UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

DIPARTIMENTO DI SCIENZE FISICHE,
INFORMATICHE E MATEMATICHE

Tesi di Laurea in Informatica

# Progettazione e implementazione di un applicativo di raccolta dati, ottimizzazione e monitoraggio per un cluster web

**Relatore**
Ing. Riccardo Martoglia

**Laureando**
Anna Fossali

Anno Accademico 2017-2018

# Ringraziamenti

Grazie all'ingegnere Riccardo Martoglia, il mio relatore, per la disponibilità mostrata in questi mesi e per il sostegno durante il tirocinio e la stesura della tesi.
Grazie ad ADmantX, che mi ha accolto e mi ha permesso di crescere, mettendomi alla prova ogni giorno.
Grazie alle mie amiche di sempre, le quali non hanno mai smesso di credere in me e con le quali, oggi come allora, una semplice serata si trasforma in qualche folle avventura.
Grazie a tutti gli amici conosciuti proprio qui, tra queste mura, che in questi ultimi anni hanno reso il raggiungimento di questo traguardo un periodo pieno di risate, viaggi, scherzi ed imprese, oltre che di studio e lezioni.
Grazie inoltre a chi, soprattutto in questo ultimo periodo, mi è rimasto accanto nelle mie giornate più pesanti, riuscendo ad alleggerirle con un semplice sguardo, e sorprendendomi ogni giorno di più.
Grazie, grazie, grazie infine alla mia straordinaria famiglia, sempre pronta a consigliarmi, ad insegnarmi, ad incoraggiarmi, a sostenermi e ad amarmi.

*Grazie*

# Indice

# Introduzione

L'esponenziale crescita del mercato della pubblicità online degli ultimi anni ha messo in luce la mancanza di soluzioni intelligenti per la pianificazione e il targeting avanzato del marchio. La pubblicità sul web, o web advertising, rappresenta oggi una delle forme di comunicazione a pagamento più utilizzate a tal punto che sono sempre di più le aziende, grandi o piccole, che destinano parte del budget pubblicitario sui canali online. L'azienda ADmantX, in quanto data provider, offre a editori online, network e centri media una tecnologia per l'erogazione automatica della pubblicità.

In questo elaborato di tesi si descrive il progetto, svolto durante il tirocinio presso ADmantX, nato dalla necessità di interagire in modo facile e intuitivo con i dati interni dell'azienda, per aggiornare e migliorare la tecnologia di targeting semantico, utilizzata per capire il vero contesto dei contenuti online e creare profili completi dei consumatori. L'obiettivo è quindi quello di fornire una documentazione del lavoro svolto nello sviluppo del progetto. In particolare è stato creato un web service attraverso il quale è possibile monitorare i dati interni. Il web service dispone di un'area privata accessibile solo ai dipendenti e velocizza la visualizzazione, l'aggiunta, la rimozione e la ricerca dei profili dei clienti, delle richieste ricevute e degli *admant*, gli elementi a base della tecnologia semantica. Inoltre l'implementazione di questo servizio ha reso necessario lo sviluppo di un applicativo di raccolta dei dati relativi alle richieste effettuate dai clienti e di creazione di un nuovo database ottimizzato, con una nuova struttura dati. Tale script aggiorna i conteggi ogni dieci minuti, tenendo traccia dei conteggi totali e parziali.

La tesi è strutturata su quattro capitoli. Il primo capitolo delinea i requisiti definiti dall'azienda, una breve descrizione delle tecnologie e degli strumenti di sviluppo utilizzati e la presentazione della tecnologia semantica sviluppata dall'azienda stessa, necessaria per la comprensione di alcuni concetti e di alcune scelte implementative. Il secondo capitolo descrive la progettazione del web service, specificando le strutture dati dei database utilizzati e alcuni casi d'uso, necessari per avere un visione completa delle parti da sviluppare. Il terzo capitolo delinea nel dettaglio il funzionamento e il procedimento di implementazione dell'applicazione, sulla base dei requisiti descritti nella prima parte. L'ultimo capitolo infine descrive lo sviluppo dell'applicativo di raccolta dati a sostegno del web service.

# Parte I

# Presentazione progetto

# Capitolo 1

# Requisiti e tecnologie utilizzate

Lo sviluppo del progetto mira al monitoraggio dei dati interni dell'azienda attraverso un web service che, oltre alla visione dei profili dei clienti e degli *admant*, permetta il controllo delle chiamate ricevute da ogni cliente sulle varie macchine e per ogni lingua disponibile.

La progettazione del software si è composta di tre fasi principali: la definizione dei requisiti, la scelta delle tecnologie da utilizzare e l'ottimizzazione dei database. In questo capitolo si presentano brevemente la tecnologia semantica utilizzata dall'azienda, fondamentale per la comprensione del progetto, e la ricerca e la specifica dei requisiti, al fine di avere una descrizione chiara e non ambigua delle funzionalità necessarie del futuro software, senza entrare nei dettagli implementativi. La definizione di tali requisiti ha messo in luce la necessità di sviluppare un ulteriore applicativo di raccolta e ottimizzazione dei dati delle chiamate ricevute sui vari server, per permetterne un accesso più rapido. Inoltre vengono descritte le tecnologie e gli strumenti di sviluppo e i database, scelti perché più efficienti e con maggior possibilità di incremento di performance.

## 1.1 Presentazione dell'azienda

ADmantX è un data provider intelligente che facilita la pianificazione e il targeting avanzato del marchio nel settore della pubblicità digitale. Per offrire il proprio servizio e dare una risposta alle richieste, l'azienda utilizza la tecnologia semantica di elaborazione del linguaggio naturale che permette di capire il vero contesto dei contenuti online e creare profili completi dei consumatori, permettendo ai marchi di raggiungere l'audience giusta e di abbinare la loro pubblicità al giusto contenuto, proteggendone la reputazione e migliorandone le prestazioni. Dopo un'attenta analisi semantica, questa tecnologia contestualizza il contenuto in ben venti lingue differenti. ADmantX offre quindi agli utenti la possibilità di creare, salvare, testare e pubblicare campagne pubblicitarie personalizzate attraverso l'utilizzo di *admant*. Un *admant* è una combinazione di elementi, combinati da uno o più operatori booleani, che definisce il tipo di contenuto di un testo, fornendone un'analisi che permette di capire il miglior posizionamento degli annunci. Gli elementi combinati nella creazione dell'*admant*, come ad esempio categorie, sentimenti e lemmi, possono corrispondere o meno ai risultati dell'analisi semantica

delle pagine web su cui la campagna è in esecuzione. La selezione della maggior parte degli elementi di un *admant* avviene attraverso la tassonomia. La tassonomia di ADmantX è un elenco gerarchico di categorie e sentimenti (*emotions* e *sentiment*) rilevati nel testo analizzato dal servizio di analisi semantica. L'elenco è rappresentabile come albero, i cui nodi rappresentano le categorie e i sentimenti. Un elemento di un *admant* quindi potrebbe essere un padre, un figlio o entrambi, a seconda della categoria e del livello in cui si trova nella tassonomia. Ogni volta che una categoria padre viene aggiunta a un *admant*, le categorie secondarie vengono incluse automaticamente e i risultati forniti dell'*admant* sono ampi poiché includono tutti i suoi discendenti. Per un risultato più specifico bisogna spostarsi verso il basso della tassonomia.

La piattaforma di ADmantX è un web service RESTful real-time che esegue l'analisi semantica di testi attraverso richieste HTTP/s e ritorna in risposta un descrittore semantico. La piattaforma supporta sia richieste GET che POST, che ricevono la sorgente che deve essere analizzata, sotto forma di URL di una pagina contenente testo, di semplice testo o di un file HTML, e dei parametri aggiuntivi necessari per configurare le modalità di analisi e il formato della risposta. L'analisi dei contenuti è supportata in più di venti lingue, tra cui italiano, inglese e quasi tutte le altre lingue europee, ma anche arabo, cinese, Hindi, giapponese, coreano, turco e vietnamita. L'azienda suddivide le chiamate ricevute dai vari clienti nei diversi server che ha a disposizione. Ogni server analizza il contenuto della richiesta e ritorna la risposta al cliente, tenendo traccia delle chiamate ricevute suddivise per cliente e per lingua. L'azienda inoltre dispone di un database dei clienti della piattaforma, nel quale salva i dati personali del cliente, come username id e email, e i parametri che imposta nelle chiamate all'analizzatore, e di un database di tutti gli *admant* creati, di cui salva il nome, l'id e l'espressione. Le informazioni di questo paragrafo sono tratte da [1].

## 1.2 Definizione dei requisiti

L'applicazione web si rivolge a due principali gruppi di utenti: il team IT e il team dei linguisti. I primi devono gestire e controllare le chiamate degli utenti sui server dell'azienda, i secondi invece devono gestire e monitorare gli *admant* e i profili dei clienti. Il requisiti fondamentali richiesti sono: l'autenticazione, la gestione di *admant* e profili dei clienti, il conteggio delle chiamate ricevute dai clienti e la loro gestione.

### 1.2.1 Autenticazione

Le funzionalità da implementare per l'autenticazione sono: registrazione, abilitazione, login e logout, "Forgot password" e cambiamento dei dati inseriti e/o della propria password.

#### Registrazione

La registrazione permette all'utente di effettuare una richiesta di accesso alla piattaforma. L'utente infatti dopo essersi registrato deve attendere l'abilitazione da parte dell'amministra-

tore.

- **Introduzione**: ogni utente ha la possibilità di registrarsi al sistema, inserendo nome, cognome e email.
- **Input**: i dati inseriti dall'utente.
- **Processing**: controllo della sintassi e dell'unicità della password, aggiunta dei dati del nuovo utente nel database, invio di un'email all'amministratore per una richiesta di abilitazione del nuovo utente, invio di un'email all'utente con la conferma della registrazione e la richiesta di attesa di abilitazione.
- **Output**: notifica di operazione effettuata o con la descrizione dell'errore.

#### Abilitazione

Dopo la richiesta di registrazione, l'amministratore ha la possibilità di abilitare o meno l'utente.

- **Introduzione**: l'amministratore riceve una mail di abilitazione, nella quale oltre ai dati del nuovo utente, è presente un link per l'attivazione di quest'ultimo. Cliccando sul link l'utente sarà abilitato e potrà effettuare il login.
- **Input**: conferma di abilitazione.
- **Processing**: invio di un'email all'utente con la conferma di abilitazione e la nuova password in chiaro, salvataggio della nuova password criptata e del booleano di attivazione a True nell'oggetto corrispondente nel database.
- **Output**: notifica di operazione effettuata o con la descrizione dell'errore.

#### Forgot password

Nel caso di smarrimento della password l'utente già abilitato può richiederne una nuova attraverso la funzionalità "Forgot password".

- **Introduzione**: con l'inserimento della propria email l'utente richiede una nuova password per l'accesso alla piattaforma.
- **Input**: l'email inserita.
- **Processing**: controllo della presenza dell'indirizzo email sul database e dell'abilitazione dell'utente relativo e, in caso di controlli positivi, invio di una email all'utente con la nuova password in chiaro.
- **Output**: notifica di operazione effettuata o con la descrizione dell'errore.

### Login

La procedura di login dell'utente abilitato e in possesso della propria password, redirige alla Home dell'applicazione web o dà una notifica di errore.

- **Introduzione**: l'utente può effettuare il login inserendo la propria email e la password.
- **Input**: i dati inseriti.
- **Processing**: controllo dei dati inseriti e dell'abilitazione dell'utente sul database. In caso di controlli non andati a buon fine, il login sarà bloccato, altrimenti salvataggio dei dati di sessione in cache, per effettuare l'accesso automatico all'utente già loggato.
- **Output**: redirezione alla Home page del sistema o notifica di errore con la descrizione.

### Logout

Il logout può essere effettuato sia manualmente, attraverso il bottone apposito, che automaticamente, allo scadere di un intervallo di tempo di inattività prestabilito. Ciò deve permettere all'utente già loggato di accedere direttamente alla Home page, senza il reinserimento dei propri dati.

- **Introduzione**: dalla Home dell'applicazione l'utente può accedere alla procedura di logout.
- **Input**: la conferma di logout.
- **Processing**: cancellazione delle variabili di sessione dell'utente.
- **Output**: redirezione alla pagina di login.

### Modifica dei dati personali

Oltre all'inserimento in fase di registrazione, l'utente ha la possibilità di modificare i propri dati dalla Home page.

- **Introduzione**: dalla Home dell'applicazione l'utente può modificare i propri dati personali, esclusa l'email. Per il cambiamento della password ne è richiesto l'inserimento duplice.
- **Input**: i nuovi dati inseriti.
- **Processing**: aggiornamento del database con i nuovi dati e la nuova password criptata e invio di un'email all'utente con le modifiche e la password in chiaro (se modificata).
- **Output**: notifica di operazione effettuata o con la descrizione dell'errore.

### 1.2.2 Gestione Admant

Il progetto deve mettere a disposizione dell'utente autenticato la tabella con i dati di tutti gli *admant* (Key, Name e Expression). Le funzionalità richieste, quindi, sono quelle di aggiunta, rimozione, modifica e download di uno o più *admant* da parte dell'utente, sia in locale che attraverso la connessione con il database, e la loro ricerca. I dati nella tabella inoltre dovranno essere ordinabili, selezionabili e impaginati.

#### Aggiunta e modifica di un admant

L'utente deve poter effettuare la richiesta di aggiunta o modifica di un *admant* attraverso lo stesso form. In entrambi i casi i dati inseriti devono superare specifici controlli.

- **Introduzione**: l'utente autenticato può aggiungere un nuovo *admant* con l'inserimento di chiave univoca, nome e espressione o modificarne uno esistente inserendone solo i campi da modificare. L'espressione dell'*admant* inserita deve rispettare particolari regole semantiche, descritte nell'appendice A.
- **Input**: i nuovi dati inseriti.
- **Processing**: controllo dell'unicità della chiave e controlli semantici dell'espressione dell'*admant*. In caso di controlli con esito positivo, aggiornamento dei dati sul database e invio di un'email all'amministratore con i dati relativi.
- **Output**: notifica di operazione effettuata o con la descrizione dell'errore.

#### Rimozione di uno o più admant

La rimozione deve essere disponibile per un singolo *admant* o per un insieme di *admant*.

- **Introduzione**: l'utente autenticato può rimuovere un *admant* o selezionarne alcuni ed eliminarli in gruppo. Nella richiesta di conferma dell'operazione, può vedere l'elenco dei candidati alla rimozione.
- **Input**: l'/gli *admant* selezionati per la rimozione.
- **Processing**: rimozione degli *admant* dal database e invio di un'email all'amministratore con i dati relativi.
- **Output**: notifica di operazione effettuata o con la descrizione dell'errore.

#### Ricerca di uno o più admant

La ricerca deve essere selettiva su uno o più campi, case-sensitive che case-insensitive.

- **Introduzione**: inserendo una stringa l'utente può ricercare gli *admant* combacianti, selezionando il campo su cui effettuare la ricerca e se eseguirla case-sensitive o case-insensitive.

- **Input**: la stringa di ricerca.
- **Processing**: ricerca del dato corrispondente.
- **Output**: tabella filtrata con gli *admant* trovati.

#### Download

Infine è richiesta anche la funzionalità di scaricamento dei dati, nella loro totalità o in base alla selezione.

- **Introduzione**: l'utente può scaricare i dati selezionati e/o tutti i dati del database su un file in formato .csv .
- **Input**: elenco degli *admant* da scaricare.
- **Processing**: generazione del file e download.
- **Output**: file in formato .csv .

### 1.2.3 Profili dei clienti

La visione e la gestione dei profili dei clienti si rifà a quella degli *admant*. È quindi necessaria la visione della tabella, l'aggiunta, la rimozione, la modifica e la ricerca di uno o più clienti. In questo caso i dati di un profilo hanno tipi diversi: stringhe per la chiave, il nome e l'email del cliente, interi per il conteggio massimo delle chiamate, enumerati per la selezione dei tipi e dei filtri attivi e booleani per l'abilitazione e l'analisi.

#### Aggiunta e modifica di un profilo

Così come per gli admant, l'utente autenticato può inserire o modificare un profilo, ma nel caso dell'inserimento la chiave viene generata in automatico dal sistema.

