



UNIVERSITÀ DEGLI STUDI DI MILANO - BICOCCA
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea in Informatica

Chatbot per il monitoraggio ferroviario

Relatore: Dott.ssa Daniela Micucci

Co-relatore: Dott. Davide Ginelli

Relazione della prova finale di:
Fabrizio Olivadese
Matricola 820864

Anno Accademico 2017-2018

A mio papà Antonello, a mia mamma Cristina, a mio fratello Andrea e a tutta la mia famiglia, fonte di ispirazione grazie alla quale sono la persona che sono e che mi hanno sempre sostenuto per raggiungere questo traguardo.

Ai miei compagni di corso Alex, Cristian, Marco, Michele, Simone G. e Simone V. che ogni giorno hanno condiviso con me gioie, sacrifici e successi; l'affetto e il sostegno che mi hanno dimostrato rendono questo traguardo ancora più prezioso.

Ai miei amici, quelli che ci sono ancora e quelli che ora sono lontani, e a tutte le persone con cui ho condiviso le esperienze più belle che io ricordi e che hanno avuto un peso determinante nella mia vita e in questo risultato; possa il destino non separarmici mai.

Ai miei professori, in particolare ai miei relatori Dott.ssa Daniela Micucci e Dott. Davide Ginelli, per avermi guidato nella stesura di questo lavoro.

Indice

Introduzione	vii
1 Progettazione	1
1.1 Requisiti	1
1.1.1 Requisiti funzionali	1
1.1.2 Requisiti non funzionali	2
1.1.3 Requisiti di dominio	2
1.2 Casi d'Uso	3
1.3 Architettura Software	4
1.3.1 Diagramma delle classi	4
1.3.2 Pacchetto TelegramResponder	5
1.3.3 Pacchetto alertChecker	8
1.3.4 Modulo starter.py	10
1.4 Architettura Dati	11
1.4.1 Tabella trains	12
1.4.2 Tabella users train	14
1.4.3 Tabelle per il monitoraggio delle direttrici	15
2 Infrastruttura e Tecnologie	17
2.1 Implementazione	17
2.1.1 Librerie e Frameworks	17
2.2 Persistenza dati	18
2.3 Deployment Operating System	19
2.4 Deployment Cloud Platform	19
2.5 Analisi delle prestazioni	20
2.6 Documentazione e Controllo Versione	21
3 Fonte Dati e Recupero delle Informazioni	23
3.1 Fonte dei dati	23
3.2 Recupero delle informazioni in tempo reale	25
3.2.1 Implementazione del recupero delle informazioni in tempo reale	30
3.3 Ricerca delle soluzioni di viaggio tra due stazioni	32
3.3.1 Implementazione della ricerca di soluzioni di viaggio	35
4 Monitoraggio delle Direttrici	37
4.1 Fonte dei dati	38
4.2 Implementazione del Monitoraggio Direttrici	39
5 Conclusioni e Sviluppi Futuri	43

6 Manuale Utente	45
6.1 Primo avvio e menù principale	45
6.2 Informazioni in tempo reale di un treno	45
6.3 Ricerca di una soluzione di viaggio	47
6.4 Aggiunta di un treno alla lista	48
6.5 Aggiunta una direttrice alla lista	50
6.6 Riepilogo completo dell'account	51
6.7 Esempi di avvisi automatici	52
Bibliografia	55

Introduzione

L'idea di realizzare un *chatbot* per il *monitoraggio ferroviario* nasce a fine 2016, per necessità personali. Da universitario pendolare, le frequenti problematiche delle reti ferroviarie italiane, mi hanno portato a voler trovare una soluzione semplice e puntuale che mi consentisse di rimanere aggiornato sull'andamento dei treni che predo regolarmente.

In caso di ritardo, soppressione o modifica di un treno un *utente informato in tempo reale* può preventivamente decidere come modificare i suoi piani di viaggio: uno studente può rimanere a studiare senza recarsi in stazione per un treno che non partirà, un lavoratore può avvisare preventivamente del ritardo con cui si recerà sul posto di lavoro oppure un viaggiatore qualsiasi può essere informato della modifica del treno su cui sta viaggiando anche in mancanza di comunicazione. Tutti scenari, da me stesso vissuti, che mi hanno portato a voler costruire un sistema di notifiche automatizzato puntuale e preciso.

Nell'era dell'informazione molte applicazioni ci forniscono utili informazioni per poter vivere la nostra vita in maniera più semplice: traffico, news, meteo, reminder, calendari e applicazioni. Tutte soluzioni molto interessanti che si basano, quasi in tutti i casi, su messaggi di broadcast *non personalizzati per l'utente*. Nel caso del monitoraggio ferroviario, le applicazioni per smartphone già presenti sul mercato (ufficiali e non) hanno il grande difetto di non notificare in tempo reale l'utente su quello che sta accadendo al proprio treno sia prima che durante il viaggio: esse si limitano infatti ad esporre informazioni realtime solo in risposta all'input dell'utente utilizzatore.

La soluzione più immediata che possa venire in mente che soddisfi i requisiti appena citati è una applicazione mobile multi-piattaforma, con integrato un sistema di notifiche intelligenti. La realizzazione di un'app però comporta *diverse complicazioni* sia dal punto di vista tecnico (molteplicità di sistemi operativi, compatibilità, connessione di rete) che dal punto di vista pratico per l'utente (spazio sul dispositivo, consumo di batteria, necessità di imparare ad usare una nuova applicazione).

L'essenza di una applicazione di questo genere è sicuramente la *notifica* tramite la quale l'utente viene avvisato. Sarebbe stato sprecato sviluppare e mantenere una applicazione vera e propria da pubblicare sui convenzionali app-store mobile.

La grande espansione di app di messaggistica avanzate come Telegram mi ha permesso di valutare la possibilità di realizzare un chatbot.

Telegram è un servizio di messaggistica istantanea basato su cloud ed erogato senza fini di lucro. I client ufficiali di Telegram sono distribuiti come software libero per qualsiasi dispositivo e sistema operativo immaginabile (in ambito consumer). Telegram è stato uno dei primissimi applicativi di messaggistica istantanea a portare su larga scala l'utilizzo dei BOT, ovvero di istanze di chat la cui conversazione non avviene con una persona ma bensì con un software che simula, tramite l'invio di normali messaggi di testo, una conversazione umana.

Sviluppare un chatbot per soddisfare i requisiti di cui sopra mi è subito sembrata l'idea migliore. I vantaggi sono molteplici: l'applicazione di Telegram ha una compatibilità vastissima, è già mantenuta dagli sviluppatori, è progettata verso progetti di intelligenza artificiale, implementa già una solida politica di sicurezza ed è già installata nei dispositivi degli utenti che la utilizzano. Questo significa che l'utente può evitare l'installazione di applicazioni esterne sul dispositivo, non spreca spazio su di esso, non ha problemi di sicurezza, non ha consumo di batteria extra (la computazione del chatbot è in cloud) o di malfunzionamenti generali dovuti ad un errato sviluppo dell'app oltre ad essere già in confidenza con l'applicazione stessa.

La chatbot per Telegram, come già anticipato, nasce a Novembre 2016 in una versione primordiale e che soddisfaceva le mie esigenze personali. Non era infatti usufruibile da chiunque ma era programmata solamente per me: era statica e mi inoltrava le informazioni da me volute impostate a livello di codice.

Alcuni miei conoscenti, vedendola, mi hanno chiesto di realizzarne una versione usufruibile, quindi aggiunsi una piccola funzione per personalizzare la chatbot e programmare i propri treni. Senza accorgermene *l'app nel giro di pochissimi mesi contava 500 utenti*, nonostante il sistema fosse davvero molto arrangiato, poco efficiente e pieno di problemi.

Inevitabilmente, qualche mese più tardi, il sistema è definitivamente andato in sovraccarico ed è rimasto inattivo fino al 2018, anno in cui ho realizzato, in occasione dello stage universitario, una chatbot per Telegram *stabile e totalmente rinnovata* che in questa relazione vi descrivo nei dettagli.

Riassunto dei Capitoli:

1. **Progettazione:** Requisiti funzionali, non funzionali, di dominio e casi d'uso. Il diagramma delle classi, analizzato per pacchetti e come esso interagisce con il database, analizzato per tabelle.
2. **Infrastruttura e Tecnologie:** Come e perché sono state scelte le tecnologie e l'infrastruttura per ospitare il chatbot. Linguaggio di programmazione, librerie, framework, database relazionale, sistema operativo e servizio di cloud computing.
3. **Fonte Dati e Recupero Informazioni:** Da dove provengono e come sono recuperate le informazioni per i treni in real time e per la ricerca di una soluzione di viaggio. Come funzionano le API del servizio fornitore e come vengono implementate dalla chatbot.
4. **Monitoraggio sulle direttive:** Come è stato implementato il sistema di avvisi delle direttive tramite parsing HTML.
5. **Conclusioni e Sviluppi Futuri**
6. **Manuale Utente**

Capitolo 1

Progettazione

1.1 Requisiti

Fondamentale per la buona riuscita di un progetto è stabilire i *requisiti* e i *casi d'uso* da rispettare per poter gestire al meglio tempo e risorse. Questa fase è stata la prima che ho affrontato, stabilendo quali requisiti minimi andassero rispettati e i casi d'uso da realizzare.

1.1.1 Requisiti funzionali

I requisiti funzionali sono elenchi di funzionalità o servizi che il sistema deve fornire. Essi descrivono il comportamento del sistema a fronte di particolari input e come esso dovrebbe reagire in determinate situazioni.

I requisiti funzionali sono espressi nella forma: *Requisito /Complessità, Progresso/*

Complessità può assumere un valore compreso tra 1 e 4

1. Consentire all'utente di visualizzare informazioni real time su uno specifico treno */2, completato/*
2. Consentire all'utente di rimanere aggiornato durante il viaggio del suo treno
 - durante il viaggio */3, completato/*
 - prima del viaggio */3, completato/*
 - specificando in quali giorni della settimana */2, completato/*
3. Consentire all'utente di rimanere aggiornato sugli avvisi di broadcast relativi alle direttive */4, completato solo per Trenord/*
4. Cercare le soluzioni di viaggio
 - specificando data e ora */3, completato/*
 - senza specificare data e ora */3, completato/*
5. Stilare una classifica dei treni più affidabili */4, mancante/*
6. Dedurre, a partire dalle informazioni rilevate, possibili disagi imminenti su una linea e sui relativi treni non ancora partiti */4, parziale/*

7. Consentire all'amministratore di

- bloccare un utente *[1, Parziale, rimpiazzato successivamente da un sistema di autoban]*
- visualizzare le statistiche del sistema *[2, completato]*
- inviare messaggi di broadcast a tutti gli utenti registrati *[3, completato]*
- forzare l'aggiornamento del database *[3, completato]*

L'unico requisito funzionale *non completato* è il numero 5: non mi è stato possibile soddisfare questo requisito poiché, per il momento, il sistema su cui il chatbot è eseguito (vedi capitolo 2.4) *non è abbastanza performante* per poter stilare una classifica sulla totalità dei treni circolanti (oltre 10.000 al giorno)

1.1.2 Requisiti non funzionali

In questa sezione vengono elencati i requisiti non funzionali.

I requisiti non funzionali sono tutte quelle caratteristiche del software non richieste dall'utilizzatore, ma che influenzano il lavoro di sviluppo, poiché non descrivono cosa, ma come il sistema fa ad eseguire certi compiti.

Performance

- Dare una risposta a qualsiasi tipo di richiesta entro 4 secondi
- Rilevare le variazioni dei treni/direttrici dopo al massimo 2 minuti dalla comparsa

Capacità

- Il sistema deve poter gestire agilmente 500 utenti e 3000 richieste al giorno

Disponibilità

- Il sistema deve essere disponibile 24h ore al giorno, 7 giorni a settimana a patto che le API esterne siano online

Affidabilità

- Il sistema deve gestire errori derivanti dalle API esterne
- Il sistema deve riavviarsi automaticamente in caso di crash

Integrità dei dati

- L'integrità dei dati deve essere garantita dall'utilizzo di database appropriati

1.1.3 Requisiti di dominio

- Minimizzare il numero di richieste esterne implementando un sistema di cache senza far perdere valore alle informazioni fornite (cache di massimo 2 minuti)

1.2 Casi d'Uso

La figura 1.1 mostra il diagramma dei casi d'uso

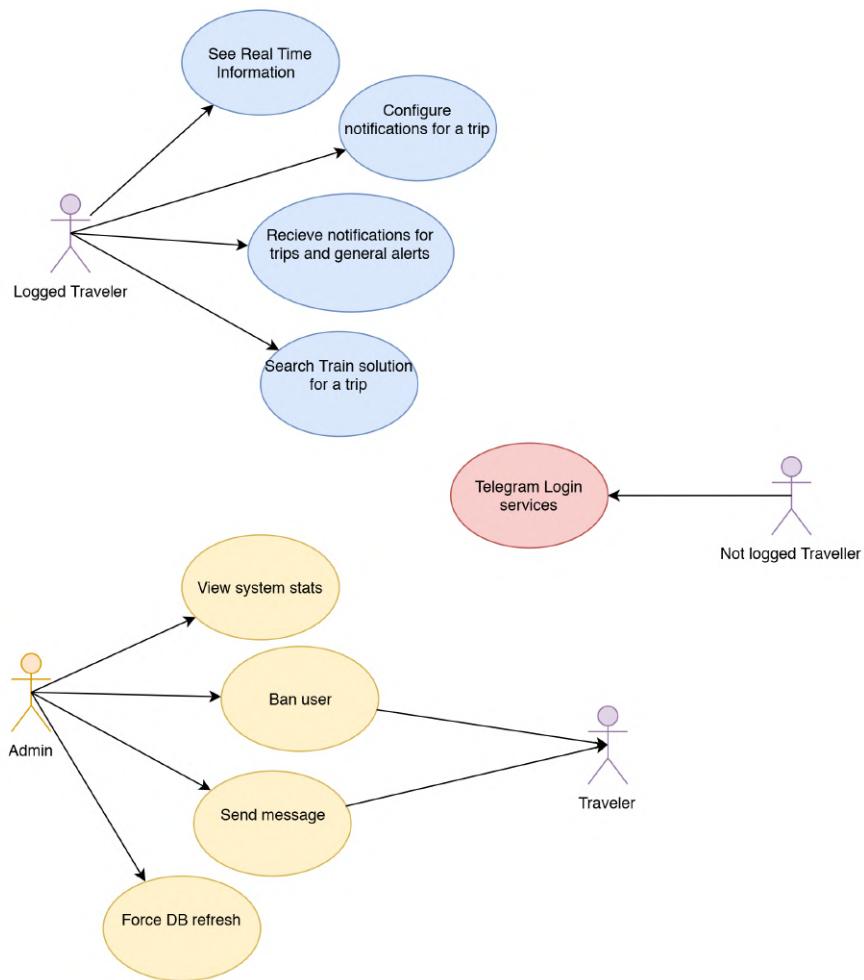


Figura 1.1: Diagramma dei casi d'uso

1.3 Architettura Software

In questa sezione viene analizzata l'architettura software, divisa in due parti principali:

- I pacchetti e le classi Python
- Il database Relazionale mariaDB

In questa sezione vengono analizzate entrambe le architetture e le motivazioni che hanno portato alla realizzazione di questi schemi.

1.3.1 Diagramma delle classi

I pacchetti e le classi verranno analizzate in questo capitolo tramite il diagramma delle classi che le rappresenta.

La figura 1.2 mostra il diagramma delle classi completo. Nelle sottosezioni seguenti il diagramma viene analizzato per pacchetti, in modo da rendere più agevole la comprensione dello stesso.

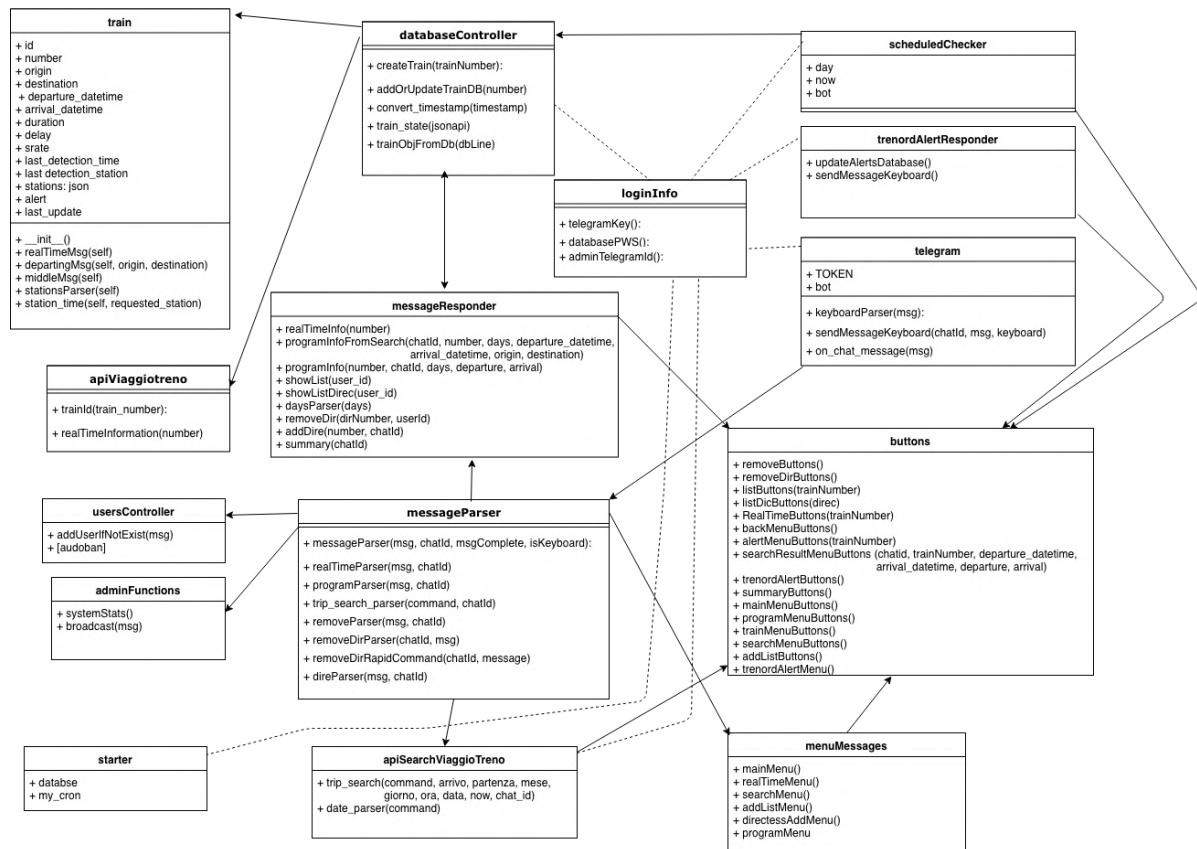


Figura 1.2: Diagramma completo delle classi utilizzate per la realizzazione del chatbot

1.3.2 Pacchetto TelegramResponder

La figura 1.3 mostra il diagramma delle classi del pacchetto telegramResponder con solo le classi di maggior interesse ai fini di questa relazione.

