

Generatore di frequenze con vari accessori gestito via web

Titolo del progetto: Generatore di frequenze con vari accessori gestito via web
Alunno/a: Giulio Bosco
Classe: SAMT I4AA
Anno scolastico: 2019/2020
Docente responsabile: Fabrizio Valsangiacomo



SAMT – Sezione Informatica

Gestore di frequenze con accessori gestito via web

Pagina 2 di 71

1	Introduzione.....	5
1.1	Informazioni sul progetto.....	5
1.2	Abstract.....	5
1.3	Scopo.....	5
2	Analisi.....	6
2.1	Analisi del dominio.....	6
2.2	Analisi e specifica dei requisiti.....	6
2.3	Use case.....	8
2.4	Pianificazione.....	9
2.4.1	Pianificazione Progetto Primo Semestre.....	9
2.5	Analisi dei mezzi.....	10
2.5.1	Software.....	11
2.5.2	Hardware.....	11
3	Progettazione.....	12
3.1	Design dell'architettura del sistema.....	12
3.1.1	Restful API.....	13
3.2	Design dei dati e database.....	14
3.3	Design delle interfacce.....	17
3.4	Design procedurale.....	18
3.4.1	Applicativo (back-end).....	18
3.4.1.1	connection to database (jdbc).....	19
3.4.1.2	data access object (dao).....	20
3.4.1.3	Queries.....	22
3.4.1.4	dati (model).....	23
3.4.1.5	json (modeljson).....	25
3.4.1.6	servlets (servlets & servlets. Help).....	26
3.4.1.7	data servlets (servlets.data).....	27
3.4.1.8	actions servlets (servlets.action).....	28
3.4.1.9	authentication (auth).....	28
3.4.1.10	arduino connection controll (acc).....	29
3.4.2	Applicativo (front-end).....	29
3.4.3	Acc Client (Arduino).....	29
4	Implementazione.....	31
1.1	Database.....	31
1.1.1	Tables.....	31
1.1.2	Triggers.....	33
4.1	Backend.....	35
1.1.3	Gradle.....	35
1.1.4	JDBC.....	37
1.1.5	DAO.....	40
1.1.6	Queries:.....	46
1.1.7	Models:.....	48
1.1.8	Servlets.....	49
4.2	Frontend.....	57
4.3	Arduino.....	60
4.3.1	Arduino – python.....	60
4.3.2	Arduino – c, ino.....	62
4.4	Installazione Raspberry.....	64
5	Test.....	66
5.1	Protocollo di test.....	66
5.2	Risultati test.....	68
5.3	Mancanze/limitazioni conosciute.....	68
6	Consuntivo.....	69
7	Conclusioni.....	70

7.1	Sviluppi futuri.....	70
7.2	Considerazioni personali.....	70
8	Glossario	70
9	Bibliografia.....	71
9.1	Sitografia	71
10	Allegati	71

1 Introduzione

1.1 Informazioni sul progetto

Docente Responsabile: Fabrizio Valsangiacomo
 Apprendista: Giulio Bosco, SAM I4AA
 Azienda formatrice: Scuola Arti e Mestieri di Trevano
 Orientamento: 88602 – Informatica Aziendale

Progetto diviso in due fasi, primo e secondo semestre:

Primo semestre:

Data inizio progetto: 05.09.2019
 Data consegna progetto: 20.12.2019
 Ore a disposizione: 174

Secondo semestre:

Data inizio progetto: 23.01.2020
 Data consegna progetto: 06.04.2020
 Ore a disposizione: 154

1.2 Abstract

The scope of the project is to create a WEB interface for manage an ultrasonic frequencies generator. There is available a circuit based on Arduino UNO and a chip AD9833 for generation of ultrasonic waves and the code working with the Arduino UNO. We want to remote manage with options, with a remote that can start and stop the generator. A microphone, if he listens for a sound higher than limit in decibels, it starts the generator and a timer after that it will turn back off. The last options is a WEB application, with that you can turn it on or off, change the frequency, the decibels of the microphone and his timer.

For develop the application have been used some different technologies, MySQL (data storage), Java (web-app – server side), AngularJS (web app – client side) and a mix of Python and C (the Arduino version) on the Arduino Side.

The project is not working because of a communication problem between Arduino YUN and the chip AD9833 for the generation of the waves. Probably the problem is in the use of the SPI library.

1.3 Scopo

Creare un'interfaccia di gestione per un generatore di frequenze ultrasoniche, con la possibilità di essere gestito da più utenti, quindi si necessita anche una piattaforma per gestire gli utenti.

Il generatore deve poter essere acceso e spento con un telecomando, captando un suono (un suono qualunque più alto di una determinata soglia, in decibel), tramite interfaccia WEB.

2 Analisi

2.1 Analisi del dominio

Questo prodotto serve per gestire un generatore di onde ad ultrasuoni tramite una pagina WEB, un telecomando ed un suono. Attualmente sul mercato non ho trovato nessun prodotto simile.

Il generatore è già esistente, ma per regolarlo bisogna ruotare un potenziometro. Il quale deve venir sostituito dall'applicativo e bisogna aggiungere la possibilità di accenderlo e spegnerlo tramite un telecomando, inoltre all'avvio tramite suono. **Parte del prodotto è stata già sviluppata nel progetto del primo semestre. Il quale non è stato finito. Per cui è stato esteso anche al secondo semestre.**

2.2 Analisi e specifica dei requisiti

ID: REQ-01	
Nome	Gestione generatore tramite WEB
Priorità	1
Versione	1.0
Note	-
	Sotto requisiti
Sub REQ 1	0-25'000 Hz
Sub REQ 2	Gestione Wireless del generatore
Sub REQ 3	Regolazione della frequenza
Sub REQ 4	Accensione/Spegnimento del generatore
Sub REQ 5	Mantenimento memoria
Sub REQ 6	Scatola finale cablata e protetta (Priorità 3)
Sub REQ 7	Minima attenuazione segnale (Priorità 3)

ID: REQ-02	
Nome	Gestione generatore tramite Telecomando
Priorità	2
Versione	2.0
Note	-
	Sotto requisiti
Sub REQ 1	Tasto Start
Sub REQ 2	Tasto Stop

ID: REQ-03	
Nome	Gestione generatore tramite Rumore
Priorità	2
Versione	2.0
Note	-
	Sotto requisiti
Sub REQ 1	Gestione tramite decibel
Sub REQ 2	Timer per spegnimento

ID: REQ-04	
Nome	Pagina WEB - Amministrazione utenti
Priorità	1
Versione	2.0
Note	-
	Sotto requisiti
Sub REQ 1	Creare utente
Sub REQ 2	Modifica utente
Sub REQ 3	Eliminare utente
Sub REQ 4	Modificare tipo di utente (amministratore, normale)
Sub REQ 5	Minimo un amministratore
Sub REQ 6	Password provvisoria (Priorità 3)

ID: REQ-05	
Nome	Wireless
Priorità	1
Versione	1.0
Note	Rete wireless autonoma
	Sotto requisiti
Sub REQ 1	Sicurezza di base per una rete wireless

ID: REQ-06	
Nome	Rifiniture del progetto
Priorità	1
Versione	1.0
Note	Riprendere i precedenti requisiti e rieseguire uno studio di fattibilità, per finire il progetto.
	Sotto requisiti
Sub REQ 1	Rieseguire lo studio di fattibilità
Sub REQ 2	Implementare parti mancanti
Sub REQ 3	Migliorare documentazione

2.3 Use case

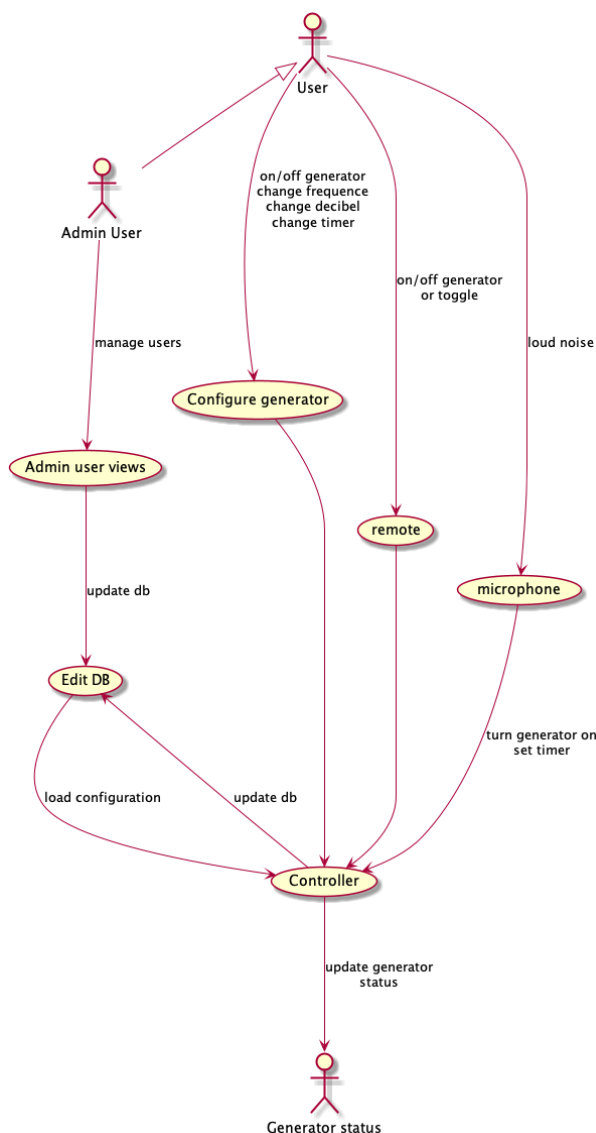


Figura 1 - Use Case

Questo progetto ha come scopo di sviluppare la gestione di un generatore di frequenze tra 0 e 25'000 Hz, il quale deve poter essere gestito in maniere diverse. La frequenza deve poter essere regolata tramite una pagina web, solamente dall'utente amministratore. Mentre per quanto riguarda l'accensione e lo spegnimento del generatore, deve poter essere fatto tramite la pagina web (da un utente di base), tramite un telecomando e tramite un suono regolato in decibel (che di deve poi spegnere allo scadere di un timer).