- **Introduzione**: l'utente può aggiungere un nuovo profilo con l'inserimento di tutti i campi, con eccezione della chiave, o modificarne uno esistente inserendo solo i campi da modificare.
- **Input**: i nuovi dati inseriti.
- **Processing**: generazione della chiave in modo casuale e univoco (in caso di aggiunta), aggiornamento dei dati sul database e invio di un'email all'amministratore con i dati relativi.
- **Output**: notifica di operazione effettuata o con la descrizione dell'errore.

#### Rimozione, ricerca e download di uno o più profili

La rimozione, la ricerca e il download hanno gli stessi requisiti descritti per gli *admant*.

### 1.2.4 Check Admant

Come anticipato, gli *admant* sono particolari espressioni che devono seguire specifiche regole semantiche, descritte nell'appendice A. Oltre al controllo effettuato al momento dell'aggiunta al database, è necessario fornire la possibilità di effettuare il controllo dell'espressione di un *admant* generico, in via di sviluppo o di test. Per questo specifico caso è quindi necessaria la funzionalità di inserimento dell'espressione e di ricezione di una risposta con lo stato del controllo.

- **Introduzione**: l'utente inserisce l'espressione dell'*admant* da controllare.
- **Input**: la stringa inserita.
- **Processing**: controlli semantici dell'espressione.
- **Output**: Correct, Warning o Error Status in base al risultato dei controlli. In caso di Warning o Error specifica della posizione del warning o dell'errore.

Per maggiori informazioni sulle regole vedere l'appendice A.

### 1.2.5 Gestione chiamate

In questa sezione del progetto, oltre al web service per la visione delle chiamate ricevute, è richiesta anche l'implementazione di un programma per la raccolta dei dati e la loro ottimizzazione. Il web service deve mostrare una tabella con i dati delle chiamate per ogni cliente sulle lingue scelte e un'altra tabella con le chiamate ricevute su ogni server dai vari clienti. La visione delle tabelle deve essere mese per mese e si deve visualizzare il totale delle chiamate mensile e di ogni giorno. Per ogni giorno inoltre è richiesta la visione di un grafico che rappresenti il numero di chiamate ricevute in funzione di un certo intervallo di tempo, prestabilito e modificabile e seconda delle esigenze. Infine è necessaria la funzionalità di selezione e scaricamento dei dati in vari formati.

#### Download

La funzionalità di scaricamento dei dati si rifà a quelle descritte in precedenza, ma con l'aggiunta della possibilità di scelta di scaricamento dei conteggi totali o di quelli di dettaglio di un singolo cliente o server.

- **Introduzione**: l'utente può scaricare i dati selezionati e/o tutti i dati del database su un file in formato .csv, specificando se richiede il download dei dati relativi alle chiamate totali del cliente o anche della suddivisione di tali chiamate nelle lingue per le quali ha richiesto l'analisi.
- **Input**: elenco dei clienti (con o senza lingue) da scaricare.
- **Processing**: generazione del file e download.
- **Output**: file in formato .csv.

#### Applicativo raccolta dati

I dati delle chiamate sono salvate su due database locali Redis di ogni server dell'azienda. Pertanto è richiesto lo scaricamento di tali dati da ogni server e la creazione di un nuovo database con delle strutture dati ottimali per le richieste specificate. Il programma deve aggiornare i dati in modo periodico allo scadere di un intervallo di tempo di dieci minuti e tenere traccia dei conteggi parziali ad ogni scaricamento. La scelta della struttura dati deve considerare la necessità di accesso di tali dati raggruppati in base a parametri diversi a seconda dei casi.

### 1.2.6 Altri requisiti

L'interfaccia del progetto deve essere user-friendly, con componenti intuitive e interattive, e cross-platform, adatto all'accesso da dispositivi diversi. Il sistema deve poter gestire più utenti alla volta in modo indipendente l'uno dall'altro e le richieste arrivate da utenti diversi vengono gestite in modo sequenziale una alla volta.

## 1.3 Tecnologie e strumenti di sviluppo

Le principali tecnologie utilizzate per lo sviluppo dell'applicativo sono il web server Tornado, il web server Nginx, la libreria jQuery di Javascript e il framework Bootstrap di CSS, unite ai database MongoDB, Redis e SimpleDB.

### 1.3.1 Tornado Web Server

Tornado è un server web che supporta un elevato numero di connessioni simultanee, con integrato al suo interno, un web-framework python, costruito principalmente per gestire processi asincroni. Tornado è stato ideato principalmente per gestire processi asincroni. Sviluppato originariamente da FriendFeed, usando network I/O non bloccanti, può supportare un elevato numero di connessioni simultanee, rendendolo ideale per le comunicazioni long-polling, WebSocket e tutte le applicazioni che richiedono una connessione duratura con ogni utente. Si distingue dagli altri framework di Python perché non si basa su WSGI e normalmente esegue solo un thread per processo. Anche se presenta un supporto per l'utilizzo di WSGI attraverso il modulo tornado.wsgi, è suggerito l'uso delle interfacce interne di Tornado, proprio perché in generale il suo codice non è thread-safe. Gli unici metodi implementati per fare chiamate da altri thread sono IOLoop.add_callback e IOLoop.run_in_executor, ricordando che in quest'ultima si dovrebbero evitare riferimenti a un qualunque oggetto di Tornado. Tornado è inoltre integrato con la libreria *asyncio*, con la quale condivide lo stesso loop di eventi. Le componenti principali di Tornado sono:

- un web framework, in particolare con le classi RequestHandler per creare web service e gestire le richieste e gli eventi;

- un'implementazione del protocollo HTTP sia client-side che server-side, con le classi HTTPServer e AsyncHTTPClient;
- una libreria di networking asincrono, con le classi IOLoop e IOStream, le basi per le componenti HTTP e per implementare altri protocolli;
- la libreria tornado.gen che permette di scrivere il codice asincrono in modo più chiaro

Un'applicazione web sviluppata in Tornado generalmente consiste in una o più sottoclassi di *RequestHandler*, un oggetto *Application*, che gestisce le richieste entranti e la configurazione iniziale, e di una funzione *main()* per far partire il server. L'oggetto *Application* configura quindi la "routing table" che mappa le richieste con le sottoclassi dei gestori. La "routing table" è una lista di oggetti *URLSpec* o tuple, ciascuno contenente almeno una stringa (dell'URL) e la classe handler corrispondente. Il metodo costruttore di *Application* prende in input diversi argomenti (keyword arguments) che possono essere usati per personalizzare il comportamento dell'applicazione e abilitare funzionalità opzionali. La maggior parte del lavoro di un'applicazione Tornado è quindi svolto dalle sottoclassi di *RequestHandler*, che implementano uno o più metodi il cui nome si rifà ai metodi delle richieste HTTP. Nel progetto in questione tutte le classi implementano un metodo *post()*, che riceve i parametri di input con il metodo *get_argument()* e producono una risposta con *write()*, che accetta stringhe, bytes e dizionari (codificati come JSON).

### 1.3.2 Nginx web server

Nginx è un web server open source gratuito che, a cominciare dal suo successo iniziale come server, è ora principalmente utilizzato anche come proxy inverso, cache HTTP, bilanciatore di carico e media streaming. Originariamente creato da Igor Sysoev, con il suo primo rilascio al pubblico nell'ottobre del 2004 e come risposta al problema C10k, riguardante le anomalie delle prestazioni nella gestione di 10.000 connessioni simultanee, è progettato per garantire un basso consumo di memoria e un'elevata concorrenza, spesso superando le prestazioni di altri server web popolari, specialmente in situazioni con contenuto statico e/o numerose richieste simultanee. Anziché creare nuovi processi per ogni richiesta web, Nginx utilizza un approccio asincrono, basato sugli eventi, in cui le richieste vengono gestite in un singolo thread. Un processo master può controllare più processi worker. Il master mantiene i processi worker, mentre i worker eseguono l'elaborazione effettiva. Poiché Nginx è asincrono, ogni richiesta può essere eseguita dal worker contemporaneamente senza bloccare altre richieste. Grazie alla sua versatilità - funziona su Unix, Linus, macOS, Solaris e Windows, e alla sua semplicità di utilizzo, attualmente è uno dei più utilizzati al mondo e in Italia. Mantiene tempi di risposta notevoli anche in presenza di richieste molto elevate, e riesce a tollerare senza rallentarsi fino a quattro volte in più delle connessioni di Apache, il suo web server concorrente più popolare, e supporta già a livello nativo il load balancing, permettendo al sistema di gestire in maniera più rapida ed efficiente il carico di richieste, e di configurare immediatamente eventuali nuovi

stack che decideremo di inserire per far fronte a un'esigenza. Alcune caratteristiche comuni di Nginx sono:

- Reverse proxy con caching,
- IPv6,
- Bilanciamento del carico,
- Supporto FastCGI con caching,
- WebSockets,
- Gestione di file statici, file indice e autoindicizzazione,
- TLS/SSL con SNI.

Nel progetto è usato principalmente come server reverse-proxy per inviare le richieste delle chiamate Ajax con i rispettivi parametri a Tornado.

### 1.3.3 Bootstrap

Bootstrap è un insieme di elementi grafici, stilistici e di impaginazione per lo sviluppo con HTML, CSS e Javascript. È uno dei framework CSS più utilizzati e imitati, oggi indipendente ma nato ad opera degli sviluppatori Mark Otto e Jacob Thornton internamente a Twitter, con obiettivo quello di spingere il team di ingegneri a utilizzare lo stesso framework per minimizzare le incoerenze Crea pagine responsive, utilizzando il "Grid system", ovvero lavorando tramite una griglia formata da righe (row) e colonne (col), rendendole adattabili a tutte le ultime versioni dei principali browser e a tutti i dispositivi desktop, tablet o smartphone. Sono disponibili diversi temi e template pronti all'uso, che permettono di avere a disposizione una gran quantità di funzionalità e di stili modificabili e adattabili a seconda delle esigenze del programmatore. Può essere quindi descritto come una libreria multidispositivo e multipiattaforma. Gli elementi personalizzabili contenuti in Bootstrap sono una combinazione di HTML, CSS e JavaScript e grazie alla comodità dell'open-source, Bootstrap viene migliorato continuamente. Le componenti principali di Bootstrap sono:

- Grid System: un insieme di fogli che considerano il contenitore generale disposto su una griglia, fissa o fluida, con una larghezza base di 960 px, nella quale distribuire i contenuti in varie righe e colonne, che definiranno il layout del template. Di base, la griglia è costituita da 12 colonne e ogni elemento del layout può espandersi su una o più colonne, semplicemente applicando una classe di stile che in qualche modo definisce la sua "estensione".
- CSS Base: area contenente una serie di stili predefiniti per tutta la parte tipografica, come i titoli (H1, H2, ...), le tabelle, i paragrafi, i form e i pulsanti.

- Componenti: area con elementi quali menù dropdown, tooltip, alert, slider, banner di navigazione e popover che ci aiutano nell'implementazione degli elementi dinamici della pagina, senza la necessità di scrivere codice Javascript, ma solo aggiungendo dei data-attributes, che Bootstrap interpreta e gestisce autonomamente. Fra i componenti è compreso anche un set di glifi (icone rappresentate in formato di carattere) di uso comune, messe a disposizione da Glyphicons.
- Javascript: area che contiene diversi plug-in jQuery per realizzare effetti come transizioni, finestre modali, popup, carousel, accordion e tab.

### 1.3.4 jQuery

jQuery è una libreria JavaScript veloce e leggera, che semplifica l'attraversamento dei documenti HTML, la gestione degli eventi, le animazioni e le interazioni Ajax per migliorare e velocizzare lo sviluppo web. Il motto di jQuery "write less, do more" sottolinea la semplicità e la potenza di questo framework. La prima versione del framework jQuery risale al 2006 ed è frutto del lavoro dello sviluppatore John Resign, il quale aveva il preciso intento di rendere il codice più sintetico e di limitare al minimo l'estensione degli oggetti globali per ottenere la massima compatibilità con altre librerie. La sua sintassi semplificata e flessibile permette di realizzare interfacce user friendly in modo efficace, fattore che ha contribuito all'enorme successo di questo framework, che in breve tempo è diventato il più utilizzato e diffuso tra i webmaster di tutto il mondo. La libreria quindi è in grado di offrire un'ampia gamma di funzionalità, che vanno dalla manipolazione degli stili CSS e degli elementi HTML, agli effetti grafici, a comodi metodi per chiamate Ajax cross-browser, senza toccare gli oggetti nativi JavaScript e potendo utilizzare contemporaneamente anche altre librerie Javascript, come per esempio Bootstrap. jQuery gestisce tutte le eventuali differenze tra i browser, e presenta quindi un ambiente standard multipiattaforma su cui sviluppare. Le funzionalità fornite della libreria jQuery sono tutte precedute dal simbolo "$" e molti metodi di jQuery sono concatenabili, per rendere la scrittura e la lettura del codice molto più lineare. La comunità di jQuery inoltre mette a disposizione numerosi plugin e rende il framework in continuo sviluppo e rilasciando aggiornamenti con una certa frequenza, al fine di rendere il codice sempre più veloce e di mantenere il passo con i nuovi motori javascript che vengono integrati all'interno delle nuove versioni dei browser.

#### Datatables

Datatables, plug-in di jQuery, è uno strumento altamente flessibile, che aggiunge funzionalità avanzate a qualsiasi tabella HTML. L'obiettivo dichiarato di DataTables è "migliorare l'accessibilità dei dati nelle tabelle HTML", sia per gli end-users, coloro che utilizzano l'interfaccia creata, che per gli sviluppatori. Per i primi implementa funzionalità come la ricerca, l'ordinamento, il download e la paginazione dei dati, mentre per i secondi facilita e velocizza il caricamento dei dati e la loro gestione. L'ampia gamma di opzioni di configurazione, impostate al momento dell'inizializzazione, può essere utilizzata per personalizzare il modo in

cui presenterà la sua interfaccia e le funzionalità disponibili all'utente finale. I dati per una DataTable invece possono provenire da tre luoghi diversi:

- HTML: la tabella è creata e popolata direttamente nel file HTML,
- array Javascript: utilizzato quando i dati sono presenti in un array Javascript,
- chiamata Ajax: implementata nel caso di dati dinamici, provenienti per esempio da database esterni e caricati server-side in una struttura dati JSON.

In questo progetto tutte i dati vengono caricati e modificati attraverso delle chiamate Ajax al web server Tornado.

#### HighCharts

HighCharts è una libreria Open Source Javascript che può girare sia su jQuery che su Standalone. Nata nel 2009, è pensata per aggiungere alle applicazioni web le funzionalità di creazione di grafici interattivi. Attraverso la tecnologia SVG e VML, offre una vasta gamma di grafici, a linee, spline, ad area, a barre, a torta e molti altri, disponibili su tutti i browser moderni, su Internet Explorer, dalla versione 6, e sulle piattaforme basate su touchscreen, di cui supporta il multitouch. La configurazione iniziale di un grafico HighCharts avviene tramite un JSON, ma è modificabile dinamicamente. È possibile impostare più assi, zoommare su una parte del grafico per vedere dati più precisi, stampare o scaricare il grafico in diversi formati e caricare i dati in diversi formati o dal server, fornendo ulteriori funzioni di callback per il controllo dei dati.

### 1.3.5 Altri strumenti

Oltre agli strumenti appena descritti, l'applicazione presenta anche l'utilizzo del framework CSS *Font awesome* e delle librerie di Python *email* e *datetime*.

*Font awesome* permette di aggiungere più di 1500 icone vettoriali facilmente personalizzabili attraverso le regole del vostro CSS ed è ampiamente utilizzato per la sua velocità di caricamento e la scalabilità delle immagini, ridimensionabili senza alcuna perdita di qualità.

Il package *email* gestisce i messaggi di email, inclusi MIME e gli altri documenti basati sulla RFC 2822, dividendo l'analisi e la generazione del messaggio di email dal modello oggetto di rappresentazione interno dell'email. In sequenza un messaggio di email viene letto come testo da un file o un'altra fonte, il testo viene analizzato per produrre la struttura dell'oggetto del messaggio di email, questa struttura viene manipolata ed infine renderizzata nuovamente in puro testo.

Il modulo *datetime* fornisce delle classi per manipolare date e tempi, supportando l'aritmetica di tempi e date e implementando un'efficiente estrazione delle componenti, per la manipolazione e la formattazione dell'output.