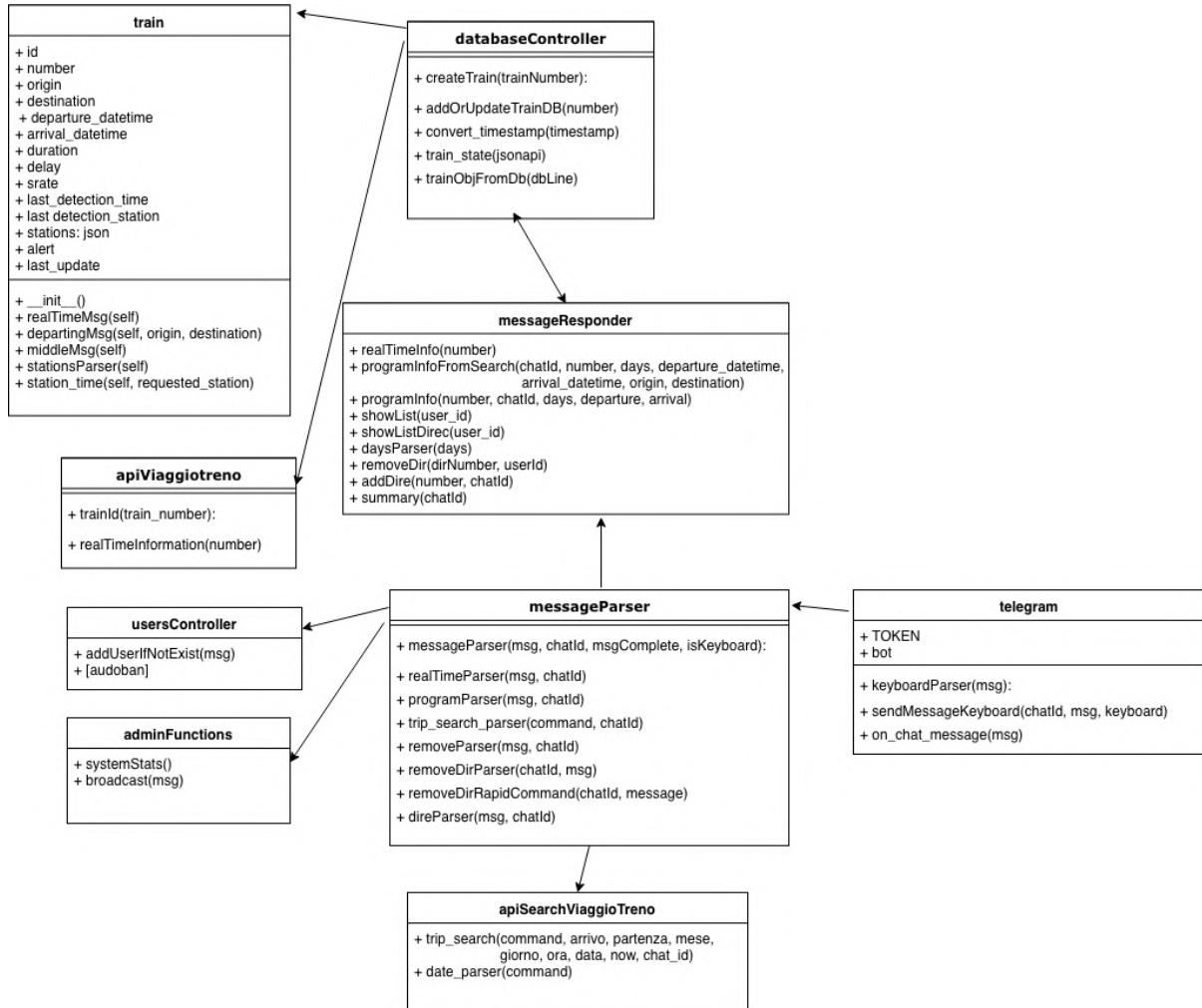


Figura 1.3: Principali classi del pacchetto Telegram Responder

- **telegram** contiene il framework *telepot* che serve per l’interfacciamento con il BOT di Telegram. In particolare si occupa della *ricezione e invio* dei messaggi testuali del BOT. Quando un utente invia al bot un messaggio, la classe *onChatMessage*, dopo aver controllato che il messaggio sia solo testuale, inoltra il testo del messaggio e l’ID dell’utente che l’ha inviato a *messageParser*.
- **messageParser**, ricevuto il messaggio testuale esegue inizialmente un controllo dell’utente tramite la classe **usersController** per applicare eventuali azioni di ban nel caso in cui l’utente abbia superato il limite massimo di richieste giornaliere. La classe *usersController* si interfaccia direttamente con il database, in particolare con la tabella *users*. Se l’utente è autorizzato ad effettuare la richiesta il metodo *messageParser* rileva le *parole chiave* contenute nel messaggio e identifica a quale tipo di richiesta l’utente sta facendo riferimento, passando al metodo competente il messaggio testuale e l’ID dell’utente stesso. Il metodo

competente, se individuato, scompongo il messaggio testuale individuando le parti che compongono il comando

(es. "Ricerca da Milano a Roma alle 20:30 il 10-3" viene scomposto in partenza, arrivo, orario, data)

- Il messaggio così scomposto viene passato alla classe **messageResponder** che si occupa di costruire il messaggio di risposta per l'utente a partire dai parametri in input su uno specifico metodo (individuato da messageParser). Tutte le operazioni che vengono eseguite da *messageResponder* fanno uso del Database e, di conseguenza, della classe **databaseController**.
- **databaseController** è la classe che si occupa di rispondere a *messageResponder* con le informazioni che esso richiede, in base al comando ricevuto. **databaseController** è fondamentale per rispettare i requisiti non funzionali precedentemente individuati poiché, implementando il meccanismo di cache, è la classe da cui dipendono le performance del sistema.

databaseController è la classe tramite la quale *tutte* le richieste devono passare. **databaseController** si occupa di rispondere con l'oggetto **treno** a **messageResponder**.

Ogni richiesta a **databaseController** genera un oggetto **treno** che viene restituito alla classe chiamante.

Se il treno richiesto dal chiamante non è presente nel database locale, **databaseController** tramite la classe **apiViaggiotreno** effettua una richiesta esterna e aggiunge l'oggetto **treno** al database.

Se il treno richiesto alla classe **databaseController** è già presente nel database locale ed è stato aggiornato entro due minuti dal momento della richiesta l'oggetto che viene restituito da **databaseController** è generato a partire dalle informazioni localmente già presenti: in questo modo si evita di effettuare una richiesta esterna quando ho già un'informazione in locale abbastanza recente.

Se l'informazione è presente ma non è aggiornata, **databaseController** interella comunque la classe **apiViaggiotreno** e aggiorna la riga corrispondente del database. Dal quel momento, per i successivi due minuti, qualsiasi richiesta relativa a quel treno verrà prelevata localmente.

Le figure 1.4 e 1.5 evidenziano le differenze tra una richiesta soddisfatta tramite il sistema di cache e una richiesta esterna.

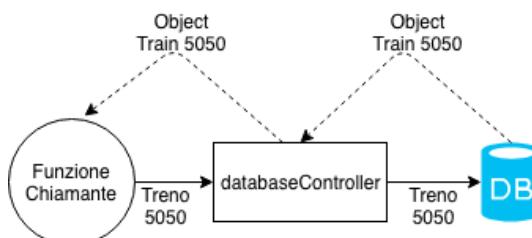


Figura 1.4: Richiesta di un treno già presente localmente e aggiornato

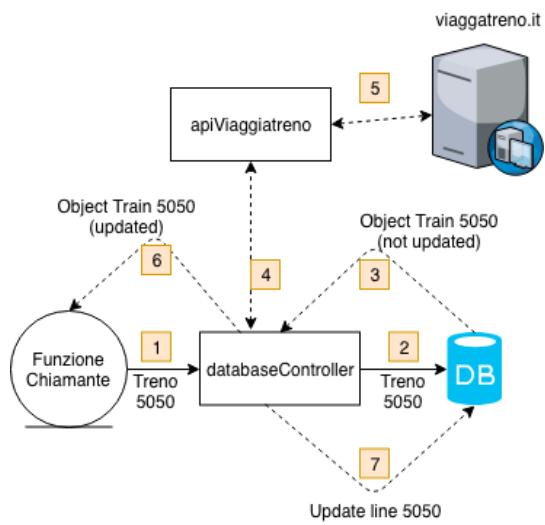


Figura 1.5: Richiesta di un treno non presente localmente o non aggiornato

1.3.3 Pacchetto alertChecker

La figura 1.6 mostra il diagramma delle principali classi del pacchetto alertChecker, che si occupa dei servizi di allerta automatica per treni o direttive monitorate dall'utente.

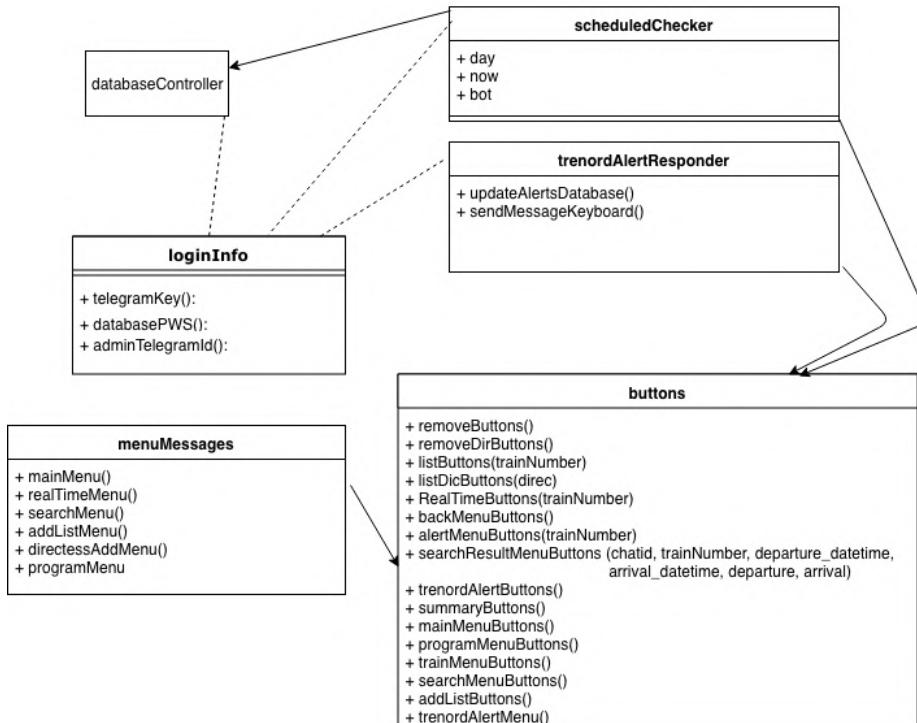


Figura 1.6: Richiesta di un treno già presente localmente e aggiornato

- **loginInfo** è una classe condivisa anche con il pacchetto TelegramResponder che contiene e gestisce l'accesso alle informazioni sensibili per il login al database, al bot di Telegram e all'ID dell'amministratore.
- **menuMessage** e **buttons** sono classi condivise anche con il pacchetto precedente e contengono le informazioni ausiliarie da applicare ai messaggi come le inline keyboard (pulsanti di risposta ai messaggi) e il testo dei messaggi di menù. Queste informazioni vanno applicate a tutti i messaggi di risposta e di conseguenza sono contenuti in queste due classi in modo che possano essere prelevati con facilità evitando duplicazioni di codice.

Le classi più importanti di questo pacchetto sono **scheduledChecker** e **trenordAlertResponder**

- **scheduledChecker** è una classe che viene *richiamata ogni due minuti* dal crontab di Debian 9, e ha accesso diretto al database contenente tutti i treni che ogni utente ha aggiunto alla propria lista, ovvero i treni su cui gli utenti vogliono essere allertati.

Essa, ogni due minuti, scansiona il database con i treni di tutti gli utenti ed esegue una logica di funzionamento così strutturata, basata sull'orario di partenza del treno stesso:

- *30 minuti prima* della partenza del treno, la classe, ogni volta che viene richiamata, inizia a monitorare eventuali disagi legati al treno stesso. Sopravvenimenti parziali/completi o ritardi superiori ai 10 minuti verranno segnalati all'utente non appena verranno rilevati.

- 5 minuti prima della partenza del treno la classe invia un messaggio all’utente segnalando che il suo treno è in partenza, indicando il binario, l’eventuale ritardo e altre informazioni utili.
- a metà viaggio (calcolato in base all’orario previsto di arrivo) la classe invia all’utente un messaggio riepilogativo sull’andamento del treno
- durante tutto il viaggio, in caso di gravi ritardi, soppressioni complete o parziali oppure avvisi da parte delle compagnie ferroviarie la classe allerta l’utente

La classe scheduledChecker fa uso intenso della classe databaseController descritta nel pacchetto TelegramResponder, in modo da poter sfruttare il sistema di cache delle richieste.

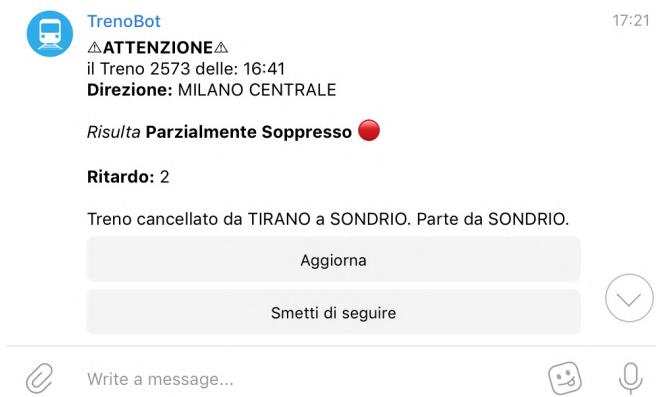


Figura 1.7: Esempio di allerta automatica di un treno monitorato dall’utente

- **trenordAlertResponder**, similmente alla classe precedente, viene richiamata dal crontab di Debian 9 ogni otto minuti e ha accesso diretto al database contenente tutti gli avvisi delle direttrici Trenord su cui gli utenti vogliono essere allertati.

Essa, ogni otto minuti, esegue un parsing HTML della pagina Trenord "direttrici in tempo reale"^[1] e rileva eventuali criticità presenti su una specifica direttrice. Se viene rilevata una criticità l’utente viene subito avvisato.

La funzione di trenordAlertResponder, come suggerisce il nome stesso, è disponibile solo per le direttrici *Trenord* poiché non esiste un servizio simile sul sito di Trenitalia.

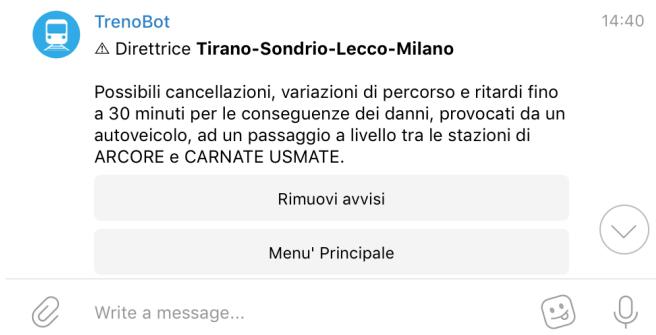


Figura 1.8: Esempio di allerta su una direttrice monitorata dall’utente

Entrambe le classi appena descritte, per non avvisare l'utente ogni volta che vengono richiamate, confrontano il messaggio da inviare con l'ultimo già inviato: in questo modo l'utente viene avvisato solo in caso di cambiamenti della situazione.

La fase di aggiunta da parte degli utenti dei treni e delle direttrici che vogliono monitorare *non è gestita da queste classi* ma dalle classi del pacchetto TelegramResponder, che interpreta le richieste via messaggio.

1.3.4 Modulo starter.py

Il modulo starter.py prepara il sistema alla prima esecuzione della chatbot e deve essere eseguito solamente su un sistema vergine, una volta sola.

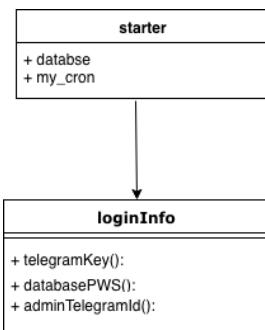


Figura 1.9: Classe starter

starter fa uso solamente della classe **loginInfo**, condivisa da tutti i pacchetti, per poter accedere alle credenziali del database.

In un sistema vergine, con già installato mariaDB, python e le librerie/framework citati nel capitolo 2.1.1 il modulo **starter.py** prepara il sistema all'esecuzione del chatbot, in particolare:

- Crea il database "TRENOBOT" all'interno di mariaDB
- Crea le tabelle necessarie all'interno del database (capitolo 1.4)
- Crea le regole di crontab necessarie a:
 - Avviare il chatbot all'avvio del sistema
 - Richiamare la classe `scheduledChecker` ogni due minuti
 - Richiamare la classe `trenordAlertResponder` ogni otto minuti

Eseguendo da terminale `starter.py` si prepara quindi il sistema ad accogliere ed eseguire l'intero applicativo (il cui modulo di avvio è rappresentato dalla classe `telegram.py`). Questa classe è di grande utilità in caso di trasferimenti di macchina e per gli utenti che, tramite GitHub, vogliono provare ad eseguire il chatbot sul proprio sistema senza dover creare manualmente l'infrastruttura.