Il diagramma mostra al centro il **controller**, che sarebbe il sistema che gestisce il nostro prodotto. Il quale si basa sulle informazioni del database. Un utente dell'applicativo può, configurare i parametri di base del generatore, che sono la frequenza che deve venir generata dal generatore, i decibel a cui deve rispondere il microfono, il timer del microfono e deve poter accendere o spegnere esso.

Inoltre deve poter accendere o spegnere tramite un telecomando (**remote**) e tramite un microfono il generatore. Quando viene acceso il generatore tramite il microfono deve partire un timer, dopo il quale automaticamente il generatore viene spento.

Mentre per quanto riguarda gli utenti amministratori, devono poter creare, modificare ed eliminare altri utenti.

2.4 Pianificazione

2.4.1 Pianificazione Progetto Primo Semestre

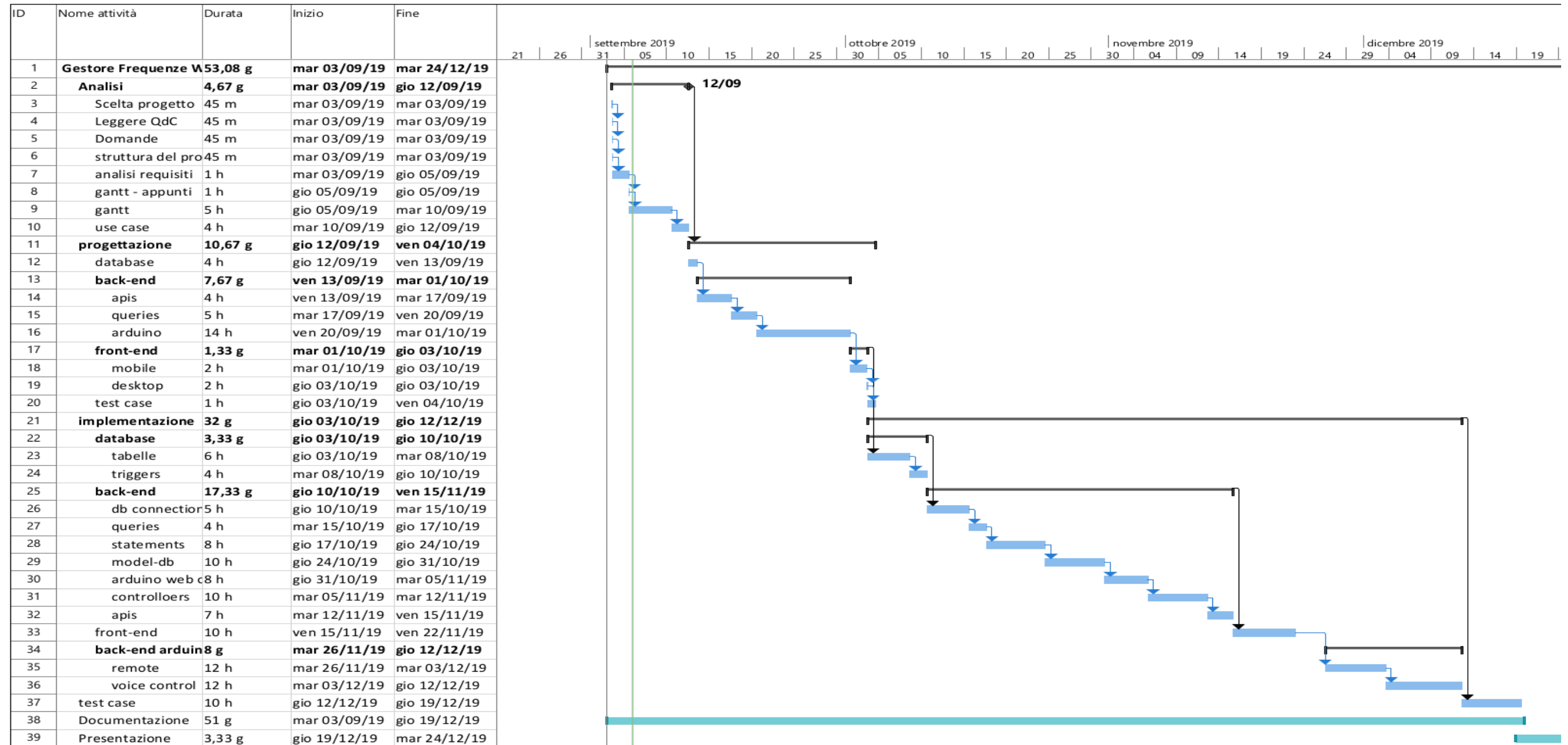
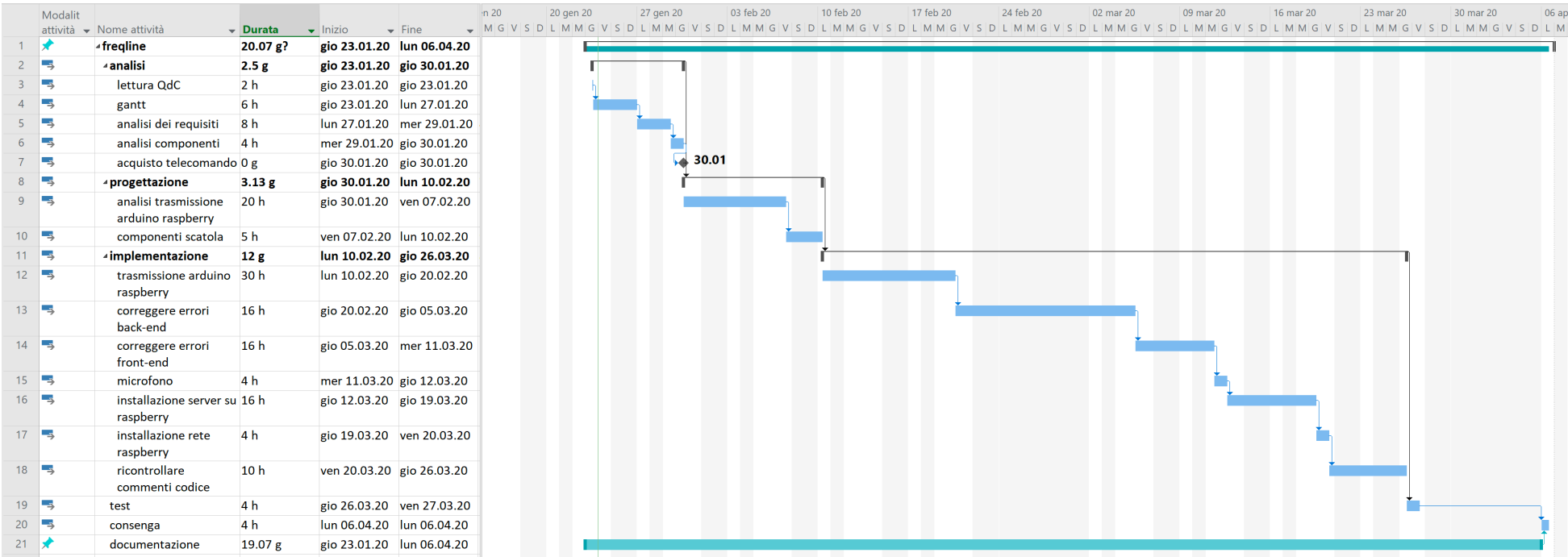


Figura 2: Diagramma di GANTT

Per lo sviluppo di questo progetto, ho pianificato lo sviluppo con 5 giorni di analisi, 10 di progettazione e 32 di implementazione, per giorni si intendono i giorni di progetto, che sono il martedì, giovedì e venerdì, in tutti e tre i giorni saranno a disposizione 4 ore di lezione (45 minuti per ora), ovvero un totale di 3 ore. La prima pianificazione che avevo pensato, sarebbe finita a metà novembre, nella quale sarebbe stato compreso l'utilizzo di un framework nello sviluppo del progetto nel lato back-end. Siccome avrei finito il progetto molto prima ho deciso che per una volta avrei scritto tutto il codice da solo. Quindi il codice di astrazione del database lo ho scritto a mano, così come la costruzione delle interfacce di comunicazione che vi sono fra l'applicativo back-end e front-end. Ho fatto questa scelta siccome ho un progetto relativamente semplice, quindi sviluppare tutto il codice che solitamente è in un framework mi avrebbe permesso di capire come funzionano la maggior parte dei framework a basso livello, questo mi porterebbe molta esperienza e conoscenza.