### 1.3.6 Chiamate Ajax

L'acronimo Ajax, Asynchronous JavaScript And XML (JavaScript asincrono ed XML), è stato enunciato per la prima volta da Jesse Garrett, nel 18 Febbraio 2005, come titolo di un post all'interno del suo blog. Una chiamata asincrona ad una risorsa non interferisce con l'esecuzione della risorsa chiamante e il caricamento avviene in background. I risultati ricevuti dalla risorsa esterna sono quindi utilizzabili solo alla ricezione della risposta, ma l'utilizzatore non subisce tempi di attesa e non deve necessariamente attendere che sia stata ultimata per effettuare altre operazioni. Il concetto descrive quindi un utilizzo asincrono di Javascript, che attraverso l'interfacciamento con XML, può permettere ad un client di richiamare informazioni lato server in modo veloce e trasparente. L'oggetto principale di una chiamata Ajax è XMLHttpRequest, che a seconda del browser usato prende nomi differenti o viene richiamato in maniera differente. XMLHttpRequest permette di effettuare la richiesta di una risorsa, statica o dinamica, (con HTTP) ad un server web in modo indipendente dal browser, permettendo di inviare informazioni sotto forma di variabili di tipo GET o di tipo POST, e in modo asincrono. E sempre attraverso Javascript avviene la gestione di eventuali errori e la manipolazione del risultato ricevuto. Generalmente il flusso è racchiuso in due passaggi alla volta:

- richiesta dell'utente (link, form o refresh),
- risposta da parte del server, attraverso un oggetto di tipo XML o testuale,

ma mentre l'utente è all'interno della stessa pagina, le richieste sul server possono essere numerose e completamente indipendenti. La risorsa contattata tramite Ajax deve essere una risorsa locale. In base alle restrizioni del Same Origin Policy, infatti, non è possibile effettuare chiamate Ajax a risorse presenti all'interno di altri domini per questioni di sicurezza.

Nel progetto ogni volta che viene effettuata una chiamata Ajax, Nginx invia la richiesta alla classe `Application` di `tornado.web`. `Application` poi richiama la classe del web service corrispondente in base alla lista di tuple (url, classe) alla quale è applicata. Quest'ultima classe gestisce la richieste e ritorna una risposta alla chiamata Ajax, che gestirà l'output.

I risultati di questo capitolo sono tratti e riadattati da [13], [9], [3], [7], [4], [6], [5], [11], [10].

## 1.4 Database utilizzati

I database in uso dall'azienda sono di tipo NoSQL, ovvero non relazionali. NoSQL è un termine universalmente accettato per raggruppare tutti quei database che non utilizzano il modello relazionale, che usano tabelle e campi ma senza schemi fissi, che non permettono vincoli di integrità referenziale o che non garantiscono transazioni ACID. Tra questi i più usati sono quelli orientati ai documenti, oggetti complessi senza uno schema rigido, quelli a grafo, con relazioni libere tra i nodi del grafo, quelli chiave-valore, che utilizzano il modello

dell'array associativo o che fanno corrispondere alle chiavi dei valori formati da famiglie di colonne anch'esse indicizzate. Ciò comporta dei requisiti maggiori per il livello di applicazione, ma consente di distribuire le serie di dati e i processi di lavoro tra più server, rendendo questi database moderni quasi scalabili in maniera illimitata. Il salvataggio dei dati degli *admant* e dei profili dei clienti avviene attraverso SimpleDB, mentre le chiamate ricevute dai clienti sono salvate sui database Redis, installati su ogni server. Oltre ai database già in uso dall'azienda, il progetto ha richiesto l'utilizzo di MongoDB, per il salvataggio degli utenti della piattaforma creata e la raccolta dei dati relativi alle chiamate su ogni server.

### 1.4.1 MongoDB

MongoDb è un database document-oriented storage, scalabile e flessibile, che salva i dati sotto forma di documenti. Nato nel 2007 in California, il suo nome deriva dall'aggettivo huMONGOus (termine inglese che significa gigante, enorme, gigantesco), in riferimento alla capacità di immagazzinare grosse quantità di dati, ed ora è uno dei più noti database non relazionali, indipendente e open-source. I documenti sono memorizzati in formato JSON e vengono raggruppati all'interno di *collections*, che a loro volta sono suddivise in database; un server MongoDB può gestire più database e un database può essere distribuito su più server. Ogni *document* contiene dei campi che possono variare tra documento e documento e la loro struttura dati interna può cambiare nel tempo, rendondoli molto più flessibili di un qualsiasi database SQL. Attraverso l'indicizzazione, applicabile a un qualsiasi campo, e le aggregazioni, l'accesso e l'analisi dei dati sono veloci e semplici anche in ambienti in cui vengono salvati milioni di informazioni. MongoDB come impostazione standard sceglie il compromesso CP ovvero fortemente consistente e tollerante alle partizioni, ma modificandone le impostazioni, permette di scegliere il compromesso AP (Accessibilità e tollerante alle Partizioni). Mongo inoltre ha la proprietà di auto-sharding, ovvero la possibilità di distribuire i suoi dati su più server in maniera automatizzata, lasciando che sia il motore di MongoDB a gestire l'interrogazione e la scrittura dei dati su più macchine. MongoDB infine è integrato con i più comuni linguaggi di programmazione per il web, da PHP a Python a Ruby.

### 1.4.2 SimpleDB

SimpleDB è un datastore non relazionale ospitato da Amazon, nato nel 2007 come integrazione agli strumenti AWS. Amazon SimpleDB è ottimizzato per fornire flessibilità e disponibilità elevate, facilitando il lavoro di amministrazione dei database e permettendo di pensare solo all'archiviazione e all'esecuzione di query su dati strutturati in tempo reale. Il database è scalabile e accessibile attraverso i web services, crea e gestisce in automatico una serie di repliche dei dati distribuendole su più aree geografiche, e indicizzandoli automaticamente ad ogni modifica, contribuendo in modo significativo a disponibilità e durabilità dei dati. SimpleDB usa un linguaggio di query string-based, con una sintassi abbastanza lineare. Inoltre fornisce un endpoint https che garantisce comunicazioni protette e crittografata tra l' applicazione o il client e il dominio.

### 1.4.3 Redis

Redis, acronimo di REmote DIctionary Server, è lo store di strutture dati chiave-valore più usato, rapido e open-source. Rilasciato per la prima volta nel 2009, con licenza BSD, si basa su una struttura a dizionario, nella quale ogni valore immagazzinato è abbinato a una chiave univoca che ne permette il recupero. Scritto in C, il codice è ottimizzato e supporta diverse sintassi di sviluppo e tutte le operazioni sono atomiche, ovvero i dati sono costantemente aggiornati anche in presenza di più accessi contemporanei. I tipi di valori supportati sono diversi, come stringhe, hash, liste, set, set ordinati, bitmap, hyperlog, indici geospaziali e stream e il servizio offre la possibilità di implementare configurazioni multi-nodo, cluster e replication. Redis è estremamente veloce sia in lettura che scrittura, conservando i dati in memoria RAM e salvandoli in maniera persistente solo in seguito e la persistenza su memoria dei dati assicura il mantenimento del database anche in caso di riavvio del server o di guasti a livello di hardware. Le maggiori applicazioni di Redis sono applicazioni Web, videogiochi, tecnologie pubblicitarie, IoT e app per dispositivi mobili che necessitano di elevate prestazioni.

I risultati di questo capitolo sono tratti e riadattati da [8], [2] e [12].

# Parte II

# Sviluppo web service

# Capitolo 2

# Progettazione web service

In questo capitolo si presenta la progettazione vera e propria del software, che pone le fondamenta per la fase di implementazione, descritta nel capitolo successivo.
In particolare si descrivono le strutture dati di tutte le informazioni citate e i principali casi d'uso da parte di un futuro utente. La scelta delle nuove strutture dati è stata guidata dalla necessità di minimizzazione dei tempi di accesso.

Per rendere l'interfaccia il più simile possibile agli altri web service già in funzione nell'azienda si è scelto di utilizzare alcuni dei framework e delle tecnologie già in uso. Il programma quindi si basa sul web framework Tornado server-side ed è scritto nel linguaggio Python. Utilizza Nginx come server reverse-proxy e front-side è implementato con il supporto di Bootstrap e jQuery. Per i database su cui salvare i dati degli utenti e dei conteggi, la scelta di MongoDB è stata effettuata in base alle performance sia nel caso di accesso a un singolo dato, sia all'accesso a un insieme di dati.

Tutte le funzionalità di interazione con i database e i controlli semantici sono implementati server-side, mentre l'interazione dell'utente con l'interfaccia è implementata front-side.

## 2.1 Strutture dati

I nuovi dati da inserire nei database riguardano l'elenco degli utenti della piattaforma e le chiamate ricevute dai clienti sui server aziendali. A tale scopo sono stati creati due database su Mongo DB: *Monitor* e *Counters*. I dati degli *admant* e dei profili sono da ricercare sugli account aziendali di Amazon SimpleDB.

### 2.1.1 User

L'insieme degli utenti registrati al servizio è salvato nella collezione *User* del database *Monitor*. La struttura dati di ogni utente contiene tutte le informazioni di registrazione, abilitazione e autenticazione.

```
{
    "_id" :  ObjectId( '5c87d6ce2452be07e0386a1a') ,
    "name" :  "user_name" ,
    "surname" :  "user_surname" ,
    "enable" :  True/False,
    "email" :  "user_email" ,
    "password" :  "cripted_password"
}
```

- *_id*: assegnata in automatico da Mongo all'inserimento del documento
- *name*, *surname* e *email*: dati inseriti in fase di registrazione.
- *password*: password criptata con l'algoritmo SHA256 - stringa vuota se l'utente non è ancora stato abilitato.
- *enable*: booleano di controllo dell'abilitazione all'uso del servizio da parte dell'amministratore.

### 2.1.2 Counters

Le chiamate ricevute su un server, da un utente, durante uno specifico giorno sono salvate nella collezione *clientserver* del database *Counters*. La struttura dati tiene traccia dei conteggi totali delle chiamate andate a buon fine, dei KO, dei conteggi parziali in ogni intervallo di tempo e dei conteggi suddivisi per lingue su cui sono state effettuate tali chiamate.

```
{
    "day" : yyyymmdd,
    "server" :  "server_name" ,
    "client" :  "client_key" ,
    "name" : "client_name",
    "sync": "(a)sync",
    "count": count,
    "count_ko": count_ko,
    "partial": {
        "time1": count1,
        "time2": count2,
        ...
    },
    "partial_ko": {
        "time1": count1_ko,
        "time2": count2_ko,
        ...
```

```
    },
    "languages_ok": {
        "language_name": {
            "count": language_count_ok,
            "partial": {
                "time1": count1_ok,
                "time2": count2_ok,
                ...
            }
        },
        ...
    },
    "languages_ko": {
        "language_name": {
            "count": language_count_ko,
            "partial": {
                "time1": count1_ko,
                "time2": count2_ko,
                ...
            }
        },
        ...
    },
    "languages_unsupported": {
        "language_name": language_count,
        ...
    }
}
```

- *day*: il giorno di ricezione delle chiamate,
- *client*: la chiave del cliente che ha effettuato le chiamate,
- *server*: il nome della macchina su cui sono state effettuate le chiamate,
- *name*: il nome del cliente,
- *sync*: sync o async in base alla tipologia sincrona o asincrona delle chiamate,
- *environment*: l'ambiente in cui è inserita la macchina,
- *count*: il conteggio totale delle chiamate ricevuto, sia OK che KO,
- *count_ko*: il conteggio delle chiamate in KO,

- *partial*: mappa dei conteggi delle chiamate ricevute in ogni intervallo di tempo,
- *partial_ko*: mappa dei conteggi delle chiamate KO ricevute in ogni intervallo di tempo,
- *languages_ok*: mappa delle lingue con i conteggi totali e parziali delle chiamate OK ricevute,
- *languages_ko*: mappa delle lingue con i conteggi totali e parziali delle chiamate KO ricevute,
- *languages_unsupported*: mappa delle lingue non supportate con i conteggi totali delle chiamate.

Per la ricerca dei dati necessari nella visione delle tabelle si è notato essere necessaria anche una mappa delle lingue su cui ogni cliente effettua le chiamate. Questi documenti sono salvati nella collection *clientlanguages* dello stesso database.

```
{
    "client" : "client_key",
    "sync" : "(a)sync",
    "ok_ko" : ["lang1", "lang2", ...] ,
    "unsupported" : ["lang1", "lang2", ...],
}
```

- *client*: chiave del cliente,
- *sync*: tipologia sincrona o asincrona delle chiamate,
- *ok_ko*: lista delle lingue su cui il cliente ha effettuato chiamate OK o KO,
- *unsupported*: lista delle lingue non supportate su cui il cliente ha effettuato chiamate.

### 2.1.3 Admant

L'azienda dispone di diversi gruppi di *admant*, salvati ognuno in un dominio diverso su SimpleDB. Le strutture dati utilizzate sono composte dalle informazioni finora citate:

```
{
    "user$name" :  {
        "admant" :  "espressione_admant"
        }
}
```

- *user*: chiave dell'*admant*,
- *name*: nome dell'*admant*,
- *admant*: espressione dell'*admant*.

### 2.1.4 Profili dei clienti

I profili dei clienti sono tutti salvati nello stesso dominio di SimpleDB. Le strutture dati utilizzate sono composte dalle informazioni precedentemente citate:

```
{
 "profilename" :  {
    "profile" :  {
        "userID" :  "id",
        "userName" :  "name",
        "email" :  "email",
        "types" :  URL/TXT/HTML,
        "filters": text/categories/feelings/entities/geo/
                    image/relation/domains/lemmas,
        "maxCount" :  maxcount,
        "analytics" :  True/False,
        "enabled" :  True/False
    }
  }
}
```

- *userID*: chiave del cliente,
- *userName*: nome del cliente,
- *email*: email del cliente,
- *types*: tipi di documenti sui quali il cliente effettua le chiamate (URL, TXT, HTML o una loro combinazione),
- *filters*: filtri da attivare nell'analisi sui documenti (text, categories, feelings, entities, geo, image, relation, domains, lemmas o una loro combinazione),
- *maxCount*: il numero massimo di chiamate effettuate dal cliente,
- *analytics*: booleano di controllo per una specifica dell'analisi testuale,
- *enabled*: booleano di controllo per l'abilitazione del cliente all'accesso agli *admant*,

## 2.2 Casi d'uso

In seguito si descrivono nello specifico due scenari d'uso dell'applicazione.

### 2.2.1 Aggiunta e modifica di un admant

**PRECONDIZIONI** l'utente ha effettuato con successo il login.

**SEQUENZA EVENTI**
- l'utente accede all'area di visione della tabella dell'*admant*,
- l'utente clicca su 'Add admant' o sul bottone 'Update' di fianco all'*admant* che desidera modificare,
- si apre un form con i campi vuoti o con i dati dell'*admant* corrispondente,
- l'utente inserisce chiave, nome e espressione del nuovo *admant* o modifica i dati con i nuovi valori,
- se lo richiede, clicca sul bottone 'Check admant' per controllare la semantica dell'espressione inserita prima dell'invio della richiesta di aggiunta o modifica,
- se cliccato 'Check admant', il server risponde con una notifica di 'OK', 'WARNING' o 'ERROR' e la posizione del warning o dell'errore, in base al risultato dei controlli,
- in caso di WARNING o ERROR l'utente modifica l'espressione inserita,
- l'utente clicca 'Add' o 'Update',
- il server controlla l'unicità della chiave e in caso contrario ritorna un messaggio di errore, altrimenti
  - in caso di ERROR il server ritorna il messaggio di notifica e blocca la chiamata di aggiunta o modifica,
  - in caso di WARNING il server ritorna il messaggio di notifica con un bottone 'Continue anyway?' per permettere all'utente di caricare l'*admant* anche in presenza di warning,
  - in caso di OK o di 'Continue anyway?' il server aggiunge i dati dell'*admant* nel database, aggiorna la tabella aggiungendo o modificando la riga corrispondente e ritorna una notifica di avvenuta operazione.