1.4 Architettura Dati

In questa sezione viene illustrata l'architettura dati della chatbot. Il database viene così composto dal modulo starter.py (capitolo 1.9)

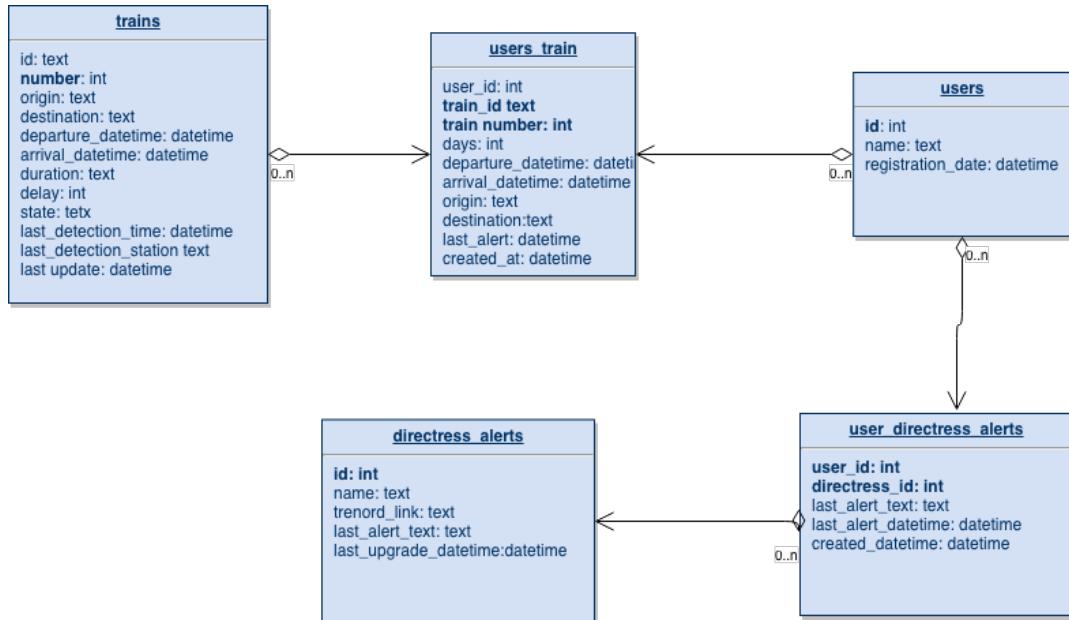


Figura 1.10: Schema del Database

1.4.1 Tabella trains

La tabella trains in figura 1.11 contiene tutti i treni che sono stati richiesti nel tempo, ed è quella che viene sfruttata per il sistema di cache.

SELECT id, number, origin, destination, departure_datetime, arrival_datetime, duration, delay, state, last_detection_time, last_detection_station, alert, last_update FROM train;														
id number origin destination departure_datetime arrival_datetime duration delay state last_detection_time last_detection_station alert last_update														
501529 5038 BERGAMO LECCO 2018-07-25 10:08:00 2018-07-25 10:48:00 0:40 0 Regolare 2018-07-25 10:48:00 LECCO 2018-07-25 11:12:33														
501520 5039 LECCO BERGAMO 2018-07-25 10:12:00 2018-07-25 10:52:00 0:40 25 Regolare 2018-07-25 11:13:30 Bergamo Ospedale 2018-07-25 11:13:02														
501529 5040 BERGAMO LECCO 2018-07-25 11:08:00 2018-07-25 11:48:00 0:40 0 Regolare - -- 2018-07-25 11:11:24														
501520 5049 LECCO BERGAMO 2018-07-24 15:12:00 2018-07-24 15:52:00 0:40 0 Regolare - -- 2018-07-24 14:49:09														
501520 5051 LECCO BERGAMO 2018-07-20 16:12:00 2018-07-20 16:52:00 0:40 0 Regolare - -- 2018-07-20 15:59:46														
501700 9619 MILANO CENTRALE NAPOLI CENTRALE 2018-07-25 10:00:00 2018-07-25 14:15:00 4:15 -2 Regolare 2018-07-25 11:11:00 PDE MONTE BIBELE 2018-07-25 11:11:30														

Figura 1.11: Tabella trains

E' identificata dai campi **id** e **number** che identificano il treno (il numero, tra le diverse compagnie ferroviarie italiane, non è univoco quindi va associato all'ID della stazione di riferimento per creare una chiave).

Altri campi significativi sono **last detection time** e **last detection station** che indicano quando e dove il treno è stato rilevato l'ultima volta.

Il campo **state** indica lo stato di un treno, che può essere:

- Regolare
- Soppresso
- Parzialmente Soppresso

In caso di treno Soppresso o Parzialmente soppresso, nel campo **alert** viene indicato il motivo, come in figura 1.12

1:23 1B10 0 Regolare 0 Cosimo 2018-08-07 09:51:00 PIACENZA Treno cancellato da TREVIGLIO a MILANO CERTOSA. Parte da MILANO CERTOSA.	1:11 5048 0 Regolare 0													
1:15 Vuoi 0 Parzialmente Soppresso :red_circles: 0	1:57 5049 0 Regolare 0													
0:57 work 0 Parzialmente Soppresso :red_circles: 0	3:44 5050 0 Regolare 0													
3:44 2 Regolare 0	6:55 5051 0 Regolare 0													
6:55 1 Regolare 0	19:38 5052 0 Regolare 0													
3:46 41 Regolare 0	19:38 5053 114 Regolare 0													
19:38 114 Regolare 0	19:38 5054 114 Regolare 0													

Figura 1.12: Esempio di treno parzialmente soppresso con utilizzo del campo alert

Il campo **last update** indica quando la suddetta riga è stata aggiornata l'ultima volta. Questo è il campo che discrimina l'utilizzo della cache al posto della richiesta esterna, come descritto nel capitolo 1.3.2 .

Ai fini di una rappresentazione grafica leggibile, in queste immagini è volutamente omesso il campo **stops** in formato JSON, che contiene, *per ogni fermata di un treno* un file strutturato come in figura 1.14 . All'interno del suddetto campo si trovano diverse informazioni utili ai fini del bot, per avvisare l'utente riguardo a modifiche, binari, stazioni e molto altro. Molti campi sono *null* se non hanno rilevanza per quel treno, o se il treno è già partito o già arrivato a destinazione. La formattazione del capo 'stops' dipende dalle API viaggiautreno descritte nel capitolo 3.

```

"fermate": [
    {
        "orientamento": null,
        "kcNumTreno": null,
        "stazione": "BERGAMO",
        "id": "S01529",
        "listaCorrispondenze": null,
        "programmata": 1531818480000,
        "programmataZero": null,
        "effettiva": null,
        "ritardo": 0,
        "partenzaTeoricaZero": null,
        "arrivoTeoricoZero": null,
        "partenza_teorica": 1531818480000,
        "arrivo_teorico": null,
        "isNextChanged": false,
        "partenzaReale": null,
        "arrivoReale": null,
        "ritardoPartenza": 0,
        "ritardoArrivo": 0,
        "progressivo": 1,
        "binarioEffettivoArrivoCodice": null,
        "binarioEffettivoArrivoTipo": null,
        "binarioEffettivoArrivoDescrizione": null,
        "binarioProgrammatoArrivoCodice": null,
        "binarioProgrammatoArrivoDescrizione": null,
        "binarioEffettivoPartenzaCodice": null,
        "binarioEffettivoPartenzaTipo": null,
        "binarioEffettivoPartenzaDescrizione": null,
        "binarioProgrammatoPartenzaCodice": null,
        "binarioProgrammatoPartenzaDescrizione": "1 Tronco OVEST",
        "tipoFermata": "P",
        "visualizzaPrevista": true,
        "nextChanged": false,
        "nextTrattaType": 2,
        "actualFermataType": 0
    }
]

```

Figura 1.13: Campo 'stops' della tabella trains



Figura 1.14: Esempio di utilizzo della tabella per una richiesta Real Time

1.4.2 Tabella users train

Ogni utente ha associata una lista, che esso può modificare, contenente i treni che esso desidera monitorare (classe scheduledChecker, capitolo 1.3.3). La tabella user train in figura 1.15, identificata da user id, trainId e trainNumber contiene ogni treno che va monitorato e il rispettivo utente che lo ha aggiunto alla propria lista.

MariaDB [TRENOBOT]> SELECT * FROM user_train;										
user_id	train_id	train_number	days	departure_datetime	arrival_datetime	origin	destination	last_alert	created_at	
4260245	S01003	51	1 12345	2018-08-07 09:17:00	2018-08-07 10:37:00	DOMODOSSOLA	MILANO CENTRALE	NULL	2018-08-07 11:03:36	
4260245	S01520	5055	1 34	2018-08-07 18:12:00	2018-08-07 18:52:00	LECCO	BERGAMO	NULL	2018-08-07 10:41:43	
4260245	S01529	5042	1 2 M	2018-08-07 12:08:00	2018-08-07 12:48:00	BERGAMO	LECCO	NULL	2018-08-07 10:51:13	
4260245	S01529	5050	1 12345	2018-08-19 16:08:00	2018-08-19 16:48:00	BERGAMO	LECCO	NULL	2018-08-19 10:57:42	
4260245	S01529	5052	1 12345	2018-08-20 17:08:00	2018-08-20 17:48:00	BERGAMO	LECCO	NULL	2018-08-20 12:18:11	

Figura 1.15: Tabella user train

I treni hanno associato un numero che li identifica univocamente a livello giornaliero. Il giorno successivo, lo stesso treno con lo stesso numero, percorrerà la stessa tratta al medesimo orario.

Il capo **days** contiene i giorni della settimana in cui l'utente deve essere avvisato (1=lunedì, 2=martedì, ... 7=domenica). Il campo di default è impostato a 12345 se non diversamente specificato.

Il campo **last alert** contiene l'ultimo avviso inviato all'utente riguardo un determinato treno: viene usato come confronto con messaggi successivi per determinare se avvisare nuovamente l'utente in caso di cambiamenti. Il campo è *NULL* quando il treno è regolare.

La tabella è in relazione con la tabella *trains*, tramite le chiavi esterne **train id** e **train number** e di conseguenza sfrutta indirettamente il meccanismo di cache legato a quest'ultima.



Figura 1.16: Esempio di aggiunta di un treno alla lista da parte di un utente

1.4.3 Tabelle per il monitoraggio delle direttrici

Le tabelle **directress alerts** e **user directress alerts** sono utilizzate dalla classe trenordAlertResponder (capitolo 1.3.3).

Ogni utente, oltre alla lista treni, ha a disposizione una lista di intere direttrici che esso può monitorare.

```

Novara-Milano-Treviglio http://www.trenord.it/IT/servizi/direttrici/D001/novara---milano---treviglio.aspx
Saronno-Seregno-Milano-Albairate http://www.trenord.it/IT/servizi/direttrici/D038/saronno---seregno---milano---albairate.aspx
Domodossola-Gallarate-Milano http://www.trenord.it/IT/servizi/direttrici/D002/domodossola---gallarate---milano.aspx
P.toCeresio-Varese-Gallarate-Milano http://www.trenord.it/IT/servizi/direttrici/D003/porto-ceresio---varese---gallarate---milano.aspx
Luino-Gallarate-Malpensa http://www.trenord.it/IT/servizi/direttrici/D004/luino---gallarate---malpensa.aspx
Chiasso-Como-Monza-Milano http://www.trenord.it/IT/servizi/direttrici/D005/chiasso---como---monza---milano.aspx
Tirano-Sondrio-Lecco-Milano http://www.trenord.it/IT/servizi/direttrici/D006/tirano---sondrio---lecco---milano.aspx
Lecco-Molteno-Monza-Milano http://www.trenord.it/IT/servizi/direttrici/D007/lecco---molteno---monza---milano.aspx
Chiavenna-Colico http://www.trenord.it/IT/servizi/direttrici/D008/chiavenna---colico.aspx
Lecco-Molteno-Como http://www.trenord.it/IT/servizi/direttrici/D009/lecco---molteno---como.aspx
Lecco-Bergamo-Brescia http://www.trenord.it/IT/servizi/direttrici/D010/lecco---bergamo---brescia.aspx
Bergamo-Carnate-Milano http://www.trenord.it/IT/servizi/direttrici/D011/bergamo---carnate---milano-.aspx
Seregno-Carnate http://www.trenord.it/IT/servizi/direttrici/D039/seregno---carnate.aspx
Bergamo-Treviglio http://www.trenord.it/IT/servizi/direttrici/D040/bergamo---treviglio.aspx
Bergamo-Pioltello-Milano http://www.trenord.it/IT/servizi/direttrici/D012/bergamo---pioltello---milano.aspx

```

Figura 1.17: Alcune delle direttrici disponibili, con il relativo url per il parsing HTML

In particolare la tabella **directress alerts** contiene tutte le direttrici Trenord disponibili, con l'ultimo avviso prelevato (ogni 8 minuti) dal sito dell'omonima azienda. Contiene inoltre i link per il parsing HTML, il nome della direttrice e la data/ora dell'ultimo avviso rilevato.

La tabella **user directress alerts**, similmente a *users train*, contiene invece la lista di ogni singolo utente, quindi le direttrici che ogni utente sta monitorando. Il campo **last alert** di 'user directress alerts' viene confrontato con il campo **last alert** di 'directress alerts' e l'utente viene avvisato solo se i due campi non coincidono (ovvero se c'è stato un aggiornamento dell'avviso).

Ne viene omessa l'illustrazione poiché, il campo contenete gli avvisi, è troppo lungo per essere visualizzato correttamente come tabella.

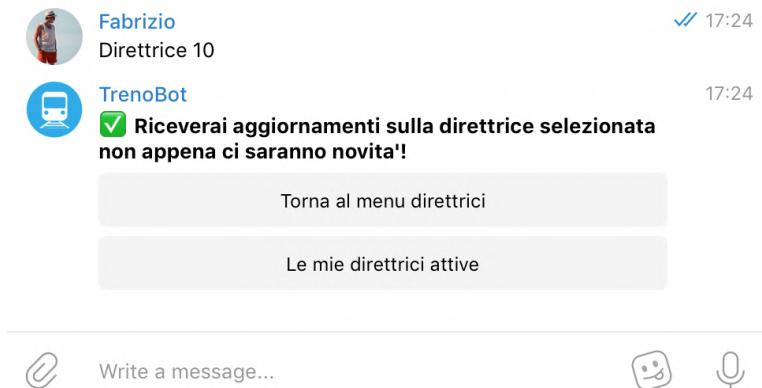


Figura 1.18: Esempio di aggiunta di una direttrice alla lista da parte di un utente

Capitolo 2

Infrastruttura e Tecnologie

2.1 Implementazione

La prima decisa non appena finita la fase di progettazione è stata il linguaggio da utilizzare. In seguito ad una attenta analisi la scelta è ricaduta su Python 2.7 grazie alle librerie disponibili per gestire tutte le esigenze individuate in fase di progettazione.

Python è un linguaggio ad alto livello orientato agli oggetti, moderno, dalla sintassi semplice e potente. Gli ambiti di applicazione di questo linguaggio di programmazione sono svariati, e mi è sempre interessato poterlo imparare. L'adeguatezza alle esigenze individuate e l'opportunità di poter imparare un linguaggio così interessante sono stati fattori determinanti per scegliere di utilizzarlo fin dalla prima versione del chatbot.

2.1.1 Librerie e Frameworks

Elenco delle principali librerie esterne utilizzate

- **Telepot**^[2] è il più famoso framework Python per l'integrazione con le API dei Telegram BOT. Il framework è open source, disponibile su GitHub e sviluppato da Nick Lee. Il funzionamento è molto intuitivo e l'unico parametro da configurare è il token identificativo del bot, rilasciato da Telegram durante la creazione dello stesso tramite BotFather (un bot di Telegram tramite il quale, chiunque, può registrare il proprio bot sulla omonima piattaforma)
- **MySQLdb**^[3] Python permette agli sviluppatori di interfacciarsi con MySQL in maniera relativamente semplice: il team di sviluppo di Python mette a disposizione di tutti una linea guida standard per l'interfacciamento con i database, chiamata DB-API, di cui fa parte MySQLdb per i database basati su Mysql (copreso il fork mariaDB).
- **urllib**^[4] Modulo Python che fornisce una semplice interfaccia per accedere a risorse di rete tramite protocolli gopher, ftp e http.
- **emoji**^[5] Libreria Python, facente parte del pacchetto emoji, che associa degli alias ai codici delle emoji definiti dall'unicode consortium <http://www.unicode.org/emoji/charts/full-emoji-list.html>

- **json**^[6] Libreria Python, che serve a deserializzare file JSON per renderli compatibili con i formati dati di Python secondo la tabella di conversione in figura 2.1

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Figura 2.1: Tabella di conversione per la deserializzazione di file JSON

2.2 Persistenza dati

La persistenza dei dati, soprattutto in un contesto così variegato e imprevedibile, è fondamentale per assicurare al progetto la necessaria stabilità. Inizialmente si è valutata la possibilità di utilizzare **MySQL** essendo un database relazionale già studiato e approfondito ma che possiede un grosso limite: supporta JSON come formato dati di un attributo *soltanto dalla versione 5.7.8*.

La compatibilità con JSON, visto il formato delle risposte delle API di Trenitalia e di Telegram, è necessaria ma purtroppo la versione MySQL 5.7.8 (o addirittura la beta della versione 8.0) nel momento in cui scrivo è incompatibile con Debian 9 o Ubuntu 18.04 (gli unici sistemi operativi utilizzabili sulla piattaforma descritta nella sezione 2.4) nonostante svariati tentativi di procedere comunque all'installazione.

Ho dovuto quindi cambiare i piani iniziali, orientandomi verso **mariadb**^[7].

MariaDB è nato nel 2009 come fork di MySQL. L'ideatore e responsabile del progetto è Michael Widenius, programmatore svedese pioniere e fondatore di MySQL. Quando, all'inizio del 2008, la Sun Microsystems ha acquistato MySQL AB, Widenius si è trovato in conflitto con la nuova società a partire da MySQL 5.1 decidendo quindi di creare l'ormai famoso fork mariadb.

I vantaggi di mariadb, ai fini del chatbot, sono molteplici:

- Stessa sintassi di MySQL
- Stesso interfacciamento con Python tramite MySQLdb
- Compatibilità con attributi JSON a partire dalla versione 10.2
- Compatibilità delle ultime versioni con Debian 9 e Ubuntu 18.04

2.3 Deployment Operating System

Inizialmente, nei primi test, il sistema operativo su cui il chatbot era installato fu Raspian, una versione modificata di Debian 9 ottimizzata per Raspberry Pi. Successivamente, il chatbot è stato spostato sul servizio di cloud computing Amazon Web Services (AWS) con una istanza di Ubuntu 18.04 .

Il servizio di cloud computing attualmente utilizzato è Google Cloud Platform, con una istanza di Debian 9.

Debian è un sistema operativo multi-architettura, composto interamente da software libero anche se tramite appositi repository non-free è possibile includere nativamente software proprietario gratuito o non libero. Debian GNU/Linux utilizza al suo interno programmi di utilità provenienti dal sistema operativo GNU, utilizzando Linux come kernel.

E' un sistema molto stabile e leggero, adatto ad istanze molto piccole di cloud computing e compatibile con tutto il software citato precedentemente.

2.4 Deployment Cloud Platform

Come descritto nel capitolo 2.3, inizialmente la prima piattaforma di cloud computing utilizzata per il chatbot è stata Amazon Web Services, una delle più famose. Durante le settimane in cui questo è stato il servizio utilizzato l'esperienza è stata molto positiva: è una piattaforma molto completa ma economicamente (per uno studente) insostenibile anche con le istanze più piccole disponibili.

Cercando delle valide alternative ho provato **Google Cloud Platform** che fornisce un'istanza base completamente gratuita (730 ore al mese) composta da una singola *vCPU e 600Mb di RAM*, localizzata (obbligatoriamente) in America centrale. Nonostante questo limite geografico il ping risulta molto limitato e non limita eccessivamente le prestazioni di risposta.

