'applicazione da eseguire;
- * cmd: String
 Parametro contenente il comando da dare all'applicazione app;
- * params: ResponseBody
 Parametro contenente l'insieme dei dati relativi alla risposta ricevuta;

+ setFrame(frame: HTMLElement): void
 Metodo che permette di effettuare la dependency injection dell'elemento del DOM al quale sarà appesa l'interfaccia dell'applicazione da eseguire. Il nuovo HTMLElement sostituirà quello già presente all'interno di Manager, permettendo così di cambiare la posizione dell'interfaccia utente dell'applicazione all'interno della pagina;
 Parametri:

- * frame: HTMLElement
 Parametro che contiene l'elemento del DOM;

- Relazioni con le altre classi:

- OUT State
- OUT <> ApplicationRegistryClient
- OUT ApplicationManagerObserver

State

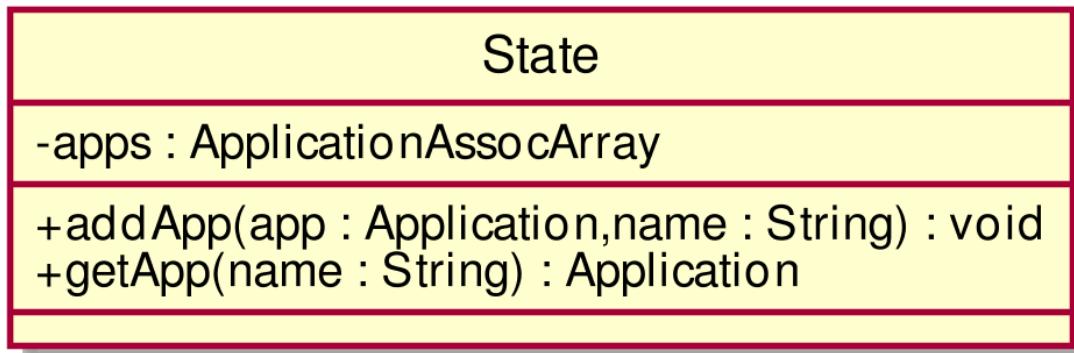


Figura 96: Client::ApplicationManager::State

- **Nome:** State;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di salvare lo stato attuale delle applicazioni la cui esecuzione vuole essere sospesa;
- **Utilizzo:** fornisce al Manager un meccanismo per salvare lo stato attuale delle applicazioni la cui esecuzione è stata sospesa, in modo da poterlo recuperare una volta riattivata. Lo stato comprende tutti gli attributi e tutti i metodi dell'istanza dell'applicazione;

- **Attributi:**

- `apps: ApplicationAssocArray`

Array associativo contenente le applicazioni che sono state sospese o in esecuzione. ;

- **Metodi:**

- + `addApp(app: Application, name: String): void`

Metodo che permette di aggiungere un'applicazione allo State, sovrascrivendo quella già esistente se presente;

Parametri:

- * `app: Application`

Parametro contenente l'Application da aggiungere allo State;

- * `name: String`

Nome sotto il quale l'applicazione verrà salvata;

- + `getApp(name: String): Application`

Metodo che permette di estrarre un'applicazione dallo State. Nel caso l'applicazione col nome specificato non sia presente, questo metodo restituisce null;

Parametri:

- * `name: String`

Nome dell'applicazione della quale si vuole recuperare l'istanza in esecuzione;

- **Relazioni con le altre classi:**

- IN `Manager`

- OUT `Application`

Client::Logic

Package contenente le classi che gestiscono la logica del client e che si occupano della comunicazione con il back-end.

Classi

DataArrivedObservable



Figura 97: Client::Logic::DataArrivedObservable

- **Nome:** DataArrivedObservable;
- **Tipo:** Class;
- **Descrizione:** classe che rappresenta un Observable che notifica gli observer interessati quando sono arrivati dei dati dall'API Gateway;
- **Utilizzo:** fornisce il meccanismo che permette agli observer di iscriversi per essere notificati quando arriva una risposta dall'API Gateway. Viene creata a partire dal DataArrivedSubject, per evitare di dover condividere quest'ultimo;
- **Relazioni con le altre classi:**
 - IN Logic
 - OUT ApplicationManagerObserver
 - OUT PlayerObserver

DataArrivedSubject

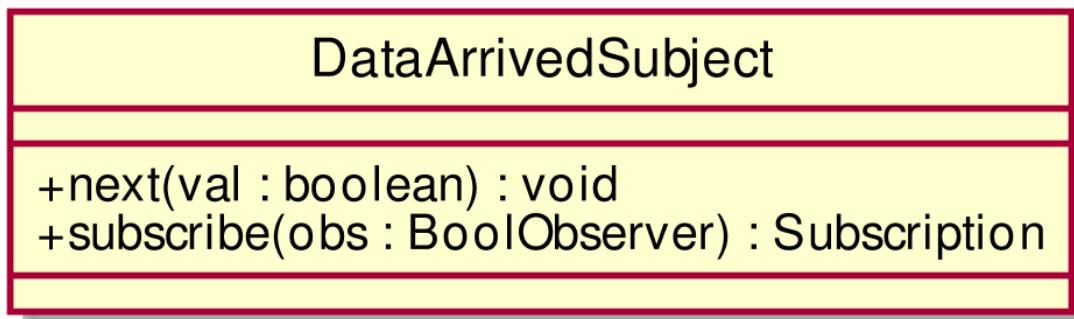


Figura 98: Client::Logic::DataArrivedSubject

- **Nome:** DataArrivedSubject;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di notificare gli observer collegati che i dati relativi ad una risposta sono arrivati;
- **Utilizzo:** fornisce un meccanismo che permette l'invio dei dati, il testo della risposta dell'assistente virtuale, una volta che questi ultimi sono stati forniti dal (sistema). Gli observer collegati sono:
 - ApplicationManagerObserver;
 - PlayerObserver.
;
- **Padre:** Subject;
- **Metodi:**
 - + next(val: boolean): void
Questo metodo permette di notificare il PlayerObserver e l'ApplicationManagerObserver;
Parametri:

```

* val: boolean
  Parametro contenente il valore con il quale notificare il PlayerObserver e l'ApplicationManagerObserver;

+ subscribe(obs: BoolObserver): Subscription
  Questo metodo permette di iscrivere il PlayerObserver e l'ApplicationManagerObserver;
  Parametri:
    * obs: BoolObserver
      Parametro contenente il valore con il quale il PlayerObserver e l'ApplicationManagerObserver devono essere iscritti;

```

- **Relazioni con le altre classi:**

- IN [Logic](#)

HttpError

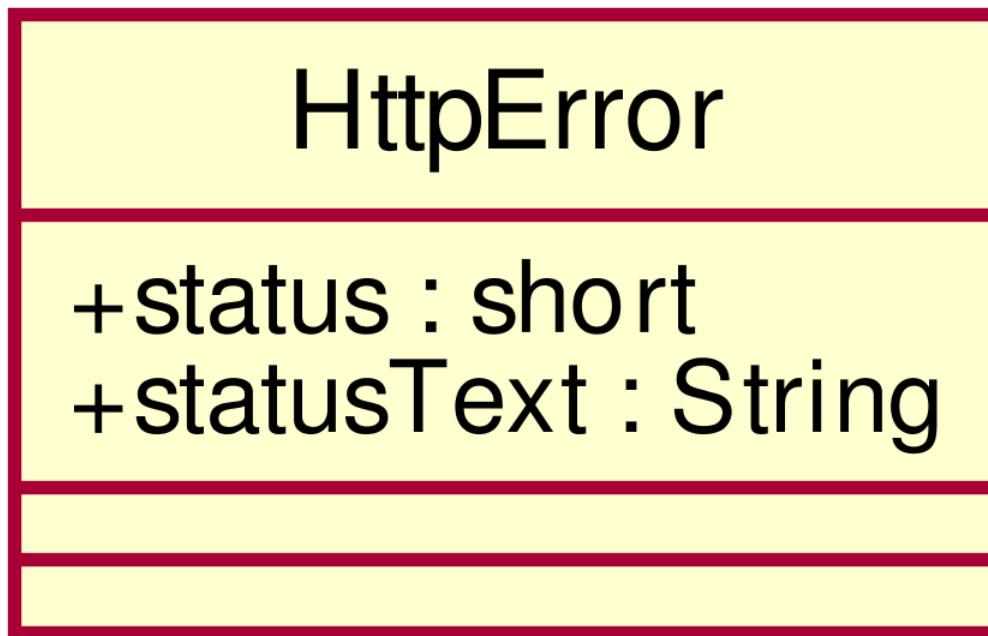


Figura 99: Client::Logic::HttpError

- **Nome:** HttpError;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare un errore HTTP;
- **Utilizzo:** viene utilizzata per fornire un messaggio di errore in seguito a richieste HTTP fallite;
- **Attributi:**

- + status: short**

Attributo contenente il codice di stato HTTP, come definito nella sezione 6 dello standard rfc7231 (<https://tools.ietf.org/html/rfc7231#section-6>);

```
+ statusText: String
Attributo contenente un messaggio descrittivo dell'errore;
```

- Relazioni con le altre classi:

- IN [HttpPromise](#)
- IN [Logic](#)

[HttpPromise](#)

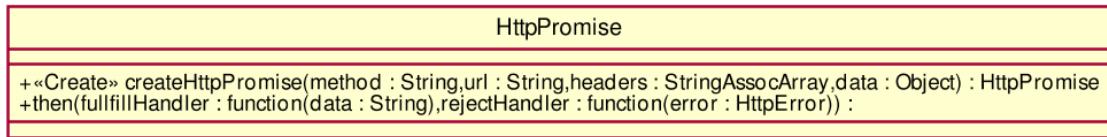


Figura 100: Client::Logic::HttpPromise

- **Nome:** [HttpPromise](#);
- **Tipo:** Class;
- **Descrizione:** promise relativa ad una richiesta HTTP;
- **Utilizzo:** viene utilizzata per interfacciarsi alle richieste HTTP utilizzando l'approccio delle Promises al posto di quello dei callback fornito dalle API JavaScript. ;
- **Metodi:**

```
+ <<Create>> createHttpPromise(method: String, url: String, headers: StringAssocArray,
data: Object): HttpPromise
```

Constructor, si occupa di creare la promessa relativa ad una richiesta HTTP fatta con metodo, headers e url specificati;

Parametri:

- * **method:** String
Metodo della richiesta HTTP;

- * **url:** String
Url a cui fare la richiesta;

- * **headers:** StringAssocArray
Array associativo contenente gli headers. La chiave rappresenta il nome dell'header;

- * **data:** Object
Dati da mandare con la richiesta;

```
+ then(fullfillHandler: function(data : String), rejectHandler: function(error : HttpError)):
```

Metodo utilizzato per specificare cosa fare nel caso in cui la Promise sia soddisfatta oppure rigettata;

Parametri:

- * **fullfillHandler:** function(data : String)
Metodo da chiamare in caso di promessa soddisfatta;

- * **rejectHandler:** function(error : HttpError)
Funzione da chiamare in caso di promessa rigettata;

- Relazioni con le altre classi:

- IN [Logic](#)

- OUT `HttpError`

Logic

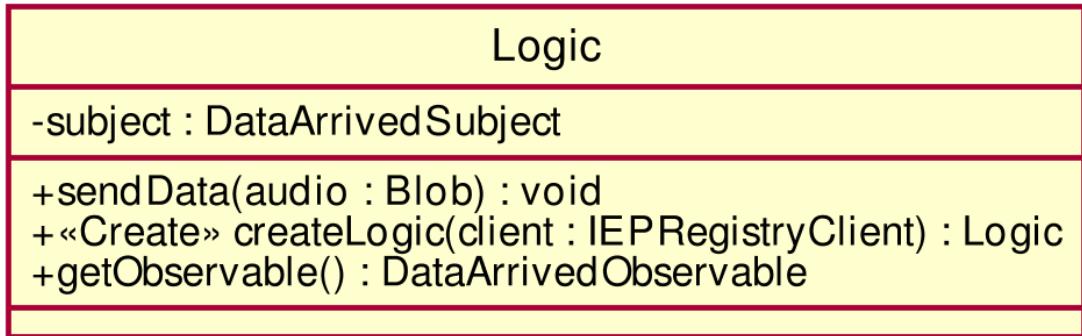


Figura 101: Client::Logic::Logic

- **Nome:** Logic;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di comunicare con l'API Gateway;
- **Utilizzo:** fornisce dei meccanismi per:
 - la comunicazione con l'API gateway;
 - pubblicare eventi all'arrivo di una risposta dall'API Gateway.
- ;
- **Attributi:**
 - subject: DataArrivedSubject
Attributo contenente il DataArrivedSubject per notificare il DataArrivedObservable che sono arrivati dei dati dall'API Gateway;
- **Metodi:**
 - + sendData(audio: Blob): void
Metodo che permette di inviare l'audio all'API Gateway;
Parametri:
 * audio: Blob
 Parametro contenente l'audio da inviare all'APIGateway;
 - + <<Create>> createLogic(client: IEPRegistryClient): Logic
Metodo che permette di istanziare Logic a partire da un IEPRegistryClient;
Parametri:
 * client: IEPRegistryClient
 Parametro contenente l'interfaccia per interrogare l'EndPointRegistryAdapter;
 - + getObservable(): DataArrivedObservable
Metodo utilizzato per ottenere un Observable per l'arrivo dei dati dall'API Gateway;
- **Relazioni con le altre classi:**
 - OUT `HttpError`