**POSTCONDIZIONI** al termine della procedura l'utente torna alla schermata di visione della tabella.

### 2.2.2 Visione e download dei dati sulle chiamate

**PRECONDIZIONI** l'utente ha effettuato con successo il login.

**SEQUENZA EVENTI**
- l'utente accede all'area di visione della tabella delle chiamate suddivise per cliente,
- l'utente seleziona dal calendario il mese di cui desidera vedere le chiamate,
- i dati della tabella sono ordinati in ordine alfanumerico sul campo del nome dell'utente. Ogni riga presenta il nome e la chiave dell'utente, il numero di chiamate totali ricevute durante il mese e le chiamate ricevute per ogni giorno del mese. In ogni casella con il numero di chiamate è presente anche la percentuale di KO,

- l'utente clicca sul bottone per aprire i dettagli di un utente,
- nella riga corrispondente si apre il dettaglio con un elenco delle lingue, in ordine alfabetico, su cui l'utente ha effettuato le chiamate, ognuna con il totale del mese e di ogni giorno, sia OK che KO.
- l'utente seleziona i clienti di cui desidera scaricare i dati,
- l'utente clicca su 'Download selected' e seleziona se scaricare i dati con o senza i dettagli sulle lingue,
- il server elabora la stringa da inserire nel file con i dati corrispondenti e scarica il file nel formato csv.

**POSTCONDIZIONI** al termine della procedura l'utente torna alla schermata di visione della tabella.

# Capitolo 3

# Implementazione web service

In questo capitolo viene presentata la fase implementativa del web service, realizzata sulla base della progettazione descritta nel capitolo 2.
In particolare sono descritti nel dettaglio gli script server-side in Python, che gestiscono le interazioni con i database e i controlli dei dati più complessi, gli script Javascript, che gestiscono gli eventi generati dall'interazione dell'utente con l'applicazione, e il codice HTML e CSS.

## 3.1 Configurazione iniziale

All'avvio dell'applicazione, tornado carica la configurazione iniziale contenuta nel file `config.json` attraverso la chiamata alle classi del file `configuration.py`. In particolare la classe `Configuration` setta nelle variabili di classe i dati dei parametri iniziali, quali host, port, le informazioni per l'invio delle email, i parametri di accesso ai database utilizzati e la lista di prefissi degli *admant*, carica i file delle tassonomie e setta il logger con i parametri contenuti in `logging.json`. La classe `MongoConfig` setta la connessione con i database e salva nelle variabili di classe i puntatori alle collezioni. La classe `SimpleDbConfig` setta la connessione con il database di SimpleDB e salva il dominio nella variabile relativa. Infine la classe `AdmantRegex` compila le espressioni regolari utilizzate dalla funzione `check_admant()`.

## 3.2 Autenticazione

Come richiesto in fase di progettazione, l'accesso all'applicazione è disponibile solo per gli utenti correttamente registrati e abilitati. La gestione degli utenti è implementata principalmente server-side, attraverso le classi di tornado `Register(RequestHandler)`, `Enable(RequestHandler)`, `Login(RequestHandler)`, `ForgotPsw(RequestHandler)` e `ChangePsw(RequestHandler)`, sottoclassi di `RequestHandler` e dagli script `login.js`, `logout.js`, `register.js`, `user_management.js` e `cookie.js`. Ogni classe prende in input i dati inviati dalla rispettiva chiamata Ajax, accede alla collezione *Users* del database *Monitor*, aggiunge o aggiorna il documento relativo all'utente in questione, imposta e invia l'email

all'amministratore e/o all'utente e risponde con un messaggio di notifica di avvenuta operazione o di descrizione dell'eccezione occorsa. Attraverso l'interfaccia quindi l'utente può inserire i dati richiesti e al termine della chiamata Ajax vedrà il messaggio di notifica.

All'avvio dell'applicazione, l'utente accede alla pagina `index.html`, che redirige l'utente alla pagina di Login `login.html` o direttamente alla Home, `home.html`, nel caso in cui nel browser fosse già presente il cookie corrispondente all'email dell'utente.

### 3.2.1 `login.js`

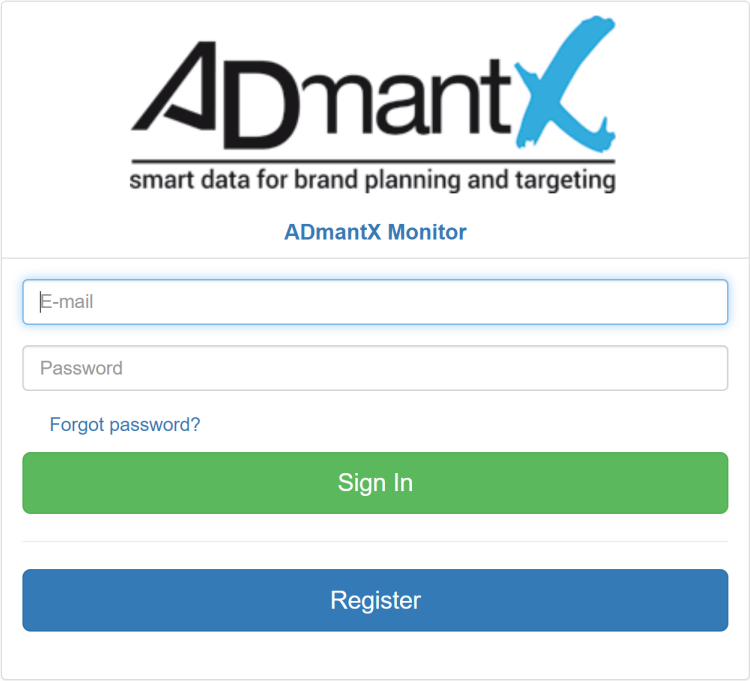


Figura 3.1: Login Panel

Lo script `login.js` gestisce le possibili azioni dell'utente effettuate dalla pagina iniziale `login.html` (figura 3.1), ovvero la richiesta di login, di forgot password e la redirezione alla pagina di registrazione.

#### Inserimento email e password nel form

Attraverso la chiamata jQuery Ajax, la classe `Login` di tornado riceve i due dati in input e in sequenza:

- controlla la presenza di `email` nel database; se non la trova ritorna un messaggio di non avvenuta registrazione.
- Controlla se il parametro `enable` del documento trovato è a `True`; in caso contrario ritorna un messaggio di non abilitazione.

- Genera la codifica della password inserite e controlla che combaci con il campo `password` del documento; in caso contrario ritorna un messaggio di errore nell'inserimento dei campi.
- Se tutti i controlli sono andati a buon fine ritorna un messaggio di successo e il nome e cognome dell'utente.

La chiamata Ajax quindi mostra la notifica di errore con la relativa descrizione o redirige l'utente alla pagina `Home` e salva email, nome e cognome dell'utente nei cookie.

### Forgot password

Nel caso di click sul bottone di 'Forgot password', si apre il modale di inserimento della email di registrazione (figura 3.2). Tale azione imposta una chiamata Ajax che invia la sua richiesta alla classe `ForgotPsw` di tornado, la quale

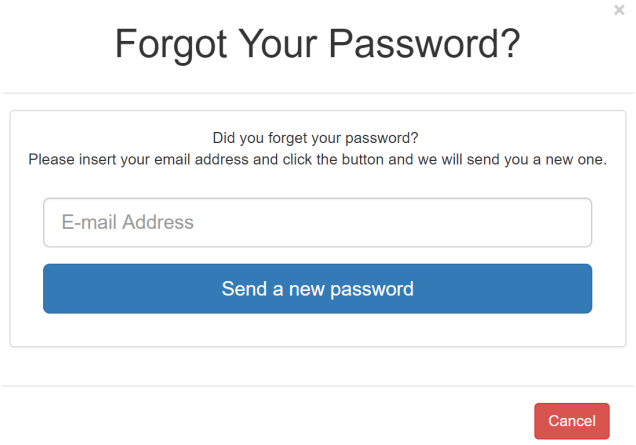


Figura 3.2: Forgot Password Modal

- controlla la presenza di `email` nel database. Se non la trova ritorna un messaggio di non avvenuta registrazione.
- Controlla se il parametro `enable` del documento trovato è a `True`. In caso contrario ritorna un messaggio di non abilitazione.
- Genera una password casuale attraverso la funzione `generatepsw()` e la codifica attraverso la funzione `encryptpsw()`.
- Aggiorna il campo `password` nel documento relativo della *collection*.
- Genera e invia la mail con la nuova password,
- Ritorna un messaggio di avvenuta operazione.

La chiamata Ajax infine mostra il messaggio di notifica ricevuto.

### Registration

Un click sul bottone 'Register', reindirizza l'utente alla pagina `register.html`, gestita dallo script `register.js`.

### 3.2.2 register.js

Dalla pagina di registrazione l'utente ha la possibilità di tornare alla pagina di Login o inserire i propri dati e registrarsi alla piattaforma. Con l'inserimento dei propri dati, la chiamata Ajax li invia alla classe `Register` di tornado, che

- controlla la presenza di `email` nel database; se la trova controlla il parametro `enable` e ritorna un messaggio di utente già registrato o registrato ma non abilitato,
- crea un nuovo documento nel database con `email`, `nome` e `cognome` inseriti e `enable` a False,
- setta e invia la mail all'amministratore con la richiesta di abilitazione e il link per effettuare tale operazione,
- ritorna un messaggio di avvenuta operazione.

La chiamata Ajax riceve il messaggio di notifica e lo mostra all'utente.

```
class Register(tornado.web.RequestHandler):
  def post(self):
    logger = Configuration.get_logger()
    try:
      # checking if user is already in the database
      user = self.get_argument('email').lower()
      found = MongoConfig.users_coll().find_one({"email": user})
      if found is not None:
        response = {"status": "KO"}
        if not found["enable"]:
          logger.info('Registration error: already registered')
          response["info"] = Notification.registered_not_enable
        else:
          logger.info('Registration error: already registered')
          response["info"] = Notification.registered
      else:
        # setting user's data and adding it to the database
        name = self.get_argument('name')
        surname = self.get_argument('surname')
        new_user = dict(name=name, surname=surname,
                        email=user, enable=False)
```

```
        MongoConfig.users_coll().insert_one(new_user)
        # sending email
        msg = texts.USER_REGISTRATION \
            .replace("<"+Globals.USERNAME+">", user)\
            .replace("<"+Globals.NAME+">", name)\
            .replace("<"+ Globals.SURNAME+">", surname)\
            .replace("<address>", Urls.ENABLE_EMAIL+user)
        info = {'To': Configuration.get_data("mail_to"),
            'Subject': texts.TITLE_USER_REGISTRATION,
            'Message': msg}
        send_email(info)
        # sending response
        response = {"status": "OK",
                    "info": Notification.registration}
        logger.info("Registration: " + user + " registered")
    self.write(json.dumps(response))
  except Exception, e:
    logger.error(e, exc_info=True)
    self.set_status(500)
    self.write(traceback.format_exc().replace("\n", "<br>"))
```

**Enable**

A un click dell'amministratore sul link contenuto nell'email di abilitazione ricevuta, viene chiamata la classe `Enable` di tornado che, in sequenza:

- genera la password e la sua codifica attraverso le funzioni già citate,
- aggiorna il campi `enable` a True e aggiunge il campo `password` con la password codificata al documento con la corrispondente `email`,
- setta e invia la mail all'utente con la notifica di avvenuta abilitazione e la nuova password in chiaro,
- ritorna un messaggio di avvenuta operazione all'amministratore.

### 3.2.3 `user_management.js`

Dalla barra di navigazione in cima alla pagina a cui accede l'utente dopo il login è possibile aprire il modale per la gestione dei propri dati o effettuare il logout (figura 3.3). Cliccando sul bottone 'Account' si apre il modale gestito dalla funzione `open_user_details()`, che recupera email, nome e cognome dell'utente loggato dai cookie della pagina e li inserisce nei campi corrispondenti del form. Il campo dell'`email` è di sola lettura e l'utente non potrà modificarlo. Con il bottone 'Change password' il modale si estende e compaiono altri due

campi del form per l'inserimento duplice della nuova password. Il cambiamento di uno dei

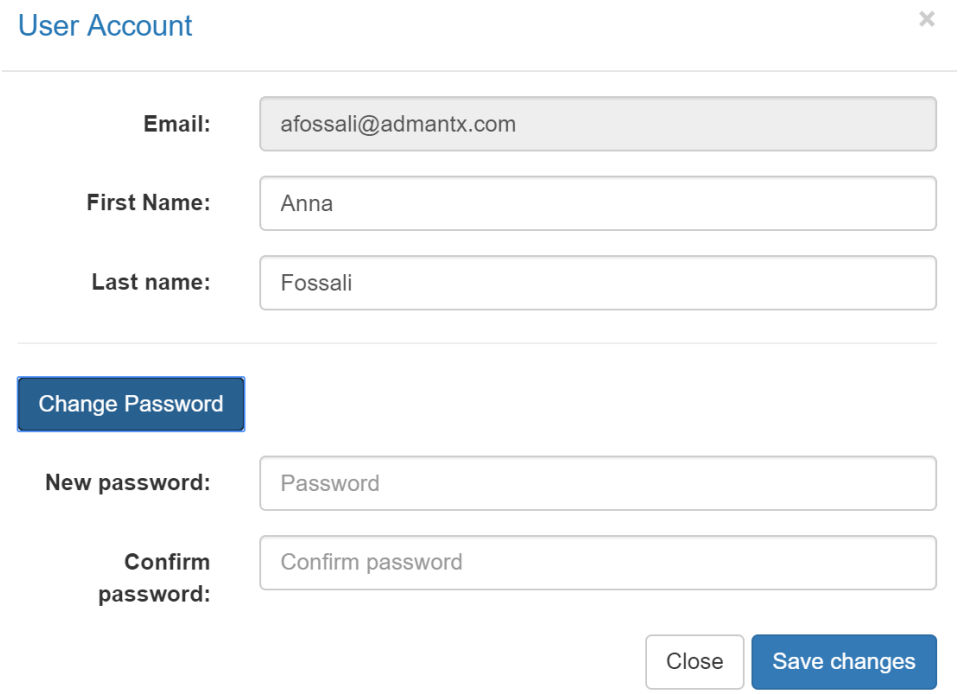


Figura 3.3: User Management Modal

dati e il click su 'Save Changes' viene chiamata la funzione che aggiorna i dati dei cookie e imposta una chiamata Ajax alla classe `ChangePsw()` di tornado, alla quale sono passati `email`, `nome`, `cognome` e la nuova `password` se cambiata, con il parametro booleano `changed` di cambiamento. La classe quindi

- codifica la password ricevuta, se `changed` è a True,
- aggiorna il documento relativo all'utente nel database,
- setta e invia l'email all'utente con i dati personali e la nuova password, se modificata,
- ritorna un messaggio di avvenuta operazione.

La funzione Ajax infine nasconde il modale e mostra la notifica di successo o mostra l'errore con la relativa descrizione.

### 3.2.4 `logout.js`

Lo script `logout.js` gestisce il logout manuale da parte dell'utente e quello automatico da parte del server, allo scadere di un intervallo di tempo di inattività.

Cliccando sul bottone 'Logout', posto nella barra di navigazione in cima alla pagina, viene chiamata la funzione `logout()` che elimina i parametri salvati nei cookie e redirige alla pagina di Login.

Una seconda funzione aggiunge alla pagina degli `EventListener` per tenere traccia dell'attività dell'utente sulla pagina. Ad ogni azione dell'utente viene resettato il timer di salvataggio dei dati dei cookie. Allo scadere di tale timer, il sistema effettua il logout automatico dell'utente, che sarà rediretto alla pagina di Login. Gli eventi di cui l'applicazione tiene traccia sono i movimenti del mouse, i click, gli scroll, la digitazione di un pulsante della tastiera e il refresh della pagina.

```
/* EventListener to reset timer */
window.addEventListener('load', resetTimer);
window.addEventListener('mousemove', resetTimer);
window.addEventListener('mousedown', resetTimer);
window.addEventListener('click', resetTimer);
window.addEventListener('scroll', resetTimer);
window.addEventListener('keypress', resetTimer);
/* Set timer  */
var t = setTimeout(logout, MAX_AGE_COOKIE*1000);
function logout() {
    clearCookie();
    window.location = "login.html"; }
function resetTimer() {
    clearTimeout(t);
    t = setTimeout(logout, MAX_AGE_COOKIE*1000);
    //reset cookie age
    setCookie("email", getCookie("email"), MAX_AGE_COOKIE);
    setCookie("name", getCookie("name"), MAX_AGE_COOKIE);
    setCookie("surname", getCookie("surname"), MAX_AGE_COOKIE);}
```

### 3.2.5 `cookie.js`

Per il salvataggio e il reset dei dati salvati nei cookie, l'applicazione utilizza le funzioni dello script `cookie.js`.

`setCookie()` salva un cookie per volta, con una `maxage` impostata da un parametro globale e per ogni `path` dell'applicazione.