Ovviamente l'istanza gratuita (quella su cui attualmente è installato il chatbot) è molto limitata come performance. Questa limitazione mi ha ulteriormente motivato a cercare di sviluppare il tutto nel modo più *ottimizzato* possibile rimanendo nei limiti della macchina.

2.5 Analisi delle prestazioni

Grazie agli strumenti messi a disposizione da *Google Cloud Platform* è stato possibile monitorare accuratamente le prestazioni della macchina virtuale. Con questi strumenti si è potuto verificare che non ci fossero mai picchi di utilizzo di rete/cpu/disco e rilevare eventuali anomalie di programmazione.

A bot completamente operativo, questi sono i valori di utilizzo rilevati negli ultimi 30 giorni:

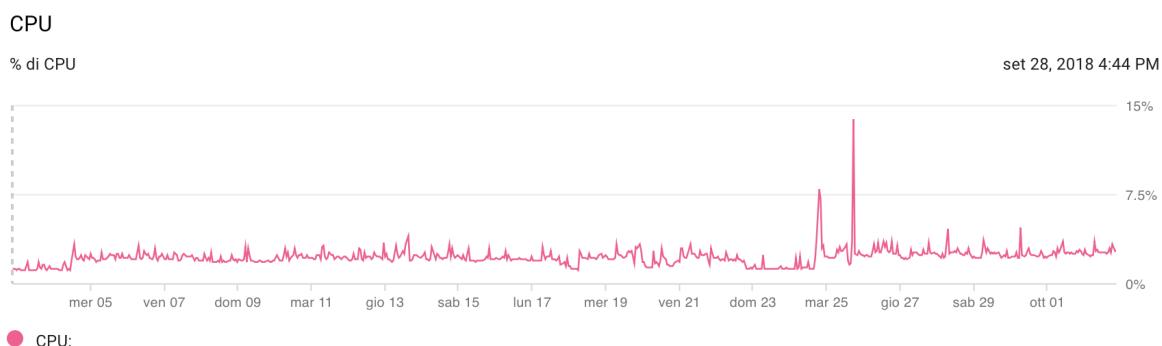


Figura 2.2: Utilizzo della singola vCPU dal 3 Settembre al 3 Ottobre 2018



Figura 2.3: Byte di rete dal 3 Settembre al 3 Ottobre 2018

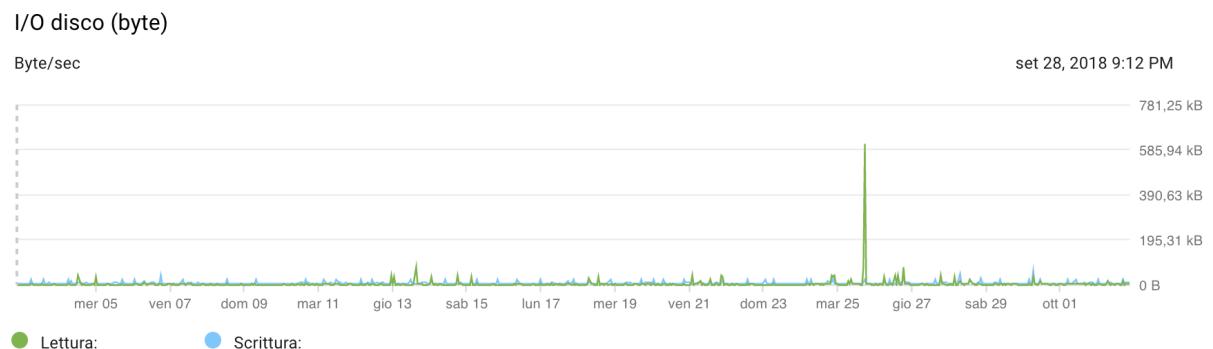


Figura 2.4: I/O disco (byte) dal 3 Settembre al 3 Ottobre 2018

Diminuendo la scala da 30 giorni a meno di un'ora come in figura 2.5 è possibile notare come ci siano picchi di rete ogni 8 minuti (ogni volta che viene attivata la classe trenordAlertResponder,

capitolo 1.3.3). I suddetti picchi come è possibile notare sono comunque **molto contenuti**, considerando che al momento di questa analisi sono in monitoraggio circa 10 direttive (su un massimo di 50 totali monitorabili)



Figura 2.5: CPU e Rete durante il richiamo della classe trenordAlertResponder. Cpu a 2,42% di utilizzo e scambio di dati inferiore agli 8Kb/s

2.6 Documentazione e Controllo Versione

Durante il periodo di stage universitario, fin dall'inizio dello sviluppo, il progetto è stato inserito in **GitHhub**. GitHub è un servizio di hosting per progetti software, in particolare si tratta di una implementazione dello strumento di controllo versione distribuito Git.

L'utilizzo di GitHub agevola molto lo sviluppo, il ripristino e la gestione delle varie versioni di ogni singola classe. E' inoltre un'ottima vetrina personale, tramite la quale mostrare i propri lavori.

Parallelamente allo sviluppo del codice su GitHub è stata inserita la **documentazione** relativa alle classi principali, il database, il funzionamento delle API, i casi d'uso e le informazioni necessarie al primo utilizzo.

Il codice è quindi **open-source** e disponibile all'indirizzo <https://github.com/Fabrolly/TrenoBot>

Capitolo 3

Fonte Dati e Recupero delle Informazioni

Il principale ostacolo nella realizzazione del progetto è stato il **recupero delle informazioni** necessarie al chatbot per poter fornire tutti i *servizi* richiesti.

Purtroppo, nonostante le ferrovie siano un servizio pubblico, **non esistono delle API di Trenitalia** che consentano di usufruire dei dati relativi alla circolazione ferroviaria. Con delle semplici ricerche sul web è immediato notare come, *in altri paesi europei*, i servizi pubblici mettano a disposizione i propri dati liberamente.

Un esempio su tutti è quello delle ferrovie tedesche (**Deutsche Bahn**) che, tramite il loro sito web, **espongono liberamente le API** documentate per tutti i *servizi legati alla circolazione ferroviaria e correlati*^[8].

Per questo motivo, il recupero delle informazioni sulla **circolazione dei treni in Italia**, si è rilevato *meno banale di quanto previsto* ed è argomento di approfondimento di questo capitolo.

3.1 Fonte dei dati

La fonte dei dati da utilizzare nel chatbot **non è stata una scelta immediata** poiché esistono diversi siti che espongono le informazioni di nostro interesse: il sito ufficiale di *Trenitalia*, quello di *Trenord*, oppure servizi forniti da terzi.

La soluzione più efficiente per le esigenze del chatbot è quella di utilizzare un servizio che possibilmente **integri i dati di tutte le compagnie ferroviarie**, evitando quindi l'interfacciamento con diversi sistemi che comporterebbero problemi di *eterogeneità* dei dati.

Il servizio identificato a questo scopo, dopo l'analisi di tanti siti potenzialmente utili, è www.viaggiatreno.it^[9]

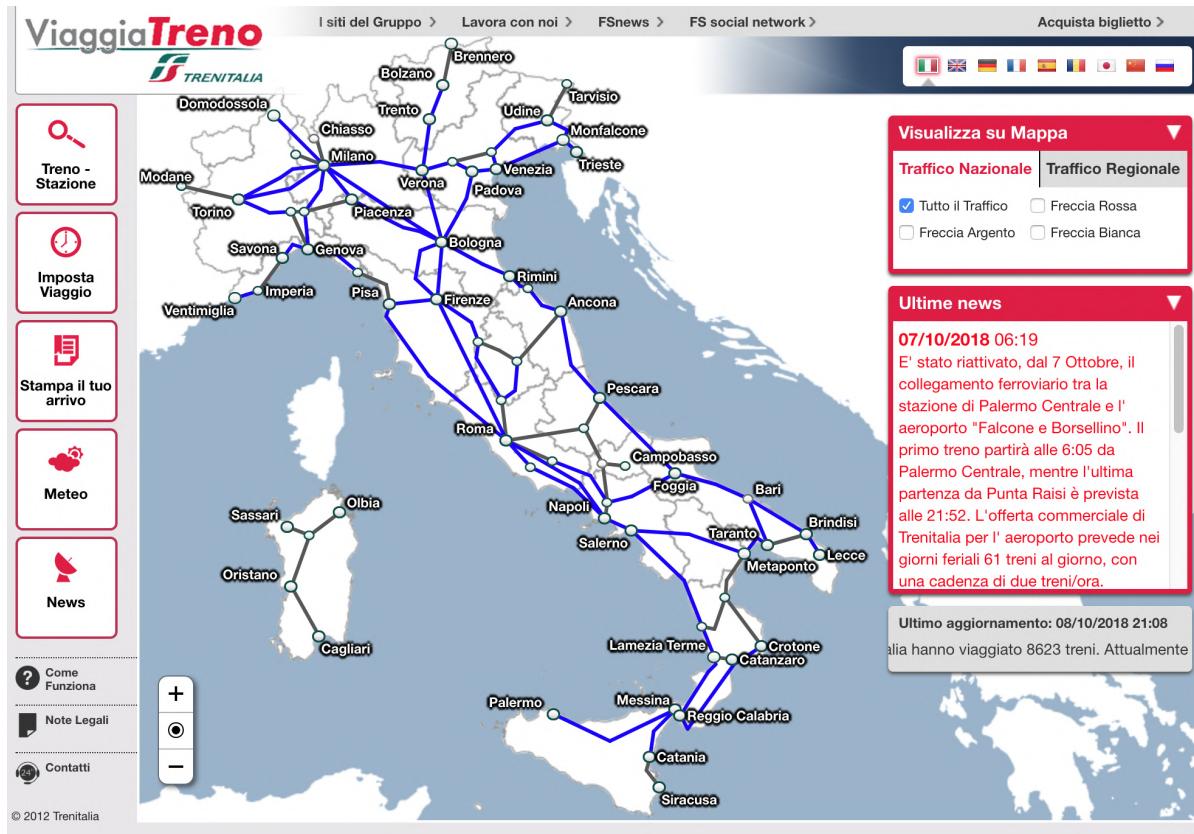


Figura 3.1: Pagina principale del sito *viaggiatreno.it*

ViaggiaTreno è un **servizio offerto da Trenitalia** a partire dal 2012 per *cercare e monitorare* i treni in real time, fornendo all’utente tutte le informazioni di *cui la chatbot*, per come progettata, ha bisogno. **ViaggiaTreno è un sito web**, disponibile anche in versione mobile all’indirizzo <http://mobile.viaggiatreno.it>

Il servizio è pensato per essere usato via browser dall’utente e non è in **alcun modo personalizzabile**: ad ogni richiesta di un utente corrisponde una risposta via web, *senza notifiche* e senza ulteriori automazioni possibili.

3.2 Recupero delle informazioni in tempo reale

Il primo caso d'uso individuato (capitolo 1.2), consiste nel mostrare all'utente informazioni sull'**andamento di un treno in tempo reale**.

Tramite il pulsante "cerca" in alto a destra della Figura 3.1 è possibile *inserire il numero di un treno* per visualizzarne i dettagli.

La Figura 3.2 mostra invece l'interfaccia che viene mostrata all'utente dopo la richiesta di un treno, in questo caso il treno numero 5050.

The screenshot shows the ViaggiaTreno website interface. At the top, there is a navigation bar with links to 'I siti del Gruppo', 'Lavora con noi', 'FSnews', 'FS social network', and 'Acquista biglietto'. Below the navigation bar, there is a language selection bar with icons for various countries. On the left side, there is a sidebar with several buttons: 'Treno - Stazione', 'Imposta Viaggio', 'Stampa il tuo arrivo', 'Meteo', 'News', 'Come Funziona', 'Note Legali', and 'Contatti'. The main content area displays the following information:

- Andamento in Tempo Reale**: Shows 'REG 5050 delle 16:08'.
- Home**: The current page is 'REG 5050 delle 16:08 da BERGAMO a LECCO'.
- STAZIONE DI PARTENZA**: BERGAMO. Partenza programmata: 16:08. Partenza effettiva: 16:09.
- FERMATE INTERMEDI**: LECCO. Arrivo Programmato: 16:48 Bin.reale: 1 Binario. Arrivo Effettivo: 16:49 Orientamento: --.
- STAZIONE DI ARRIVO**: LECCO. Arrivo programmato: 16:48. Arrivo Effettivo: 16:49.
- INFORMAZIONI TEMPO REALE**: Arrivato con un ritardo di 1 min. Ultimo rilevamento a LECCO Alle ore 16:49.
- CORRISPONDENZE**: A table showing train correspondences:

Treno	Per	Ore	Bin.prev.	Bin.reale	Ritardo
REG 2581	MILANO CENTRALE	21:01	4	4	● in orario
REG 2578	SONDRIO	21:02	3	3	● in orario
REG 10885	MILANO PORTA GARIBALDI	21:07	5	5	non partito
REG 5061	BERGAMO	21:12	1 TRONCO		● in orario
- Mappa +**, **Precedente**, **Successivo** buttons at the bottom.

Figura 3.2: Andamento del treno "5050" in realtime su viaggiatreno.it

In questa schermata le informazioni interessanti per il chatbot sono molteplici:

- Gli orari di **Partenza** e **Arrivo** effettivi e programmati (ovvero quelli previsti)
- Il **ritardo** in tempo reale (o il ritardo di arrivo, se il treno è già giunto a destinazione)
- I **binari** e i ritardi per ogni singola fermata del treno
- Eventuali **avvisi** per modifiche o cancellazioni
- La data e la posizione dell'**ultimo rilevamento**

Una prima soluzione, quella più semplice da progettare, consiste nell'effettuare un **parsing HTML** della pagina. Il parsing HTML consiste nell'individuare manualmente, da una pagina HTML, i tag e le posizioni in cui compaiono dei dati di interesse per estrarli dalla pagina e usarli come dati primitivi. Questa soluzione è piuttosto **brutale** e **poco efficiente**: anzitutto

necessita di caricare l'intera pagina web ad ogni richiesta e non assicura assolutamente una stabilità temporale: una piccola variazione del codice da parte degli sviluppatori di *ViaggiaTreno* comporterebbe il malfunzionamento di questa soluzione.

Esclusa, per le ragioni sopra, la possibilità di effettuare un parsing HTML della pagina ho cercato una soluzione migliore per poter accedere a questi dati, in un formato più comodo da gestire.

Utilizzando la **Google Chrome DevTools Console**^[10], come in figura 3.3, ho analizzato le richieste esterne della pagina di ViaggiaTreno quando ad essa viene richiesto lo stato di un treno. Nella figura 3.3 e nelle seguenti di questa sezione, prenderemo in esempio la richiesta del treno numero **5040**

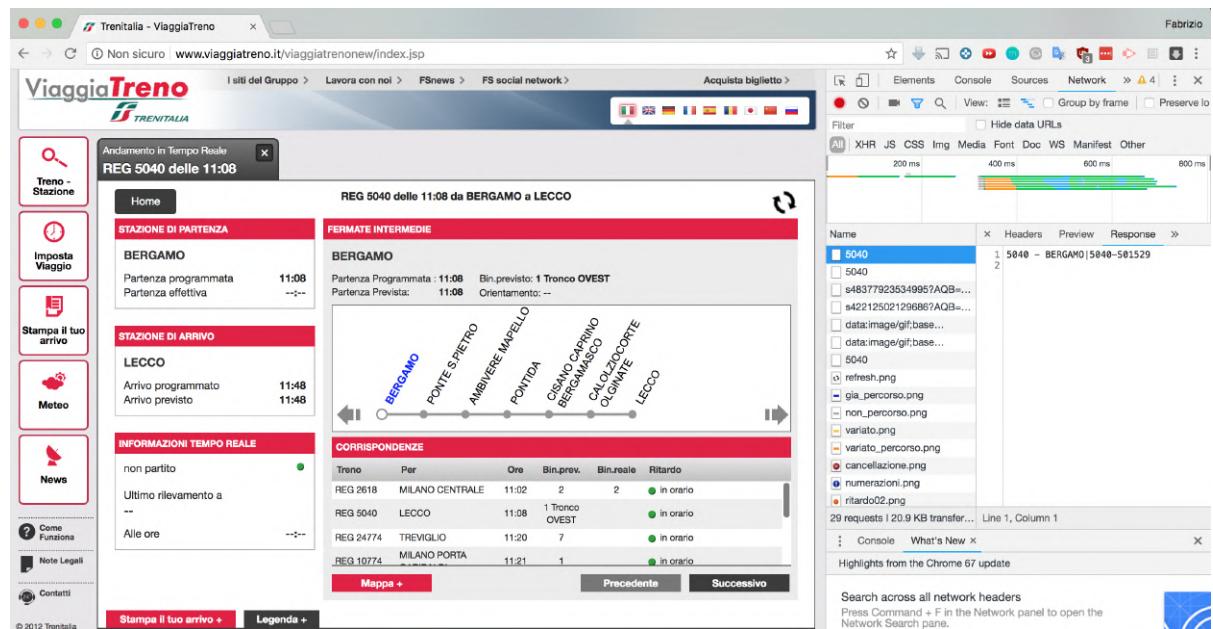


Figura 3.3: Google Chrome DevTools sulla pagina web di ViaggiaTreno

La **DevTools di Google Chrome** è una **console** che viene utilizzata dagli sviluppatori per analizzare vari aspetti di una pagina web ma può essere chiaramente usata anche per analizzare pagine realizzate da terzi, come in questo caso. La Google Chrome DevTools, tra le tante possibilità offerte, permette di analizzare tutte le *richieste esterne* che vengono fatte dalla pagina web dinamica di *ViaggiaTreno*.

I file interessanti al nostro scopo, in riferimento alla parte destra della figura 3.3, sono i due file nominati "**5040**": essi infatti hanno il nome uguale al numero del treno richiesto durante il caricamento della pagina stessa.

Il primo file, mostrato in figura 3.4, è generato da un URL che presenta nella parte finale il numero del treno cercato. Esso contiene un'informazione che sarà **fondamentale** nel prossimo passaggio ovvero il *codice della stazione di riferimento* (in questo caso "**S0529**")

Capitolo 3. Fonte Dati e Recupero delle Informazioni

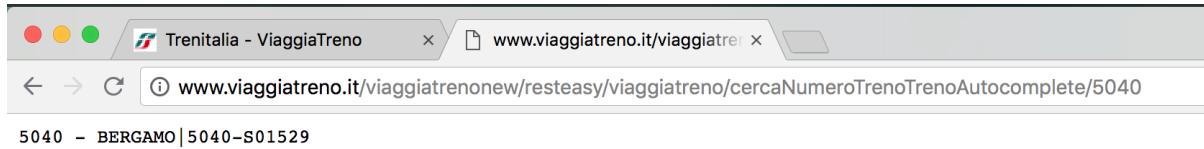


Figura 3.4: Url e contenuto del primo file con il numero del treno cercato (5040)

Il codice della stazione indica la **stazione di riferimento** di quel treno, ovvero la stazione di partenza. Questa soluzione è molto curiosa, poiché, come vedremo nel secondo file "5040", ogni treno è identificato dalla coppia "**numero treno - idStazioneRiferimento**". Questo significa che *il numero del treno non è sufficiente*, seppur sia univoco, *a identificare il treno stesso*.

