AtAVi

Definizione di Prodotto v1.0.0

Sommario
Questo documento le scelte progettuali effettuate dal gruppo Co.Code per la realizzazione del *progetto*$_{\text{g}}$ 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

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 *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;
  - 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

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 sul dati non comportano modifiche sul client cche 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:**

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:**

Librerie esterne 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, anzichè 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 concatenere successive chiamate a questo meotodo. 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 suguenti caratteristiche:

- 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

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 aggiornta

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 utiizzo di DynamoDb.