`getCookie(c_name)` ritorna il valore del cookie corrispondente a c_name o una stringa vuota se non esistente.

`clearCookie()` svuota i campi dei valori dei cookie, impostandoli a stringhe vuote con `maxage` negativa, in modo che alla fine dell'operazione il browser li elimini completamente.

## 3.3 Gestione Admant e Profili clienti

Dopo l'autenticazione, l'azienda ha richiesto l'implementazione della parte di gestione degli *admant* e dei profili dei clienti. A questo scopo sono stati creati lo script `admant_profile.js`,

per la creazione e l'interazione con le tabelle e le classi di tornado per gli *admant* `PrintAdmant`, `NewAdmant` e `DeleteAdmant`, `CheckAdmant` e per i profili `PrintProfile`, `NewProfile` e `DeleteProfile`. La selezione del database degli *admant* o dei profili su cui si desidera lavorare avviene attraverso la barra laterale impostata nel file `home.html`, che con un menu dropdown presenta l'elenco completo. Cliccando su uno dei database al centro della pagine si visualizza la tabella con i dati relativi e le funzionalità richieste.

### 3.3.1 Admant

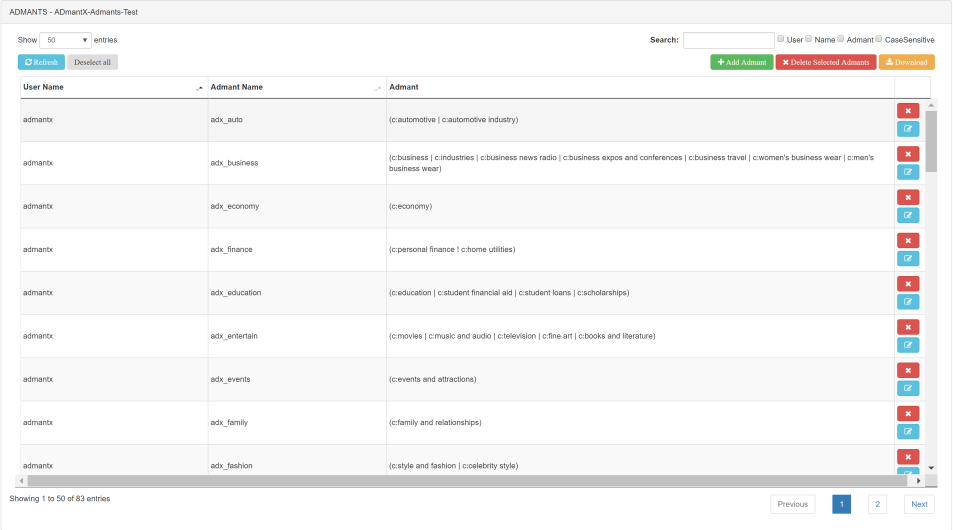


Figura 3.4: Admants table

La richiesta di visione della tabella di uno specifico database degli *admant* chiama la funzione javascript `createtable_admants(db_name)`, che crea l'oggetto `Datatable` e gli `EventListener` di jQuery per ogni possibile azione di interazione dell'utente (figura 3.4).

La creazione della tabella, salvata nella variabile `admant_table`, imposta quattro colonne: "Admant key", "Admant Name", "Admant" e l'ultima per l'inserimento dei bottoni di aggiornamento e rimozione di un *admant* specifico. Per maggiori informazioni sull'inizializzazione della tabella, si consulti l'appendice B.

#### Caricamento dei dati

La chiamata Ajax invia la richiesta con il parametro `db_name` alla classe `PrintAdmant` di tornado, che

- accede al database relativo su SimpleDb,
- scarica tutti i documenti attraverso la query `'select  from db_name'`,
- crea un dizionario per ogni *admant* scaricato, con i campi relativi alla chiave, al nome e all'espressione e un ulteriore campo booleano `added` che servirà per il riordinamento della tabella dopo l'aggiunta o la modifica di un dato,

- appende i dizionari creati a una lista `admant_list`
- ritorna un JSON in formato `"data": admant_list`

La chiamata Ajax infine accede ai dati nel campo `"data"` e li carica nella tabella, facendo corrispondere i campi con quelli dei parametri `data` delle colonne.

### Refresh

Attraverso la chiamata `Ajax.reload()` i dati vengono ricaricati nella stesso oggetto *Datatable*, precedentemente creato.

### Aggiunta e modifica di un admant

Il click sul bottone 'Add Admant', sopra la tabella, apre il modale con il form per l'inserimento dei campi necessari, mentre il click sul bottone di modifica nell'ultima cella della riga corrispondente, apre lo stesso form con i campi precompilati con i dati attuali dell'*admant* (figura 3.5). Il form, costruito nel file `home.html`, presenta i tre input per chiave, nome ed espressione, i bottoni di conferma e annullamento operazione, un terzo bottone 'Check Admant' per controllare la sintassi dell'espressione appena inserita e un checkbox per scegliere la tassonomia da utilizzare.

Sia la richiesta di aggiunta/modifica, attraverso un click sul bottone di conferma, sia la richiesta di 'Check Admant', vengono mandate con una funzione Ajax, alla classe `CheckAdmant`, che controlla la sintassi dell'espressione attraverso la funzione `check_admant()` e ritorna un messaggio di `OK`, `WARNING` o `ERROR`, in base all'output della funzione.

- Nel caso di `ERROR`, la funzione Ajax mostra il messaggio di errore, con la descrizione riguardo alla posizione di dove l'espressione risulta essere sbagliata.
- Nel caso di `WARNING`, viene mostrato un messaggio con la descrizione del warning trovato e, nel caso di richiesta di aggiunta, un bottone 'Continue anyway?', che se cliccato richiama la funzione `new_admant()` di javascript.
- Nel caso di `OK` viene chiamata la funzione `new_admant()` o solo mostrato il messaggio di notifica.

La funzione `new_admant()` recupera dai cookie l'email dell'utente che ha effettuato la richiesta, i valori dei campi e, nel caso di modifica dell'*admant*, il precedente username (key$name), e li invia attraverso una chiamata Ajax alla classe `NewAdmant`. Il metodo `post` della classe:

- controlla l'unicità del nuovo username, se lo username è stato cambiato, e in caso contrario ritorna un messaggio di errore,
- in caso di modifica, salva l'espressione dell'*admant* caricata sul database e cancella il vecchio documento dal database, imposta il contenuto dell'email con i dati del vecchio *admant* e di quello nuovo e imposta il messaggio di risposta,

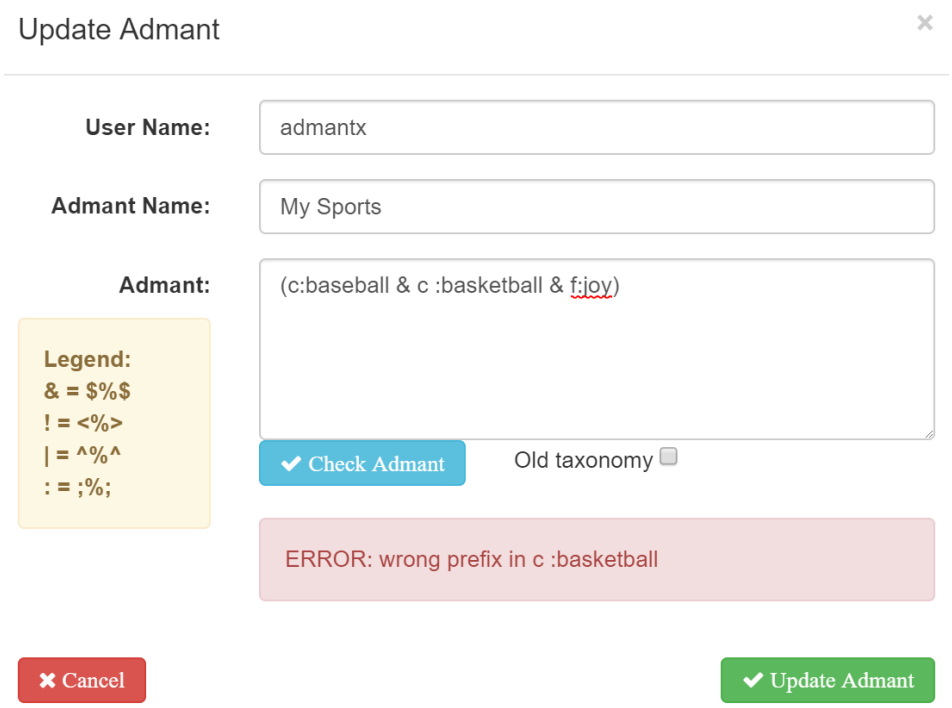


Figura 3.5: Update Admant Modal

- in caso di aggiunta imposta il contenuto dell'email con i dati del nuovo *admant* e il messaggio di notifica,
- aggiunge il nuovo *admant* al database,
- invia l'email all'amministratore,
- ritorna la risposta.

La chiamata Ajax, appena riceve la risposta, mostra il messaggio di avvenuta operazione, aggiunge la riga con i dati inseriti all'inizio della tabella o modifica i campi esistenti spostando la riga in cima ed effettua uno scroll all'inizio della tabella, oppure mostra il messaggio di errore di non unicità.

La descrizione dei controlli effettuati dalla funzione `check_admant()` si trova nell'appendice A.

### Rimozione di uno o più admant

Cliccando sul bottone rosso nell'ultima cella di una riga o selezionando più righe con un semplice click e cliccando sul bottone 'Delete selected admant' si apre un modale con le informazioni degli *admant* in questione e il bottone di conferma o di annullamento dell'operazione (figura 3.6). In caso di conferma, viene passata alla funzione `deleteadmant()` la lista degli username degli *admant* da rimuovere dal database e la chiamata Ajax invia la richiesta alla classe `DeleteAdmant`, che, per ogni *admant* nella lista,

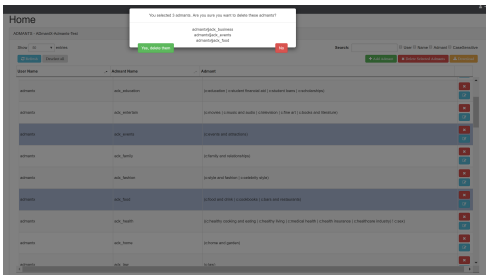


Figura 3.6: Delete Admants Modal

- salva l'espressione dal database di ogni *admant*,
- rimuove tutti i documenti relativi,
- aggiunge i dati degli *admant* cancellati al contenuto della email e invia l'email all'amministratore
- ritorna un messaggio di avvenuta operazione

La chiamata Ajax infine rimuove le righe corrispondenti e mostra il messaggio di notifica ricevuto.

### Ricerca di uno o più admant

Digitando una stringa nel campo di input in alto a destra, è possibile ricercare gli *admant* che combaciano con tale stringa. Attraverso i checkbox posti a fianco si seleziona i campi su cui effettuare la ricerca e se effettuarla *case sensitive* o *case insensitive*. Nel caso di nessuna selezione dei campi, la ricerca è effettuata su tutti. Ad ogni evento `keyup` della tastiera, la funzione ricerca la stringa nei campi desiderati e filtra i dati della tabella, mostrando solo le righe che combaciano alla ricerca.

```
/* Search admant */
$('#admant_filter_input').on('keyup', function() {
    var sensitive = $('#checkbox_sensitive').prop('checked');
    if(! $('#checkbox_user').prop('checked') &&
        !$('#checkbox_name').prop('checked') &&
        !$('#checkbox_admant').prop('checked'))
    // search on every fields, if none checked
    table.search($('#admant_filter_input').val(),
        true, false, !enable_sensitive).draw();
    else { // search only on checked fields
        if($('#checkbox_user').prop('checked'))
            {$('.col1').toggleClass('searched');}
        if($('#checkbox_name').prop('checked'))
            {$('.col2').toggleClass('searched');}
```

```
        if($('#checkbox_admant').prop('checked'))
            {$('.col3').addClass('searched');}

        table.columns('.searched')
            .search($('#admant_filter_input').val(),
                true, false, !enable_sensitive).draw();
    }
});
```

### 3.3.2 Profili dei clienti

La funzione javascript `createtable_profile(db_name)` inizializza e costruisce l'oggetto `Datatable` relativo alla tabella dei profili dei clienti e lo salva nella variabile `profile_table`. Le colonne in questo caso sono "UserID", "UserName", "email", "Types", "Filters", "Methods", "Max Count", "Analytics" e "Enabled" più un'ultima aggiuntiva con i bottoni di aggiornamento e rimozione di uno specifico profilo. Per maggiori informazioni sull'inizializzazione della tabella, si consulti l'appendice B.

Il caricamento dei dati, il refresh, la modifica e la rimozione di uno o più profili sono gestite dalle classi `PrintProfile`, `NewProfile` e `DeleteProfile` funzionano come già descritto in precedenza per gli *admant*. Anche l'aggiunta di un profilo si rifà all'aggiunta dell'*admant*,

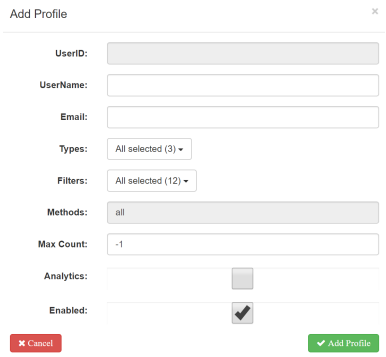


Figura 3.7: Add Profile Modal

ma lo "UserID" è aggiunto in automatico dalla classe `NewProfile` e all'apertura del form di aggiunta alcuni campi sono precompilati con dei valori di default, ma modificabili su richiesta (figura 3.7).

La ricerca infine è disponibile solo sui primi due campi del profilo, ID e Name.

### 3.3.3 Check Admant

Il pannello del check admant predispone una textarea dove inserire l'espressione che si desidera controllare, due checkbox per la scelta della tassonomia e per abilitare il controllo sulla lunghezza della stringa e il bottone per inviare la chiamata alla classe `CheckAdmant`, gestisce la richiesta come già descritto in precedenza e ritorna una notifica con un messaggio di ERROR,

WARNING o OK in base al risultato dei controlli effettuati, descritti nell'appendice A (figura 3.8).

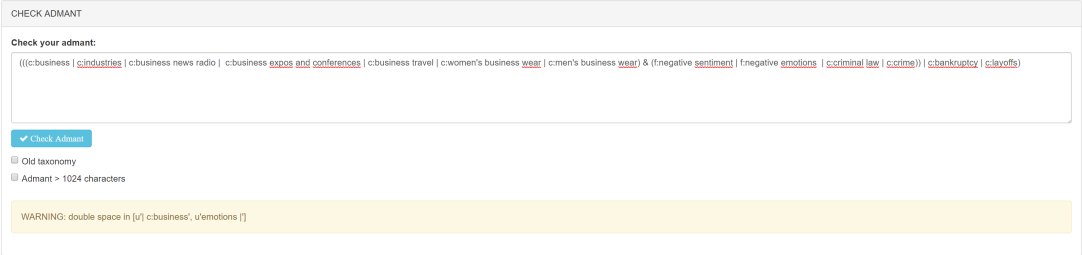


Figura 3.8: Check Admant Panel

## 3.4 Conteggi delle chiamate

L'ultima fase di implementazione del progetto ha riguardato la gestione delle chiamate ricevute da ogni cliente sulle varie macchine a disposizione dell'azienda e nelle varie lingue su cui è effettuabile l'analisi. Attraverso l'elenco degli indirizzi dei server che gestiscono tali chiamate, è stata necessaria l'implementazione delle classi di tornado e con i relativi EventListener del web service e di un applicativo di raccolta dati, del quale se ne descrive lo sviluppo nel prossimo capitolo.