Una volta ottenuto l'**id della stazione di riferimento** recuperato tramite l'URL in figura 3.4 è possibile procedere alla fase di analisi del **secondo file** nominato "5040" individuato in figura 3.3.

Ricapitolando, al momento, abbiamo ottenuto le seguenti **informazioni**:

- Numero del treno (inserito dall'utente) -> 5040
 - Id della stazione di riferimento -> S01529

Aprendo il secondo file "5040" individuato in figura 3.3, otteniamo il risultato in figura 3.5

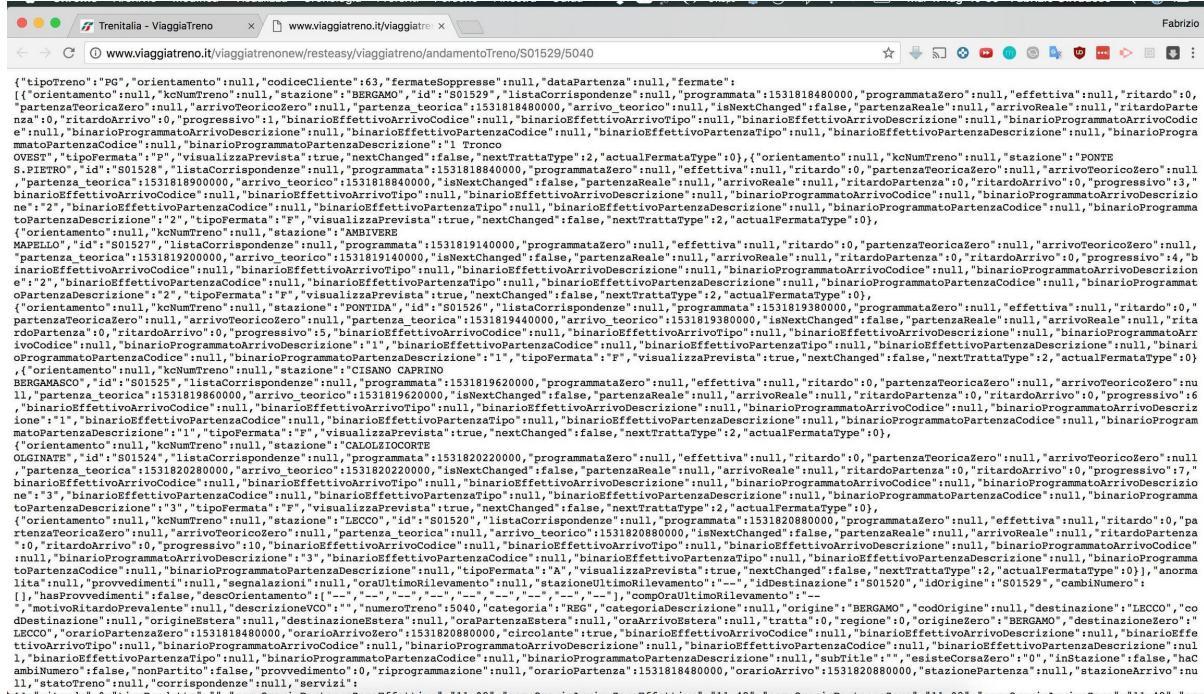


Figura 3.5: Url e contenuto del secondo file nominato "5040"

Iniziando ad analizzare il risultato partendo dall'URL notiamo subito la presenza della stringa **.../SO1529/5040** in fondo all'URL stesso. Essa corrisponde esattamente al formato

id stazione di riferimento/numero treno

Per questo motivo, l'id della stazione di riferimento di un treno **è fondamentale per identificare il treno stesso**: in caso contrario *non avremmo modo* di ottenere l'URL di questo secondo file nominato "5040".

Il contenuto del suddetto URL, in figura 3.5, è un file in **formato JSON** contenente tutte le *informazioni di cui il chatbot ha bisogno*, nonché tutte le informazioni che il sito dinamico ViaggiaTreno utilizza per la generazione della pagina web in risposta all'utente (figura 3.2).

Le **informazioni di interesse** di un singolo treno, in particolare, si riferiscono ai campi del formato JSON di seguito elencati:

- 'numeroTreno'
- 'origineZero'
- 'destinazioneZero'
- 'orarioPartenza'
- 'orarioArrivo'
- 'compTipologiaTreno'
- 'compDurata'
- 'oraUltimoRilevamento'
- 'stazioneUltimoRilevamento'
- 'subTitle' (*in questo campo compaiono, se presenti, avvisi di variazioni o disagi*)
- 'ritardo'
- 'provvedimento' (*stato del treno: Regolare, Parzialmente Soppresso o Soppresso*)
- 'fermate'

In particolare, il campo **fermate**, contiene al suo interno i valori che sono stati analizzati in figura 1.14 al capitolo 1.4.1 e che faranno parte, così come sono formattati, del campo "stops" del database interno utilizzato dal chatbot. I campi di "fermate" sono presenti *per ogni fermata che effettua il treno*.

A questo punto, tramite i passaggi riassunti in figura 3.6, *sono state ottenute tutte le informazioni necessarie al chatbot*.

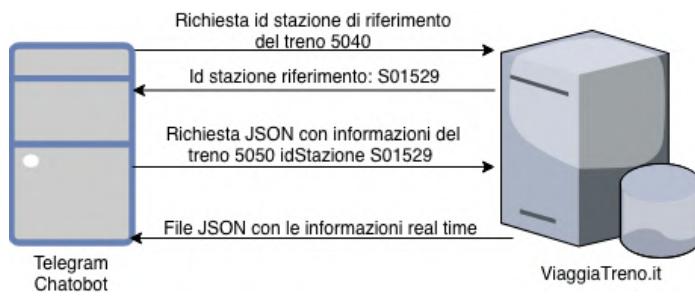


Figura 3.6: Schema delle richieste necessarie ad ottenere le informazioni in real time sul treno 5040



Figura 3.7: Esempio di richiesta informazioni real time del treno 5040 al chatbot

3.2.1 Implementazione del recupero delle informazioni in tempo reale

A livello implementativo, la classe `apiViaggiaTreno` (capitolo 1.3.2) contiene due metodi:

- `stationId(trainNumber)`
- `realTimeInformation(trainNumber)`

Il primo metodo, `stationId`, dato in input il numero di un treno (`trainNumber`) restituisce l'**id della stazione di riferimento** del treno stesso (`stationId`).

Listing 3.1: Implementazione Python del metodo `stationId` nella classe `apiViaggiaTreno.py`

```

1 def stationId (trainNumber):
2     try:
3         stationId=urllib.urlopen ('http://www.viaggiaTreno.it/viaggiaTrenoNew/resteasy/
4                                     viaggiaTreno/cercaNumeroTrenoTrenoAutocomplete/%s' % trainNumber).read()
5     except Exception ,e:
6         print e #ViaggiaTreno.it offline
7
8     if(len(stationId)!=0):
9         stationId=stationId[(stationId.index('-')+1):(stationId.index('\n'))]
10        return stationId #Id stazione di riferimento del treno
11    else:
12        print 0 #train_number non esistente

```

Alla **riga numero 3** è presente la composizione dell'URL individuato in figura 3.4, necessaria a generare la pagina da cui la classe `trainNumber` estrae l'**id della stazione di riferimento**, tramite l'indicizzazione di alcuni caratteri di standard contenuti nella risposta.

Questo metodo, è **fondamentale** per le ragioni descritte al capitolo 3.2.0, infatti ci consente di identificare uno specifico treno ottenendo la coppia:

<id stazione di riferimento, numero treno>

Il secondo metodo della classe `apiViaggiaTreno`, `realTimeInformation`, dato in input il numero di un treno (`trainNumber`) restituisce il **file JSON** (figura 3.5) contenente tutte le **informazioni real time di un treno**. Il metodo `realTimeInformation` fa uso del primo metodo (`stationId`), per ricavare la coppia di valori necessaria a comporre la richiesta a `viaggiaTreno.it`

Listing 3.2: Implementazione Python di `realTimeInformation`

```

1 def realTimeInformation (trainNumber):
2     try:
3         stationId=stationId (trainNumber) #recupero l'id della stazione di riferimento
4                                     tramite il metodo
4         url='%s/%s'%(stationId , trainNumber)
5         raw_json= urllib.urlopen ('http://www.viaggiaTreno.it/viaggiaTrenoMobile/resteasy/
6                                     /viaggiaTreno/andamentoTreno/%s' % url).read()
7
8         parsed_json = json.loads(raw_json)
9         return parsed_json
10    except Exception ,e:
11        print e

```

La pagina JSON, generata tramite l'URL composto alla riga numero 4, viene **deserializzata** tramite la libreria JSON^[6] (capitolo 2.1.1) alla riga 7. Viene infine ritornato il file JSON deserializzato alla classe chiamante (`databaseController`) che verrà utilizzato per generare l'**oggetto treno** e aggiornare il database locale.

La classe `apiViaggiaTreno`, contenente i due metodi appena descritti, è interamente controllata dalla classe **databaseController** che sceglie di interpellare le API di `viaggiaTreno.it` solamente

Capitolo 3. Fonte Dati e Recupero delle Informazioni

se le informazioni già presenti nel database locale non sono abbastanza *recenti*, come approfondito nel capitolo 1.3.2, classe *databaseController*).

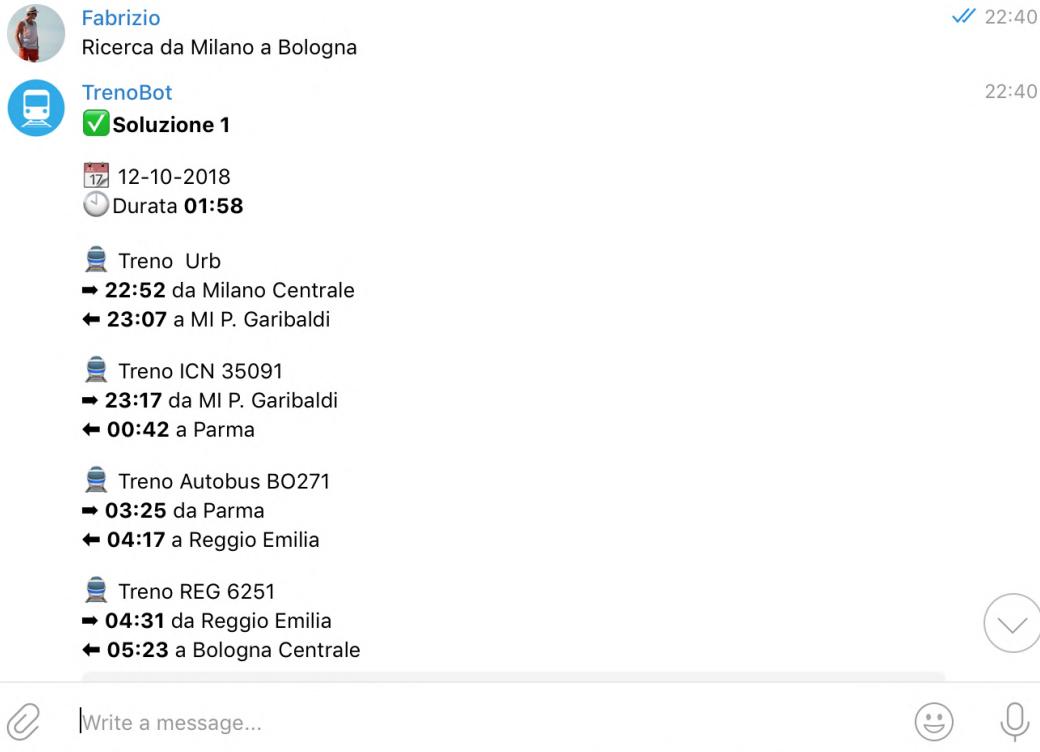


Figura 3.8: Esempio di soluzione di viaggio da Milano a Bologna. Se la data/ora non viene specificata, viene considerata quella attuale

3.3 Ricerca delle soluzioni di viaggio tra due stazioni

Il servizio viaggiaTreno consente inoltre di **cercare una soluzione di viaggio** comprendendo più compagnie ferroviarie, fornendo in input:

- Stazione di Partenza
- Stazione di Arrivo
- Fascia oraria
- Giorno

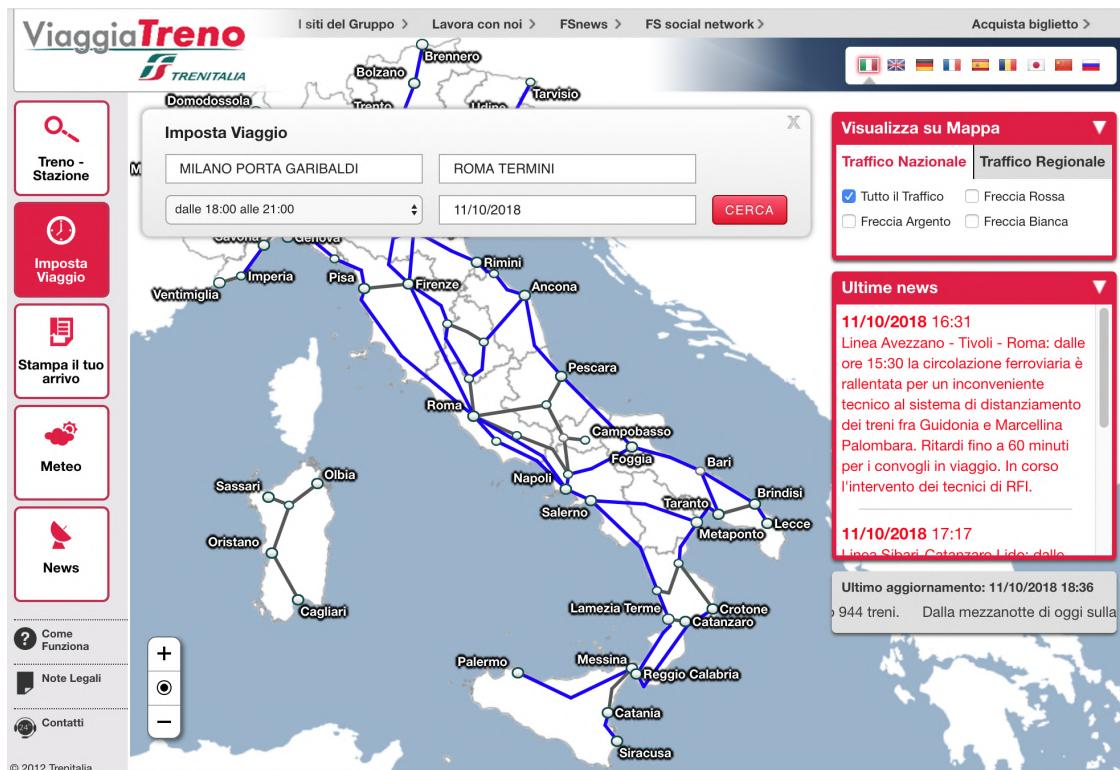


Figura 3.9: Interfaccia per la ricerca di una soluzione di viaggio su viaggiatreno.it

Supponendo di voler cercare un treno da **Milano Porta Garibaldi** a **Roma Termini** e sfruttando nuovamente la **Google Chrome DevTools Console**^[10] è possibile *analizzare* le richieste esterne della pagina web viaggiatreno.it

Le *richieste esterne*, per questo tipo di input, sono in particolare **tre**:

- Richiesta del *codice identificativo* della stazione di partenza (Milano Porta Garibaldi)
- Richiesta del *codice identificativo* della stazione di arrivo (Roma Termini)
- Richiesta delle *soluzioni di viaggio* compatibili

Capitolo 3. Fonte Dati e Recupero delle Informazioni

Le prime due richieste, relative al *codice identificativo delle stazioni* inserite dall'utente ,sono molto simili, e fanno entrambe riferimento alla figura 3.10.

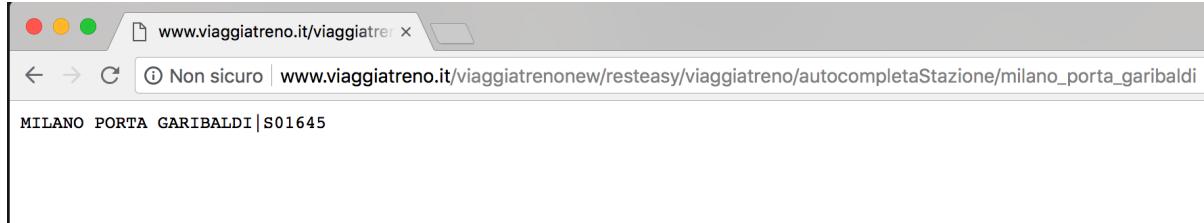


Figura 3.10: Url e contenuto della richiesta per l'id della stazione "Milano Porta Garibaldi"

Analizzando l'**URL** in figura 3.10, si nota come l'input dell'utente venga **formattato** in *minuscolo* e gli spazi vengano *sostituiti* dal carattere "_" e messi in coda all'URL per la richiesta del codice identificativo della stazione.

La **sintassi** dell'indirizzo web per la richiesta è quindi:

```
1 http://www.viaggiatreno.it/viaggia... /STATION_NAME
```

Analizzando invece il **contenuto** in figura 3.10, si nota come il **codice identificativo** della stazione è rappresentato dalla stringa successiva al carattere "|" che lo separa dal nome della stazione. In questo esempio quindi, per Milano Porta Garibaldi, il codice è **S01645**.

In figura 3.11 viene invece mostrato l'output per lo *stesso tipo di richiesta* nel caso in cui l'input dell'utente sia *generico* e presenti **più corrispondenze**.