- OUT `DataArrivedSubject`
- OUT `DataArrivedObservable`
- OUT `HttpPromise`

LogicObserver

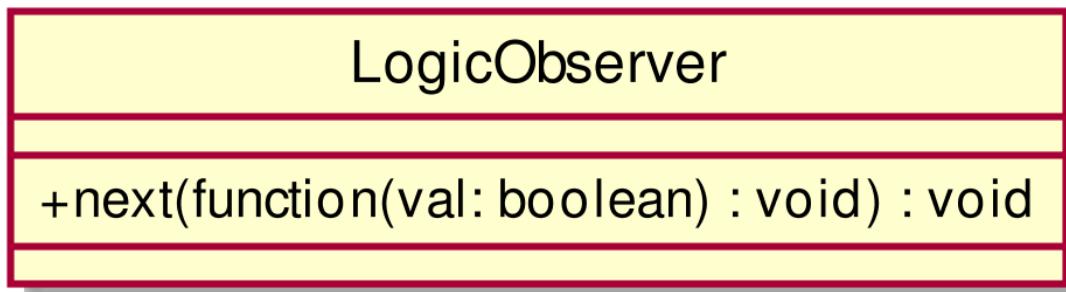


Figura 102: Client::LogicObserver

- **Nome:** LogicObserver;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di inviare l'audio relativo a ciò che l'utente ha comunicato a Logic. ;
- **Utilizzo:** fornisce un meccanismo per eseguire una funzione all'arrivo di una notifica da SpeechEndObservable;
- **Padre:** ObserverAdapter;
- **Metodi:**
 - + next(function(val: boolean): void)

Questo metodo, all'arrivo della notifica dall'EndSpeechDataSubject, permette di passare al LogicObserver un'operazione da eseguire. ;
Parametri:

 - * function(val: boolean): void

Parametro contenente la funzione da eseguire;
- **Relazioni con le altre classi:**
 - IN `SpeechEndObservable`

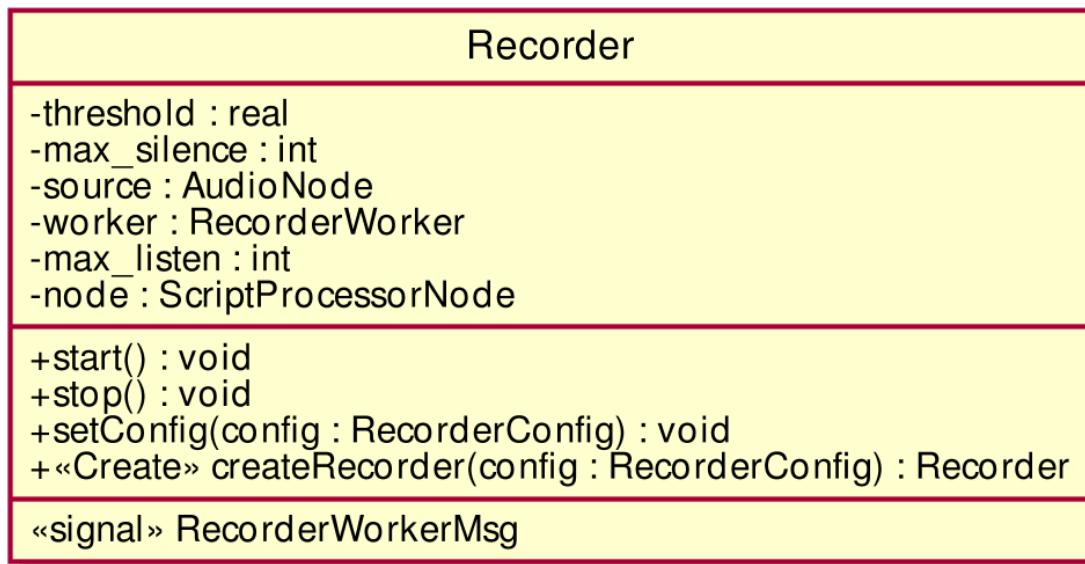
Client::Recorder

Package contenente le classi che realizzano la registrazione audio.

Classi

Recorder

- **Nome:** Recorder;
- **Tipo:** Class;

**Figura 103:** Client::Recorder::Recorder

- **Descrizione:** questa classe si occupa della registrazione della conversazione;
- **Utilizzo:** fornisce al client un meccanismo per registrare ciò che l'ospite comunica. Utilizza il volume rilevato per capire se l'ospite sta parlando, e genera i relativi comandi da mandare al **RecorderWorker**. Quando inizia una nuova registrazione, l'audio della registrazione precedente viene eliminato chiamando il comando clear del Worker;
- **Attributi:**
 - threshold: real
Attributo contenente la soglia di volume per la registrazione;
 - max_silence: int
Attributo contenente il tempo in ms atteso prima che venga fermata la registrazione a causa del silenzio (volume del suono registrato al di sotto di threshold);
 - source: AudioNode
Attributo contenente la sorgente audio dalla quale vogliamo registrare;
 - worker: RecorderWorker
Attributo contenente l'oggetto **RecorderWorker** sfruttato per la registrazione;
 - max_listen: int
Attributo contenente il tempo massimo di registrazione, espresso in ms;
 - node: ScriptProcessorNode
Attributo contenente il nodo utilizzato per la registrazione dell'audio. ;
- **Metodi:**
 - + start(): void
Metodo che mette il **Recorder** in ascolto, in attesa che la soglia minima di volume venga superata. Quando tale soglia viene superata, viene generato un evento;
 - + stop(): void
Metodo che permette di terminare l'ascolto del **Recorder**;

+ `setConfig(config: RecorderConfig): void`
 Metodo che permette la modifica della configurazione del **Recorder**;
 Parametri:
 * `config: RecorderConfig`
 È un oggetto che rappresenta la configurazione della registrazione. ;
+ `<<Create>> createRecorder(config: RecorderConfig): Recorder`
 Metodo che permette di creare un oggetto di tipo **Recorder** a partire da un parametro
 di configurazione in formato JSON;
 Parametri:
 * `config: RecorderConfig`
 È un oggetto che rappresenta la configurazione della registrazione;

- **Relazioni con le altre classi:**

- OUT `RecorderConfig`
- OUT `RecorderMsg`
- OUT `RecorderWorker`
- OUT `RecorderWorkerMsg`
- OUT `BoolObserver`

- **Eventi gestiti:**

- `RecorderWorkerMsg`
 Messaggio mandato da `RecorderWorker` a `Recorder` quando l'encoding dell'audio è terminato. Contiene il blob del file audio codificato.

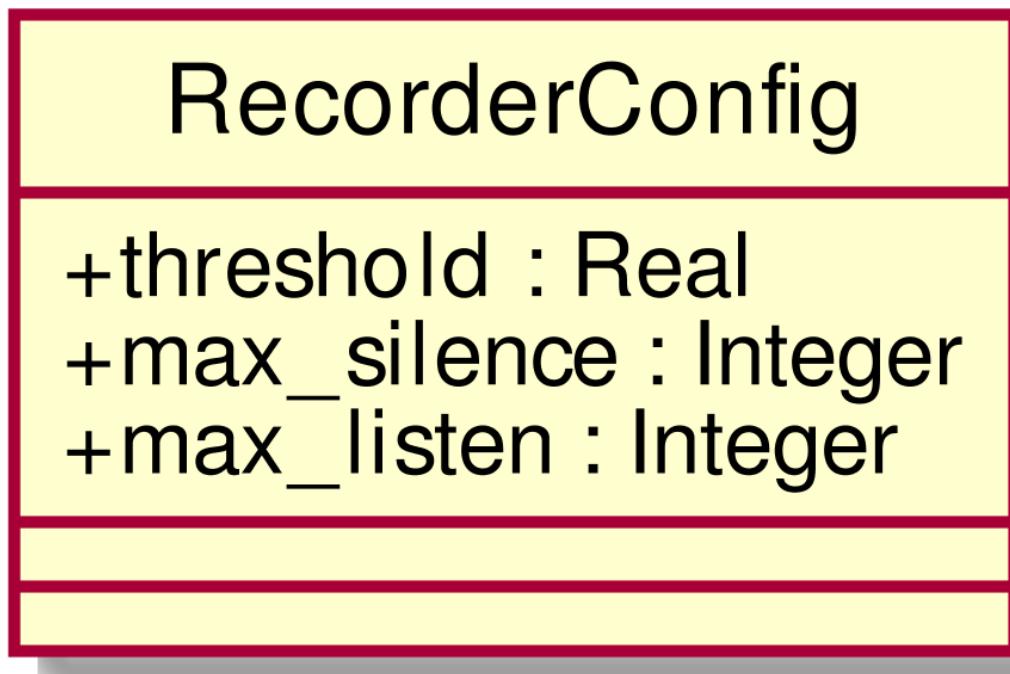
RecorderConfig

- **Nome:** `RecorderConfig`;
- **Tipo:** `Class`;
- **Descrizione:** questa classe contiene gli attributi necessari a configurare **Recorder**;
- **Utilizzo:** fornisce un meccanismo per configurare i parametri di **Recorder**. ;
- **Attributi:**

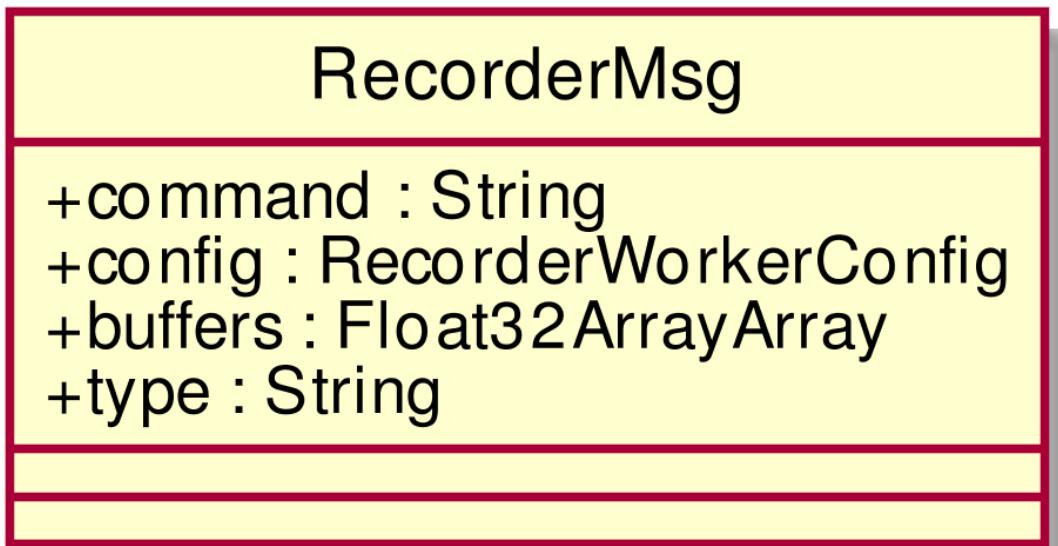
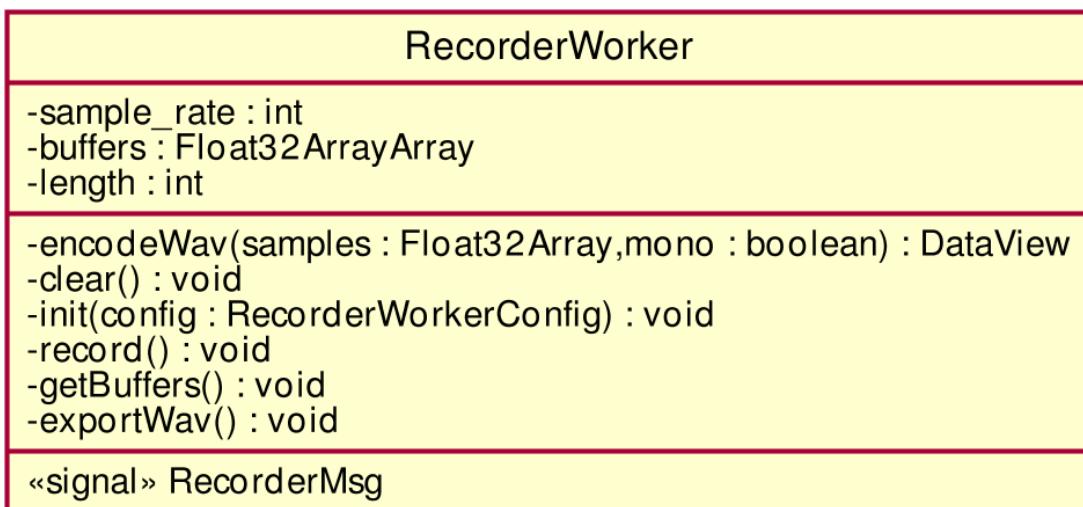
+ `threshold: Real`
 Indica la soglia di volume al di sopra della quale il `Recorder` deve iniziare la registrazione;
+ `max_silence: Integer`
 indica il tempo in ms che deve trascorrere prima che il `Recorder` smetta di registrare a causa di un silenzio (volume al di sotto di `threshold`). In caso il valore sia -1 (default), il `recorder` continuerà a registrare per un tempo indefinito;
+ `max_listen: Integer`
 indica il tempo massimo di registrazione, indicato in millisecondi;

- **Relazioni con le altre classi:**

- IN `Recorder`

**Figura 104:** Client::Recorder::RecorderConfig**RecorderMsg**

- **Nome:** RecorderMsg;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare un messaggio che viene mandato dal Recorder al RecorderWorker;
- **Utilizzo:** fornisce al Recorder un meccanismo per organizzare le informazioni che esso vuole passare a RecorderWorker;
- **Attributi:**
 - + command: String
Attributo contenente il comando da dare al RecorderWorker;
 - + config: RecorderWorkerConfig
Attributo contenente i parametri necessari alla configurazione del RecorderWorker. ;
 - + buffers: Float32ArrayArray
Questo attributo contiene dati grezzi della registrazione audio, da aggiungere ai buffer del Recorder. Contiene un buffer per ogni canale;
 - + type: String
Attributo contenente il tipo del messaggio;
- **Relazioni con le altre classi:**
 - IN RecorderWorker
 - IN Recorder

**Figura 105:** Client::Recorder::RecorderMsg**RecorderWorker****Figura 106:** Client::Recorder::RecorderWorker

- **Nome:** RecorderWorker;
- **Tipo:** Class;
- **Descrizione:** questa classe è una classe Worker che si occupa dell'effettiva registrazione dell'audio;
- **Utilizzo:** fornisce a Recorder un thread che si occupa di registrare effettivamente l'audio in modo da non appesantire il thread dell'interfaccia grafica. Eredita la classe Worker di JavaScript e ne ridefinisce il metodo onmessage(Event e). (tutti i metodi sono privati perché vengono usati messaggi per comunicare);

- **Attributi:**

- `sample_rate: int`

Questo attributo contiene la sample rate della registrazione audio nei buffer;

- `buffers: Float32ArrayArray`

Questo attributo contiene i buffer per i dati dell'audio. sono presenti due buffer, uno per ogni canale;

- `length: int`

Questo attributo contiene la lunghezza dei buffer;

- **Metodi:**

- `encodeWav(samples: Float32Array, mono: boolean): DataView`

Metodo che permette la codifica dell'audio in formato wav a uno o due canali;

Parametri:

- * `samples: Float32Array`

Attributo contenente i dati dei sample audio che verranno utilizzati per la codifica;

- * `mono: boolean`

Attributo contenente un valore booleano che indica se il file audio ha uno o due canali.