2.4.2 Pianificazione Progetto Secondo Semestre



Questo progetto è la continuazione del progetto del primo semestre, lo scopo di esso è finire le parti mancanti del progetto e migliorare le parti già presenti. Il progetto comprende una parte di analisi, dove verranno analizzati i componenti mancanti e verranno scelti i componenti da acquistare, dopo di che verrà fatto uno studio sulle soluzioni possibili per implementare le parti mancanti, in particolare la trasmissione fra il Raspberry PI e l'Arduino. Per continuare con la progettazione di esso e della scatola nella sua versione finale.

Vi è anche presente una buona parte di implementazione, dove verranno implementate le parti precedentemente progettate e verranno sistemati i problemi presenti nella vecchia versione del progetto.

Nel diagramma si possono notare dei giorni bianchi, che sono i giorni in cui si lavora al progetto e dei giorni grigi che sono di vacanza, nei quali sono comprese le vacanze di carnevale, le 2 giornate di reclutamento al militare che sono tenuto a fare e la gita di 4°.

2.5 Analisi dei mezzi

2.5.1 Software

- **Arduino IDE (v1.8.9)**
Software Sviluppato ufficiale per lo sviluppo di codice per Arduino sviluppato dalla Arduino Foundation. Il quale è stato utilizzato per compilare e caricare il codice sulla scheda.
- **VS Code (v1.41.1)**
Code editor, utilizzato per scrivere la maggior parte del codice.
- **plantuml (Version 1.2019.9)**
Software utilizzato per fare il diagramma Use-case e i diagrammi delle classi. Utilizzato per la progettazione
- **WPS Writer (Version: 6.2.5.2)**
Software per modifica di documenti di testo, utilizzato per scrivere la documentazione ed i diari.
- **Java (v1.8)**
 - **Libreria org.gretty (v2.2.0)**
Libreria utilizzata per la creazione di un web server di base per gradle e libreria java servlet.
 - **Libreria Junit (v4.12)**
Libreria per la scrittura dei test.
 - **Libreria JDBC (MySQL Connector Java – v8.0.11)**
Libreria di connessione Java al database.
 - **Libreria JSON (org.json:json:20171018)**
Libreria gestione Oggetti JSON.
 - **Guava (v27.0.1)**
Libreria di Google, per l'ottimizzazione di Gradle.
 - **Servlets API (v3.1.0)**
Libreria per api delle servlet java.
- **AngularJS (v1.6.9)**
Framework front-end.
- **jQuery (v3.2.1)**
Libreria java-script, per velocizzare azioni sugli elementi html.
- **Bootstrap (v4.3.1)**
Libreria CSS.

2.5.2 Hardware

- Mac Book Pro 2018 Intel Core i7 CPU 3.1GHz RAM 16GB
- Arduino UNO (Rev 3)
- Arduino YUN (Rev 2)
- Raspberry PI 3 Model B
- Circuito AD9833
- Circuito amplificatore

3 Progettazione

3.1 Design dell'architettura del sistema

Il progetto è sviluppato su quattro diversi elementi, che sono messi in evidenza nello schema sottostante:

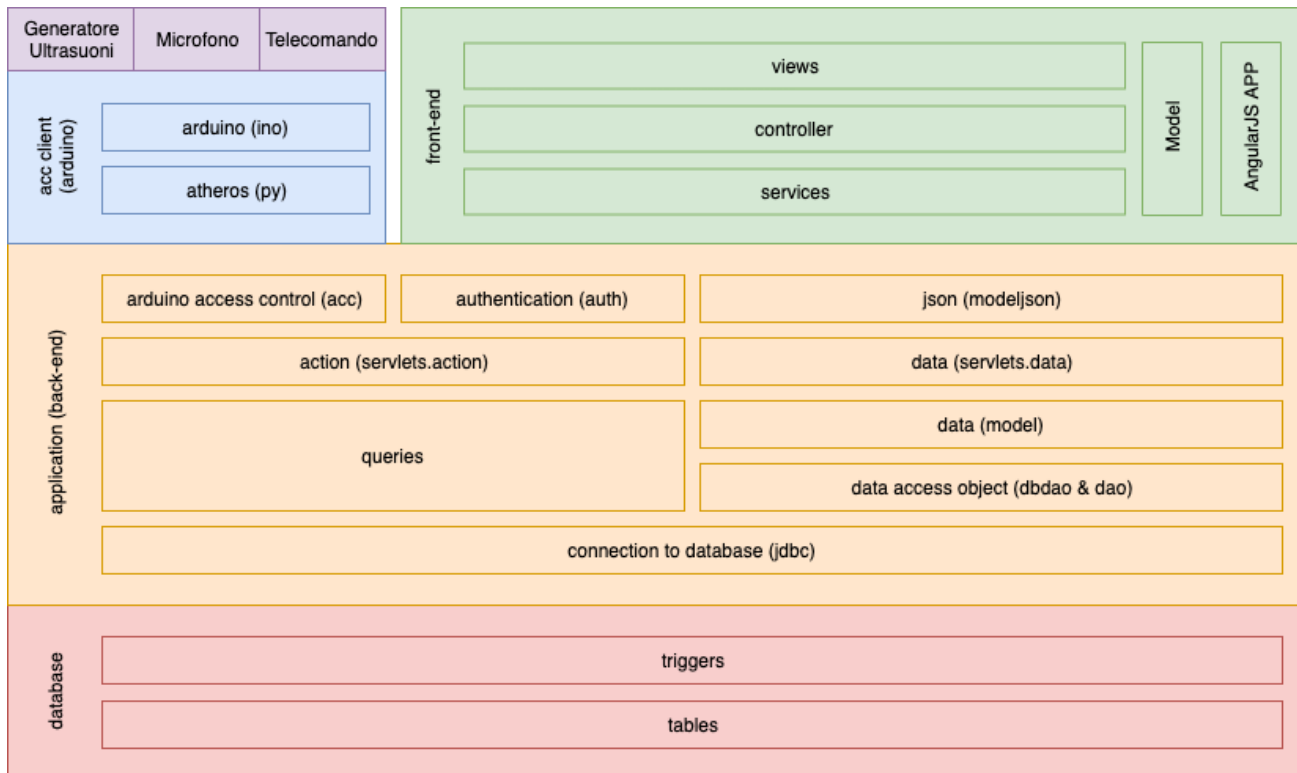


Figura 3 - Schema componenti del progetto

Alla base di tutto vi è il database nel quale vengono salvati tutti i dati dell'applicativo, il quale è diviso in 2 due parti, il back-end ed il front-end.

Tutto il sistema, verrà strutturato in maniera modulare, in modo che tutti gli elementi del progetto siano indipendenti, quindi siano più facili da testare, singolarmente, per cercare di avere meno problemi possibili durante il corso del progetto, che siano anche più facili da mantenere e modificare in futuro e che possano essere riutilizzati in progetti futuri. Alcuni moduli del progetto verranno ripresi da vecchi progetti che ho svolto in questi anni alla SAMT e probabilmente riscritti siccome ora ho più conoscenze ed esperienza, viene utilizzato lo stesso concetto, ma riscrivendo il codice solamente in parte oppure totalmente.

I vari moduli, possono avere dei sotto moduli, per semplificare ancora lo sviluppo ed il mantenimento dell'applicativo.

Per ogni modulo è stata fatta una progettazione, in alcuni più approfondita mentre in altri meno. Questo perché a dipendenza dei moduli vi sono più o meno complessi, mentre per quelli che si pensa di prendere da dei vecchi progetti non è proprio stata fatta, verranno fatti degli adattamenti direttamente durante lo sviluppo.

Per esempio, la parte relativa al *acc client (Arduino)* verrà preso dal progetto *domotics* (<https://github.com/giuliobosco/domotics>), come il modulo *application (front-end)*.

3.1.1 Restful API

Questo applicativo WEB, è sviluppato suddiviso in 2 elementi, front-end e back-end, nel lato front-end vi sono le grafiche dell'applicativo e l'interpretazione dei dati. Mentre nel lato back-end vi è l'interazione con il database, quindi la creazione e l'aggiornamento dei dati. Per costruire l'applicativo con questa struttura ho deciso di utilizzare le best practices del trend attuale dello sviluppo web.

Quindi per comunicare fra i due elementi dell'applicativo utilizzerò le Restful API che fondamentalmente sono delle stringhe in formato JSON, le quali possono essere richieste tramite delle richieste http, con i suoi vari metodi. Per progettare le API, mi sono documentato su restapitutorial.com.