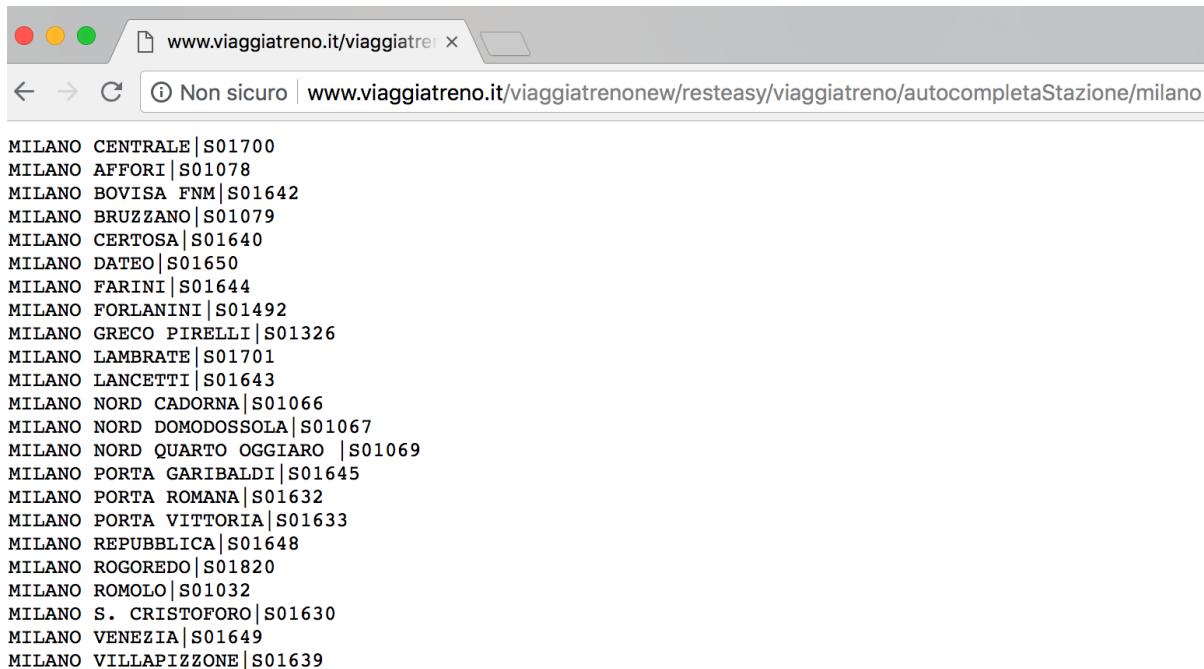


Figura 3.11: Url e contenuto della richiesta per l'id della stazione "milano"

Capitolo 3. Fonte Dati e Recupero delle Informazioni

Eseguendo la **medesima richiesta** della figura 3.10 utilizzando al posto di "**milano_porta_garibaldi**" la stringa "**roma_termini**" otteniamo il *codice identificativo* della stazione di arrivo dell'esempio che stiamo seguendo, ovvero **S08409**.

L'ultima informazione necessaria per procedere alla richiesta delle soluzioni di viaggio è la *data e l'ora di partenza* nel formato

1 | AAAA-MM-GGTHH:MM: SS

dove il carattere "T" separa la data dall'ora.

Se nessuna informazione su data e ora viene specificata, il chatbot **imposta automaticamente** la stringa con la data e l'ora della richiesta, mostrando quindi *la prima soluzione disponibile*.

Abbiamo quindi a disposizione, in riferimento all'esempio fin qui seguito:

- Codice identificativo della stazione di Milano Porta Garibaldi: **S01645**
 - Codice identificativo della stazione di Roma Termini: **S08409**
 - Data e ora di partenza nel formato AAAA-MM-GGTHH:MM:SS: **2018-10-11T18:00:00**

A questo punto siamo a conoscenza di tutto quello che ci serve per procedere alla **terza ed ultima richiesta**, ovvero quella che ci fornirà le **soluzioni di viaggio** vere e proprie.

Sempre tramite la **Google Chrome DevTools Console**^[10] è possibile individuare l'URL della richiesta del nostro esempio, ottenendo l'output in figura 3.12

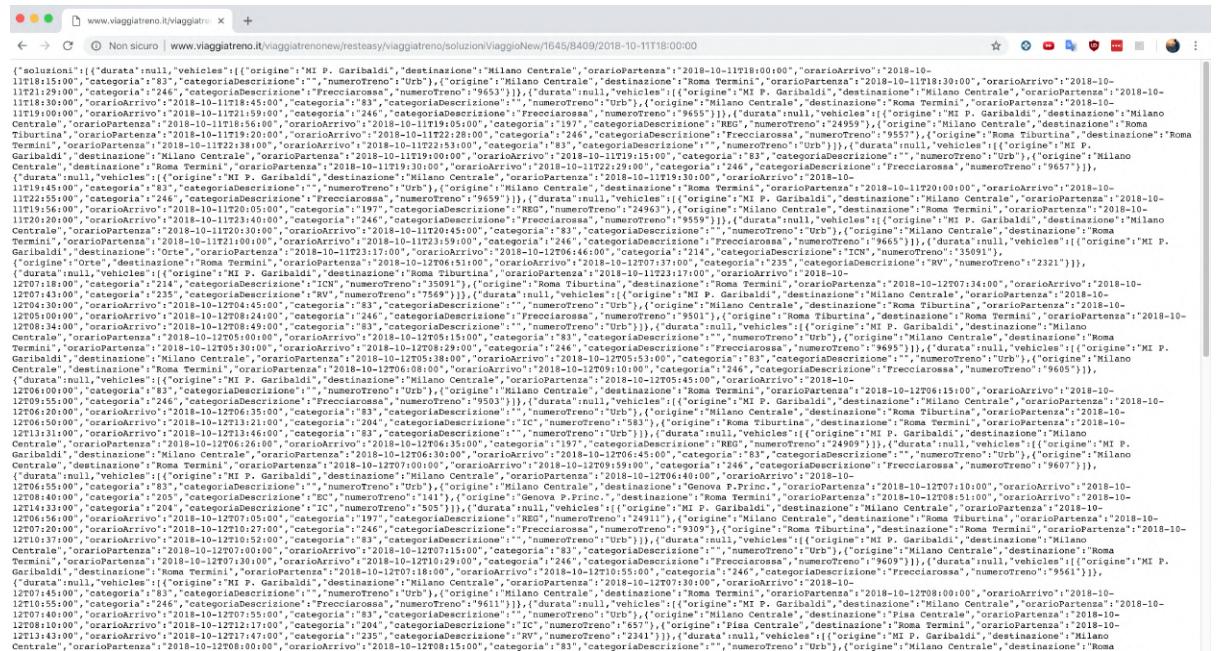


Figura 3.12: URL e contenuto della richiesta per la soluzione di viaggio da Milano Porta Garibaldi a Roma Termini alle ore 18 del 10 Ottobre 2018

L'URL in figura 3.12 è riportato anche di seguito per garantire una miglior comprensione:

1 http://www.viaggiautreno.it/viaggiautrenonew/resteasy/viaggiautreno/soluzioniViaggioNew /1645/8409/2018-10-11T18:00:00

E' facile notare che l'ultima parte che compone l'indirizzo è **esattamente la successione dei valori individuati precedentemente al netto del prefisso "S0"** per l'identificazione dei codici delle due stazioni.

La sintassi della richiesta è quindi la seguente:

1 `.... / soluzioniViaggioNew / CODICE_STAZIONE_PARTENZA / CODICE_STAZIONE_ARRIVO / DATETIME`

Il contenuto della richiesta in figura 3.12, è in formato **JSON** e contiene diverse soluzioni di viaggio possibili a partire dall'orario di *DATETIME* e per le *3 ore successive*.

Nell'**output** in formato JSON appena individuato, di particolare interesse per il chatbot sono i seguenti **campi** di ogni singola soluzione:

- origine
- destinazione
- orarioPartenza
- orarioArrivo
- durata
- numeroTreno (*da cui, tramite quanto visto nel capitolo 3.2.1, posso ricavare tutte le informazioni del treno stesso*)

Le soluzioni che **prevedono dei cambi**, all'interno della stessa soluzione di viaggio, comprendono i suddetti campi per *tutti i treni necessari* per raggiungere la destinazione inserita.

3.3.1 Implementazione della ricerca di soluzioni di viaggio

Tramite il **modulo urllib** di Python^[4], nel seguente frammento di codice facente parte della classe `trip_search.py`, vengono individuati i **i codici identificativi delle stazioni**:

Listing 3.3: Implementazione Python della ricerca dei codici delle stazioni

```
1 #Creo url per la richiesta del CODICE DELLE STAZIONI e interrogo
2 urlPartenza=('http://www.viaggiatreno.it/viaggiatrenonew/resteasy/viaggiatreno/
   autocompletaStazione/%s' %partenza)
3 urlArrivo=('http://www.viaggiatreno.it/viaggiatrenonew/resteasy/viaggiatreno/
   autocompletaStazione/%s' %arrivo)
4
5 #Interrogo per ottenere i codici della stazione
6 CodiceStazionePartenza=urllib.urlopen(urlPartenza).read()
7 CodiceStazioneArrivo=urllib.urlopen(urlArrivo).read()
```

I codici delle stazioni vengono poi utilizzati per effettuare la **terza** richiesta del capitolo 3.3, quella che ci fornirà in output le informazioni con **tutte le soluzioni di viaggio possibili**.

Listing 3.4: Implementazione Python della richiesta delle soluzioni di viaggio

```
1 DataOra='%s-%s-%sT%s:00:00'%(now.year, mese, giorno, ora)
2 urlRicerca=( 'http://www.viaggiatreno.it/viaggiatrenonew/resteasy/viaggiatreno/
   soluzioniViaggioNew/%s/%s/%s' % (CodiceStazionePartenza, CodiceStazioneArrivo,
   DataOra))
3
4 parsed_json = json.loads(urllib.urlopen(urlRicerca).read())
```

L'output in formato **JSON** contenete le soluzioni di viaggio possibili viene poi **decomposto** per poter generare il *messaggio testuale* in risposta all'utente. In questo caso è stato scelto di mostrare **le prime due** soluzioni disponibili.

Listing 3.5: Pseudocodice decomposizione della risposta di viaggiaTreno

```

1  for sol in range (0,2): #I am interested in the first two solutions proposed
2      for cont in range(0,len(parsed_json[ 'soluzioni '][ sol ][ 'vehicles '])): #if the
3          solution has more trains
4              departure_time=parsed_json[ 'soluzioni '][ sol ][ 'vehicles '][ cont ][ 'orarioPartenza ']
5              departure_day=departure_time[departure_time.index( 'T' )-2:departure_time .
6                  index( 'T' )]
7              departure_month=departure_time[departure_time.index( '-' )+1:]
8              departure_month=departure_month[:departure_time.index( '-' )-2]
9              departure_time=departure_time[departure_time.index( 'T' )+1:departure_time .
10                 index( 'T' )+6]
11             arrival_time=parsed_json[ 'soluzioni '][ sol ][ 'vehicles '][ cont ][ 'orarioArrivo ']
12             arrival_time=arrival_time[arrival_time.index( 'T' )+1:arrival_time.index( 'T' )
13                 +6]
14
15             #In this point I have one of the train of the solution
16             #In this point I have ALL the trains of the solutions
17             #In this point I have two solutions complete

```

La **funzionalità di ricerca** della chatbot **non** prevede un sistema di *cache*, poichè le possibilità di ricerca sono troppe per poter essere immagazzinate e riutilizzate. Ogni richiesta di soluzione di viaggio **viene gestita singolarmente** e sempre tramite le tre richieste esterne alle API ViaggiaTreno^[9] descritte in questo capitolo.



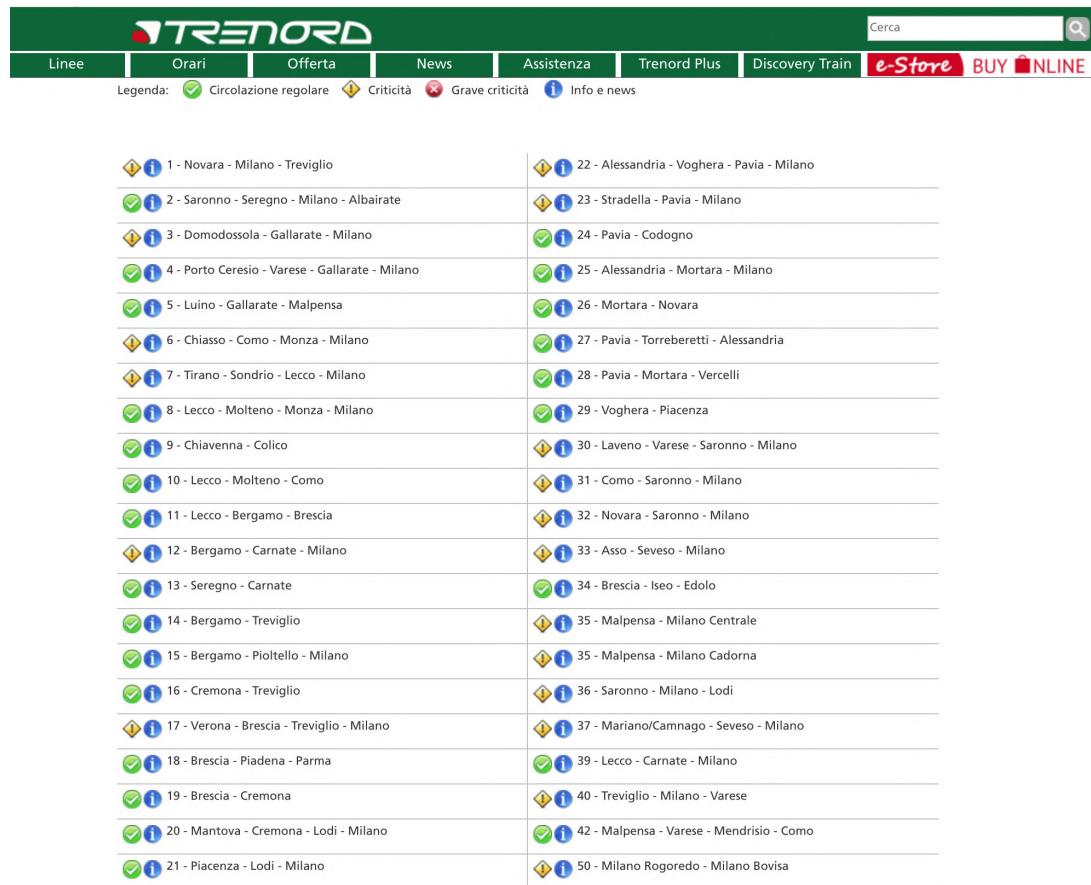
Figura 3.13: Esempio di richiesta al chatbot di una soluzione di viaggio da Milano a Bologna. In questo caso, essendo un orario notturno, la soluzione è molto complessa e prevede addirittura degli Autobus.

Capitolo 4

Monitoraggio delle Direttrici

Un'altra possibilità della chatbot è quella di poter monitorare per l'utente un'**intera direttrice Trenord**.

Una **direttrice** è una **tratta** composta da diverse stazioni sulla quale si susseguono diversi treni durante la giornata e su cui possono verificarsi dei **disagi** che coinvolgono tutti i treni che la attraversano.



The screenshot shows the Trenord website interface. At the top, there is a navigation bar with links for Linee, Orari, Offerta, News, Assistenza, Trenord Plus, Discovery Train, e-Store, and BUY ONLINE. A search bar is also present. Below the navigation bar, a legend provides status icons: a green circle for 'Circolazione regolare', a yellow diamond for 'Criticità', a red circle for 'Grave criticità', and a blue circle with an info icon for 'Info e news'. The main content area displays a table of 50 train routes, each with a status icon and a route number followed by its destination. The routes are listed in two columns:

1 - Novara - Milano - Treviglio	22 - Alessandria - Voghera - Pavia - Milano
2 - Saronno - Seregno - Milano - Albairate	23 - Stradella - Pavia - Milano
3 - Domodossola - Gallarate - Milano	24 - Pavia - Codogno
4 - Porto Ceresio - Varese - Gallarate - Milano	25 - Alessandria - Mortara - Milano
5 - Luino - Gallarate - Malpensa	26 - Mortara - Novara
6 - Chiasso - Como - Monza - Milano	27 - Pavia - Torreberetti - Alessandria
7 - Tirano - Sondrio - Lecco - Milano	28 - Pavia - Mortara - Vercelli
8 - Lecco - Molteno - Monza - Milano	29 - Voghera - Piacenza
9 - Chiavenna - Colico	30 - Laveno - Varese - Saronno - Milano
10 - Lecco - Molteno - Como	31 - Como - Saronno - Milano
11 - Lecco - Bergamo - Brescia	32 - Novara - Saronno - Milano
12 - Bergamo - Carnate - Milano	33 - Asso - Seveso - Milano
13 - Seregno - Carnate	34 - Brescia - Iseo - Edolo
14 - Bergamo - Treviglio	35 - Malpensa - Milano Centrale
15 - Bergamo - Pioltello - Milano	35 - Malpensa - Milano Cadorna
16 - Cremona - Treviglio	36 - Saronno - Milano - Lodi
17 - Verona - Brescia - Treviglio - Milano	37 - Mariano/Cannago - Seveso - Milano
18 - Brescia - Piadena - Parma	39 - Lecco - Carnate - Milano
19 - Brescia - Cremona	40 - Treviglio - Milano - Varese
20 - Mantova - Cremona - Lodi - Milano	42 - Malpensa - Varese - Mendrisio - Como
21 - Piacenza - Lodi - Milano	50 - Milano Rogoredo - Milano Bovisa

Figura 4.1: Direttrici in tempo reale sul sito Trenord

4.1 Fonte dei dati

Monitorare una direttrice consente all’utente di avere aggiornamenti precisi sulle comunicazioni Trenord in **anticipo** rispetto al proprio treno: infatti, manutenzioni programmate o altri tipi di avvisi particolari vengono comunicati solo tramite questa pagina.

Trenord srl è l’unica compagnia ferroviaria che permette un **monitoraggio** di questo tipo, attraverso il proprio **sito web**^[1].

Come si nota in figura 4.6, su 50 direttrici **ben 18 direttrici** al momento della realizzazione dello screenshot presentano una qualche tipo di **criticità**.

Cliccando su una direttrice tra quelle presenti, viene caricata una pagina come in figura 4.2, dove è possibile vedere i **dettagli relativi alla tratta selezionata**.

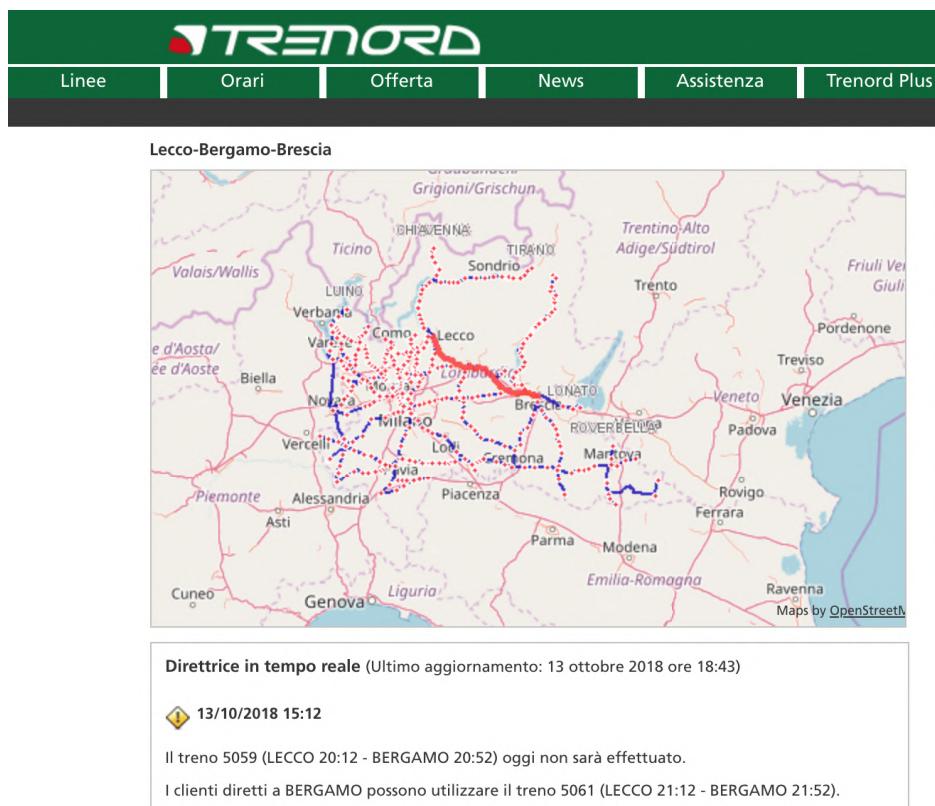


Figura 4.2: Esempio di criticità segnalata sulla pagina Trenord della direttrice Lecco-Bergamo-Brescia

La difficoltà riscontrata nel reperire queste informazioni è dovuta al fatto che la pagina in questione è **statica, generata lato server**: di conseguenza **non esiste alcuna API** interna da poter analizzare tramite **Google Chrome DevTools Console**^[10].