Anche in questo caso la visione delle chiamate è stata sviluppata attraverso l'utilizzo di due oggetti `Datatable`, uno per le chiamate effettuate da ogni cliente, totali e suddivise per lingue e l'altro per le chiamate ricevute su ogni server, totali e suddivise per cliente. Le colonne di tali oggetti mostrano le informazioni del cliente, nome e tipologia sincrona o asincrona di chiamata, con un tooltip per la chiave, o il nome del server, il totale delle chiamate del mese, il numero di chiamate ricevute giorno per giorno e una colonna iniziale con il bottone per la visione della suddivisione per lingua. In ogni cella è visibile la percentuale di KO ricevuti e il numero totale, approssimato nel caso fosse maggiore di 100000 e con un tooltip indicante il valore esatto. La selezione del mese è possibile grazie all'oggetto `datetimepicker` di javascript posto in alto alla tabella e inizializzato al mese corrente. La richiesta di visione delle chiamate o una nuova selezione del mese è gestita dalla funzione `get_counters()` che legge il valore di `datetimepicker`, distrugge la tabella precedentemente creata e ne crea un'altra con i nuovi parametri. In particolare l'inizializzazione crea dinamicamente le colonne in base ai giorni del mese selezionato, gestibili on la funzione `daysInMonth()`, e carica i dati attraverso la chiamata Ajax, che invia la richiesta alla classe `Counters` di tornado con i parametri del mese e dell'anno e un booleano per indicare se è stata richiesta la visione della tabella dei clienti o dei server. La classe poi, attraverso `print_clients()` utilizza le funzionalità di MongoDB per recuperare e raggruppare i dati richiesti, sommando i parametri `count` e `count_ko` dei vari giorni e successivamente del mese e resettandone il formato, e restituisce un dizionario. Il codice che effettua il raggruppamento dei dati è presentato in seguito.

```
client_day = clientcol.aggregate([
    {"$match": {"day":{"$gt":year_month, "$lt":year_month+32}}},
    { "$group": {
        "_id": {"client": "$client", "name": "$name",
                    "sync": "$sync", "day": "$day"},
        "day_ok": {"$sum": "$count"},
        "day_ko": {"$sum": "$count_ko"}
        }
    },
    { "$group": {
        "_id": {"client": "$_id.client", "name": "$_id.name",
        "sync": "$_id.sync"},
        "month_ok": {"$sum": "$day_ok"},
        "month_ko": {"$sum": "$day_ko"},
        "days": {"$push": {"day": "$_id.day",
                    "day_ok": "$day_ok", "day_ko": "$day_ko"}}
        }
    }
],
    allowDiskUse=True
)
```

### 3.4.1 Chiamate per lingua e per cliente

Un doppio click sul nome del cliente o sul bottone nella prima colonna, apre l'oggetto `child_row` di Datatables, che permette di mostrare ulteriori informazioni riguardanti la riga. In particolare si manda una richiesta Ajax alla classe `CounterInfo` di tornado, che ritorna l'elenco delle chiamate di uno specifico cliente suddivise per lingua o di uno specifico server suddivise per cliente. Nel caso delle chiamate di uno specifico cliente, la funzione `print_languages()` prende in input i parametri `name` e `sync` del cliente e il mese e l'anno, recupera la lista delle lingue su cui il cliente ha effettuato chiamate dalla *collection* di Mongo `clientlanguages` e per ogni lingua della lista trova e raggruppa i dati come già descritto in precedenza. Infine ritorna i dati recuperati o un stato di errore se non ne ha trovati. La chiamata Ajax mostra una stringa di dati non esistenti nel caso di status KO o crea una tabella con una riga per ogni lingua e la allinea alla riga del cliente, elencando prima le chiamate OK e KO ricevute e in seguite quelle sulle lingue non supportate dall'analizzatore (figura 3.9). Nel caso delle chiamate di uno specifico server, la funzione `print_clients()` lavora come già descritto e alla fine della chiamata Ajax sarà visibile l'elenco dei clienti che hanno effettuato chiamate sul server.

Refresh | Deselect all | Download all | Download selected | Download languages

| | Month Total | 20190301 | 20190302 | 20190303 | 20190304 | 20190305 | 20190306 | 20190307 | 20190308 | 20190309 | 20190310 | 20190311 | 20190312 | 20190313 | 20190314 | 20190315 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **admantx** | | | | | | | | | | | | | | | | |
| sync | | | | | | | | | | | | | | | | |
| | 142K (2.5%) | 1705 (0.7%) | 1320 (0.5%) | 1206 (0.5%) | 1694 (0.6%) | 1845 (0.8%) | 1797 (3.1%) | 2862 (0.3%) | 7659 (0.2%) | 1330 (0.6%) | 1213 (0.3%) | 1911 (0.7%) | 1938 (0.8%) | 12K (3.5%) | 1828 (7.8%) | 1694 (14.4%) |
| **it** | **76K (2)** | 1493 (0) | 1170 (0) | 1056 (0) | 1467 (0) | 1621 (0) | 1554 (0) | 2667 (0) | 7434 (0) | 1161 (0) | 1023 (0) | 1618 (0) | 1650 (0) | 1700 (0) | 1343 (0) | 1170 (0) |
| **en** | **61K (5)** | 158 (0) | 144 (0) | 144 (0) | 163 (0) | 165 (0) | 177 (0) | 157 (0) | 163 (0) | 143 (0) | 144 (0) | 148 (0) | 154 (0) | 10K (0) | 292 (0) | 276 (0) |
| **de** | **1227 (0)** | 32 (0) | - | - | 39 (0) | 16 (0) | 2 (0) | 13 (0) | 43 (0) | 17 (0) | 39 (0) | 108 (0) | 84 (0) | 69 (0) | 37 (0) | - |
| **es** | **167 (0)** | 9 (0) | - | - | 12 (0) | 15 (0) | 9 (0) | 16 (0) | 2 (0) | - | 2 (0) | 7 (0) | 8 (0) | 11 (0) | 13 (0) | 4 (0) |
| **da** | **23 (4)** | - | - | - | - | - | - | - | - | - | - | - | 22 (4) | - | - | - |
| **nl** | **21 (1)** | 1 (0) | - | - | - | 8 (1) | - | - | - | - | - | - | - | - | - | - |
| **fi** | **15 (3)** | - | - | - | - | - | - | - | - | - | - | 14 (3) | - | - | - | - |
| **fr** | **10 (0)** | - | - | - | - | - | - | - | - | - | - | 3 (0) | - | - | - | - |
| **no** | **10 (0)** | - | - | - | 1 (0) | 6 (0) | - | - | - | - | - | - | - | - | 1 (0) | - |
| **pt** | **6 (0)** | - | - | - | 1 (0) | - | - | - | - | 1 (0) | - | - | 1 (0) | - | - | - |
| **cs** | **2 (0)** | - | - | - | - | - | - | - | - | - | - | - | 2 (0) | - | - | - |
| **sk** | **1 (0)** | - | - | - | - | - | - | - | - | - | 1 (0) | - | - | - | - | - |
| **pl** | **1 (0)** | - | - | - | - | - | - | - | - | - | - | - | 1 (0) | - | - | - |
| **ja** | **1 (0)** | - | - | - | - | - | - | - | - | - | - | - | 1 (0) | - | - | - |
| | UNSUPPORTED | | | | | | | | | | | | | | | |
| **ca** | **98** | 1 | 3 | - | 2 | 3 | 1 | 1 | 3 | 3 | - | 2 | 2 | 2 | 1 | 3 |
| **ro** | **5** | 1 | - | - | 1 | - | - | - | 1 | - | - | - | 1 | - | - | - |
| **hr** | **4** | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |

Figura 3.9: Clients Counts Table

### 3.4.2 Download

Il download dei dati è possibile sia con le informazioni dettagliate sulle lingue e i clienti che solo con i totali. Per implementare il download solo dei dati delle chiamate Ok e non dei KO, come richiesto dall'azienda, sono state aggiunte delle colonne non visibili, sia alle tabelle dei totali che alle tabelle nelle `child_rows`, con i dati richiesti. Il download delle tabelle con i dati delle `child_rows`, non già presente tra le funzionalità di Datatables, è stato customizzato in modo da aprire tutte le `child_rows` delle righe selezionate, se non già aperte, di scaricarne i dati e inserire nel file solo le colonne non visibili.

### 3.4.3 Show Partial

L'ultima funzionalità aggiunta al web service è la visione dei conteggi parziali delle chiamate ricevute (figura 3.10).
Con un doppio click su una cella relativa a un giorno, sia della tabella dei totali che di quella delle lingue e dei clienti nelle `child_rows`, apre il modale con un grafico, costruito e inizializzato attraverso l'oggetto `HighCharts`.
I dati di tale `HighCharts` sono scaricati attraverso una chiamata Ajax alle classi di tornado `Partials` e `PartialInfo`.
Le due classi lavorano allo stesso modo di `Counters` e `CountersInfo`, raggruppando e sommando i dati dei parziali dalla *collection* di Mongo `clientserver`.
Il grafico presenta sull'asse x i tempi dei parziali e sull'asse y il numero di chiamate e mostra due serie di dati: il totale delle chiamate, sia OK che KO, in blu, e le chiamate KO, in nero.
Il passaggio del mouse su un punto di una delle due serie mostra un tooltip con i conteggi esatti di entrambe le serie, mentre la selezione di un'area del grafico permette di zoommare un particolare intervallo di tempi parziali o di conteggi, per avere una visualizzazione dei dati più precisa.

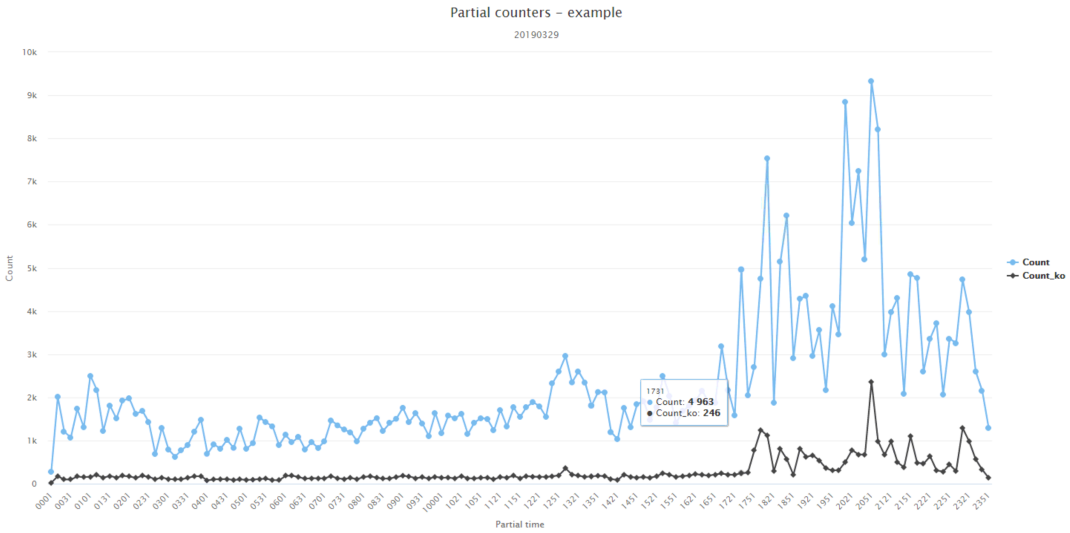


Figura 3.10: Server Line Chart

Per maggiori informazioni sull'inizializzazione di `HighCharts` si veda l'appendice B.

# Parte III

# Sviluppo software di appoggio

# Capitolo 4

# Software di raccolta e ottimizzazione dei dati

Come già specificato precedentemente, i conteggi delle chiamate ricevute dai clienti sono salvati sui database Redis installati su ogni server, la cui complessità architetturale ha reso necessario lo sviluppo di un software di sostegno, che gestisse la raccolta di tali dati. L'applicativo di raccolta dati deve quindi accedere ai database di ogni server, scaricare i dati di ogni cliente e salvarli nel database appositamente creato su MongoDB, utilizzando la struttura dati presentata nella sezione 2.1.2, necessaria per la visione delle chiamate ricevute, descritta nel capitolo 3.4. In particolare il software deve chiamare la funzione di raccolta dati allo scadere di un intervallo di tempo, impostato inizialmente. Ad ogni chiamata tale funzione cerca i dati della giornata corrente e aggiorna i conteggi su Mongo, mentre all'avvio dell'applicazione è possibile impostare la modalità di recovery per la raccolta dei conteggi delle chiamate effettuate tra due date, impostate come parametri di input.

## 4.1 Struttura dati

Ogni server presenta due database: uno per i conteggi totali delle chiamate ricevute da uno specifico cliente, l'altro per i conteggi suddivisi per le lingue di cui ogni chiamata ha richiesto l'analisi. I dati delle chiamate sono salvati in set ordinati di Redis.

### 4.1.1 Database 1

Il primo database presenta un elenco di chiavi dalla struttura

`counters:<mode>:<clientkey>`

o

`ko:<mode>:<clientkey>`,

dove `<mode>` può essere `sync` o `async` a seconda della tipologia della chiamata e `<clientkey>` indica la chiave del cliente. Le chiavi che iniziano con `"counters"` presentano i conteggi totali delle chiamate ricevute, sia andate a buon fine che KO, mentre le altre presentano solo le

chiamate KO. Ad ogni chiave corrisponde uno zset, ovvero un insieme di coppie chiave-valore, in cui la chiave rappresenta i giorno e il valore il conteggio, ordinato secondo i conteggi.

### 4.1.2 Database 2

Il secondo database salva le chiamate di ogni cliente suddivise per lingue. Le chiavi hanno la seguente struttura:

`languages:<mode>:<clientkey>`

o

`unsupported-languages:<mode>:<clientkey>`,

dove `languages` raggruppa le chiamate ricevute, sia OK che KO, da `clientkey` in modalità `mode`, suddivise per lingua e per giorno, mentre `unsupported-languages` tiene traccia solo delle chiamate ricevute sulle lingue non supportate. Ad ogni chiave corrisponde un zset, come nel caso precedente, dove la chiave ha il formato `day:OK/KO:language` o `day:language`, nel caso delle lingue non supportate. Un esempio di dato quindi potrebbe essere 20190410:OK:it.

## 4.2 Progettazione

Per rendere l'applicazione il più veloce possibile, lo script scarica i dati da Redis di ogni server e crea le relative strutture dati in locale, caricandole solo al termine sulla *collection* di Mongo. All'avvio dell'applicazione, deve essere implementata una funzione che scarichi i dati presenti su Mongo relativi al giorno corrente, in modo che se i server dei conteggi dovessero spegnersi, non si perdono i dati precedentemente salvati con una sovrascrittura. Inoltre, nel caso fosse richiesta, deve essere disponibile una funzione di recovery che, date due date in input, scarichi tutti i conteggi dei giorni compresi tra le due date e carichi o aggiorni i relativi documenti di Mongo sul database. Ad ogni chiamata effettuata allo scadere dell'intervallo invece l'applicazione lavora solo con i dati relativi al giorno corrente. Infine è necessaria una funzione che tenga traccia dell'ora locale di ogni server, poiché essendo in fusi orari diversi, c'è il rischio di perdere alcuni dati dei giorni precedenti o di non avere il database su Mongo totalmente aggiornato ai dati delle macchine in fusi orari seguenti a quello della macchina su cui gira l'applicativo.

## 4.3 Implementazione

L'applicazione presenta un file JSON con i parametri di configurazione iniziali, il file di configurazione del logger, lo script per la gestione dell'intervallo di tempo e il file principale, con l'effettiva classe di scaricamento dei conteggi e caricamento dei nuovi documenti. Il file di configurazione `config.json` contiene i parametri riguardanti l'elenco degli indirizzi, degli indici dei database e degli ambienti di ogni server aziendale, l'intervallo di tempo per la gestione delle chiamate, impostato a 10 minuti, e i parametri di accesso ai database di Mongo

e Redis e al database di SimpleDB dei profili dei clienti, per il salvataggio dei nomi dei clienti oltre che della loro chiave.

La classe `RedisToMongo` lavora principalmente su tre variabili d'istanza rappresentanti tre dizionari: `self.servers_dates`, `self.server_client` e `self.client_languages`. Nella prima tiene traccia dell'ultimo giorno in cui è stata effettuata una chiamata, in modo da non perdere dati se la macchina dovesse passare al giorno successivo nel caso si trovasse in un fuso orario diverso. La struttura dati di ogni elemento del dizionario è una lista di tre valori: il giorno, un booleano settato a True se il giorno è già stato aggiornato all'ultima chiamata effettuata e un secondo booleano settato a True se il giorno è già stato aggiornato nelle ultime 24 ore. Al passare di un giorno, se il primo parametro non è mai stato aggiornato, l'applicazione lo aggiorna in automatico al giorno successivo. Nel secondo dizionario salva gli effettivi documenti che saranno poi scaricati dal web service per la visione delle chiamate, con struttura dati semplificata rispetto a quella realmente caricata sul database, la quale, oltre ai parametri `day`, `server`, `client`, `name`, `sync`, `count`, `count_ko`, tiene traccia solo dell'ultimo `partial: {last_time: last_partial_count}`, dell'ultimo `partial_ko` e degli ultimi parziali nella mappatura delle lingue. Infine l'ultimo parametro salva l'elenco di tutti i clienti e delle lingue per cui ogni cliente richiede l'analisi, come già presentato nella sezione 2.1.2.