In caso il canale sia uno, allora questo attributo dovrà contenere il valore true;

- `clear(): void`

Metodo che permette di eliminare tutti i dati relativi all'audio registrato fino a quel momento;

- `init(config: RecorderWorkerConfig): void`

Metodo che permette di inizializzare la configurazione del `RecorderWorker` in seguito alla ricezione di un messaggio dal `Recorder`;

Parametri:

- * `config: RecorderWorkerConfig`

Questo parametro contiene la configurazione;

- `record(): void`

Metodo che permette di iniziare a registrare l'audio dal microfono in seguito ad un messaggio ricevuto dal `Recorder`;

- `getBuffers(): void`

Metodo che permette di ottenere i buffer nei quali sono salvati i dati relativi ai samples dell'audio;

- `exportWav(): void`

Metodo che permette di esportare al `Recorder` il Blob del file wav contenente l'audio della registrazione;

- **Relazioni con le altre classi:**

- IN `Recorder`

- OUT `RecorderWorkerConfig`

- OUT `RecorderWorkerMsg`

- OUT `RecorderMsg`

- **Eventi gestiti:**

- **RecorderMsg**

Messaggio mandato da Recorder a RecorderWorker per comunicare l'operazione da eseguire in background.

RecorderWorkerConfig

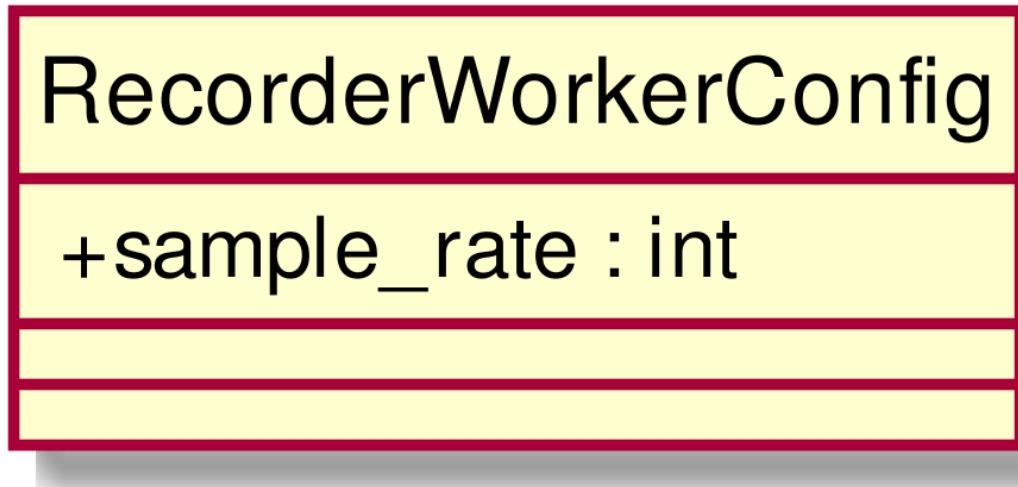
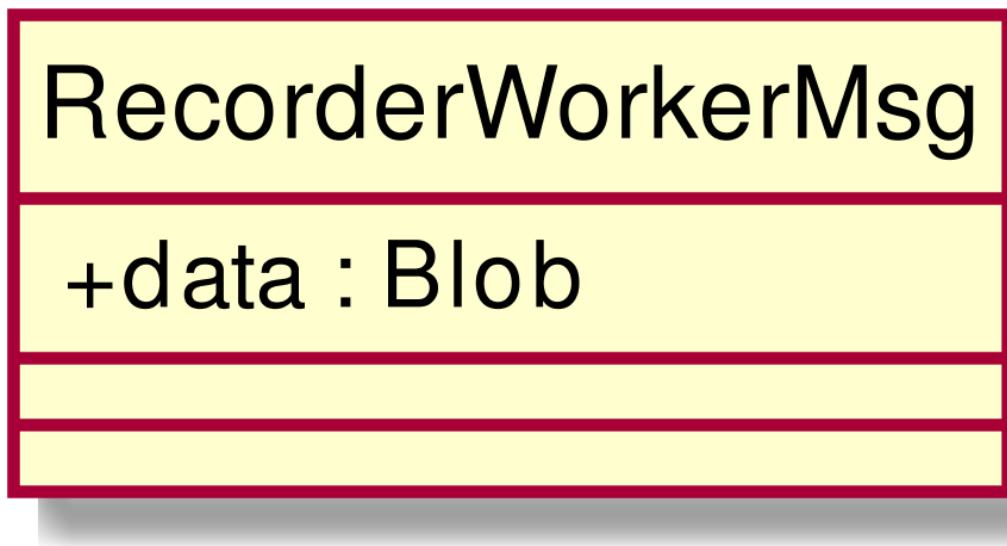


Figura 107: Client::Recorder::RecorderWorkerConfig

- **Nome:** RecorderWorkerConfig;
- **Tipo:** Class;
- **Descrizione:** questa classe contiene tutti i parametri relativi alla configurazione di un RecorderWorker;
- **Utilizzo:** fornisce al RecorderWorker un meccanismo per organizzare le informazioni che esso vuole passare a Recorder;
- **Attributi:**
 - + sample_rate: int
Attributo contenente il valore definito per il sample rate della registrazione in Hz;
- **Relazioni con le altre classi:**
 - IN RecorderWorker

RecorderWorkerMsg

- **Nome:** RecorderWorkerMsg;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di rappresentare un messaggio che viene mandato dal RecorderWorker al Recorder;
- **Utilizzo:** fornisce al RecorderWorker un meccanismo per organizzare le informazioni che esso vuole passare a Recorder;
- **Attributi:**

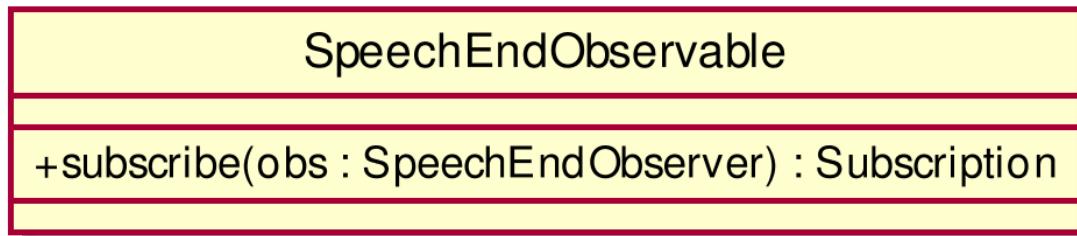
**Figura 108:** Client::RecorderWorkerMsg

+ data: Blob
Attributo contenente il file audio in formato WAV;

- **Relazioni con le altre classi:**

- IN [RecorderWorker](#)
- IN [Recorder](#)

SpeechEndObservable

**Figura 109:** Client::Recorder::SpeechEndObservable

- **Nome:** `SpeechEndObservable`;
- **Tipo:** `Class`;
- **Descrizione:** classe che rappresenta un `Observable` che notifica gli observer interessati quando l'ospite ha finito di parlare;
- **Utilizzo:** fornisce il meccanismo che permette agli observer di iscriversi per essere notificati quando l'ospite finisce di parlare. Viene creata a partire dal `SpeechEndSubject`, per evitare di dover condividere quest'ultimo;
- **Metodi:**

+ subscribe(obs: SpeechEndObserver): Subscription

Metodo che permette di iscrivere un Observer all'Observable;

Parametri:

* obs: SpeechEndObserver

Parametro che rappresenta l'observer che si vuole iscrivere;

- Relazioni con le altre classi:

- OUT LogicObserver

SpeechEndSubject

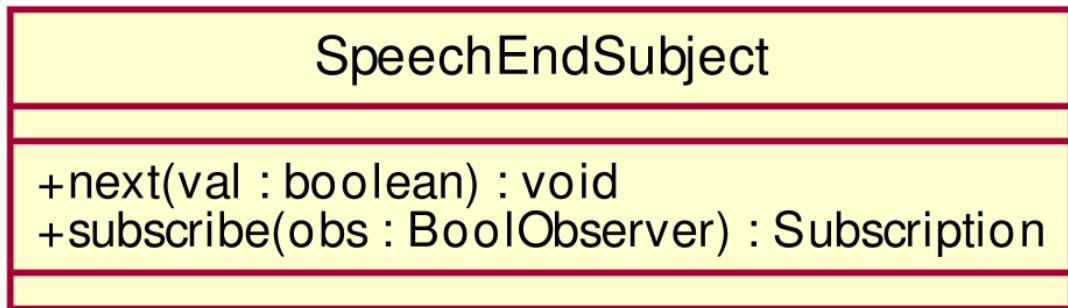


Figura 110: Client::Recorder::SpeechEndSubject

- **Nome:** SpeechEndSubject;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di notificare l'observer collegato che l'utente ha finito di parlare;
- **Utilizzo:** fornisce un meccanismo che permette l'invio del file audio contenente il messaggio dell'utente una volta che quest'ultimo ha terminato di parlare. L'observer collegato è LogicObserver;
- **Padre:** Subject;
- **Metodi:**

+ next(val: boolean): void

Questo metodo permette di notificare il LogicObserver;

Parametri:

* val: boolean

Parametro contenente il valore con il quale notificare il LogicObserver;

+ subscribe(obs: BoolObserver): Subscription

Questo metodo permette di iscrivere il LogicoObserver;

Parametri:

* obs: BoolObserver

Parametro contenente il valore con il quale il LogicObserver dev'essere iscritto;

Client::TTS

Package contenente le componenti che realizzano il text to speech.

Classi

Player

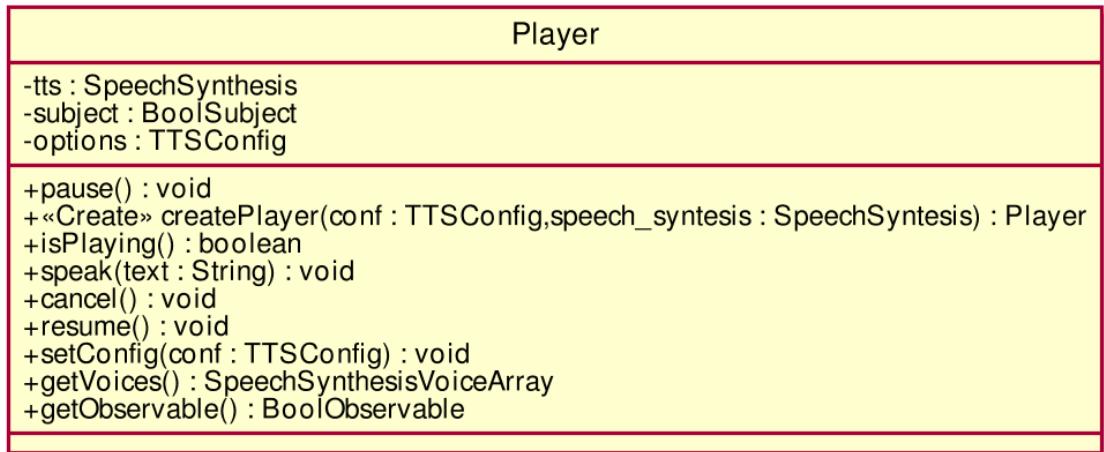


Figura 111: Client::TTS::Player

- **Nome:** Player;
- **Tipo:** Class;
- **Descrizione:** questa classe si occupa di riprodurre la risposta, fornita dal sistema, all'ospite;
- **Utilizzo:** fornisce al client funzionalità di riproduzione del file audio di risposta, da parte del sistema, relativo a ciò che l'ospite comunica. Permette all'ospite di fermare e riavviare la riproduzione audio e di impostare il volume. ;
- **Attributi:**
 - **tts:** SpeechSynthesis
Attributo contenente uno SpeechSynthesis di JavaScript Web Speech API il quale fornisce dei metodi per la gestione del file audio. Per la relativa documentazione, consultare questa pagina <https://developer.mozilla.org/en-US/docs/Web/API/SpeechSynthesis>;
 - **subject:** BoolSubject
Attributo contenente un BoolSubject, il quale emette item del tipo:
 - * TRUE quando l'utente sta parlando;
 - * FALSE altrimenti.