L’unico metodo, per poter reperire queste utilissime informazioni per l’utente, è utilizzare il metodo del **parsing HTML**.

Come affrontato nel capitolo 3.2.1, il parsing HTML **non è una buona pratica** per diverse ragioni. **In questo caso**, però, la presenza della stringa "13/10/2018 15:12" (della figura 4.2) prima del messaggio di allerta ci aiuta nella realizzazione di un parsing HTML piuttosto solido.

La stringa infatti ci fornisce un **riferimento statico** per recuperare il messaggio sottostante oltre alle informazioni su **quando l'avviso è stato inserito**.

Il codice sorgente della pagina in figura 4.2 è riportato in figura 4.3. Si notano immediatamente le due **informazioni di interesse**: la prima alla riga 240, tra i tag <H4>, contenente la stringa con data e ora dell'avviso; la seconda, dentro i tag <p> alla riga 244 contenente l'avviso vero e proprio.

```
235
236
237
238
239
240
241
242
243
244
245
246
247
248
<div style="float: left; position: relative" class='aggiornamento first'>
    <div style="float:left; position: relative; padding-right: 5px;">
        <img src='/img/web/direttrici_critical.png' alt='Criticit&agrave;' title='Criticit&agrave;' />
    </div>
    <div style="float:left; position: relative">
        <h4> 13/10/2018 15:12</h4>
    </div>
    <div style="clear: both"></div>
    <div style="float:left; position: relative">
        <p> <p>Il treno 5059 (LECCO 20:12 - BERGAMO 20:52) oggi non sarà effettuato.</p>
    </div>
<div>I clienti diretti a BERGAMO possono utilizzare il treno 5061 (LECCO 21:12 - BERGAMO 21:52).</p></div>
</div>
```

Figura 4.3: Sorgente della pagina in figura 4.2

4.2 Implementazione del Monitoraggio Direttrici

La classe **trenordAlertResponder.py** analizzata nel capitolo 1.3.3 si occupa di effettuare, **ogni otto minuti**, il *parsing HTML* di tutte le direttrici Trenord.

La classe, ogni volta che viene richiamata, genera una stringa nel formato "**DD/MM/AAAA HH:**" contenente la data e l'ora corrente. Successivamente, **per ogni direttrice**, viene scaricata la pagina web ed eseguita una ricerca della suddetta stringa all'interno del codice sorgente della pagina stessa.

- Se la stringa non viene trovata si passa alla direttrice successiva
- Se la stringa viene trovata viene effettuato il parsing della "**<div>**" contenente la stringa

Il *parsing HTML* viene effettuato mediante il codice in figura 5.2 della classe *trenordAlertResponder*, che seleziona il contenuto del messaggio presente tra i due **tag** <p><p>, come illustrato nel codice sorgente di esempio alla figura 4.3 . Inoltre, durante l'operazione di parsing HTML, eventuali altri tag HTML vengono **cancellati**, non essendo direttamente supportati da Telegram per l'invio del messaggio. In questo modo, il messaggio di nostro interesse, **è estrapolato** dalla pagina web e può essere utilizzato per notificare l'utente.

```

if(dateString in page):
    page=page[page.index(dateString):]
    lastAlert=page[page.index("<p> <p>") + 7 : page.index("</p></p>")]

    if('disagi' in lastAlert.lower() or '<strong>' in lastAlert.lower() or 'cancel' in lastAlert.lower()):
        lastAlert = lastAlert.replace("<p>", "")
        lastAlert = lastAlert.replace("<strong>", "")
        lastAlert = lastAlert.replace("</strong>", "")
        lastAlert = lastAlert.replace("</p>", "")
        lastAlert = lastAlert.replace("<br/>", "")
        lastAlert = lastAlert.replace("&nbsp;", "")

else:
    lastAlert=None

```

Figura 4.4: Operazione di parsing della DIV contenente la stringa target

Alla riga 5 del codice in figura 5.2 è presente una condizione che serve a segnalare all'utente i messaggi contenenti **soltamente** delle parole chiave relative a disagi importanti. Questa soluzione è resa necessaria dal fatto che molti degli avvisi presenti sulle pagine delle direttive Trenord **non sono particolarmente rilevanti**, e produrrebbero un continuo invio di messaggi di poco interesse all'utente. Alcune volte, i suddetti messaggi, sono addirittura di **natura pubblicitaria** (come in figura 4.5).

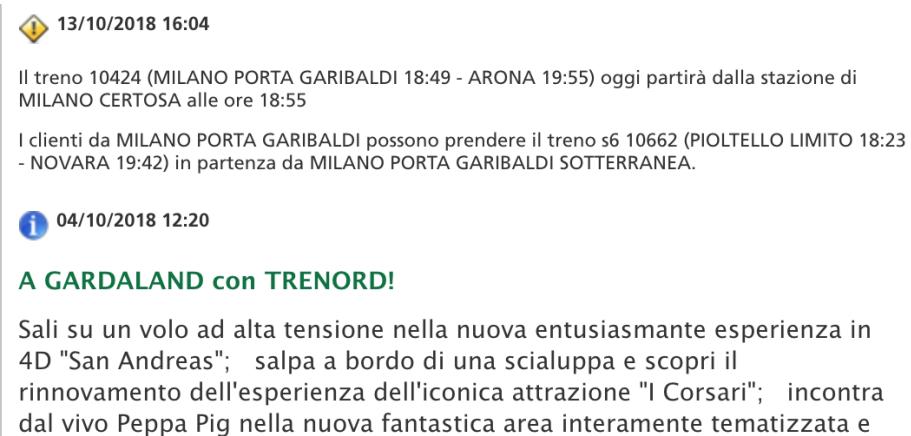


Figura 4.5: Esempio di messaggio rilevante e non rilevante sulla pagina delle direttive Trenord

Il messaggio, una volta estratto dalla pagina web, viene *immagazzinato* nel database locale e **invia** all'utente secondo le logiche già descritte al capitolo 1.4.3 .

Capitolo 4. Monitoraggio delle Diretrici



Figura 4.6: Esempio di criticità di una direttrice segnalata dalla chatbot

Capitolo 5

Conclusioni e Sviluppi Futuri

Il progetto è stato completato durante i *tre mesi* di stage previsti presso l'Università di Milano-Bicocca. Tutte le funzionalità previste in fase di progettazione e di presentazione della proposta di stage *sono state sviluppate e sono correttamente testate e funzionanti*.

La chatbot è stata resa disponibile su telegram con il nome di "**TrenoBot**" e con username **@trainOllyBot**. Il codice sorgente e la documentazione, in inglese, è disponibile su GitHub^[11].

Durante la sua attività (considerando anche la versione precedente) la chatbot ha servito oltre **500 utenti unici**, eseguendo oltre **65.000 query** e informando giornalmente gli utenti su **230 treni** inseriti nella lista dei treni monitorati.

stats.txt							
19/5/2017	21:0	64570	539	232	11	11	
20/5/2017	21:0	64616	539	230	11	11	
21/5/2017	21:0	64624	540	231	1	1	
21/5/2017	21:0	64624	540	231	1	1	
22/5/2017	21:0	65043	542	233	194	192	
22/5/2017	21:0	65043	542	233	194	192	
23/5/2017	21:0	65450	543	231	197	196	
23/5/2017	21:0	65450	543	231	197	196	
24/5/2017	21:0	65852	544	231	193	193	
24/5/2017	21:0	65852	544	231	193	193	
25/5/2017	21:0	66247	545	232	196	196	
25/5/2017	21:0	66247	545	232	196	196	
26/5/2017	21:0	66663	544	230	199	199	
26/5/2017	21:0	66663	544	230	199	199	
27/5/2017	21:0	66686	544	230	10	10	

Figura 5.1: Statistiche nei primi 6 mesi di vita della chatbot, alla versione originaria

Per essere stato un progetto non pubblicizzato, la cui diffusione si deve solamente al *passaparola* degli utenti, questi numeri sono di grande **soddisfazione**. Ho avuto modo di discutere con molti degli utenti che utilizzavano il servizio scoprendo che molti di essi erano *lavoratori del settore* come macchinisti, controllori o semplici appassionati. Seppure questi fossero una minoranza rispetto agli utenti **pendolari**, è stato molto interessante notare come la chatbot abbia attirato l'attenzione anche degli addetti ai lavori, con cui sono nate **discussioni** di grande aiuto e interesse.

A inizio 2017, dopo aver inserito il mio link paypal all'interno del bot per poter mantenere il server, sono *inaspettatamente* arrivate diverse **donazioni**, di pochi euro, che mi hanno però permesso di continuare a sviluppare questo progetto con grande determinazione. A tal proposito era nata anche una *community* all'interno di un gruppo Telegram, dove gli utenti del chatbot discutevano di argomenti di interesse o esprimevano pareri e consigli per **migliorare ulteriormente il servizio** offerto dal chatbot.

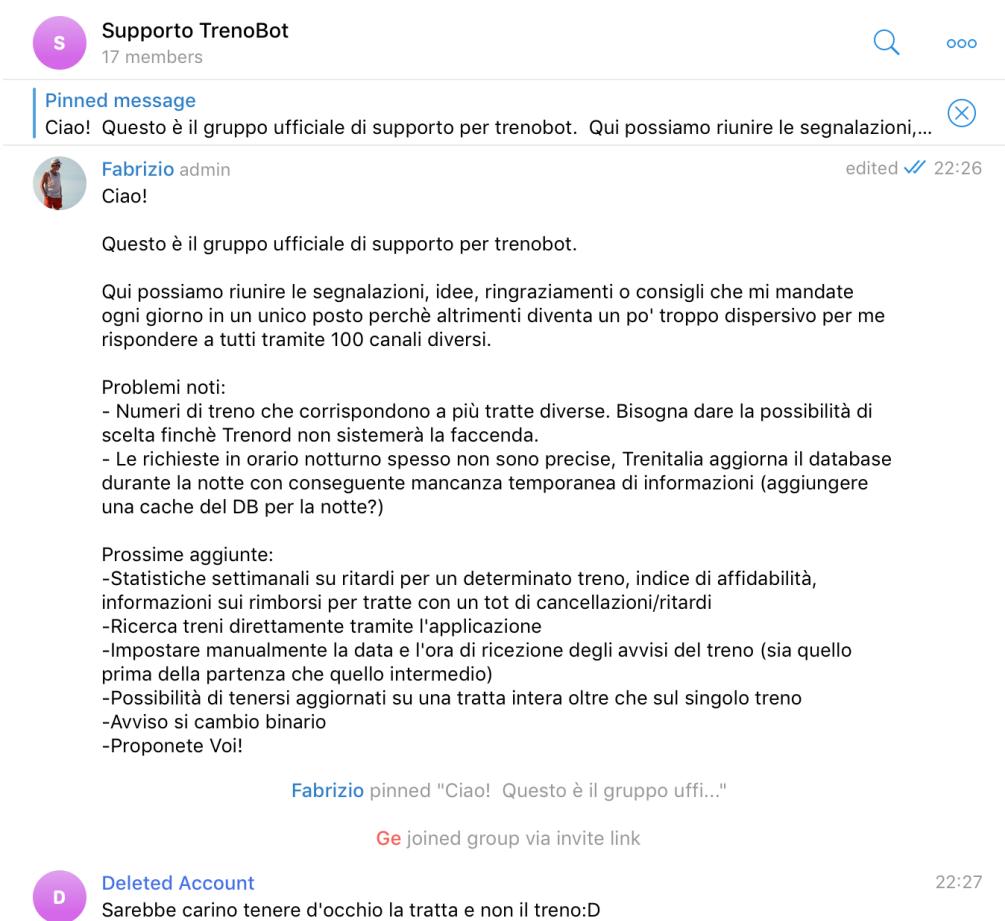


Figura 5.2: La community di Trenobot nel 2017, nella quale gli utenti proponevano aggiunte di funzioni che, allo stato dell'arte attuale, sono poi state sviluppate.

Il grande interesse nei confronti del chatbot non può che rendermi **soddisfatto del lavoro eseguito**, avendo sviluppato un sistema in grado di soddisfare le vere *necessità* dei pendolari.

Seppur tutti gli utenti non siano ancora stati avvisati che il chatbot è tornata in funzione con tutte le migliorie e le funzionalità che essi stessi avevano richiesto (riprogrammandola completamente), è mia intenzione **cercare di diffondere nuovamente il sistema** e ritornare ad avere tanti utenti, soprattutto ora che esso può *sostenere un carico più elevato*.

Guardando al **futuro**, le migliori e l'aggiunta di servizi offerti del chatbot è potenzialmente molto vasta. Sono già soddisfatto di tutto quello che il chatbot è in grado di fare allo stato attuale e sono fiducioso del fatto che con gli utenti e i loro feedback sarà semplice e stimolante **trovare nuove idee**, senza però sbilanciare l'equilibrio tra *numero* di funzionalità e *semplicità* di utilizzo.

Capitolo 6

Manuale Utente

6.1 Primo avvio e menù principale

Al primo avvio, il messaggio presentato all'utente è il **menù principale** come in figura 6.1. Il menù principale può essere richiamato in ogni momento digitando "menu principale" oppure tramite i tasti di scelta rapida a partire dagli altri sotto-menù.



Figura 6.1: Menù principale

Dal menù principale sono accessibili tutte le funzioni della chatbot che vengono di seguito descritte.

6.2 Informazioni in tempo reale di un treno

Selezionando la voce "*Treno Real Time*" dal menù principale in figura 6.1, il chatbot propone il messaggio in figura 6.2 nel quale viene spiegata la sintassi dei comandi da scrivere per ricevere le informazioni in tempo reale su un determinato treno. Il messaggio riporta anche un esempio.

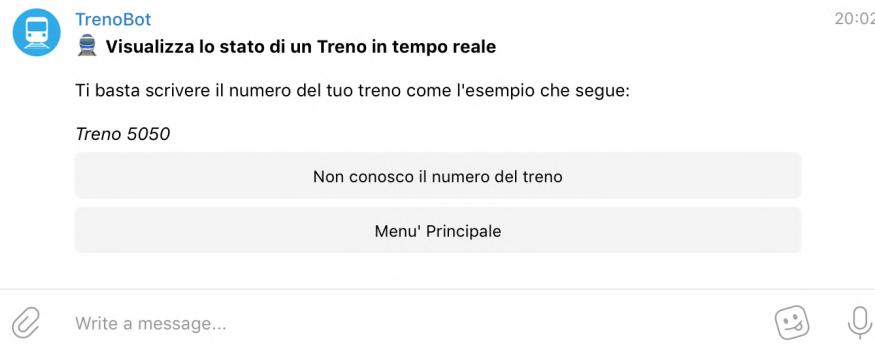


Figura 6.2: Messaggio treno real time

Sono inoltre presenti due pulsanti di scelta rapida:

- **Non conosco il numero del mio treno.** E' un richiamo rapido della funzione "*Ricerca un Treno*" in figura 6.1
- **Menù Principale.** Ritorna al menu principale.

Eseguendo il comando, come descritto dalla chatbot stessa, si ottengono le informazioni in tempo reale sul treno che desidera visualizzare, come in figura 6.3

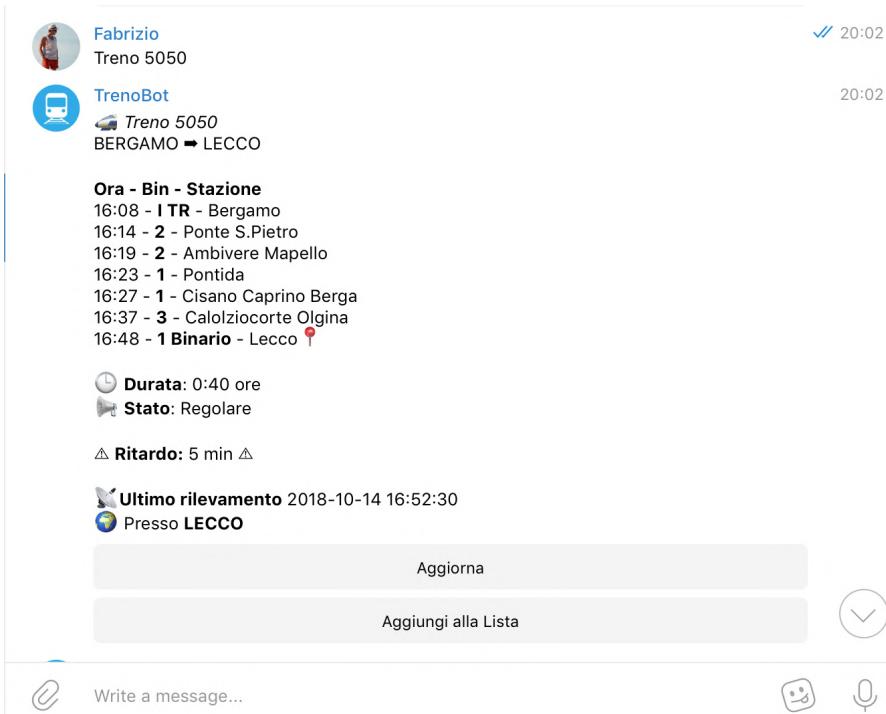


Figura 6.3: Treno 5050 in real time. Il Pallino rosso da parte alla stazione indica l'ultima stazione presso cui il treno è stato rilevato

6.3 Ricerca di una soluzione di viaggio

Selezionando la voce "Ricerca un Treno" dal menù principale in figura 6.1, il chatbot propone il messaggio in figura 6.4 nel quale viene spiegata la sintassi dei comandi da scrivere per eseguire la ricerca. Il messaggio riporta anche degli esempi.