Esempio di Restful API (<http://localhost/api/v1/users>):

```
{
  "users": [
    {
      "username": "giulio.bosco",
      "firstname": "Giulio",
      "lastname": "Bosco"
    },
    {
      "username": "fabrizio.valsangiaco",
      "firstname": "Fabrizio",
      "lastname": "Valsangiaco"
    }
  ]
}
```

Ad ogni API viene associato un indirizzo nel web server, come per esempio:

<http://localhost/api/servletAddress>

Il protocollo HTTP prevede la possibilità di implementare diversi metodi per eseguire le richieste e diverse possibili risposte per ogni richiesta. Le richieste più comuni sono GET, POST, PUT e DELETE, le quali sono implementate nei relativi metodi per ogni servlet:

- doGet: serve per eseguire la richiesta GET
serve per richiedere gli elementi della api, se termina con un numero questo deve essere l'id dell'elemento (riferimento con SQL: `SELECT * FROM table WHERE id=?`). Altrimenti ritorna tutti gli elementi (riferimento con SQL: `SELECT * FROM table`).
- doPost: serve per eseguire la richiesta POST
serve per creare un nuovo elemento (riferimento SQL: `INSERT INTO table (?)`).
- doPut: serve per eseguire la richiesta PUT
serve per aggiornare un elemento (riferimento SQL: `UPDATE table SET x=? WHERE id=?`).
- doDelete: serve per eseguire la richiesta DELETE
serve per eliminare un elemento (riferimento SQL: `DELETE FROM table WHERE id=?`).

Ognuna di queste richieste può avere una serie di risposte, le quali sono rappresentate da un numero e da una stringa di descrizione, il numero è sempre composto di 3 cifre, la prima indica il tipo di risposta, che può essere mentre le seconde 2 cifre identificano la risposta:

- 1xx Informational: risposta al client di tipo informativo
- 2xx Success: la richiesta ha una risposta con esito positivo
- 3xx Redirection: ridirezionamento su un'altra pagina.
- 4xx Client Error: il client ha fatto una richiesta non valida
- 5xx Server Error: la richiesta ha provocato un errore sul server

Qui sotto sono elencati i metodi più frequenti ed utilizzati (soprattutto per quanto riguarda le Restful API)

ID Risposta	Stringa	Descrizione
200	OK	la richiesta ha una risposta con esito positivo
201	CREATED	la richiesta ha una risposta con esito positivo, creato (p.s.: creato record MySQL)
204	NO CONTENT	risposta con esito positivo, ma non ha contenuto
304	NOT MODIFIED	redirect non modificato
400	BAD REQUEST	richiesta sconosciuta
401	UNAUTHORIZED	non autorizzato per eseguire la richiesta
403	FORBIDDEN	richiesta possibile ma non accettabile dal server, con autenticazione non cambia
404	NOT FOUND	elemento non trovato
405	NOT ACCEPTABLE	richiesta non accettabile
409	CONFLICT	richiesta non processabile, perché contiene dei conflitti (conflitti nella modifica)
500	SERVER ERROR	errore nel server, la richiesta ha provocato un errore nel server
501	NOT IMPLEMENTED	l'elaborazione della richiesta non è ancora stata implementata

3.2 Design dei dati e database

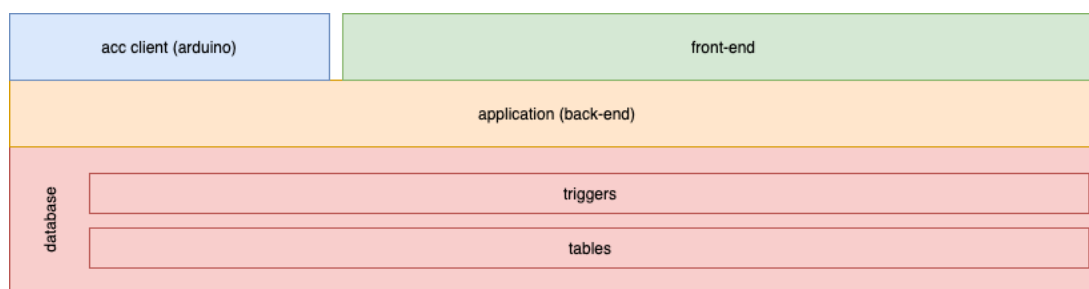


Figura 4 - Schema componenti database

Per lo sviluppo del database, ho prima di tutto creato il minimo indispensabile per il progetto, quindi tutte le tabelle di cui necessito, tutti gli attributi, senza il quale il progetto non funziona.

Lista delle tabelle:

- generators
- users
- groups
- permissions

Dopo di che ho pensato potesse essere una buona idea tenere traccia delle operazioni eseguite sul database. Questo perché è dal lato della piattaforma WEB, il progetto è piccolo. Quindi potrei investire del tempo nello sviluppare questa parte del progetto, che potrebbe comunque essere riutilizzata in qualunque progetto.

Per eseguire i log delle azioni effettuate sulle banche dati, vi sono diversi modi:

1. Eseguire il log delle azioni di tutte le tabelle in una tabella di log, nella quale si inserisce la query eseguita, l'autore, la data e l'ora.
2. Eseguire il log in una tabella dedicata per ogni tabella, nella quale si inseriscono tutte le azioni che vengono eseguite sulle tabelle.
3. Eseguire il log nell'applicativo, quindi inserire una parte del software che esegua il log del database.
4. Si potrebbe utilizzare un software di monitoraggio della banca dati.

Analizzando queste opzioni ho trovato questi pro e questi contro, di ogni metodologia.

Metodo	Punti a favore	Punti a sfavore
1	<ul style="list-style-type: none"> Semplice implementazione e integrazione con la banca dati Facile ricostruzione del database 	<ul style="list-style-type: none"> Difficile creare dei report sui log
2	<ul style="list-style-type: none"> Facilita generazione di report sulle azioni eseguite sul database mantiene separati i dati 	<ul style="list-style-type: none"> Implementazione del database più complessa
3	<ul style="list-style-type: none"> Facile implementazione del database 	<ul style="list-style-type: none"> Complica la struttura del software Nel caso in cui vi fossero in futuro più elementi che interagiscono con lo stesso database, i log non sarebbero più autentici
4	<ul style="list-style-type: none"> Semplificherebbe il codice 	<ul style="list-style-type: none"> Dovrei prendere del tempo per imparare ad utilizzare un altro software Richiederebbe più risorse

Analizzando questi punti, ho deciso di utilizzare il secondo metodo.

Del database, vi sono 2 schemi. Uno che contiene lo schema di base, quindi solamente le informazioni rilevanti per il progetto. Ed uno completo che comprenderà anche gli attributi e le tabelle legati al log.

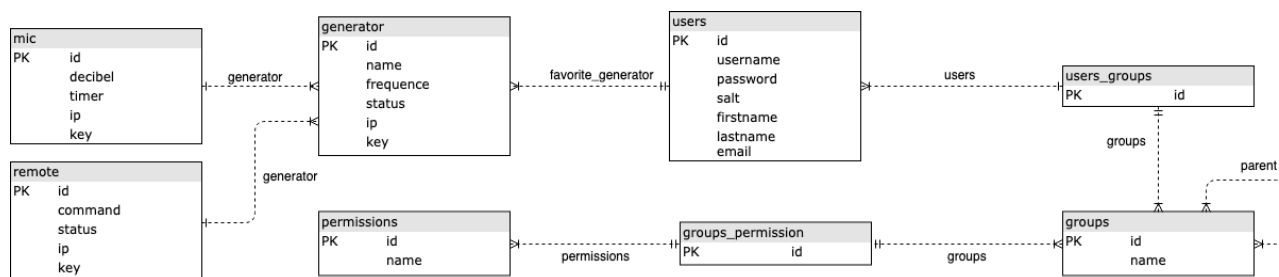


Figura 5 - Diagramma ER semplificato

In questo schema vi sono le tabelle con gli attributi minimi, per far funzionare l'applicativo. Vi è una tabella `generator` che dà la possibilità di inserire diversi generatori, i quali hanno un nome, una frequenza alla quale devono lavorare, uno stato, un IP ed una key (pensata per creare una comunicazione sicura fra il server ed i controller degli altoparlanti). Questo per permettere al progetto di essere espandibile.

Dopo di che vi sono le tabelle `users`, `groups`, e `permissions`, che servono per gestire i permessi. fra le varie tabelle vi sono anche le tabelle ponte, per permettere le relazioni molti a molti.