Questo attributo serve per disattivare il Recorder durante la riproduzione del file audio relativo alla risposta, fornita dal sistema;

- **options:** TTSCConfig

Attributo contenente l'insieme delle opzioni relative alla configurazione del Player;

- **Metodi:**

+ **pause(): void**

Metodo che permette di mettere in pausa la riproduzione dell'audio;

+ <<Create>> **createPlayer(conf: TTSCConfig, speech_syntesis: SpeechSyntesis): Player**

Metodo che si occupa di creare un oggetto di tipo Player con una determinata configurazione. Permette di fornire un'oggetto SpeechSyntesis al Player, effettuandone

quindi la dependency injection a costruttore;

Parametri:

* `conf: TTSSConfig`

Rappresenta l'audio da utilizzare nella creazione di un Player;

* `speech_syntesis: SpeechSynthesis`

Parametro che rappresenta l'oggetto `SpeechSynthesis` di cui viene effettuata la constructor-based dependency injection;

+ `isPlaying(): boolean`

Metodo che permette di capire se il Player sta riproducendo un audio o meno;

+ `speak(text: String): void`

Metodo che fa pronunciare una frase dal Player;

Parametri:

* `text: String`

Questo parametro contiene il testo da pronunciare;

+ `cancel(): void`

Metodo che permette di bloccare la riproduzione dell'audio;

+ `resume(): void`

Metodo che permette di riprendere la riproduzione audio dal punto in cui era stata interrotta;

+ `setConfig(conf: TTSSConfig): void`

Metodo che permette di modificare la configurazione;

Parametri:

* `conf: TTSSConfig`

È l'insieme delle opzioni da passare;

+ `getVoices(): SpeechSynthesisVoiceArray`

Metodo che restituisce una lista con le voci disponibili nel sistema per la sintesi vocale;

+ `getObservable(): BoolObservable`

Metodo che ritorna l'Observable relativo all'attributo `subject`;

- **Relazioni con le altre classi:**

- IN `PlayerObserver`

- OUT `TTSSConfig`

- OUT `BoolSubject`

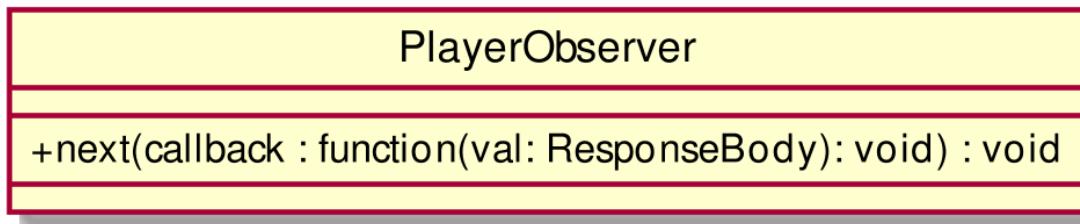
- OUT `BoolObservable`

PlayerObserver

- **Nome:** `PlayerObserver`;

- **Tipo:** Class;

- **Descrizione:** questa classe si occupa di inviare l'audio contenente la risposta fornita dal sistema al Player. È la classe che funziona da Observer;

**Figura 112:** Client::TTS::PlayerObserver

- **Utilizzo:** fornisce un meccanismo per l'invio dell'audio relativo alla risposta fornita dal sistema. ;
- **Padre:** `ObserverAdapter`;
- **Metodi:**
 - + `next(callback: function(val: ResponseBody): void)`: void
Questo metodo, all'arrivo della notifica dal `DataArrivedSubject`, permette di passare al `PlayerObserver` un'operazione da eseguire. ;
Parametri:
 * `callback: function(val: ResponseBody): void`
 Parametro contenente la funzione di callback da eseguire all'arrivo dei dati;
- **Relazioni con le altre classi:**
 - IN `DataArrivedObservable`
 - OUT `Player`

TTSTConfig

- **Nome:** `TTSTConfig`;
- **Tipo:** `Class`;
- **Descrizione:** questa classe contiene tutti i parametri relativi alla configurazione di un `Player`;
- **Utilizzo:** fornisce al `Player` un meccanismo per gestire i parametri relativi alla configurazione per la riproduzione di un file audio. Per la relativa documentazione, consultare la pagina <https://developer.mozilla.org/it/docs/Web/API/SpeechSynthesisUtterance>;
- **Attributi:**
 - + `lang: String`
Attributo contenente la lingua dell'audio che deve essere pronunciato dal `Player`. ;
 - + `pitch: float`
Attributo contenente il valore del pitch (intonazione) del file da riprodurre;
 - + `rate: int`
Attributo contenente la velocità alla quale il file audio deve essere pronunciato. ;
 - + `voice: SpeechSynthesisVoice`
Attributo contenente la voce con la quale il file audio deve essere pronunciato. ;
 - + `volume: int`
Attributo contenente il volume al quale il file audio deve essere pronunciato;
- **Relazioni con le altre classi:**

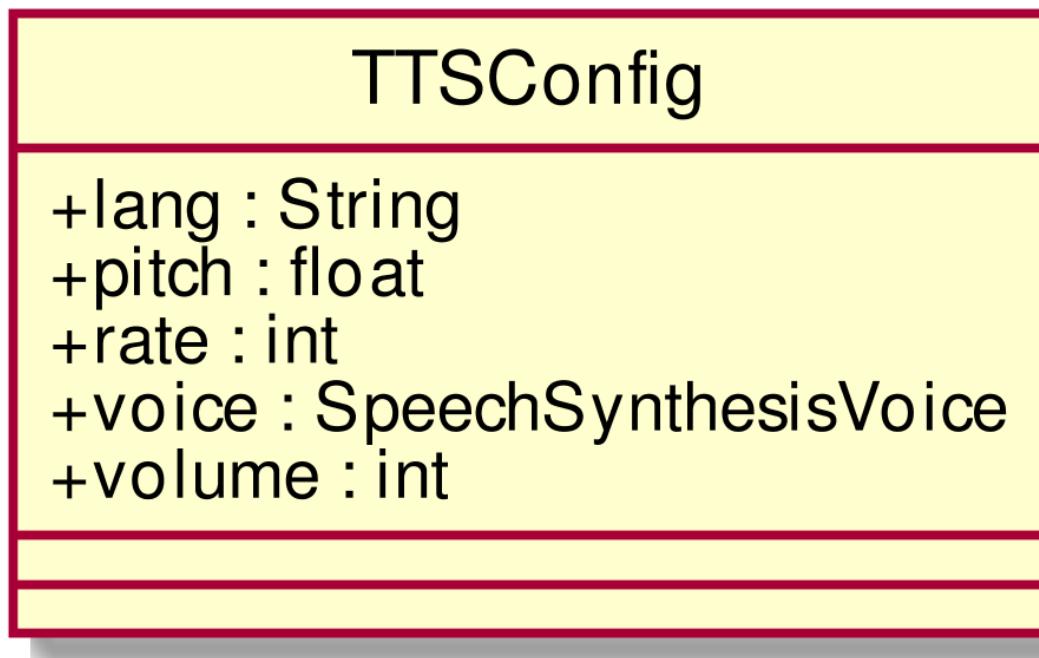


Figura 113: Client::TTS::TTSTConfig

– IN Player

Client::Utility

Package contenente classi e interfacce, dallo scopo generico, utili ad altri package del client.

Classi

BoolObservable

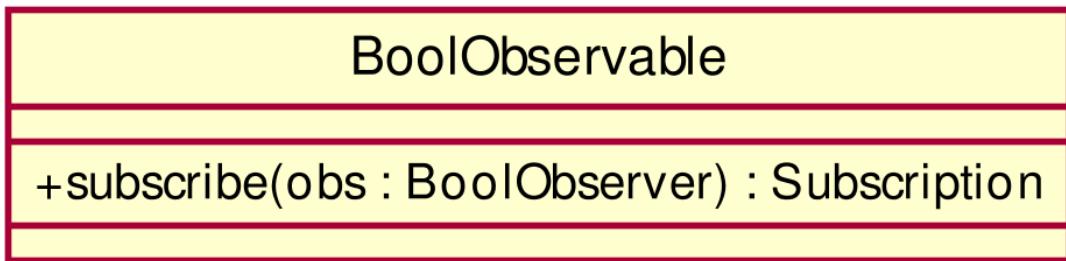


Figura 114: Client::Utility::BoolObservable

- **Nome:** BoolObservable;
- **Tipo:** Class;
- **Descrizione:** ereditata da Observable, con eventi di tipo boolean;

- **Utilizzo:** viene utilizzata come observable;

- **Metodi:**

+ subscribe(obs: BoolObserver): Subscription

Iscrive l'Observer all'Observable;

Parametri:

* obs: BoolObserver

OsservatoreBool;

- **Relazioni con le altre classi:**

– IN Player

– OUT BoolObserver

BoolObserver

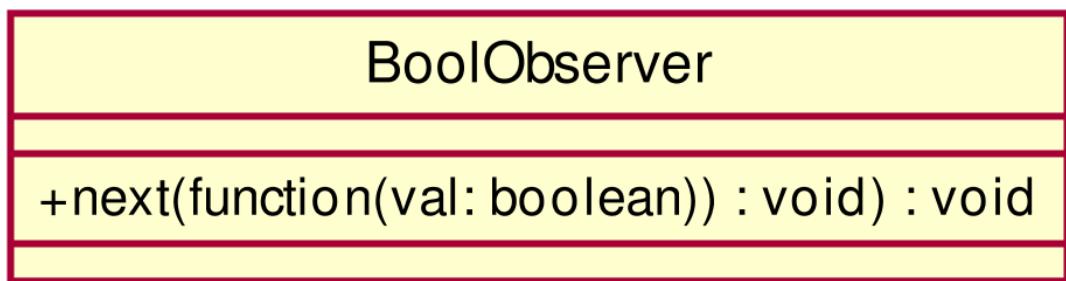


Figura 115: Client::Utility::BoolObserver

- **Nome:** BoolObserver;

- **Tipo:** Class;

- **Descrizione:** questa classe;

- **Utilizzo:** fornisce un meccanismo per eseguire una funzione all'arrivo di una notifica dal BoolObservable;

- **Metodi:**

+ next(function(val: boolean)): void

Questo metodo, all'arrivo della notifica dal BoolObservable, permette di passare al BoolObserver un'operazione da eseguire;

Parametri:

* function(val: boolean)): void

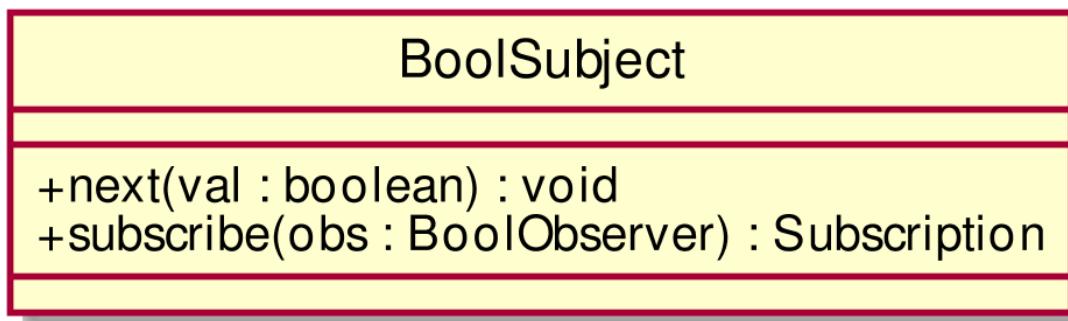
Parametro contenente la funzione da eseguire. ;

- **Relazioni con le altre classi:**

– IN Recorder

– IN BoolObservable

– IN BoolSubject

**Figura 116:** Client::Utility::BoolSubject

BoolSubject

- **Nome:** BoolSubject;
- **Tipo:** Class;
- **Descrizione:** classe che eredita da Subject, che fa uso di valori boolean;
- **Utilizzo:** viene utilizzata nella stessa maniera di Subject, con valori di tipo boolean;
- **Metodi:**
 - + `next(val: boolean): void`
Questo metodo permette di notificare il BoolSubject;
Parametri:
 * `val: boolean`
 Parametro contenente il valore con il quale notificare il BoolSubject;
 - + `subscribe(obs: BoolObserver): Subscription`
Questo metodo permette di iscrivere il BoolObserver;
Parametri:
 * `obs: BoolObserver`
 Parametro contenente il valore con il quale iscrivere il BoolObserver;
- **Relazioni con le altre classi:**
 - IN `Player`
 - OUT `BoolObserver`

RxJS

Libreria esterna

Classi

Observable

- **Nome:** Observable;
- **Tipo:** Class;
- **Descrizione:** observable di RxJs;