All'avvio dell'applicazione, viene caricato e settato il logger e, se presenti, controllato il formato delle date inserite per la modalità di recovery. Nel caso di inserimento di valori sbagliato, l'applicazione ritorna un messaggio di errore ed esce, altrimenti inizializza la classe `RedisToMongo` e avvia il primo scaricamento dei dati. Infine chiama la funzione `set_interval`, impostata sia sulla funzione di scaricamento ad ogni intervallo di tempo, sia sulla funzione di aggiornamento delle date associate ai server ogni 24 ore.

### 4.3.1 Inizializzazione

La funzione `__init__()` prende in input le date di recovery, se presenti, setta le variabili d'istanza con i valori del file `config.json` e la variabile di classe che memorizza l'intervallo di campionamento dei dati e richiama le funzioni di recovery, di settaggio delle date di ogni server e di download dei dati riguardanti il giorno corrente.

La funzione di recovery `load_past_mongo()` prende in input le due date e, per ogni server, accede in pipeline al database di Redis dei conteggi totali e passa i dati scaricati alla funzione `add_day()`, che imposta i documenti con la struttura dati richiesta, ed infine carica i dati su Mongo.

La funzione `set_dates()` imposta la mappa dei server `server_dates`: inizialmente associa ad ogni server il giorno antecedente al corrente, in seguito verifica se sono presenti chiamate per il giorno successivo o per il giorno seguente ancora. In tal caso aggiorna il parametro relativo al giorno ed esce. In questo modo si può supporre di avere recuperato il giorno effettivo impostato sui vari server. Nel caso in cui un server non ricevesse chiamate per uno o più giorni, l'applicazione aggiorna il parametro ogni 24 ore.

La funzione download_mongo(), infine, utilizza le date dei server salvati nella mappa server_dates e per ogni server scarica i documenti presenti su Mongo relativi a quei giorni e li riformatta alla struttura semplificata salvata in locale.

### 4.3.2 Aggiornamento counters

Allo scadere di ogni intervallo di tempo, viene richiamata la funzione day_counters(). Tale funzione setta il parametro self.time con il quale vengono salvati i parziali e, per ogni server, si connette al database 1 e controlla la presenza di chiamate nel giorno successivo e in quello seguente, attraverso la funzione chech_new_day(). Nel caso trovasse dei conteggi con la chiave di uno di questi due giorni, aggiorna la data del server salvata in server_dates, scarica un'ultima volta i dati da Redis relativi al giorno precedente, settando la struttura dati necessaria sempre attraverso add_day(), e infine carica su Mongo i documenti relativi a quel server e a quel giorno e li cancella dal dizionario locale server_client. Dopo il controllo dell'aggiornamento della data, day_counters() scarica il conteggio totale e quello delle chiamate KO e li passa alla funzione add_day(), il cui codice è riportato in seguito:

```
def add_day(self, key, server, ok_count, ko_count, past_day):
  try:
    client = key.split(":")
    ok_count = int(ok_count)
    ko_count = int(ko_count)
    if past_day == 0:
      day = self.servers_dates[server][0]
    else:
      day = past_day
    new_key = server + client[2] + client[1] + unicode(day)
    found = self.server_client.get(new_key)
    if found is None:
      langs_ok, langs_ko, langs_unsup =
            self.get_languages_new(server, client, day)
      if langs_ok is not None:
        new_day = {SERVER: server,
                   CLIENT: client[2],
                   DAY: day,
                   ENV: self.hosts[server],
                   NAME: self.get_clientname(client[2]),
                   SYNC: client[1],
                   COUNT: ok_count,
                   COUNT_KO: ko_count,
                   PARTIAL: {self.time: ok_count},
                   PARTIAL_KO: {self.time: ko_count},
```

```
                    LANG_OK: langs_ok,
                    LANG_KO: langs_ko,
                    LANG_UNSUP: langs_unsup}
        self.server_client[new_key] = new_day
    else:
      if ok_count >= self.server_client[new_key][COUNT]:
        redis_down = False
        self.server_client[new_key].update(
            {COUNT: ok_count,
             COUNT_KO: ko_count,
             PARTIAL: {
                    self.time: ok_count -
                        self.server_client[new_key][COUNT]},
             PARTIAL_KO: {
                    self.time: ko_count -
                        self.server_client[new_key][COUNT_KO]}})
        else:
            redis_down = True
            self.server_client[new_key].update({
                COUNT: ok_count +
                    self.server_client[new_key][COUNT],
                COUNT_KO: ko_count +
                    self.server_client[new_key][COUNT_KO],
                PARTIAL: {self.time: ok_count},
                PARTIAL_KO: {self.time: ko_count}})
        self.get_languages_old(
                server, client, day, new_key, redis_down)
  except Exception:
    traceback.print_exc()
    logger.error('ERROR add_day():', exc_info=True)
```

Tale funzione prende in input i due conteggi, la chiave del cliente, il nome del server e il booleano che indica se si tratta dei dati relativi al giorno attuale o a un giorno passato per la modalità di recovery. Ad ogni sua chiamata, cerca il file nel dizionario `server_client` con gli stessi parametri `server`, `client` e `day`. Nel caso non trovasse alcun documento corrispondente, ne crea uno nuovo e lo aggiunge al dizionario, altrimenti paragona i conteggi totali appena scaricati con quelli del documento trovato. Se il nuovo conteggio è minore significa che il database Redis è stato spento e quindi i conteggi sono stati azzerati; in questo caso aggiorna il documento sommando i conteggi e ponendo il parziale uguale al nuovo conteggio. In caso contrario aggiorna i dati sostituendo i conteggi e calcolando il parziale attraverso la loro differenza.

Con l'aggiunta o l'aggiornamento dei documenti in locale, vengono chiamate anche le funzioni di scaricamento delle lingue dal database 2 dei server. Queste due funzioni `get_languages_new()` e `get_languages_old()` utilizzano tre dizionari `lang_ok`, `lang_ko` e `lang_unsup` per il salvataggio dei conteggi specifici delle lingue. La prima funzione, chiamata nel caso di aggiunta di un nuovo documento a `server_client`, aggiunge ogni lingua trovata alla lista relativa al cliente nel dizionario `client_languages`, scarica i conteggi di ogni lingua del relativo giorno e li aggiunge ai tre dizionari. Infine ritorna i tre dizionari, che saranno aggiunti come valori relativi ai parametri delle lingue al documento appena creato. La seconda funzione, chiamata nel caso di aggiornamento di un documento, modifica direttamente i tre dizionari nel documento, scaricando i dati relativi e aggiornando o aggiungendo i conteggi delle lingue, anche nel caso in cui Redis sia stato spento.

La funzione `day_counters` infine, dopo aver settato tutti i documenti necessari, carica i dati di `server_client` sulla collection `clientserver` di Mongo e i dati di `client_languages` sulla collection `clientlanguages`, attraverso la chiamata a `load_mongo`.

Quest'ultima funzione scarica il documento corrispondente ad ogni dato presente in `server_client`, se presente, e lo aggiorna sommando o sostituendo i dati a seconda che Redis sia caduto o meno. Nel caso non trovasse il documento corrispondente, aggiunge quello appena creato in locale. Inoltre aggiorna anche la lista delle lingue su cui i clienti hanno fatto le chiamate, cercando sempre il documento corrispondente a un dato di `client_languages` e aggiornandolo o aggiungendolo alla collection.

### 4.3.3 Settaggio dell'intervallo di tempo

Per la gestione delle chiamate allo scadere dell'intervallo di tempo, si è implementata la classe `setInterval`, che, attraverso il modulo di python `threading` imposta l'azione da eseguire e l'evento che fermi la richiamata di tale azione. Con l'utilizzo di cicli while inoltre, la funzione prosegue con le chiamate ad ogni intervallo anche in caso di errori e eccezioni.

```
class setInterval:
    def __init__(self, interval, func):
        try:
            logger = logging.getLogger("log_counters")
            self.interval = interval
            self.action = func
            self.stopEvent = threading.Event()
            thread = threading.Thread(target=self.__setInterval)
            thread.start()
        except Exception:
            traceback.print_exc()
            logger.error('ERROR __init()__', exc_info=True)

    def __setInterval(self):
```

```
try:
    logger = logging.getLogger("log_counters")
    nextTime = time.time() + self.interval
    while not self.stopEvent
                  .wait(nextTime - time.time()):
        try:
            nextTime += self.interval
            self.action()
        except Exception:
            traceback.print_exc()
            logger.error('ERROR __setInterval()',
                exc_info=True)
            continue
except Exception:
    traceback.print_exc()
    logger.error('ERROR __setInterval()',
        exc_info=True)
```

## 4.4 Test

Per verificare la correttezza dello scaricamento e dell'ottimizzazione dei dati, si sono sviluppati ulteriori script di test e controllo. È fondamentale testare l'uguaglianza tra i conteggi di ogni documento di Mongo e quelli su ogni database Redis dei server, ma anche la suddivisione dei conteggi giorno per giorno nei parziali e il totale mensile. La fase di test quindi ha portato alla scrittura di diversi script python.

### 4.4.1 `check_total.py`

Per la verifica di un corretto salvataggio dei conteggi totali su Mongo, è stato sviluppato lo script `check_total.py`. Lo script presenta diverse funzioni di controllo, quelle per i conteggi totali di ogni cliente e di ogni server e quelle per i conteggi specifici di ogni cliente su uno specifico server e di ogni lingua per uno specifico cliente. Inoltre controlla che tutte i dati presenti su Redis siano presenti su Mongo con gli stessi valori. Nel primo caso, per esempio, la funzione prende in input il nome di un cliente e il mese su cui effettuare il controllo e cerca e raggruppa tutti i documenti di Mongo di uno specifico cliente per un certo mese, effettuando le stesse somme già descritte in precedenza per la visione delle chiamate. In seguito cerca la chiave del cliente su tutte le macchine aziendali utilizzate per i conteggi e somma tra loro i dati di uno stesso giorno. Infine controlla l'uguaglianza, giorno per giorno dei conteggi, sia totali che delle chiamate KO. Nel caso alcuni dati fossero diversi, ritorna un messaggio di errore e una lista di tuple `(giorno, conteggio_mongo, conteggio_redis)`.

### 4.4.2 `check_mongo.py`

Lo script `check_mongo.py` effettua un controllo più leggero, controllando solo i dati già presenti su Mongo, confrontando i totali con il dato relativo su Redis e testando la consistenza dei conteggi parziali. Le funzioni presenti lavorano sia sui conteggi totali e parziali di ogni giorno, sia su quelli delle specifiche lingue. La funzione `check_lang()`, per esempio, per ogni lingua presente nelle mappe `languages_ok`, `languages_ko` e `languages_unsupported`, controlla i parametri `count` e `count_ok` confrontandoli con i conteggi del cliente sul server nel giorno relativo al documento di Mongo. Inoltre somma tutti i parziali OK e KO confrontandoli sempre con i totali. Nel caso di errore ritorna un messaggio di errore e la lista dei documenti in cui sono stati trovati tali errori.

# Conclusione

Lo sviluppo del progetto svolto durante il tirocinio presso l'azienda ADmantX ha raggiunto gli obiettivi prefissati. L'applicazione implementata rende disponibile un'interfaccia web con la quale è possibile monitorare e gestire tutti i dati interni dell'azienda. Il web service sviluppato gestisce l'area privata dell'interfaccia e presenta le funzionalità di aggiunta, rimozione, modifica, ricerca e download dei dati attraverso l'interazione con diversi database di tipo NoSQL. L'applicativo di sostegno inoltre rende disponibili nuove strutture dati ottimizzate alla visualizzazione dei dati raggruppati per server o cliente. Ciò facilita il conteggio totale delle chiamate ricevute, che è utile nell'analisi dell'utilizzo della tecnologia semantica da parte dei clienti. In particolare grazie alla scelta delle più adatte tecnologie di sviluppo, l'applicazione presenta un'interfaccia grafica facile da comprendere e con funzionalità specifiche per ogni possibile richiesta dell'utente. Oltre al controllo e al miglioramento del codice durante la fase di sviluppo, al completamento del progetto sono stati effettuati test su tutti i servizi offerti per verificarne il corretto funzionamento. In questo modo l'applicazione finale risulta robusta ed efficiente.

L'azienda sta utilizzando le due applicazioni create per la gestione giornaliera dei dati, sia internamente che nell'interazione diretta con i clienti. In futuro mira ad un'ulteriore ottimizzazione nella raccolta dati, per abbassare il tempo necessario di accesso e scaricamento degli stessi.

# Bibliografia


[1] ADmantX. http://www.admantx.com.

[2] Amazon SimpleDB. https://aws.amazon.com/it/simpledb/.

[3] Bootstrap. https://getbootstrap.com/.

[4] DataTables. https://datatables.net/.

[5] FontAwesome. https://fontawesome.com/.

[6] HighCharts. https://www.highcharts.com.

[7] jQuery. https://api.jquery.com/.

[8] MongoDB. https://www.mongodb.com.

[9] Nginx. https://www.nginx.com/.

[10] Python, datetime package. https://docs.python.it/html/lib/module-datetime.html.

[11] Python, email package. https://docs.python.it/html/lib/module-email.html.

[12] Redis. https://redis.io.

[13] Tornado. https://www.tornadoweb.org.

# Appendice A

# Check Admant

Un'*admant* è una combinazione di elementi, combinati da uno o più operatori booleani, che definisce il tipo di contenuto di un testo. Gli elementi sono suddivisibili secondo la tipologia di cui fanno parte. Le tipologie disponibili sono:

- le *categories* descrivono il contenuto del testo,
- i *feelings* rappresentano le emozioni che il testo suscita nel lettore,
- i *lemmas* indicano le *entities* e i *mainlemmas* riconosciuti dalla tecnologia semantica,
- le *keywords* sono le parole chiave trovate,
- i *tags* rappresentano i collegamenti a liste esterne di keywords,
- le *regex* sono le espressioni regolari che corrispondono a una parte del testo,
- le *url* indicano le parti di url trovate,
- i *vocabulary* descrive la lingua nella quale un testo deve essere scritto perché venga analizzato.

Le *entities* si suddividono in *people*, *organizations* e *places* e indicano quindi le persone, le organizzazioni e i luoghi, mentre i *mainlemmas* sono elementi del testo riconosciuti come importanti, normalmente in base alla frequenza in cui occorrono, e riportati alla loro forma base. Il lemma "bello" ad esempio avrà una corrispondenza con "bello", "bella", "belli" e "belle", mentre il lemma "mangiare" trova le occorrenze di "mangiare", ma anche di tutta la coniugazione del verbo. I *tag* sono utili nel caso le *keyword* necessarie facessero aumentare la lunghezza dell'*admant* oltre la consentita, permettendo quindi di sostituirne un insieme con un singolo elemento.

## A.1 Regole sintattiche e algoritmo

La sintassi degli *admant* in generale è

`elemento operatore elemento`,

ma può essere composto da più blocchi di elementi, aggregati con delle parentesi tonde. In questo caso la sintassi per esempio diventa

`blocco operatore blocco`

Gli operatori disponibili sono `& - AND`, `| - OR` e `! - NOT`. Ogni elemento deve essere preceduto dal prefisso della tipologia a cui appartiene, che serve ad identificarlo, seguito dal carattere ":". I prefissi sono indicati dalla lettera iniziale di ogni tipologia e sono:

"c", "f", "l", "k", "t", "r", "u" e "v".

All'interno degli elementi non possono comparire una serie di caratteri riservati: `&`, `|`, `!`, `:`, `(` e `)`, poiché servono per definire i blocchi e gli operatori. Nel caso fosse necessario l'inserimento di uno di questi caratteri all'interno di un elemento, deve essere trasformato secondo la seguente legenda:

$$\& = \$\%\$, \qquad ! = <\%>, \qquad | = \hat{}\%\hat{}, \qquad : = ;\%;$$

Un *admant* inoltre non può superare i 1024*8 byte, se non esplicitamente indicato. La lunghezza dipende quindi dal numero di byte, non dal numero di caratteri presenti nell'espressione e tiene conto del fatto che i caratteri UTF-8 non occupano tutti lo stesso spazio, ma quelli che non appartengono ai caratteri ASCII occupano due o più blocchi di 8 byte. Ogni tipologia infine necessita di regole sintattiche diverse a seconda dei casi:

- *categories* e *feelings*: le categorie e i sentimenti devono essere per forza incluse nella tassonomia, un elenco gerarchico di elementi già formati. ADmantX utilizza due tassonomie, una nuova composta da un solo file e una vecchia composta dal file delle categorie in inglese e da quello per le categorie in italiano. L'*admant* può essere formato da elementi appartenenti solo a una delle due tassonomie.