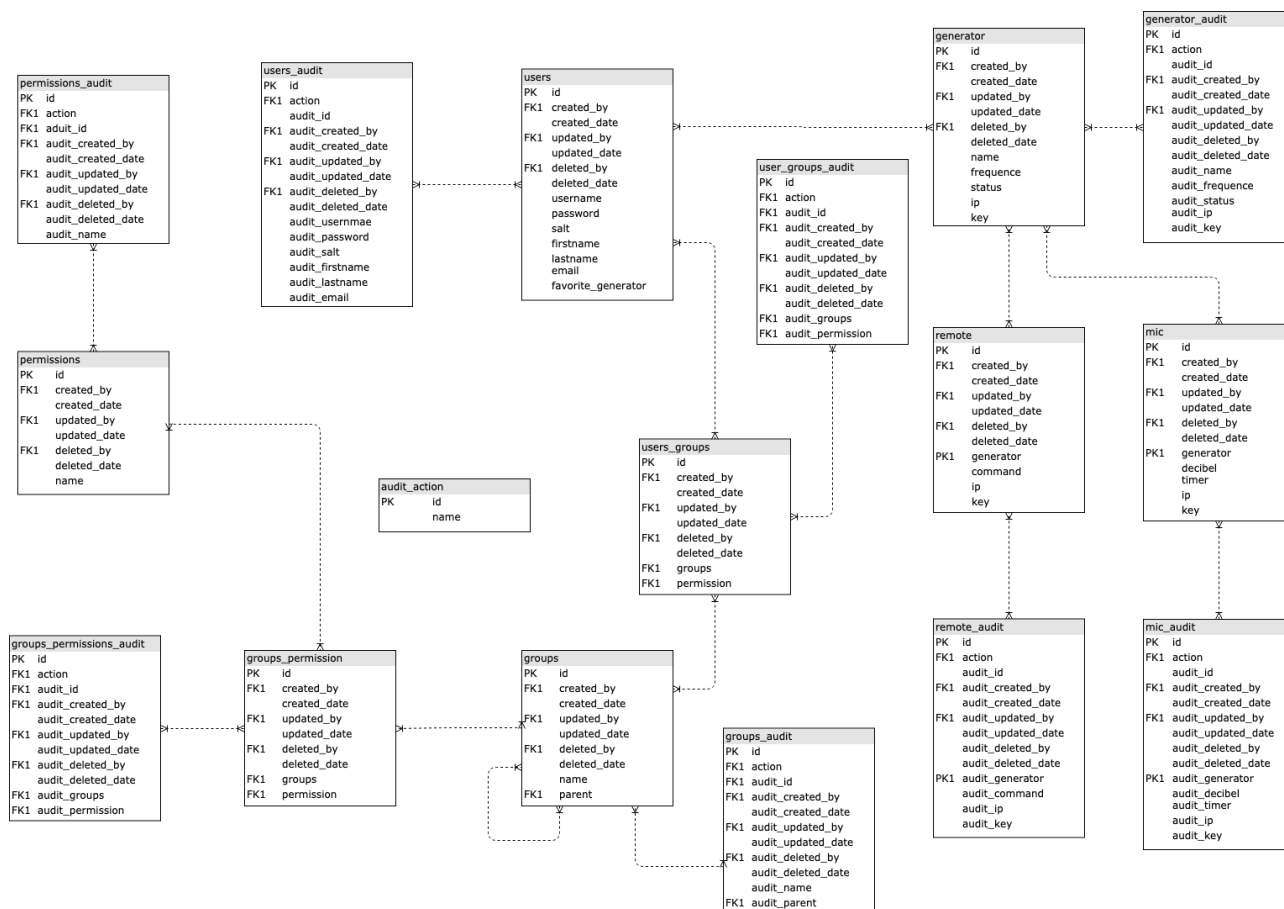


Figura 6 - Diagramma ER esteso

In questo schema si possono notare molte più tabelle e più attributi, per ogni tabella che vi era nello schema antecedente, 6 attributi, che servono per salvare la data e l'autore delle azioni principali che si possono fare sul database (create, update, delete). Inoltre, vi è un'altra tabella, nella quale si inseriscono gli audit, cioè gli stessi parametri, con in più l'azione eseguita ed un id per ogni audit. Le relazioni fra le tabelle di audit e la tabella `audit_action` non sono rappresentate, come le relazioni fra tutti i campi discussi prima e la tabella `users`, questo per permettere una miglior leggibilità dello schema.

3.3 Design delle interfacce

Le interfacce grafiche sono state pensate per dispositivi mobile, e poi adattate a desktop, questo perché la maggior parte degli utenti utilizza dispositivi mobili, con schermi lunghi e stretti che richiedono l'utilizzo delle dita come puntatore, quindi si necessitano icone grandi. Poi le interfacce sono state adattate anche ai dispositivi mobili.

Le interfacce principali sono quella di login (per loggarsi nell'applicativo), quella degli utenti (lista di utenti), quella di modifica dei dati degli utenti e quella della gestione del generatore di frequenze.

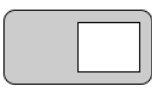
<div> Freqline Gestione Generatore di frequenze via web </div> <div> <input type="text" value="username.txt"/> <input type="password" value="password.pwd"/> <div style="color: red; font-size: small;">error</div> <input type="button" value="login.btn"/> </div>	<div> <div>status users new</div> <div>  </div> <div> <input type="text" value="5000 MHz"/> <input type="text" value="60 db"/> <input type="text" value="3600 s"/> <input type="button" value="save"/> </div> </div>	<div> <div>status users new</div> <table border="1"> <tr><td>Admin 0</td></tr> <tr><td>User 0</td></tr> <tr><td>Admin 1</td></tr> <tr><td>User 1</td></tr> </table> </div>	Admin 0	User 0	Admin 1	User 1	<div> <div>status users new</div> <div> <input type="text" value="username.txt"/> <input type="text" value="nome.txt"/> <input type="text" value="cognome.txt"/> <input type="text" value="email.txt"/> <input type="text" value="gruppo.txt"/> <input type="text" value="password.txt"/> <input type="button" value="ok"/> </div> </div>
	Admin 0						
User 0							
Admin 1							
User 1							
copy giulio bosco							

Figura 7 - Design interfacce grafiche

Le interfacce grafiche principali sono 4, una per il login nell'applicativo, una per gestire il generator, una con l'elenco delle pagine ed una per la modifica dei dati.

- Mockup 0: Si vede il nome dell'applicativo, con sotto 2 campi di input di tipo text box, una per lo username, ed una per la password. Sotto vi è un campo di testo di colore rosso solitamente nascosto, che mostra gli errori, come la password sbagliata ed infine vi è un input di tipo submit per eseguire il login.
- Mockup 1: È la pagina di gestione del generatore di frequenze, nel quel vi è un interruttore che cambia lo stato del generatore, poi vi è sino 3 input per numeri, i quali servono rispettivamente per la frequenza del generatore (in megahertz). Poi vi è un input per la regolazione dei decibel del microfono ed uno per i secondi dopo i quali viene spento il generatore (se acceso dal microfono). Infine, vi è un input di tipo submit per salvare le modifiche fatte.
- Mockup 2: La lista degli utenti, accessibile solamente se l'utente è amministratore, contiene una lista con tutti gli utenti dell'applicativo. Viene mostrato nome e cognome. Quando viene cliccato new (in alto a destra, per creare un nuovo utente), oppure il nome di un utente, viene aperto il Mockup 3, con tutti i campi vuoti, nel caso di un utente nuovo, mentre con i dati dell'utente se è già esistente, serve per modificarlo.
- Mockup 3: Vi sono i dettagli dell'utente, quindi 5 input di tipo text ed una select. Gli input di tipo text servono per lo username, il nome, il cognome, l'indirizzo e-mail e la password. La select serve per la selezione del gruppo.

3.4 Design procedurale

3.4.1 Applicativo (back-end)

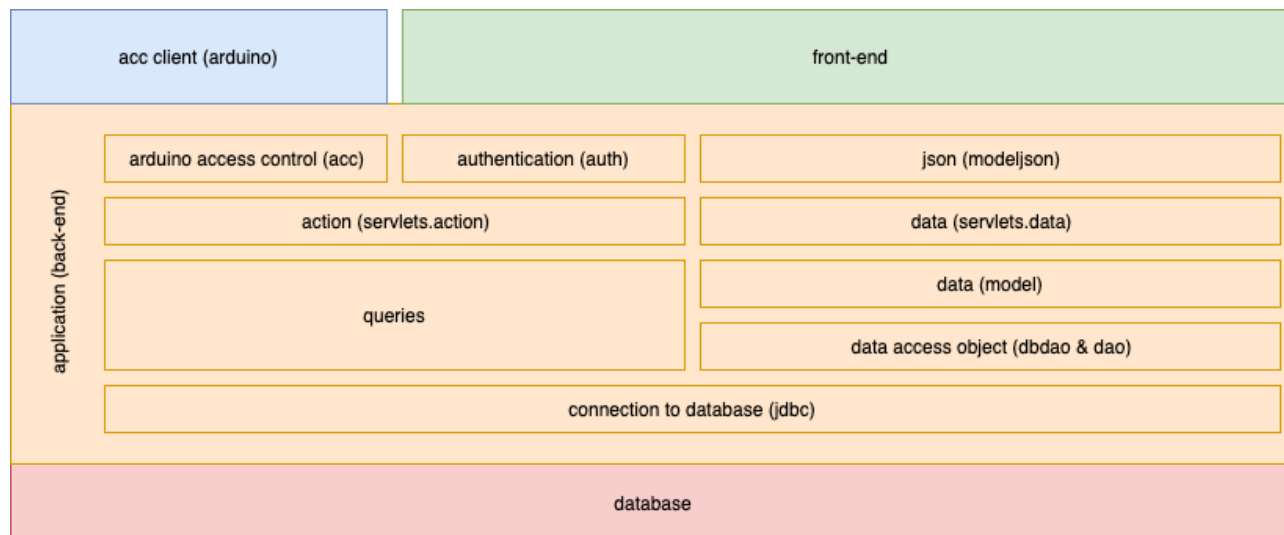


Figura 8 - Schema componenti Applicativo (back-end)

Per quanto riguarda la progettazione del back-end, che sarà abbastanza complesso, siccome ho deciso di non utilizzare un framework per interfacciarmi con il database e con l'interfaccia grafica. Quindi l'intero codice deve essere scritto a mano. L'applicativo deve contenere un sistema di gestione degli utenti, con differenti livelli di permessi basata su database e vi deve essere un sistema gestire il generatore di frequenze.

