



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	4
1.1	Scopo del documento	4
1.2	Scopo del prodotto	4
1.3	Glossario	4
1.4	Riferimenti	4
1.4.1	Riferimenti Normativi	4
1.4.2	Riferimenti Informativi	4
2	Standard di progetto	6
3	Architettura dell'applicazione	7
3.1	Microservizi	7
3.1.1	Virtual Assistant	7
3.1.1.1	Descrizione	7
3.1.1.2	Endpoints	7
3.1.2	Notifications	7
3.1.2.1	Descrizione	7
3.1.2.2	Endpoints	8
3.1.3	Users	8
3.1.3.1	Descrizione	8
3.1.3.2	Endpoints	8
3.1.4	Rules	8
3.1.4.1	Descrizione	8
3.1.4.2	Endpoints	8
4	Diagrammi riassuntivi dei package	9
5	Specifica dei componenti	10
6	Diagrammi di sequenza	11
7	Tracciamento	12
A	Design Patterns	13
A.1	Architetturali	13
A.1.1	Architettura a microservizi	13
A.1.2	Architettura event-driven	13
A.1.3	Client-side discovery	13
A.1.4	Data Access Object	14
A.1.5	Dependency Injection	14
A.2	Strutturali	14
A.2.1	Facade	14
A.2.2	Adapter	15
A.3	Creazionali	15
A.3.1	Singleton	15
A.3.2	Module	15
A.4	Comportamentali	16
A.4.1	Observer	16
B	Tecnologie utilizzate	16
B.1	Promise e Observable	16
B.1.1	Promise e bluebird	16
B.1.2	Observable e RxJS v5	17
B.2	AWS SDK per JavaScript in Node.js	17

B.2.1	Node.js	17
B.2.2	AWS Lambda	17
B.2.3	Serverless Framework	17
B.2.4	DynamoDB	18
B.2.5	API.AI	18
B.2.6	JWT	18
B.2.7	Web Speech API	18
B.2.8	Speaker recognition	18
B.2.9	STT IBM Watson	18
B.2.10	Request promise	18
B.2.11	AWS Lambda	19

Elenco delle figure

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 *Programmatore* 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*) 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>;
 - architetturali:
 - * <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.
- 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

L'architettura scelta è quella a microservizi.

Microservizi

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

Virtual Assistant

Descrizione

Il microservizio Virtual Assistant 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 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 apprendimento macchina.

Di seguito vengono esposti i vari passaggi:

- arriva una richiesta;
- interrogo il database, contenente gli agenti, utilizzando il nome dell'applicazione;
- 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

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

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

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 obbiettivi 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

Diagrammi riassuntivi dei package

Specifica dei componenti

Diagrammi di sequenza

Tracciamento

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 perchè risulta essere un sistema distribuito;
 - la gestione di più microservizi potrebbe risultare in un carico di lavoro maggiore rispetto ad una sua versione monolitica.
- **Utilizzo:**

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.
- **Svantaggi:**
 - i sistemi che utilizzano tale architettura sono spesso distribuiti: ciò comporta un maggiore livello di complessità.
- **Utilizzo:**

Client-side discovery

- **Scopo:** all'interno di un'architettura a microservizi, i singoli microservizi si trovano spesso in posizioni non fissate in quanto decise dinamicamente. Un metodo per la loro localizzazione consiste nel pattern Client-side discovery, che consiste nella richiesta della posizione di uno specifico microservizio da parte del client ad un registro, che conosce le posizioni di tutte le istanze dei microservizi.
- **Vantaggi:**
 - permette di allocare dinamicamente diverse istanze di diversi servizi.
- **Svantaggi:**
 - crea dipendenze tra il registro e il client.

- **Utilizzo:**

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 un database, o più in generale per la gestione della persistenza.
- **Vantaggi:**
 - separazione tra logica di business e dati;
 - modifiche sui dati non comportano modifiche sul client che utilizza DAO.
- **Svantaggi:**
 - un'interfaccia di questo tipo potrebbe nascondere i costi di accesso ad un database;
 - potrebbe essere necessarie molte più operazioni rispetto all'esecuzione diretta di una query su un database.
- **Utilizzo:**

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.
- **Utilizzo:**

Strutturali

Facade

- **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.
- **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 facade 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;
- **Utilizzo:**

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.
- **Utilizzo:**

Creazionali

Singleton

- **Scopo:** questo pattern ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale ad essa.
- **Vantaggi:**
 - questo pattern risulta molto utile ogni qual volta è necessaria una sola istanza di una classe.
- **Svantaggi:**
 - la classe Singleton risulta essere globale, e di conseguenza rende più difficile la definizione di test di unità;
 - aumenta il livello di accoppiamento del codice.
- **Utilizzo:**

Module

- **Scopo:** questo pattern ha lo scopo di introdurre il concetto di modularità nel 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 permette la definizione di una o più classi Observer le quali "osservano" una classe Soggetto 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 Soggetto.
- **Svantaggi:**
 - una cattiva implementazione comporta un aumento della complessità del codice;
 - l'interfaccia Observer deve essere implementata, e ciò comporta ereditarietà.
- **Utilizzo:**

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 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 le seguenti caratteristiche:

- `cross-platform`: ideale per progetti che prevedono esperienze multiplatforma;
- `compatibile con le specifiche Promise/A+`: bluebird può essere usato come rimpiazzo per Promise nativa offrendo un immediato miglioramento delle prestazioni;
- `debug facile`: gli `stack trace` sono in cui vengono riportati, quando possibile, gli errori non gestiti sono configurabili.

Observable e RxJS v5

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 perchè 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 SDK per JavaScript in Node.js per

interfacciarsi ai singoli servizi offerti da AWS. In particolare si farà utilizzo di DynamoDb.

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.

AWS Lambda

AWS Lambda è un servizio di elaborazione serverless che esegue codice in risposta a determinati eventi e gestisce automaticamente le risorse di elaborazione in uso. Può essere usato per estendere altri servizi AWS con logica personalizzata oppure creare servizi di back-end.

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.

DynamoDB

Amazon DynamoDB è un servizio database NoSQL, il quale permette di immagazzinare documenti e grafici tra i suoi dati. E' 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.

API.AI

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.

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 identifica utenti individuali, e li autentica 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 converte voce in testo scritto. 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.

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).

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;
- **context** – AWS Lambda uses this parameter to provide your handler the runtime information of the Lambda function that is executing. For more information, see [The Context Object](#) (Node.js).