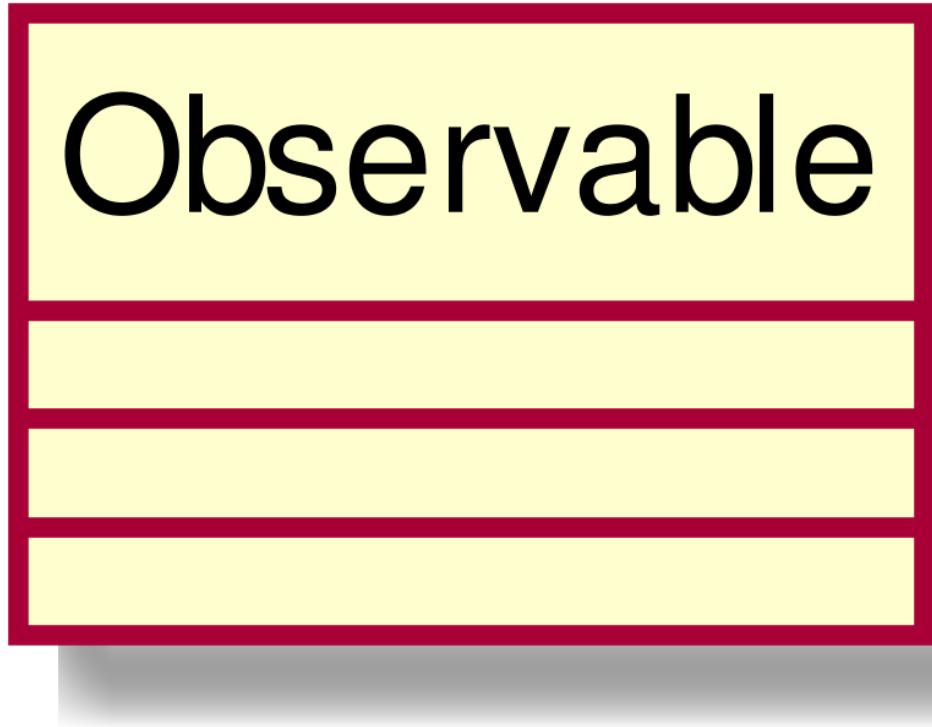


Figura 117: RxJS::Observable

- **Utilizzo:** ;
- **Figli:** UserObservable, ConversationObservable, GuestObservable, TaskObservable, RuleObservable, AgentObservable, ErrorObservable;

Observer

- **Nome:** Observer;
- **Tipo:** Class;
- **Descrizione:** observer di RxJS;
- **Utilizzo:** ;

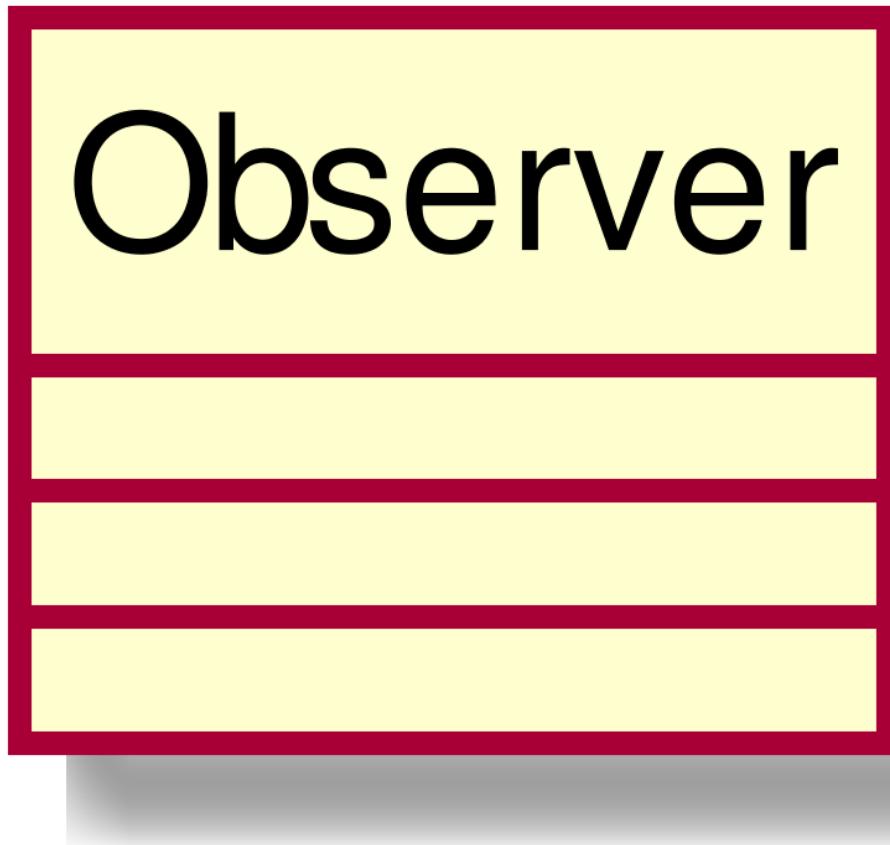


Figura 118: RxJS::Observer

Subject

- **Nome:** Subject;
- **Tipo:** Class;
- **Descrizione:** rxJS5;
- **Utilizzo:** ;
- **Figli:** SpeechEndSubject, DataArrivedSubject;

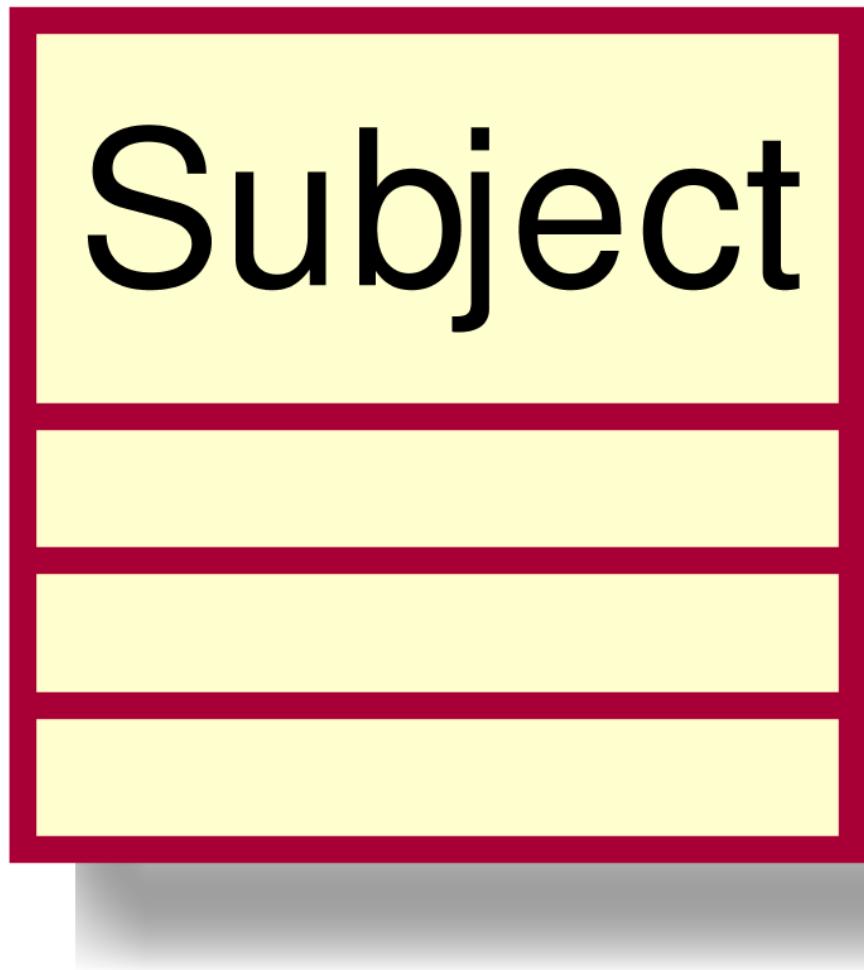


Figura 119: RxJS::Subject

Slack

Package contenente tutte le classi che fanno parte dell'SDK per Node.js di Slack.

Classi

WebClient

- **Nome:** WebClient;
- **Tipo:** Class;
- **Descrizione:** questa classe rappresenta il modulo node.js che si occupa di comunicare con le API Web di Slack;



Figura 120: Slack::WebClient

- **Utilizzo:** fornisce i metodi necessari ad usufruire delle funzionalità offerte dalle API Web di slack. Per ulteriori informazioni, fare riferimento alla documentazione di slack (<https://api.slack.com/>);

WatsonDeveloperCloud

SDK node.js per l'accesso ai servizi cloud IBM Watson.

Classi

SpeechToTextV1

- **Nome:** SpeechToTextV1;
- **Tipo:** Class;

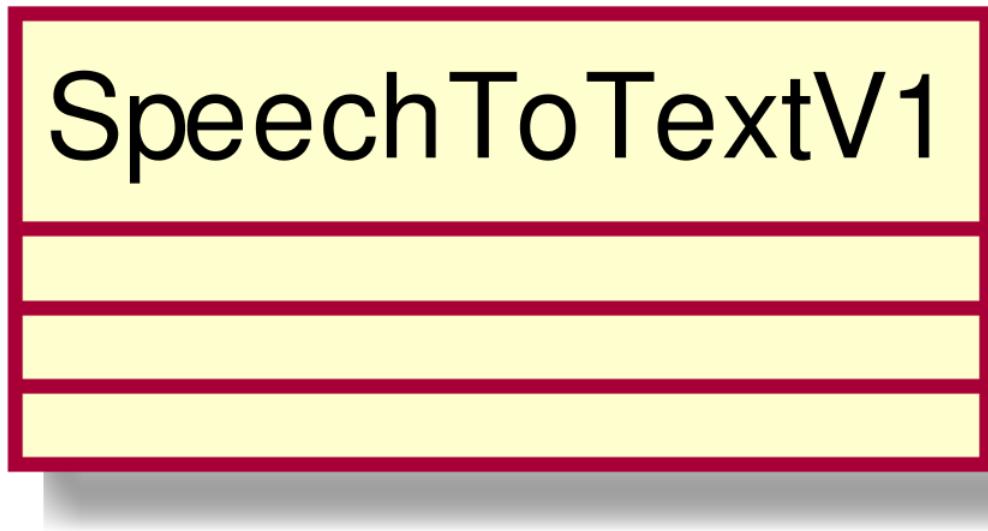


Figura 121: WatsonDeveloperCloud::SpeechToTextV1

- **Descrizione:** classe che rappresenta il modulo node.js che permette l'utilizzo del servizio di STT di IBM;
- **Utilizzo:** viene utilizzata per accedere al servizio STT fornito da IBM. Permette di interrogare tale servizio utilizzando dei callback oppure utilizzando gli streams di node.js. Per ulteriori informazioni, consultare la seguente pagina: <https://github.com/watson-developer-cloud/node-sdk#speech-to-text>;

Tracciamento

Tracciamento Classi-Requisiti

Classe	Requisiti
Back-end::APIGateway::VocalAPI	RFO1
Back-end::Auth::<<interface>> UsersDAO	RFO2.2 RFO2.2.1
Back-end::Auth::SRUser	RFO1 RFO1.1.2 RFO1.1.2.1 RFF1.1.2.2 RFO2
Back-end::Auth::User	RFO1 RFO1.1.2 RFO1.1.2.1 RFF1.1.2.2 RFF1.2
Back-end::Auth::UsersService	RFD9 RFD9.1 RFD9.1.1 RFD9.1.1.1 RFF9.1.1.2 RFD9.1.3
Back-end::Auth::VocalLoginModule	RFO1 RFO1.1.2 RFO1.1.2.1
Back-end::Notifications::Action	RFO13 RFO13.1 RFD13.2
Back-end::Notifications::Attachment	RFO13 RFO13.1 RFD13.2
Back-end::Notifications::- NotificationChannel	RFO13 RFO13.1 RFD13.2
Back-end::Notifications::- NotificationMessage	RFO13 RFO13.1 RFD13.2
Back-end::Notifications::Purpose	RFO13 RFO13.1 RFD13.2
Back-end::Notifications::Topic	RFO13 RFO13.1 RFD13.2
Back-end::Rules::Task	RFO2.1 RFO2.1.1 RFO2.1.1.1 RFO2.1.1.2 RFO2.1.1.3 RFD2.1.1.4 RFD2.1.1.4.1

Classe	Requisiti
	RFO2.1.1.5 RFO2.1.2 RFD2.1.3 RFD2.1.3.1 RFD2.1.3.2 RFD2.1.3.2.1 RFD2.1.3.2.2 RFD2.1.3.3 RFD2.1.3.4 RFD2.1.3.5 RFD2.1.3.5.1 RFD2.1.3.5.2 RFD2.1.3.6
<code>Client::ApplicationManager::-</code> <code>ApplicationPackage</code>	RVO9
<code>Client::Logic::LogicObserver</code>	RFO1 RFO1.1 RFO1.1.1 RFO1.1.1.1 RFO1.1.1.2 RFO1.1.2 RFO1.1.2.1 RFF1.1.2.2 RFO1.1.3 RFO2 RFO2.1 RFO2.1.1 RFO2.1.1.1 RFO2.1.1.1.1 RFO2.1.1.2 RFO2.1.1.3 RFO2.1.1.3.1 RFO2.1.1.3.2 RFO2.1.1.3.3 RFD2.1.1.4 RFD2.1.1.4.1 RFO2.1.1.5 RFO2.1.2 RFO2.1.2.1 RFD2.1.3 RFD2.1.3.1 RFD2.1.3.2 RFD2.1.3.2.1 RFD2.1.3.2.2 RFD2.1.3.3 RFD2.1.3.4 RFD2.1.3.5 RFD2.1.3.5.1 RFD2.1.3.5.2 RFD2.1.3.5.3 RFD2.1.3.5.4 RFD2.1.3.6
<code>Client::Recorder::Recorder</code>	RFO3 RFO3.1