Il lato back-end dell'applicativo è suddiviso in due parti, i dati (e la loro gestione) e le azioni, cioè le operazioni specifiche che possono venir eseguite. Questa suddivisione è stata fatta per rendere le azioni più performanti e per mantenere l'integrità dei dati. Entrambe le due parti, sono suddivise in moduli.

Lato dei dati, vi sono 4 livelli, uno per l'astrazione dei dati:

- data access object (dbdao & dao): interfaccia dei dati con il database
- modelli (model): rappresentazione dei dati a livello database all'interno dell'applicativo
- data (servlets.data): esposizione http dei dati
- json (modeljson): traduzione dei dati da modelli a rappresentazione JSON, per le richieste delle servlet.

Il lato delle azioni vi sono 3 livelli di cui uno diviso in 2 moduli.

- queries sul database: sono le query, per aggiornare i dati sul database
- action (servlets.action): esposizione http delle azioni
- autenticazione (auth): autenticazione nell'applicativo
- controllo arduino (acc): controllo remoto dell'Arduino

I vari moduli, non verranno sviluppati nello stesso ordine in cui sono posti ora, siccome dipendono l'uno dall'altro, verranno sviluppati in un ordine che mi permetta di sviluppare al meglio il software.

3.4.1.1 connection to database (jdbc)

La connessione al database avviene tramite il driver di default di java, che deve poter essere istanziata in due modi:

- utilizzando un costruttore e passando i vari parametri per la connessione
- utilizzando un file di properties

Una volta istanziata la connessione, bisogna poterla aprire, chiudere e bisogna poter scrivere in un file di properties le proprietà utilizzate al momento.

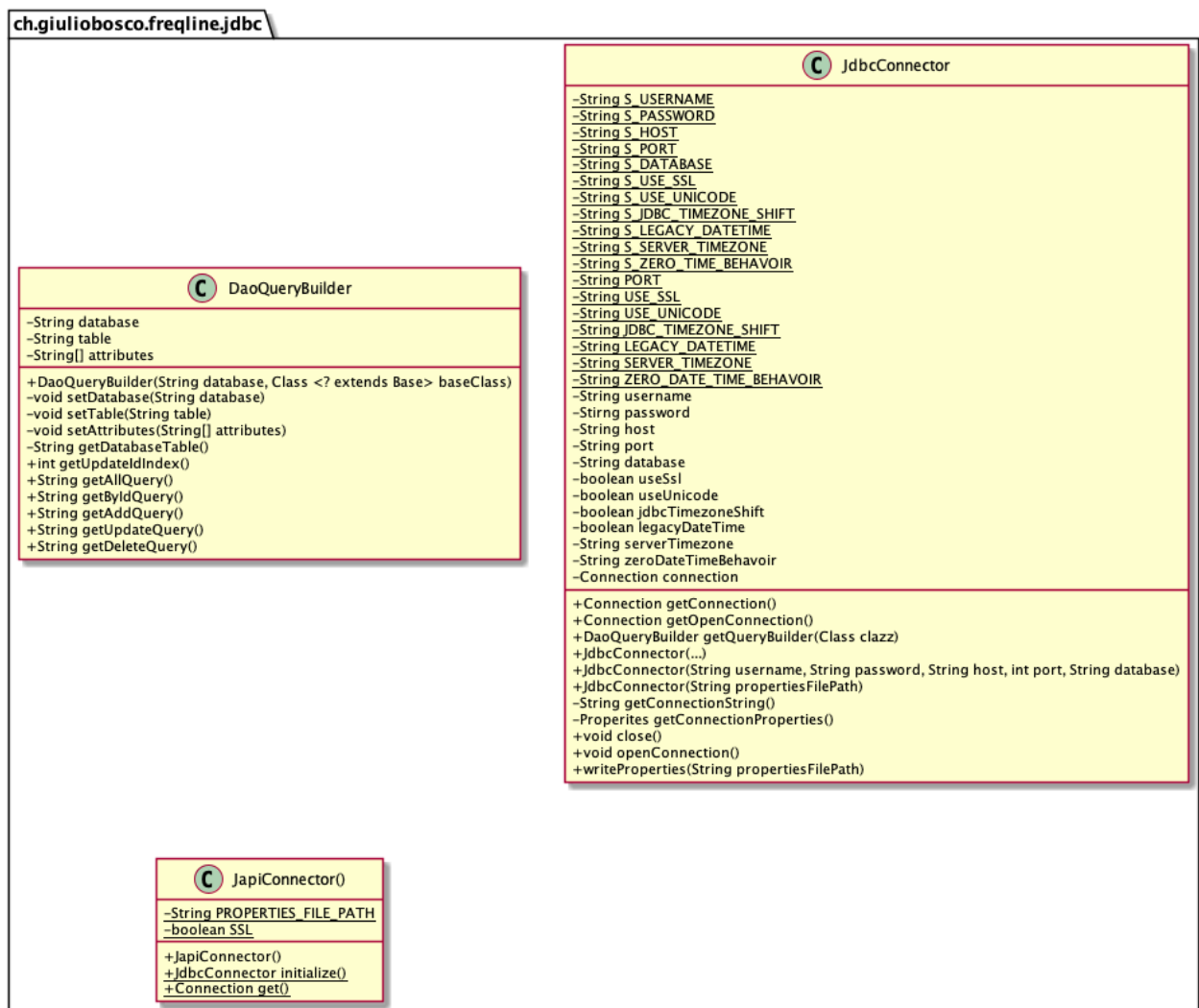


Figura 9 - Diagramma UML delle classi (jdbc)

Nel seguente diagramma si vede, il modulo di connessione al database, che è composto di una classe che serve per creare una connessione al database, che può essere utilizzata in diversi progetti, nel quale si possono configurare le opzioni di connessione più comuni, così che non vi sia il rischio di fare errori di battitura creando la stringa di connessione. Per ogni opzione vi è un valore di default, così che non sia necessario impostare tutti i valori. Mentre hostname, username, password e database sono sempre obbligatori. Siccome il codice del progetto è posto su un repository pubblico di GitHub, è buon'abitudine non salvare nel codice le credenziali in generale, per questo motivo ho deciso di implementare anche una modalità di istanziazione della connessione tramite file di properties che non viene pubblicato con il codice.

3.4.1.2 data access object (dao)

L'interfaccia dei dati fra le rappresentazioni dei modelli e il database (*dbdao*), che sfrutterà il modulo di connessione al database, per collegarsi ad esso. Verrà implementato sul modello DAO (Data Access Object), che dovrà essere sviluppato per ogni tabella. Il modello DAO comprende 5 possibili interazioni con il database:

- *getById*: seleziona un elemento dalla tabella, tramite il suo id
- *getAll*: seleziona tutti gli elementi dalla tabella
- *add*: aggiunge un elemento alla tabella
- *update*: aggiorna un elemento nella tabella, sostituisce tutti i parametri
- *delete*: Elimina un elemento dalla tabella, tramite l'id

Ognuno di questi metodi, comprende l'inizializzazione della *query*, con il *prepared statement*, l'inserimento dei dati nello *statement*, la sua esecuzione e l'analisi del suo risultato nel caso sia una *query* di selezione. Per questo motivo, ho deciso di progettare il software in maniera che sia il più astratto possibile, cioè che sia meno ripetitivo possibile.

Questo vuol dire, che ogni metodo è scritto una volta sola. Per poter sviluppare in questa maniera, alcune parti devono essere diverse per ogni tabella, quindi riscritte ogni volta per ogni tabella. Mentre per quanto riguarda le *query*, verrà automatizzato il tutto. Utilizzando un metodo laborioso, ma che permette di non dover scrivere le *query*, vengono composte automaticamente tramite il modello dei dati.

Le *query* verranno automaticamente create prendendo l'oggetto *model* che rappresenta la tabella, dal quale verranno presi tutti gli attributi e dal quale verrà generata la *query*, per preparare lo *statement*.

Il *prepared statement* non dovrà essere riempito ogni volta manualmente per ogni tabella, per questo è importante che vengano scritti gli attributi nello stesso ordine sia nell'oggetto, che vengano inseriti nello stesso ordine nel *prepared statement*.