Figura 6.4: Ricerca di una soluzione di viaggio

La data e l'ora sono parametri non obbligatori, e se vengono lasciati in bianco vengono automaticamente considerati uguali alla data e l'ora attuale. E' possibile specificare sia la data che l'ora, oppure solo una delle due.

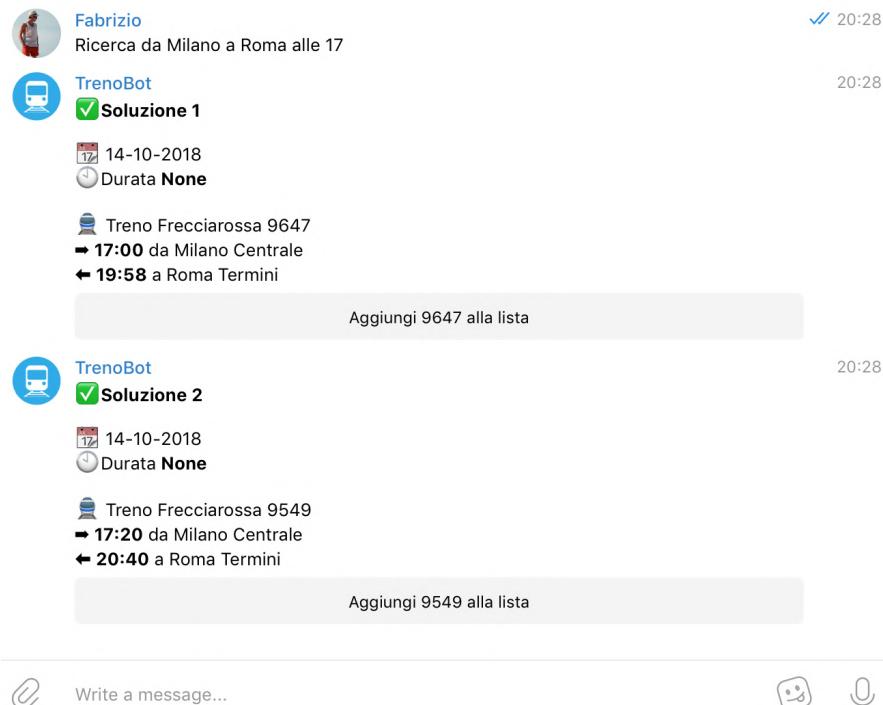


Figura 6.5: Ricerca di una soluzione di viaggio

Eseguendo il comando come in figura 6.5 la chatbot risponde proponendo due soluzioni di viaggio.

Sono inoltre presenti dei pulsanti di scelta rapida che consentono di aggiungere rapidamente il treno cercato alla propria lista dei treni monitorati, funzione che può essere anche fruibile manualmente come vedremo dettagliatamente nella sezione 6.4 .

6.4 Aggiunta di un treno alla lista

Selezionando la voce "*Menù Treni Monitorati*" dal menù principale in figura 6.1, il chatbot propone il messaggio in figura 6.6 che illustra le funzionalità disponibili in questo sotto-menù.

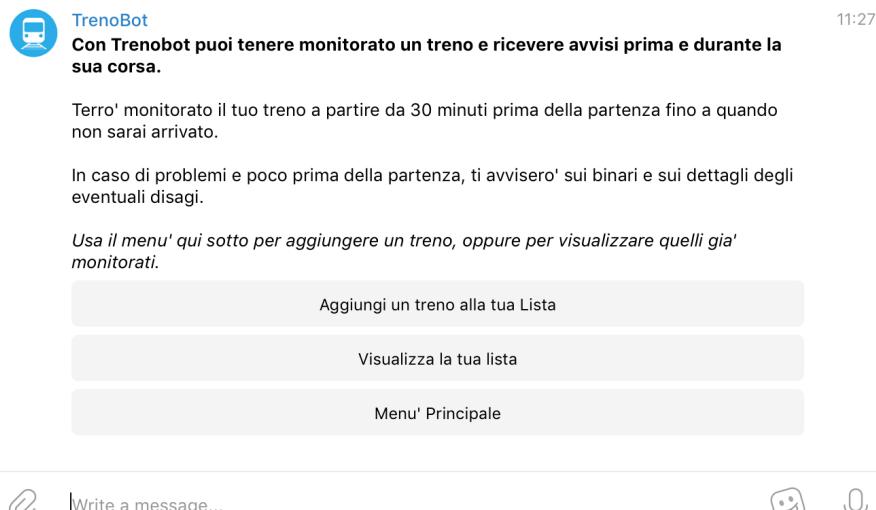


Figura 6.6: Menu per la programmazione di un treno

Selezionando la prima voce "*Aggiungi un treno alla tua lista*" viene mostrato il messaggio in figura 6.7.

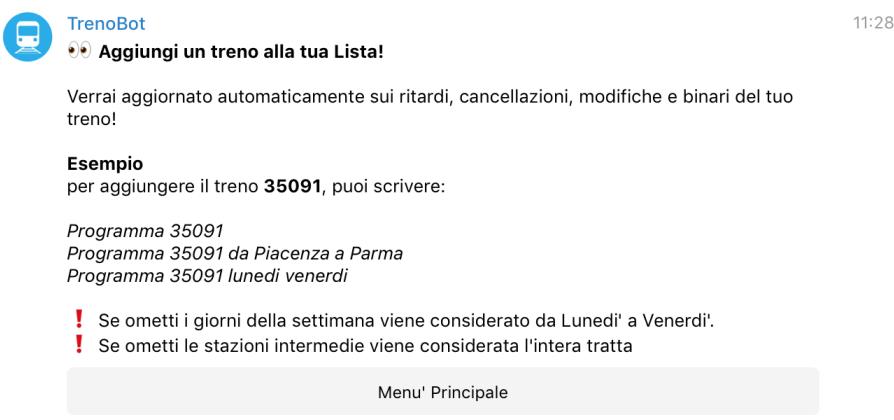


Figura 6.7: Aggiungere un treno alla lista

Capitolo 6. Manuale Utente

Eseguendo un comando, come descritto dal messaggio stesso, è possibile aggiungere un treno alla lista, come in figura 6.8 .

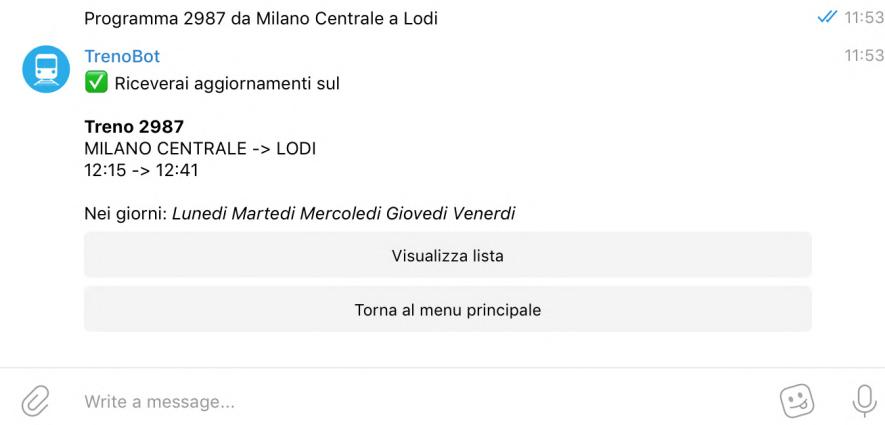


Figura 6.8: Esempio di aggiunta di un treno alla lista

Come indicato nel messaggio in figura 6.7 è possibile inserire delle stazioni intermedie (altrimenti viene considerata l'intera tratta) e i giorni della settimana in cui si è interessati a ricevere gli avvisi (altrimenti vengono considerati tutti i giorni da Lunedì a Venerdì).

Selezionando la voce "*Selezione lista*" sia dai pulsanti in figura 6.8 e in figura 6.6 viene mostrata l'intera lista dei treni monitorati, come in figura 6.9



Figura 6.9: Lista dei treni monitorati

Per ogni treno nella lista è possibile aggiornare lo stato in tempo reale oppure procedere alla rimozione dello stesso tramite i pulsanti dedicati.

6.5 Aggiunta una direttrice alla lista

Selezionando la voce "*Menù Direttrici Monitorate*" dal menù principale in figura 6.1, il chatbot propone il messaggio in figura 6.10 che illustra l'elenco di direttrici monitoratili (in questo caso limitate a 15 per comodità illustrativa).

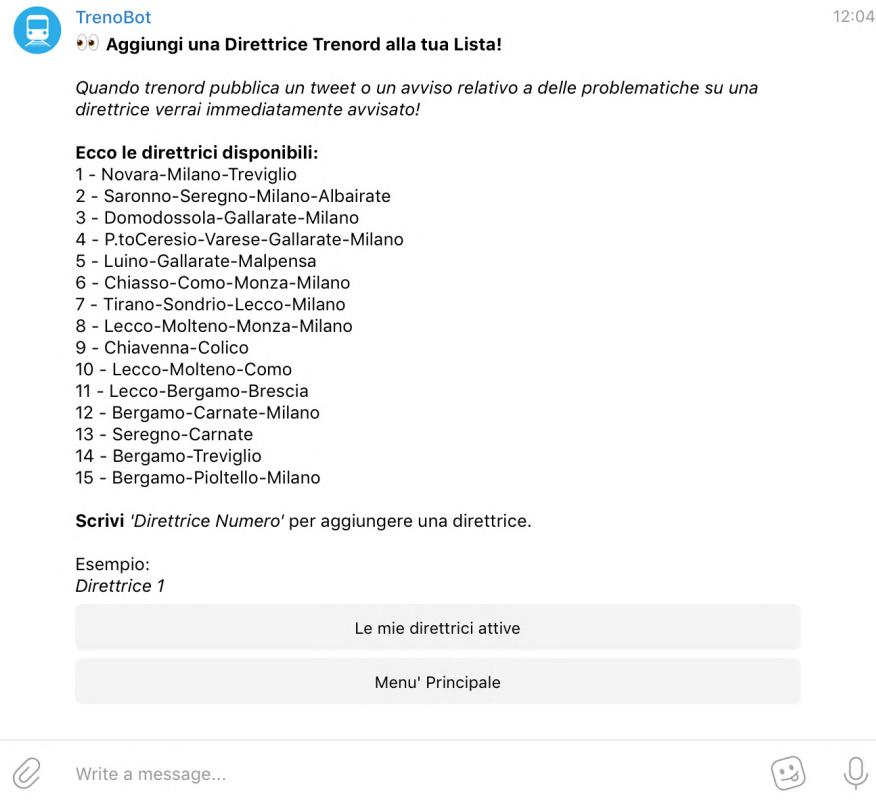


Figura 6.10: Direttrici Trenord monitoratili

Procedendo all'aggiunta di una direttrice come suggerito nell'esempio in figura 6.10 si ottiene la risposta in figura 6.11 .

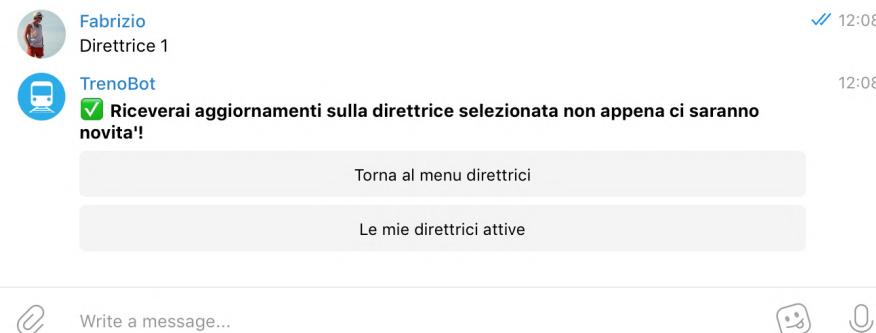


Figura 6.11: Aggiunta della direttrice 1

Selezionando la voce "*Le mie direttrici attive*" sia dai pulsanti in figura 6.11 e in figura 6.10 viene mostrata l'intera lista delle direttrici monitorate come in figura 6.12, da cui è possibile procedere

alla rimozione delle stesse.



Figura 6.12: Lista delle direttrici

6.6 Riepilogo completo dell'account

Selezionando la voce "*Riepilogo completo dei miei avvisi*" dal menù principale in figura 6.1, il chatbot propone il messaggio in figura 6.13 che illustra tutti i treni e tutte le direttrice attualmente monitorate.

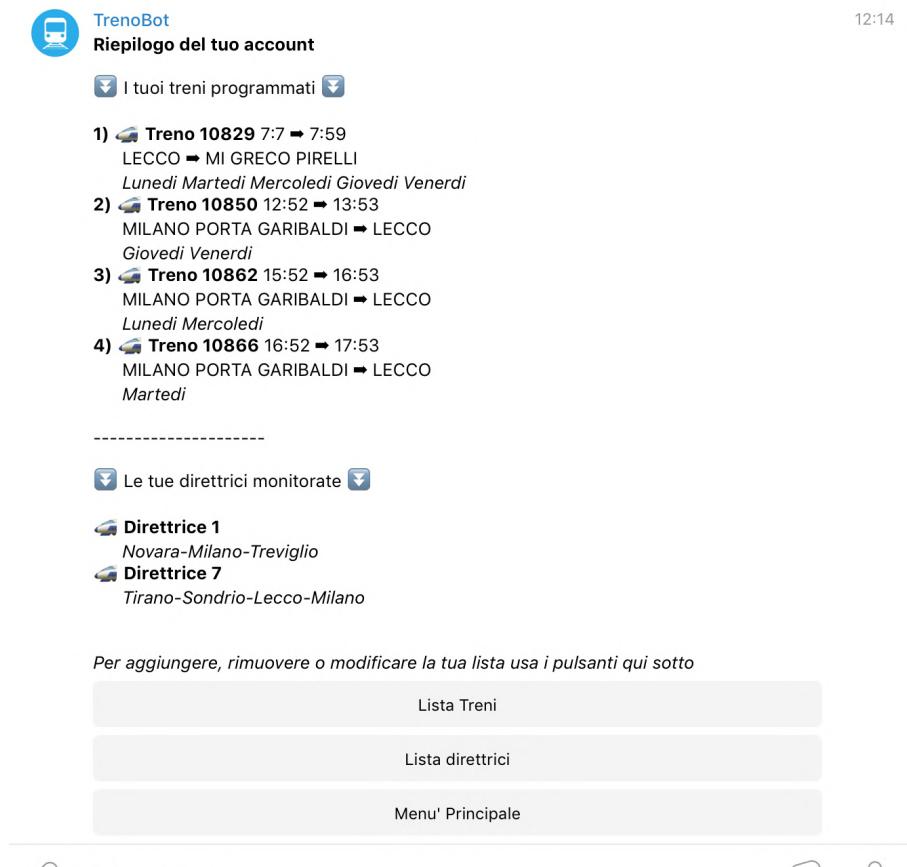


Figura 6.13: Riepilogo dell'account

Dai pulsanti sotto al messaggio è possibile visualizzare le due liste nel dettaglio e, come visto nel manuale alla sezione 6.4 e 6.5, procedere alla modifica delle stesse.

6.7 Esempi di avvisi automatici

Quando un treno viene aggiunto alla propria lista la chatbot:

- *30 minuti* prima della partenza del treno e fino alla fine della corsa monitora lo stato del treno e segnala all'utente eventuali disagi (figura 6.14)
- *5 minuti prima* della partenza del treno invia un messaggio all'utente per informarlo del binario e altre informazioni utili (figura 6.15)



Figura 6.14: Esempio di avviso automatico su una parziale soppressione di un treno monitorato dall'utente prima dell'effettiva partenza dello stesso



Figura 6.15: Esempio di avviso automatico 5 minuti prima della partenza del treno

Capitolo 6. Manuale Utente

Quando una direttrice monitorata dall'utente presenta un disagio, l'utente viene avvisato immediatamente, con un ritardo massimo di 8 minuti rispetto alla pubblicazione da parte di Trenord (figura 6.16 e 6.17)

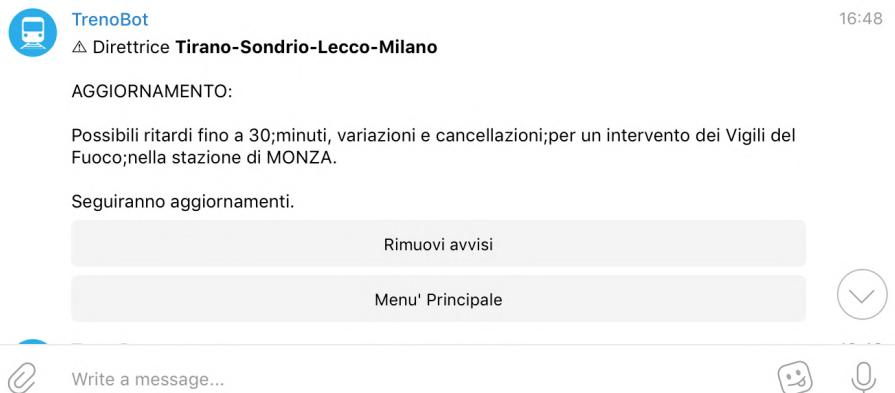


Figura 6.16: Esempio 1 di avviso automatico relativo ad un disagio sulla direttrice



Figura 6.17: Esempio 2 di avviso automatico relativo ad un disagio sulla direttrice

Bibliografia

- [1] Trenord srl. Direttrici in tempo reale. URL <http://www.trenord.it/it/circolazione-e-linee/direttrici.aspx>.
- [2] Nick Lee. Python framework for telegram bot api. URL <https://telepot.readthedocs.io>.
- [3] Mysqldb user's guide. URL <https://pypi.org/project/emoji/>.
- [4] Python. Urlib for python, . URL <https://docs.python.org/3/library/urllib.html>.
- [5] Kevin Wurster Taehoon Kim. Emoji for python. URL <http://mysql-python.sourceforge.net/MySQLdb.html>.
- [6] Python. Json encoder and decoder, . URL <https://docs.python.org/2/library/json.html>.
- [7] The MariaDB Foundation. Mariadb. URL <https://mariadb.org/>.
- [8] Deutsche Bahn. Udeutsche bahn api portal. URL <https://developer.deutschebahn.com/store/>.
- [9] Trenitalia spa. Viaggatreno. URL <http://www.viaggatreno.it/>.
- [10] Google. Devtools console, . URL <https://developers.google.com/web/tools/chrome-devtools/console/>.
- [11] Fabrizio Olivadese. Trenobot, . URL <https://github.com/Fabroll/TrenoBot>.
- [12] Telegram. Telegram apis. URL <https://core.telegram.org/api>.
- [13] Google. Google cloud platform solutions, . URL <https://cloud.google.com/solutions/>.
- [14] Fabrizio Olivadese. Aggiungi il chatbot a telegram, . URL <https://telegram.me/TrainOllyBot>.
- [15] Debian org. URL <https://www.debian.org/>.