Classe	Requisiti
	RFD3.2
	RFD3.2.1
	RFD3.2.2
	RFD3.2.3
	RFD3.2.3.1

Tabella 1: Tracciamento Classi-Requisiti

/PragmaDB/Classi/LaTeX/gettracciamentorequisiticlassi.php: Query fallita: Expression 1 of ORDER BY clause is not in SELECT list, references column 'pragmadb.h.Position' which is not in SELECT list; this is incompatible with DISTINCT;br /;i

Tracciamento Componenti-Requisiti

Componente	Requisiti
Back-end	RFO1 RFO1.1.2 RFO1.1.2.1 RFF1.1.2.2 RFF1.2 RFO2 RFO2.1 RFO2.1.1 RFO2.1.1.1 RFO2.1.1.2 RFO2.1.1.3 RFD2.1.1.4 RFD2.1.1.4.1 RFO2.1.1.5 RFO2.1.2 RFD2.1.3 RFD2.1.3.1 RFD2.1.3.2 RFD2.1.3.2.1 RFD2.1.3.2.2 RFD2.1.3.3 RFD2.1.3.4 RFD2.1.3.5 RFD2.1.3.5.1 RFD2.1.3.5.2 RFD2.1.3.6 RFO2.2 RFO2.2.1 RFD9 RFD9.1 RFD9.1.1 RFD9.1.1.1 RFF9.1.1.2 RFD9.1.3 RFO13 RFO13.1 RFD13.2
Back-end::APIGateway	RFO1
Back-end::Auth	RFO1 RFO1.1.2 RFO1.1.2.1 RFF1.1.2.2 RFF1.2 RFO2 RFO2.2 RFD9

Componente	Requisiti
	RFD9.1 RFD9.1.1 RFD9.1.1.1 RFF9.1.1.2 RFD9.1.3
Back-end::Notifications	RFO13 RFO13.1 RFD13.2
Back-end::Rules	RFO2.1 RFO2.1.1 RFO2.1.1.1 RFO2.1.1.2 RFO2.1.1.3 RFD2.1.1.4 RFD2.1.1.4.1 RFO2.1.1.5 RFO2.1.2 RFD2.1.3 RFD2.1.3.1 RFD2.1.3.2 RFD2.1.3.2.1 RFD2.1.3.2.2 RFD2.1.3.3 RFD2.1.3.4 RFD2.1.3.5 RFD2.1.3.5.1 RFD2.1.3.5.2 RFD2.1.3.6
Client	RFO1 RFO1.1 RFO1.1.1 RFO1.1.1.1 RFO1.1.1.2 RFO1.1.2 RFO1.1.2.1 RFF1.1.2.2 RFO1.1.3 RFO2 RFO2.1 RFO2.1.1 RFO2.1.1.1 RFO2.1.1.1.1 RFO2.1.1.2 RFO2.1.1.3 RFO2.1.1.3.1 RFO2.1.1.3.2 RFO2.1.1.3.3 RFD2.1.1.4 RFD2.1.1.4.1 RFO2.1.1.5 RFO2.1.2 RFO2.1.2.1 RFD2.1.3 RFD2.1.3.1

Componente	Requisiti
	RFD2.1.3.2 RFD2.1.3.2.1 RFD2.1.3.2.2 RFD2.1.3.3 RFD2.1.3.4 RFD2.1.3.5 RFD2.1.3.5.1 RFD2.1.3.5.2 RFD2.1.3.5.3 RFD2.1.3.5.4 RFD2.1.3.6 RFO3 RFO3.1 RFD3.2 RFD3.2.1 RFD3.2.2 RFD3.2.3 RFD3.2.3.1 RVO9
Client::ApplicationManager	RVO9
Client::Logic	RFO1 RFO1.1 RFO1.1.1 RFO1.1.1.1 RFO1.1.1.2 RFO1.1.2 RFO1.1.2.1 RFF1.1.2.2 RFO1.1.3 RFO2 RFO2.1 RFO2.1.1 RFO2.1.1.1 RFO2.1.1.1.1 RFO2.1.1.2 RFO2.1.1.3 RFO2.1.1.3.1 RFO2.1.1.3.2 RFO2.1.1.3.3 RFD2.1.1.4 RFD2.1.1.4.1 RFO2.1.1.5 RFO2.1.2 RFO2.1.2.1 RFD2.1.3 RFD2.1.3.1 RFD2.1.3.2 RFD2.1.3.2.1 RFD2.1.3.2.2 RFD2.1.3.3 RFD2.1.3.4 RFD2.1.3.5 RFD2.1.3.5.1 RFD2.1.3.5.2

Componente	Requisiti
	RFD2.1.3.5.3 RFD2.1.3.5.4 RFD2.1.3.6
Client::Recorder	RFO3 RFO3.1 RFD3.2 RFD3.2.1 RFD3.2.2 RFD3.2.3 RFD3.2.3.1

Tabella 2: Tracciamento Componenti-Requisiti

/PragmaDB/Package/LaTeX/gettracciamentorequisiticomponenti.php: Query fallita: Expression 1 of ORDER BY clause is not in SELECT list, references column 'pragmadb.h.Position' which is not in SELECT list; this is incompatible with DISTINCT
/i

Design Patterns

Architetturali

Architettura a microservizi

- **Scopo:** l'architettura a microservizi è un approccio allo sviluppo di una singola applicazione come insieme di piccoli servizi, ciascuno dei quali viene eseguito da un proprio processo e comunica con un meccanismo snello, spesso una HTTP API;
- **Vantaggi:**
 - ogni microservizio è relativamente piccolo, quindi più semplice da implementare e da capire per gli sviluppatori;
 - ogni microservizio è indipendente dagli altri; è quindi possibile distribuire nuove versioni più frequentemente e isolare i possibili errori.
- **Svantaggi:**
 - l'architettura risulta maggiormente complessa perché risulta essere un sistema distribuito;
 - la gestione di più microservizi potrebbe risultare in un carico di lavoro maggiore rispetto ad una sua versione monolitica.

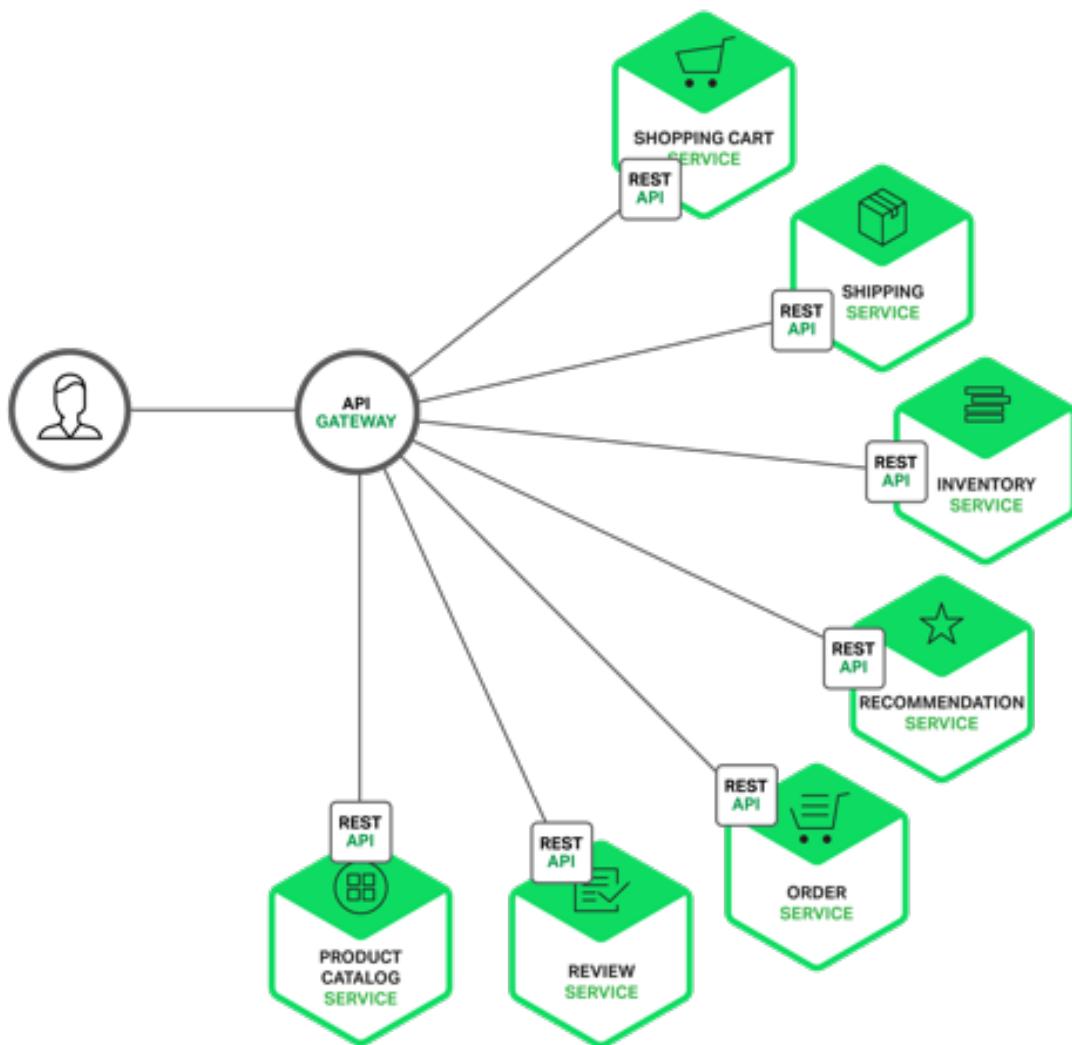


Figura 122: Architettura a microservizi

Architettura event-driven

- **Scopo:** anche se non è un vero e proprio pattern, l'architettura event-driven è un particolare tipo di architettura asincrona per sistemi distribuiti basata sugli eventi.
- **Vantaggi:**
 - per definizione, questo tipo di architettura è particolarmente adatto ad ambienti di tipo asincrono basati sugli eventi, come ad esempio l'interazione con degli utenti in tempo reale;
 - permette un alto disaccoppiamento tra le componenti.
- **Svantaggi:**
 - i sistemi che utilizzano tale architettura sono spesso distribuiti: ciò comporta un maggiore livello di complessità.

Client-side discovery

- **Scopo:** questo pattern, utilizzato in un'architettura a microservizi, permette ad un client la localizzazione dei microservizi interrogando un registro che ne conosce le posizioni.

Nel contesto della progettazione dell'architettura per il progetto AtAVi, questo pattern è stato riadattato (vedi figura 123) per uno scopo ben diverso, ovvero quello di ottenere delle applicazioni interrogando un registro che le contiene, tramite un'interfaccia per quest'ultimo. Come appena detto, lo scopo del pattern adottato è ben diverso da quello dell'originale, ma l'organizzazione delle classi del pattern Client-side discovery (vedi figura 124) modella bene la soluzione progettuale per l'ottenimento delle applicazioni;

- **Vantaggi:**

- **Originale:** permette di allocare dinamicamente diverse istanze di diversi servizi;
- **Riadattato:** permette di demandare la ricerca e recupero dell'applicazione a classi appropriate, nel rispetto del single responsibility principle.

- **Svantaggi:**

- **Originale:** crea dipendenze tra il registro e il client;
- **Riadattato:** aggiunge una leggera complessità al sistema.