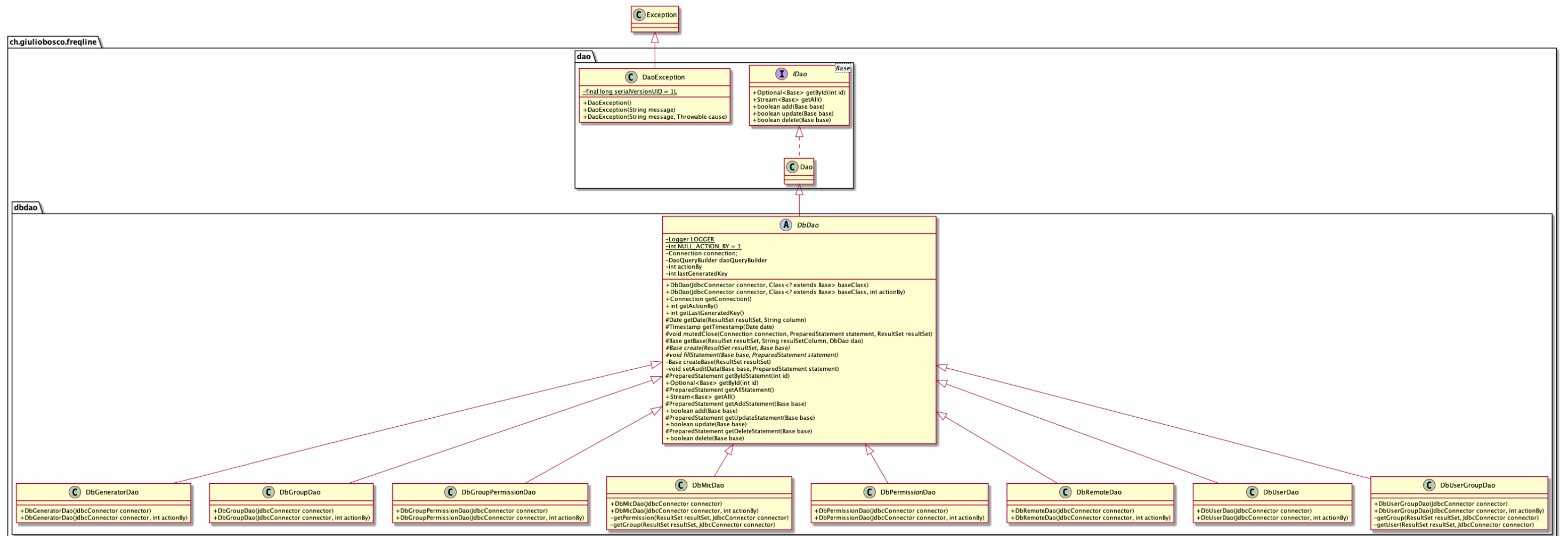


Figura 10 - Diagramma UML delle classi (dao)

Il modello DAO ha un'interfaccia che le operazioni possibili, che viene implementata per tutti i modelli che rappresentano le tabelle del database. Le azioni sono:

- insert: che inserisce nel database un dato
- update: che aggiorna un dato nel database, il quale utilizza l'ID per aggiornare tutto il resto dei parametri
- delete: elimina un elemento tramite il suo id
- getAll: seleziona tutti gli elementi dal database, del modello in questione
- getById: seleziona un elemento tramite il suo id

Questa struttura viene utilizzato per facilitare il cambio di database, quindi per ognuno di essi viene implementato un modello di dati. Nel nostro caso è stato implementato solo la classe DbDao, per MySQL.

La classe DbDao è molto complessa siccome è stata sviluppata in maniera astratta, così da poter riutilizzare praticamente tutto il codice, per poi adattarla per ogni tabella del database (e modello), la si estende, implementandone i metodi astratti, si può facilmente creare l'accesso ai dati, per ogni tabella. Come si può vedere nel diagramma UML delle classi. Per la classe DbDao contiene i metodi principali, più diversi metodi di aiuto, che servono per la costruzione degli statement SQL da eseguire sul server, per la trasformazione da tipi di dati di MySQL a tipi di dati Java.

La classe DbDao viene estesa per ogni tabella la quale contiene solamente i costruttori ed i metodi per la trasformazione da ResultSet (Ritorno dei dati da MySQL) a oggetti Java e la trasformazione da oggetti Java a statement (inserimento dati nei prepared statement).

3.4.1.3 Queries

Per ogni tipo di query, vi è una classe. Tutte le query per i prepared statement sono delle costanti di tipo stringa, che vengono inserite negli statement e poi vengono assegnati i valori, per poi essere eseguiti. Per facilitare l'utilizzo dei metodi sono tutti statici, che possono ritornare degli oggetti. Questo per facilitarne e velocizzarne l'utilizzo.

ch.giuliobosco.freqline.queries



Figura 11 - Diagramma UML delle classi (queries)

3.4.1.4 dati (model)

Per ogni tabella del database, vi è una relativa classe che la rappresenta, che deve contenere principalmente gli stessi attributi della tabella (nello stesso ordine, per mantenere la logica), con i costruttori ed i setter ed i getter, siccome in Java è best-practice mantenere tutti gli elementi private, creandone dei metodi per settare e dei metodi per richiedere il valore, questo si fa per poter regolare i permessi. Per esempio, mettere il setter di un attributo ad un livello di accesso (come private) ed il getter ad un altro livello (come public).

Anche in questo caso il codice ripetitivo verrà riciclato, utilizzando la tecnica di programmazione ad ereditarietà. In questa parte del codice vi saranno tutti gli elementi che sono utilizzati per l'audit delle tabelle.

ch.giuliobosco.freqline.model

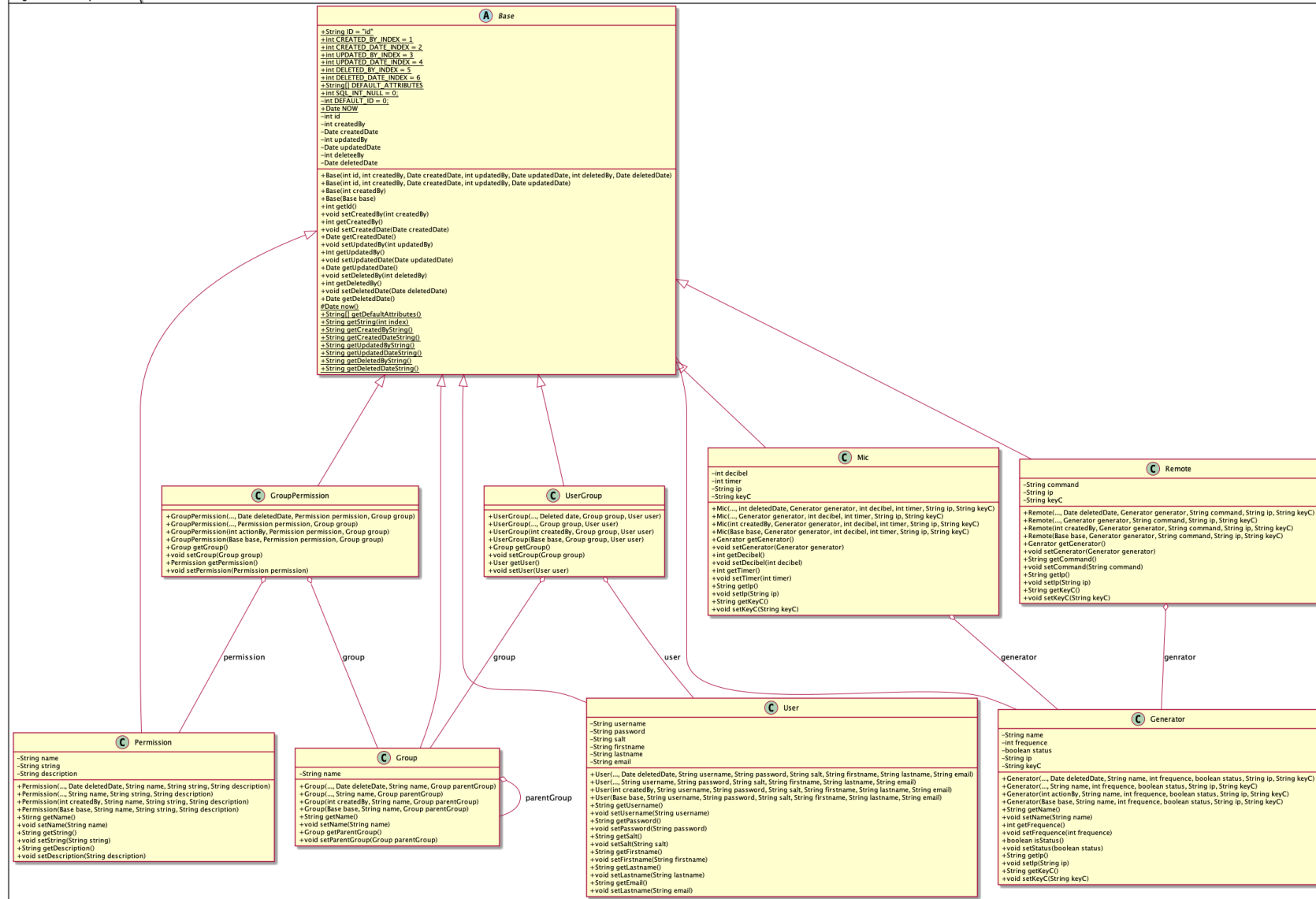


Figura 12 - Diagramma UML delle classi (model)

Nel diagramma delle classi si nota un modello **Base** che contiene l'id e gli attributi di audit. La classe viene estesa da una classe per ogni tabella del database, che contiene tutti gli attributi delle varie tabelle (eccetto id e attributi di audit, perché sono già contenuti nel modello **Base**). Sono anche rappresentati i legami fra i vari elementi, tramite delle composizioni.

3.4.1.5 json (modeljson)

Questo sotto modulo del back-end si occupa di trasformare i modelli in elementi JSON, così che siano pronti per essere inviati come risposte delle Restful API. Anche in questo modulo è stato utilizzato un approccio astratto ed ereditario.