- *keywords*: le keywords possono finire con i suffissi `[s]` e `[i]`, ma l'utilizzo di tale suffisso non è obbligatorio.

- *regex* e *url*: questi due elementi possono contenere caratteri url-codificati, che dovranno essere trasformabili nel carattere corrispondente. Il codice del carattere deve quindi corrispondere a un effettivo carattere dell'alfabeto UTF-8.
  (Esempio: u:edition.cnn.com/test%2Bprova = u:edition.cnn.com/test+prova).

- *vocabulary*: questi elementi devono essere composti solo da due caratteri, indicanti il codice della lingua.

- *lemmas* e *tags*: i lemmi non hanno regole aggiuntive.

I controlli sono stati implementati nella funzione `check_admant()`, chiamata dalla classe di tornado `CheckAdmant`. L'algoritmo applicato parte dall'intera espressione dell'*admant*,

effettuando i primi controlli sulla lunghezza in byte, sul conteggio e il posizionamento delle parentesi aperte e chiuse e sulla presenza di doppi operatori. Se questi primi controlli vanno a buon fine suddivide l'espressione in blocchi (sotto-stringhe contenute tra le parentesi tonde) ed effettua i controlli sintattici relativi sui blocchi. In particolare, per ogni blocco, controlla

- la presenza di almeno un elemento all'interno del blocco,
- l'uguaglianza tra il numero di elementi (contati con le ripetizioni del carattere ":" del prefisso) e il numero di operatori aumentato di uno,
- il numero di spazi bianchi tra gli elementi.

Se anche questi controlli vanno a buon fine, suddivide ogni blocco negli elementi da cui è composto ed effettua i controlli inerenti:

- la presenza di almeno una tipologia di elemento,
- la correttezza del prefisso, che deve appartenere a uno dei caratteri elencati in precedenza,
- la presenza di caratteri riservati,
- il rispetto delle regole di ogni tipologia, ovvero dell'appartenenza delle *categories* e dei *feelings* alla tassonomia, la correttezza dei caratteri codificati nelle *url* e nelle *regex* e la lunghezza dei *vocabularies*.

Al termine dei controlli su un blocco, sostituisce il blocco con il blocco di default `l:default` e passa al successivo. Dopo aver controllato tutti i blocchi, l'algoritmo controlla la presenza di possibili errori, identificati come `WARNING`. In particolare cerca la presenza di doppi spazi bianchi, di uno spazio bianco tra il prefisso e l'elemento o un suffisso diverso da `[s]` o `[i]` nelle *keywords*.

Al termine, la funzione ritorna un messaggio con uno status `OK`, `WARNING` o `ERROR` e la descrizione del controllo non andato a buon fine, se presente. Lo status di risposta è utilizzato dalla classe `CheckAdmant` per la visualizzazione della notifica o il blocco della chiamata di aggiunta dell'*admant*.

La ricerca dei blocchi, degli elementi e delle stringhe dei controlli è effettuata attraverso il riconoscimento di espressioni regolari salvate e precompilate nella classe `AdmantRegex` della configurazione all'avvio dell'applicazione.

## A.2 Codice implementato

In seguito è riportato il codice della funzione in python `check_admant()`, con delle funzioni di controllo dei singoli blocchi e elementi.

```
def check_admant(admant, old_taxonomy, enable_length):
    check_result = {"status": "KO", "parameters": "error"}
    first_response = first_checks(admant, enable_length)
    if first_response != "continue":
        check_result["info"] = first_response
        return check_result
    default = "l:default"
    risp = "(" + admant + ")"
    while bool("(" in risp) & bool(")" in risp):
        blocks = re.findall(AdmantRegex.BLOCK, risp)
        if not blocks:
            check_result["info"] = CheckAdmant.brackets
            return check_result
        for block in blocks:
            block_response = check_block(block)
            if block_response.startswith("ERROR"):
                check_result["info"] = block_response
                return check_result
            elements = block_response.split("(")
            for element in elements:
                element_response = check_element(element,
                                                 old_taxonomy)
                if element_response != "continue":
                    check_result["info"] = element_response
                    return check_result
            risp = risp.replace(block, default)
    check_result["info"] = admant_warning(admant)
    check_result["status"] = "OK"
    if check_result["info"] != CheckAdmant.no_error:
        check_result["parameters"] = "warning"
    else:
        check_result["parameters"] = "correct"
    return check_result

def first_checks(admant, enable_length):
    if (not enable_length)&(len(admant.encode('utf-8'))>1024):
        return CheckAdmantResponse.long
    if admant.count('(') > admant.count(')'):
        return CheckAdmant.brackets_open

    if admant.count('(') < admant.count(')'):
```

```
        return CheckAdmant.brackets_close
    if re.search(AdmantRegex.DOUBLEOP, admant) is not None:
        return CheckAdmantResponse.operators
    if re.search(AdmantRegex.BRACKETS, admant) is not None:
        return CheckAdmantResponse.position_brackets
    return "continue"

def check_block(block):
    if block.count(":") !=
      (block.count("&")+block.count("|")+block.count("!")+1):
        return CheckAdmantResponse.element
    newblock = block[1:-1].strip()
    if newblock == '':
        return CheckAdmant.empty_block
    newblock = newblock.replace(" ! ", "(")
                .replace(" & ", "(").replace(" | ", "(")
    if bool("!" in newblock) | bool("&" in newblock) |
            bool("|" in newblock):
      return CheckAdmant.spaces.replace(Globals.BLOCK, block)
    return newblock

def check_element(element, is_old_taxonomy):
    element = element.strip()
    if element == '' or element.split(":")[1].strip() is '':
        return CheckAdmant.empty_element
    if not re.search(AdmantRegex.PREFIX, element):
        return CheckAdmant.prefix.replace(Globals.EL, element)
    if re.search(AdmantRegex.RESERVEDCHARACTERS, element[2:]):
        return CheckAdmant.reschar.replace(Globals.EL, element)
    if element.startswith("c:") | element.startswith("f:"):
        if is_old_taxonomy:
            old_taxonomy = Configuration.get_oldtaxonomy()
            if element not in old_taxonomy:
                return CheckAdmant.oldtax
                        .replace(Globals.EL, element)
        else:
            new_taxonomy = Configuration.get_newtaxonomy()
            if element not in new_taxonomy:
                return CheckAdmant.newtax
                        .replace(Globals.EL, element)
    if element.startswith("v:") & (len(element[2:].strip())>2):
```

```
        return CheckAdmant.velement.replace(Globals.EL, element)
    if element.startswith("u:") | element.startswith("r:"):
        try:
            urllib2.unquote(element).decode('utf8')
        except:
            return CheckAdmant.url.replace(Globals.EL, element)
    return "continue"

def admant_warning(admant):
    double_space = re.findall(AdmantRegex.DOUBLE_SPACE, admant)
    if double_space:
        return CheckAdmant.doublespace
                .replace(Globals.BLOCK, str(double_space))
    space_element = re.findall(AdmantRegex.SPACE_EL, admant)
    if space_element:
        return CheckAdmant.spaceelement
                .replace(Globals.BLOCK, str(space_element))
    k_element = re.search(AdmantRegex.K_EL, admant)
    if k_element:
        return CheckAdmant.k_element
               .replace(Globals.EL, unicode(k_element.group(0)))
    return CheckAdmant.no_error
```

I risultati di questo capitolo sono tratti e riadattati da [1].

# Appendice B

# Datatables e Highcharts

L'implementazione del progetto ha richiesto l'utilizzo di molteplici tabelle e grafici per la visualizzazione dei dati di diversi database. Le tabelle sono state create e sviluppate con l'utilizzo del plugin jQuery *Datatables*, mentre i grafici a linea hanno utilizzato lo script di *HighCharts*. Entrambi creano oggetti javascript, con i quali è possibile interagire tramite le variabili in cui sono stati salvati e attraverso funzioni specifiche.

## B.1 Creazione delle tabelle

L'oggetto `Datatable` è creato e salvato in una variabile, `admant_table`, `profile_table`, `client_table` o `server_table`. L'inizializzazione della tabella avviene tramite l'impostazione di diversi parametri interni all'oggetto o successivamente attraverso le API. I parametri interni utilizzati per la creazione delle due tabelle sono:

- `columns` è una lista di mappe, contenenti le opzioni da settare per ogni colonna
  - `title` il titolo della colonna,
  - `data` il campo dei dati caricati da inserire nella colonna,
  - `className` la classe da associare alla colonna, per poterla recuperare in futuro,
  - `width` la larghezza della colonna,
  - `orderable` per rendere la tabella ordinabile o meno rispetto ai dati della colonna,
  - `visible` per rendere la colonna visibile o meno,
  - `defaultContent` il contenuto della cella corrispondente, nel caso in cui sia costante per tutte le righe,
  - `searchable` per rendere i dati della colonna cercabili dalla funzionalità di ricerca.
- `dom` è una stringa che indica l'aggiunta e l'ordinamento degli elementi di interazione con la tabella, quali i bottoni, il campo di ricerca e la paginazione.
- `order` è una array di array contenente una lista di colonne sulle quali effettuare l'ordinamento dei dati, con la relativa direzione (ascendente o discendente).

- `autoWidth` è un booleano per permettere alla tabella di impostare la larghezza delle colonne in automatico, in base al contenuto.
- `select` è una mappa per impostare i parametri di selezione delle righe.
- `lengthMenu` è la lista che indica i possibili numeri di righe mostrare in una singola pagina.
- `destroy` è il booleano che permette di reinizializzare la tabella con diverse opzioni.
- `fixedHeader` è il booleano che, se settato a true, fissa l'header della tabella in cima alla pagina, anche in caso di scorrimento verso il basso.
- `scrollX` e `scrollY` sono due stringhe o booleani che impostano il layout della tabella, aggiungendo una barra di scroll orizzontale e/o verticale con l'altezza indicata nella stringa o una dimensione di default nel caso di booleano.
- `scrollCollapse` è il booleano che riadatta l'altezza della tabella nel caso i dati necessitassero di una dimensione minore di quella indicata in `srollY`.
- `rowGroup` è una mappa che abilita e configura l'estensione relativa di raggruppamento dei dati mostrati. In particolare
  - `dataSrc` è un array che indica i campi per i quali raggruppare i dati, nell'ordine inserito nel caso di raggruppamento a più livelli,
  - `startRender` è la funzione che aggiunge il tooltip al campo di ordinamento.
- `buttons` è una lista di mappe contenti le opzioni di inizializzazione dei bottoni da aggiungere alla tabella:
  - `text` il testo da mostrare come label del bottone,
  - `className` la classe da associare il bottone, per impostarne lo stile,
  - `attr` gli attributi, quali l'identificatore `id`,
  - `action` la funzione da chiamare in conseguenza a un click,
  - `extend`, `fieldSeparator`, `newline` e `filename` per impostare il formato, i caratteri separatori e il nome del file per il download dei dati,
  - `exportOptions` e `modifier` per impostare le colonne e le righe da scaricare nel caso di download,
  - `customize` la funzione da applicare alla stringa impostata come contenuto del file scaricato.
- `ajax` è la chiamata AJAX da effettuare per caricare i dati nella tabella.

Attraverso le API infine si aggiungono altri bottoni, il filtro di ricerca personalizzata e si aggiusta l'allineamento dell'header fissato. In particolare `$.fn.dataTable.Buttons()` prende in input la tabella e la lista di bottoni e li crea e l'aggiunta alla pagina avviene tramite `table.buttons(1, null).container().appendTo()`, mentre 'allineamento dell'header è aggiustato con `table.fixedHeader.adjust()`.

```
table = $('#admants_table').DataTable({
    destroy: true,
    ajax: {
        url: URL_ADMANTS,
        type: "POST",
        data: {"db": db_name},
        beforeSend: function(){
            $("#overlay_loading").show();
        },
        complete: function(){
            $("#overlay_loading").hide();
            $('#admants_heading').html("ADMANTS" + db_name);
        },
        error: function (jqxhr, textStatus, errorMessage) {
            $("#overlay_loading").hide();
            $('#alert_exception').html(EXC_ERROR + "<br>" +
            errorMessage + "<br>" + jqxhr.responseText);
            $('#alert_exception').show();
        }
    },
    columns: [{ title: 'User Name', data: "user",
                 className: "col1", width: "21%"},
              { title: 'Admant Name', data: "name",
                className: "col col2", width: "20%" },
              { title: 'Admant', data: "admant",
                className: "col3", width: "57%",
                orderable: false },
              { title: '', defaultContent:ADMANT_DELETE_BUTTON,
                className: "buttons", width: "4%",
                orderable: false  },
              { data: "added", visible: false,
                searchable: false}],
    dom: '<"#tsearch.row" l <"#search.pullright">>'+
    '<"#tbuttons.row"<"#buttons.pull-left"><"pullright"B>>rtip',
    scrollY: '700px',
```

```
    select: {style: 'multi+shift',
             selector: 'td:not(:last-child)' },
    lengthMenu: [[50, 100, 200, -1],
                 [ '50', '100', '200', 'Show all' ]],
    buttons: [
        {text: 'Add Admant', className: "fa-plus btn-success",
         attr: {id: "addAdmant"}},
        {text: 'Delete Admants',className:"fa-remove btn-danger"
         attr: {id: "deleteAdmants"}},
        {text: 'Download', className:"fa-download btn-warning",
         extend: 'csv', filename: 'Monitor_admants',
         fieldSeparator: '\t', exportOptions: {columns:[0,1,2]}}
    ]
});
/* Add another group of buttons, on the left */
new $.fn.dataTable.Buttons(table, {
    buttons: [
        {text: ' Refresh', className: "fa-refresh btn-info",
         attr: {id: "refreshAdmant"}},
        {text: 'Deselect all', className: "fa btn ",
        action: function() {table.rows().deselect();}}]});
table.buttons(1,null).container().appendTo($('#buttons');
$('#search_admant').append(SEARCH_ADMANT);
```

## B.2 Creazione dei grafici

L'oggetto `HighCharts`, creato attraverso il suo costruttore, non è salvato in una variabile come nel caso delle tabelle, ma direttamente appeso a un *container* aggiunto al file html. Il costruttore prende in input il *container* e le opzioni di inizializzazione, mappe che impostano:

- `chart`: la grandezza del grafico e l'abilitazione e la configurazione dello zoom dei dati.
- `title` e `subtitle`: il testo di titolo e sottotitolo.
- `xAxis` e `yAxis`: il titolo, il formato dei dati inseriti sull'asse e, nel caso di doppio asse, i dati dell'altro con la relativa posizione.
- `tooltip`: i dati mostrati allo scorrimento del mouse su una serie e il loro formato (condiviso o separato),
- `legend`: il layout e il posizionamento della legenda.

- `series`: le serie di dati da mostrare nel grafico. Questo parametro è rappresentato da un lista di mappe, in cui ogni mappa indica una linea che sarà mostrata nel grafico, con il nome della serie e i dati, in formato JSON.

I dati dei grafici a linee utilizzati nel progetto sono tutti caricati tramite una chiamata Ajax alla classe di tornado, che una volta ricevuta la risposta, li applica alle serie presenti nelle opzioni iniziali.

```
Highcharts.chart({
    title: { text: 'Partial counters - ' + detail},
    subtitle: { text: day },
    xAxis: {
        title: { text: "Partial time"},
        categories: data["categories"],
        tickInterval: null
    },
    yAxis: {  title: { text: "Count"}, },
    tooltip: {shared: true},
    chart: {
        height: (8 / 16 * 100) + '%', /* 16:9 ratio */
        zoomType: 'xy'
        },
    legend: {
        layout: 'vertical',
        align: 'right',
        verticalAlign: 'middle'
    },
    series: [
        {
            name: 'Count',
            data: data["count"]
        },
        {
            name: 'Count_ko',
            data: data["count_ko"]
        }
    ]
    };
}
```

I risultati di questo capitolo sono tratti e riadattati da [4] e [6].