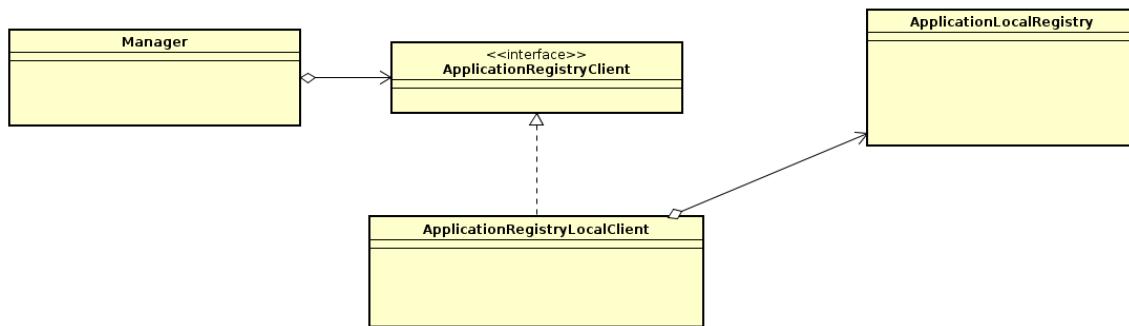


Figura 123: Pattern client-side discovery adattato alle esigenze del gruppo

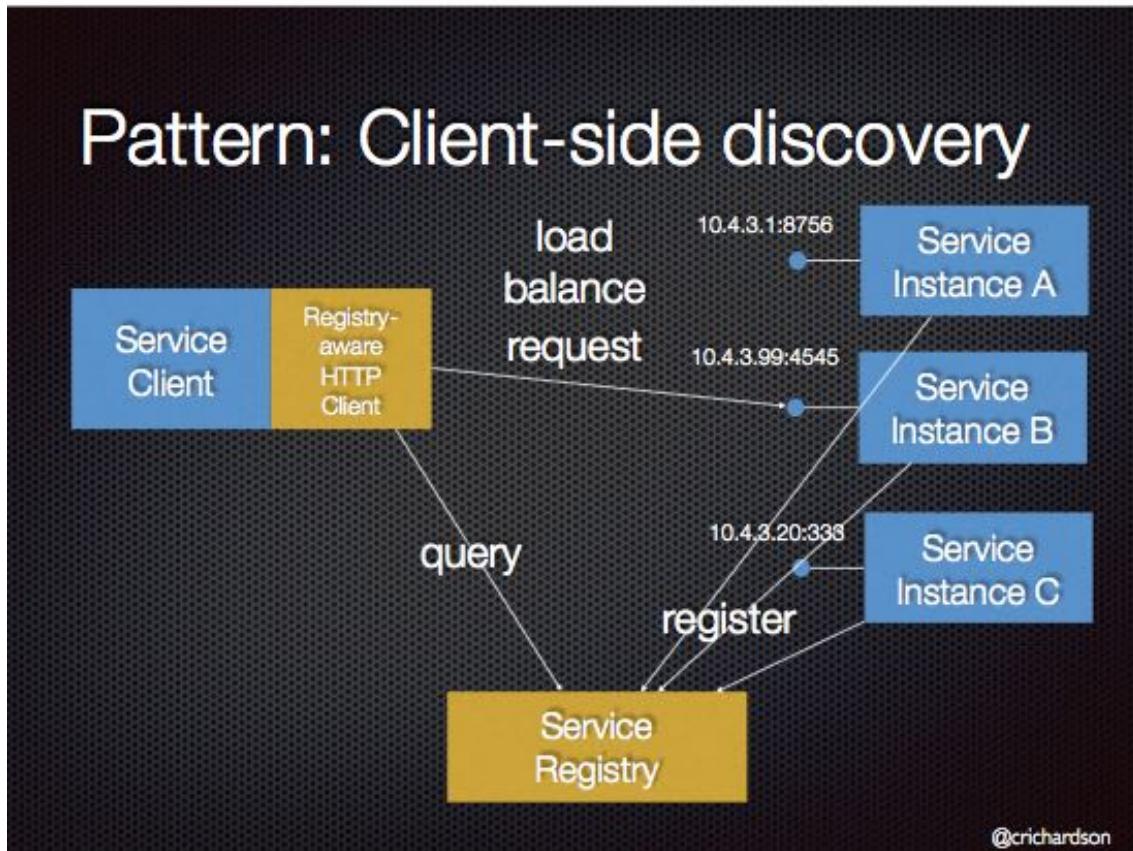


Figura 124: Pattern client-side discovery

Data Access Object

- **Scopo:** il pattern Data Access Object (DAO) consiste nell'utilizzo di un oggetto che fornisce un'interfaccia astratta per la gestione di una sorgente di dati, o più in generale per la gestione della persistenza.
- **Vantaggi:**
 - separazione tra logica di business e dati;
 - modifiche sul dati non comportano modifiche sul client che utilizza DAO.
- **Svantaggi:**
 - un'interfaccia di questo tipo potrebbe nascondere i costi di accesso ad un database;
 - potrebbero essere necessarie molte più operazioni rispetto all'esecuzione diretta di una query su un database.

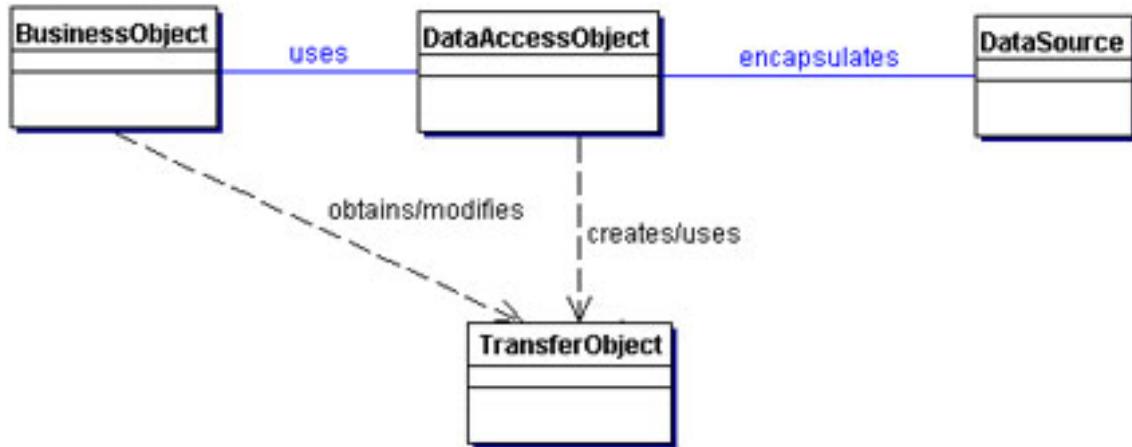


Figura 125: pattern Data Access Object (DAO)

Dependency Injection

- **Scopo:** consiste nella separazione del comportamento di una componente dalla risoluzione delle sue dipendenze.
- **Vantaggi:**
 - la separazione del comportamento dalle dipendenze rende una componente molto più flessibile;
 - rende le singole componenti maggiormente indipendenti permettendo una più facile progettazione dei test di unità.
- **Svantaggi:**
 - eventuali errori legati alla risoluzione delle dipendenze o alla loro implementazione vengono rilevati solamente a runtime;
 - rende più difficile il tracciamento del codice in quanto ne separa la costruzione dal comportamento.

Strutturali

Façade

- **Scopo:** indica un oggetto che permette, attraverso un'interfaccia più semplice, l'accesso a sottosistemi che espongono interfacce complesse e molto diverse tra loro, nonché a blocchi di codice complessi.

Nel contesto della progettazione architetturale del progetto AtAVi, questo pattern non è stato usato nella sua forma pura (visibile in figura 126). Facendo però un utilizzo di un API Gateway, il quale nasconde la complessità del back-end al client fornendone anche i meccanismi di utilizzo, si è fatto utilizzo dei concetti che questo pattern rispetta;

- **Vantaggi:**
 - permette di nascondere la complessità di un'operazione: rispetto alla chiamata diretta di un sottoinsieme di classi è possibile chiamare solamente la classe definita come façade semplificando l'operazione;
 - permette di diminuire le dipendenze tra sottosistemi;

- **Svantaggi:**

- i sottosistemi risultano essere collegati al facade: modifiche alla struttura dei sottosistemi comportano una serie di modifiche al facade stesso;

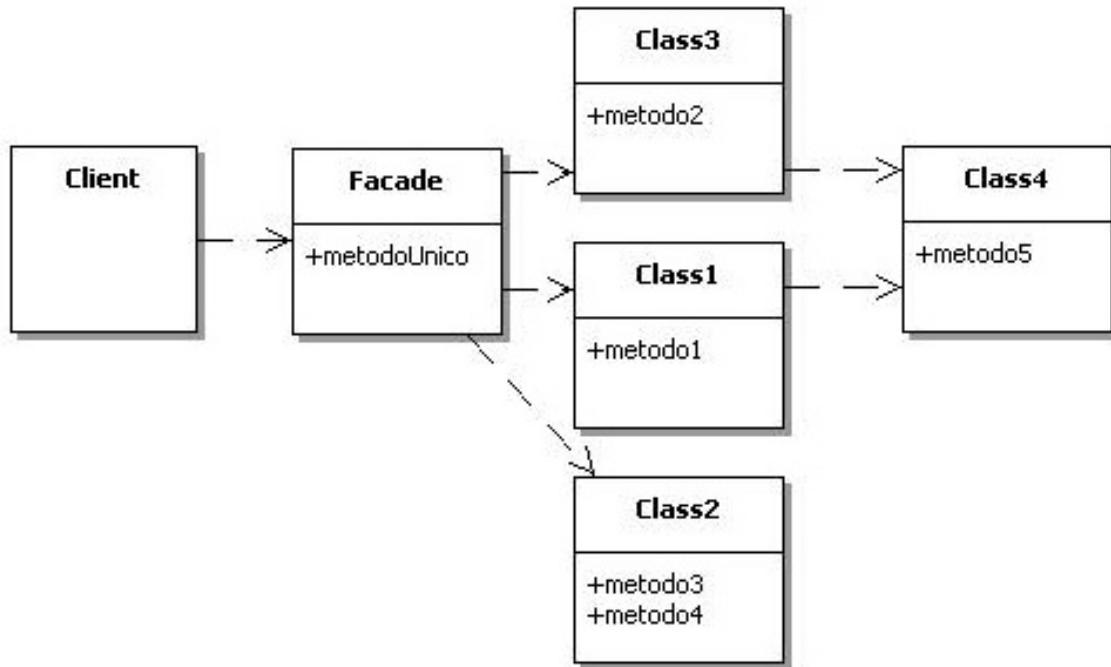


Figura 126: pattern Façade

Adapter

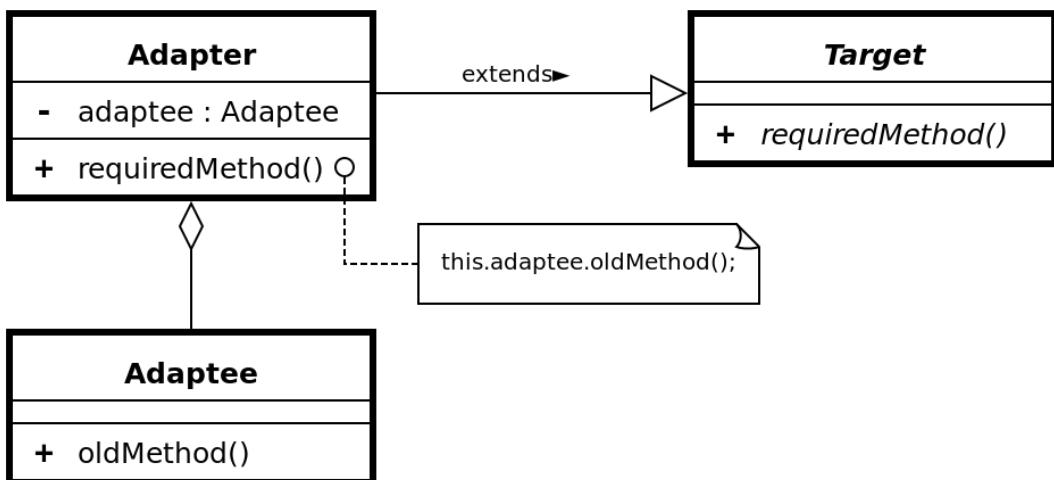
- **Scopo:** questo pattern permette la comunicazione tra due interfacce completamente differenti tramite l'utilizzo di un Adapter.

- **Vantaggi:**

- permette la conversione di una classe esistente in un'altra completamente differente senza modificarne il codice;
- maggiore flessibilità nella progettazione.

- **Svantaggi:**

- aumenta la dimensione del codice;
- a volte per interconnettere due interfacce sono necessari più Adapter.

**Figura 127:** Pattern Adapter

Creazionali

Module

- **Scopo:** questo pattern ha lo scopo di introdurre il concetto di modularità nei linguaggi di programmazione che non lo possiedono.
- **Vantaggi:**
 - come da definizione, questo pattern permette l'implementazione della modularità in ambienti privi di supporto ad essa.
- **Svantaggi:**
 - la sua implementazione richiede un maggiore carico di lavoro.
- **Utilizzo:**

Comportamentali

Observer

- **Scopo:** questo pattern, adottato nella variante offerta da ReactiveX, permette la definizione di una o più classi Observer le quali "osservano" una classe Observable e ne gestiscono gli eventi.
- **Vantaggi:**
 - permette la gestione di eventi tramite l'invio di dati ad altre classi in modo efficiente;
 - la definizione di classi Observer non causa modifiche alla classe Observable.
- **Svantaggi:**
 - una cattiva implementazione comporta un aumento della complessità del codice;
 - l'interfaccia Observer deve essere implementata, e ciò comporta ereditarietà.

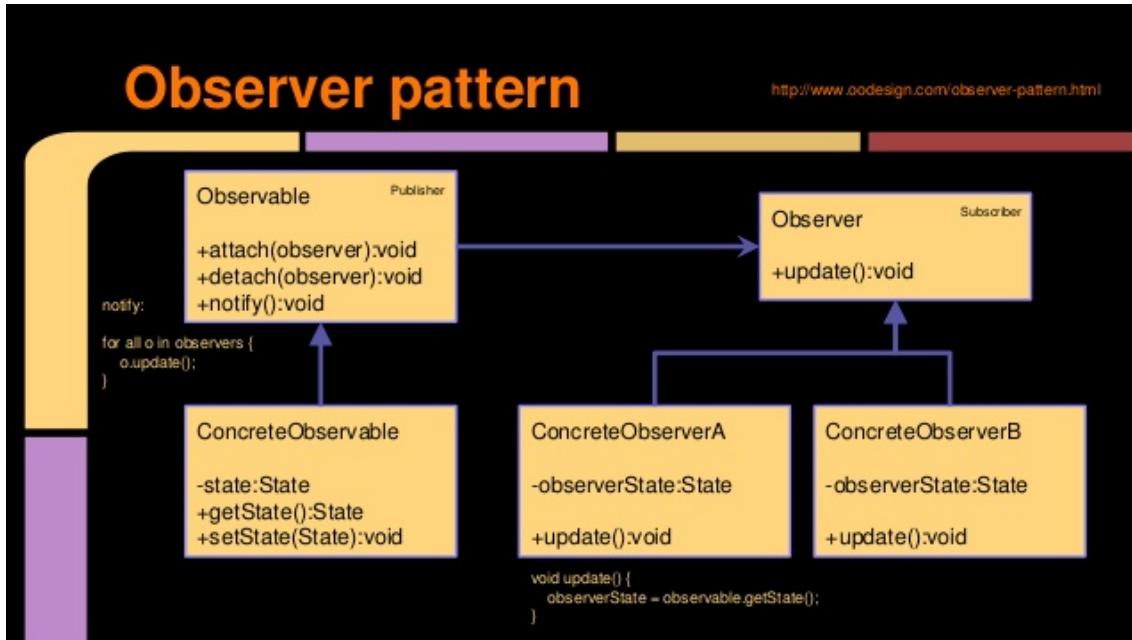


Figura 128: Pattern Observer

Tecnologie utilizzate

Promise e Observable

JavaScript è un linguaggio single-thread, quindi basato su un singolo thread in esecuzione. Questo significherebbe dover aspettare sempre il termine di un'operazione prima di passare alla successiva, quindi nel caso di operazioni di lunga durata, il flusso dell'elaborazione principale di un'applicazione JavaScript potrebbe "congelarsi".

Per aggirare questo problema, una delle soluzioni più diffuse è quella di passare una callback alla funzione in questione e, anziché aspettare il compimento dell'operazione, restituisce il controllo al chiamante. Quando l'operazione sarà terminata, la funzione di callback verrà invocata.

L'uso di callback, spesso annidate, rendono il codice di difficile comprensione e di difficile manutenzione. Due possibili soluzioni sono l'uso delle Promise di bluebird e gli Observable di RxJS.

Promise e bluebird

Una Promise è un oggetto che rappresenta il risultato pendente di un'operazione asincrona. Ciò permette ai metodi asincroni di restituire valori alla stessa maniera dei metodi sincroni: invece del valore finale, il metodo asincrono restituisce una Promise, ovvero una promessa di ottenere un valore in un momento futuro.

I principali metodi di una Promise sono:

- **then:** ritorna una Promise, in questo modo è possibile concatenare successive chiamate a questo metodo. È composto da due parametri opzionali che corrispondono a due funzioni: `onFulfill` viene richiamata se la Promise ha avuto successo, `onReject` se è stata rigettata.
- **catch:** ritorna una Promise e si occupa di gestire eventuali errori generati nella catena dei `then`.

Si è deciso di utilizzare bluebird in quanto offre i seguenti vantaggi:

- cross-platform: ideale per progetti che prevedono esperienze multipiattaforma;
- compatibile con le specifiche Promise/A+: bluebird può essere usato come rimpiazzo per Promise nativa offrendo un immediato miglioramento delle prestazioni;
- debug facile: gli stake trace sono in cui vengono riportati, quando possibile, gli errori non gestiti sono configurabili.

Observable e RxJS v5

ReactiveX è un insieme di API per estendere la programmazione asincrona facendo uso di Observable.

In base al linguaggio di programmazione che si vuole utilizzare, vengono fornite differenti librerie specifiche. La scelta quindi ricade su RxJS, ReactiveX library per JavaScript.

Un Observable è una rappresentazione di un qualsiasi insieme di valori durante un qualsiasi periodo di tempo. Viene utilizzata per le implementazioni del pattern Observer. RxJS v5 è ancora in beta ma sostituirà in breve la v4. Si è deciso di utilizzarla perché è una libreria solida nella gestione degli Observable e sarà ulteriormente aggiornata da Microsoft e altri sviluppatori di software open source.

AWS SDK per JavaScript in Node.js

Siccome il gruppo ha deciso di appoggiarsi all'infrastruttura Amazon Web Services (AWS) per lo sviluppo del sistema, è necessario utilizzare l'AWS SDK per JavaScript in Node.js per interfacciarsi ai singoli servizi offerti da AWS.

Node.js

La parte Back-End è stata sviluppata tramite la piattaforma event-driven Node.js, basata sul motore JavaScript V8. Esso permette di realizzare applicazioni Web utilizzando JavaScript, tipicamente client-side, per la scrittura server-side. La caratteristica principale di Node.js risiede nella possibilità che offre di accedere alle risorse del sistema operativo in modalità event-driven non sfruttando il classico modello basato su processi o thread concorrenti, utilizzato dai classici web server.

Request promise

E' un modello che implementa l'esistenza delle Promises come risposta a una serie di richieste che non possono essere soddisfatte immediatamente. Una promessa è un risultato che verrà reso disponibile non appena possibile se è possibile mantenere la promessa (fulfilled) oppure è un errore se non la si può mantenere (rejected).

Amazon Web Services

Di seguito sono elencati i servizi AWS che dovranno essere utilizzati.

AWS Lambda

AWS Lambda è un servizio di cloud-computing fornito da Amazon, il quale permette l'implementazione di una architettura serverless.

Esso permette l'esecuzione di codice senza preoccuparsi della gestione di server o di tutte le risorse

necessarie all'esecuzione del codice, in termini di tempo, spazio e computabilità.

AWS Lambda permette di eseguire codice su una piattaforma ad alta affidabilità, a patto che il codice sia scritto in un linguaggio supportato. Nel nostro caso, faremo utilizzo di Node.js 4.3.2, supportato da AWS Lambda.

Inoltre, questo servizio può essere utilizzato per eseguire del codice in risposta ad eventi.

La signature delle Lambda Function è la seguente:

```
function(event, context) {
    ...
}
```

dove:

- **event** è l'oggetto che contiene i dati della richiesta ricevuta dall'API Gateway. In particolare, contiene i seguenti campi:
 - **body**: campo di tipo **String** contenente il corpo della richiesta;
 - **pathParameters**: oggetto contenente i parametri passati all'API Gateway attraverso il path dell'URL;
 - **urlQueryParameters**: oggetto contenente i parametri passati all'API Gateway attraverso query nell'URL.
- **context** è l'oggetto contenente i dati relativi alla risposta e, in caso di Lambda Proxy Integration, contiene anche i seguenti metodi:
 - **succeed**: metodo da utilizzare per mandare una risposta all'API Gateway in caso di successo. La risposta dev'essere un oggetto contenente i seguenti campi:
 - * **headers**: array associativo nel quale la chiave indica il nome di un header HTTP da mandare nella risposta, il quale valore associato è una stringa contenente il valore di tale header;
 - * **statusCode**: attributo intero contenente il codice HTTP che dovrà avere la risposta;
 - * **body**: stringa contenente il corpo della risposta da mandare.

DynamoDB

Amazon DynamoDB è un servizio che fornisce un database NoSQL, il quale permette di immagazzinare documenti e grafici tra i suoi dati. È veloce e flessibile, ideato per tutte le applicazioni che richiedono una latenza costante non superiore a una decina di millisecondi, e grazie alle sue caratteristiche si presta perfettamente come supporto per applicazioni Web.

È un servizio di database NoSQL cloud interamente gestito, è quindi sufficiente creare una tabella di database, impostare il throughput desiderato e lasciare che il servizio si occupi del resto.

I vantaggi offerti sono:

- prestazioni rapidi e costanti, in media non oltre una decina di millisecondi;
- alta scalabilità, in quanto è possibile ridimensionare i requisiti di throughput,
- sicurezza, in quanto vengono forniti dei meccanismi per il controllo granulare degli accessi degli utenti;
- flessibilità, in quanto sono supportate sia le strutture di dati di tipo documento sia quelle di tipo chiave-valore;

Gli svantaggi sono:

- non è un servizio gratuito, ma i costi sono legati a tariffe basate sul consumo effettivo, senza pagamenti anticipati né impegni di lungo termine.

API Gateway

Amazon API Gateway è un servizio completamente gestito che semplifica agli sviluppatori la creazione, la pubblicazione, la manutenzione, il monitoraggio e la protezione delle API su qualsiasi scala.

In questo modo, si può fornire una porta d'entrata attraverso la quale le applicazioni possono accedere a dati, logica di business o funzionalità dei servizi di back-end.

I vantaggi offerti sono:

- alte prestazioni e scalabilità, offrendo una bassa latenza sulle richieste API e relative risposte;
- un pannello di controllo che consente di monitorare visivamente le chiamate al servizio;
- esecuzione simultanea di diverse versioni della stessa API, consentendo di iterare, testare e rilasciare nuove versioni in tempi brevi;
- strumenti per concedere le autorizzazioni di accesso alle API e controllare l'accesso alla gestione dei servizi;
- creazione di endpoint REST, permettendo di creare API avanzate basate sulle risorse, impiegando poi le dinamiche e flessibili funzionalità di trasformazione dei dati per generare le richieste nel linguaggio dei servizi a cui le richieste sono rivolte;
- esecuzione di API serverless, ista l'integrazione con AWS Lambda.

Gli svantaggi sono:

- non è un servizio gratuito, ma i costi di Amazon API Gateway sono bassi e calcolati esclusivamente sulle chiamate effettuate per le tue API e sui trasferimenti di dati in uscita.

SNS

Amazon Simple Notification Service (Amazon SNS) è un servizio di notifiche push rapido, flessibile e completamente gestito che ti consente di inviare messaggi individuali o collettivi a un numero elevato di destinatari.

Con Amazon SNS è possibile inviare notifiche a dispositivi Apple, Google, Fire OS e Windows. Inoltre, è possibile inviare notifiche push a delle lambda function altri endpoint HTTP.

I vantaggi offerti sono:

- alte prestazioni e affidabilità, infatti oltre ad avere una bassa latenza di risposta, tutti i messaggi pubblicati in Amazon SNS vengono memorizzati in modo ridondante su più server e data center;
- scalabilità, in quanto è consentita la pubblicazione di un altissimo numero di messaggi in qualsiasi momento;
- sicurezza, in quanto sono forniti meccanismi di controllo che permettono di proteggere argomenti e messaggi da accessi non autorizzati.

Gli svantaggi sono:

- non è un servizio gratuito, ma i costi sono legati a tariffe basate sul consumo effettivo, senza pagamenti anticipati né impegni di lungo termine.

CloudWatch

Amazon CloudWatch è un servizio di monitoraggio per le risorse cloud AWS e le applicazioni in esecuzione su AWS. Inoltre, tramite una semplice richiesta API, è possibile inviare parametri personalizzati generati dalle proprie applicazioni per monitorarli tramite questo servizio.

Serverless Framework

Serverless Framework è un web framework gratuito e open-source scritto tramite Node.js. Serverless è un framework per creare applicazioni esclusivamente su AWS Lambda. Un applicazione Serverless può essere composta da poche lambda functions per portare a termine semplici tasks o da molte lambda functions per creare ad esempio un intero back-end.

JWT

JWT è uno standard basato su JSON per creare token di accesso che possono sostenere richieste. I token sono firmati dalla chiave del server, quindi il client può verificare se un token è legittimo. I token sono pensati per essere compatti, senza contenere caratteri invalidi per gli URL e usabili principalmente nei contesti Single sign-on (SSO). Le richieste JWT possono essere usate solitamente per passare l'identità di un utente autenticato tra un identity provider e un service provider. I token inoltre possono essere autenticati e criptati.

Web Speech API

Web Speech API fornisce due aree distinte di funzionalità, il riconoscimento vocale e la sintesi vocale (conosciuta anche come text to speech o tts). Il riconoscimento vocale è acceduto attraverso l'interfaccia SpeechRecognition, la quale fornisce la possibilità di riconoscere il contesto vocale da un input vocale e risponde appropriatamente. La sintesi vocale è acceduta tramite l'interfaccia SpeechSynthesis, una componente text-to-speech che permette ai programmi di leggere testo.

Speaker recognition

Speaker Recognition API è un servizio Microsoft che, dopo aver costruito l'impronta vocale di un individuo, permette l'autenticazione per mezzo della voce. Al suo interno utilizza JSON per lo scambio dei dati e API Keys per l'autenticazione.

STT IBM Watson

Lo Speech to Text Watson è un servizio di IBM che, dato un file audio in input, restituisce il testo pronunciato in esso. Per trascrivere la voce umana accuratamente il servizio utilizza il machine learning per combinare informazioni riguardo la grammatica e la struttura del linguaggio con la composizione del segnale audio. Il servizio fornisce trascrizioni sempre migliori in base a quanto dialogo viene ascoltato.

api.ai

È una piattaforma di conversazione che permette interazioni sofisticate con il linguaggio naturale. Le applicazioni sviluppate su questa piattaforma sono costituite da Agent, i quali si occupano di trasformare il linguaggio naturale in dati processabili. Tali Agent sono a loro volta costituiti da Intent, che hanno il compito di associare la richiesta dell'utente ad una determinata azione del software, ed Entity, che sono strumenti per estrarre dal linguaggio naturale i parametri attesi. I vantaggi offerti sono:

- lo sviluppo di un assistente avanzato è piuttosto semplice.

Gli svantaggi sono:

- accetta solo input di tipo testuale.

HTML5

HTML5 è un linguaggio di markup per la strutturazione di pagine web, pubblicato dal W3C nell'ottobre del 2014 con lo scopo di migliorare l'HTML4, permettendo supporto a nuovi file multimediali, mantenendo la leggibilità da parte di uomini e computer. I vantaggi offerti sono:

- è uno standard W3C.

Gli svantaggi sono:

- non è supportato da tutti i vecchi browser.

CSS3

Pubblicata nel novembre 2014 dal W3C, CSS3 è una versione di CSS che va a correggere alcuni bug di interpretazione di Internet Explorer, migliorare la gestione degli sfondi e una soluzione per realizzare i bordi arrotondati.

CSS3 è suddiviso in moduli, al contrario delle versioni monolitiche precedenti, contenenti le vecchie specifiche CSS e nuove caratteristiche. Ciascun modulo tratta un tema diverso inerente i fogli di stile e, di conseguenza, ciascun tema è sviluppato in tempiistiche differenti.