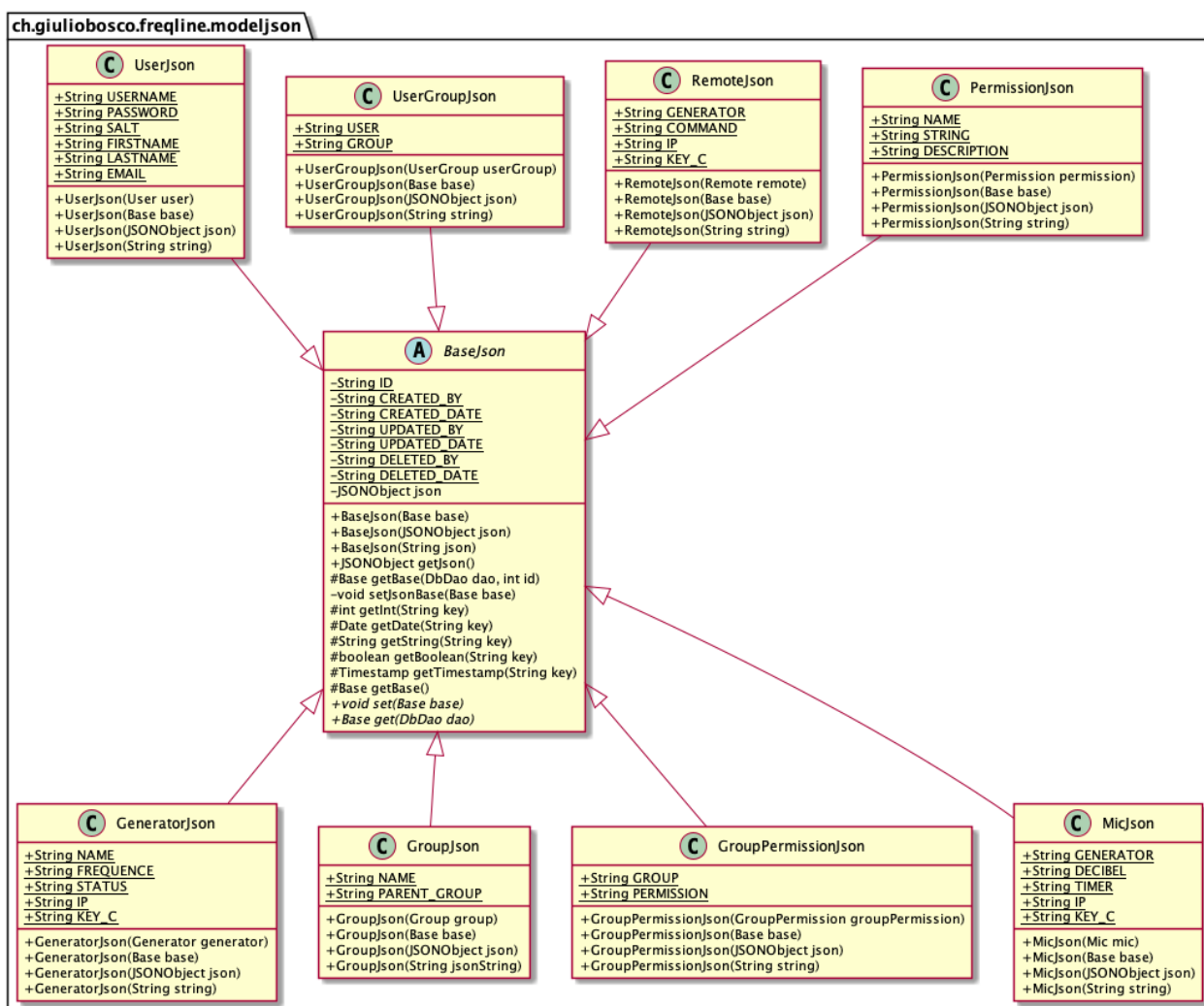


Figura 13 - Diagramma UML delle classi (modeljson)

Nel diagramma soprastante, si nota che la rappresentazione è molto simile a quella dei modelli, l'id e gli elementi di audit sono gestiti dal BaseJson dal quale discendono le classi corrispondenti ad ogni modello dei dati. Nei quali vengono gestiti gli attributi specifici.

3.4.1.6 servlets (servlets & servlets. Help)

Le servlets sono dei programmi scritti in java per elaborazione di dati e/o operazioni lato server, questi programmi sono pensati per interpretare e rispondere le richieste http. Siccome in questo caso viene utilizzato il protocollo http, ma per rispondere alle richieste con delle stringhe JSON, nelle quali deve essere sempre inserito lo status code http della richiesta ed il messaggio dello status code. Quindi è stata creata una BaseServlet, la quale contiene tutti i metodi per inviare le varie risposte, un metodo per ogni codice di errore. Per alcuni errori bisogna passare come perimetro un messaggio, mentre in altre come il metodo `notFound` necessita la richiesta http, così da automatizzare il messaggio di risposta, per esempio ritorna il messaggio, dell'elemento non trovato, anche la path dell'elemento.

Per facilitare l'analisi delle richieste http, vi è una classe che analizza i parametri inviati nella richiesta. Utilizza un array di stringhe (che sarebbero le key gli elementi) per gli elementi richiesti ed uno per gli elementi opzionali, la classe controlla che i parametri richiesti esistano e ne fa la validazione, mentre per quanto riguarda quelli opzionali se esistono li inserisce in un array contenetene tutti gli elementi presenti e li valida.

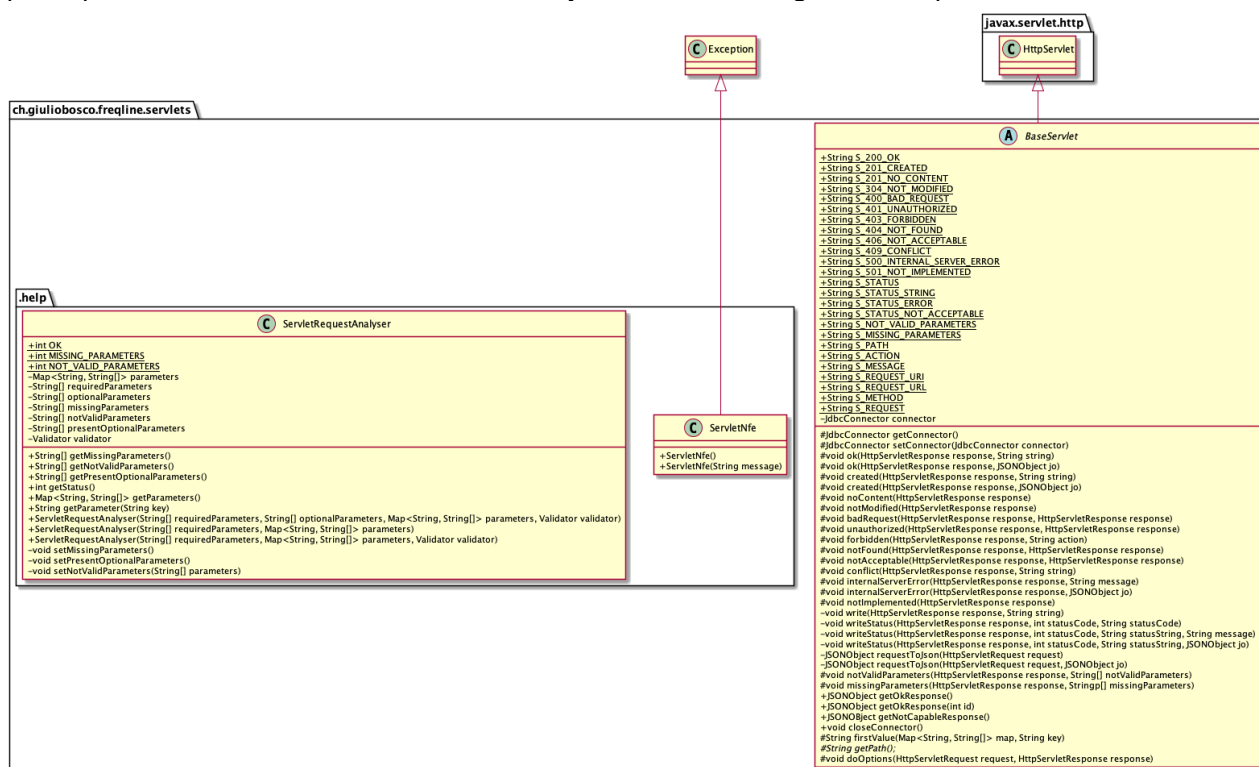


Figura 14 - Diagramma UML delle classi (servlets)

3.4.1.7 data servlets (servlets.data)

Le data servlets, sono le servlets specifiche per i dati, quindi tutte molto simili che devono eseguire le richieste dei client per quanto riguarda la gestione dei dati. Esse hanno a disposizione le operazioni CRUD (Create Read Update Delete), per le quali vi sono dei permessi da applicare per ogni operazione.

Per alcune operazioni il codice è stato suddiviso in più metodi, così da facilitarne la lettura. Le risposte delle richieste vengono inviate in formato JSON, come viene detto nelle best-practice delle Restful API (spiegate nel capitolo) per la comunicazione fra l'applicativo back-end e front-end.

Anche per questo modulo vi è una classe con tutta la logica e la struttura del codice; la quale viene estesa da una classe per ogni modello, che implementa i metodi astratti che servono per la regolazione dei permessi, per la creazione della classe di astrazione dei dati (model) e convertitore da model a json.

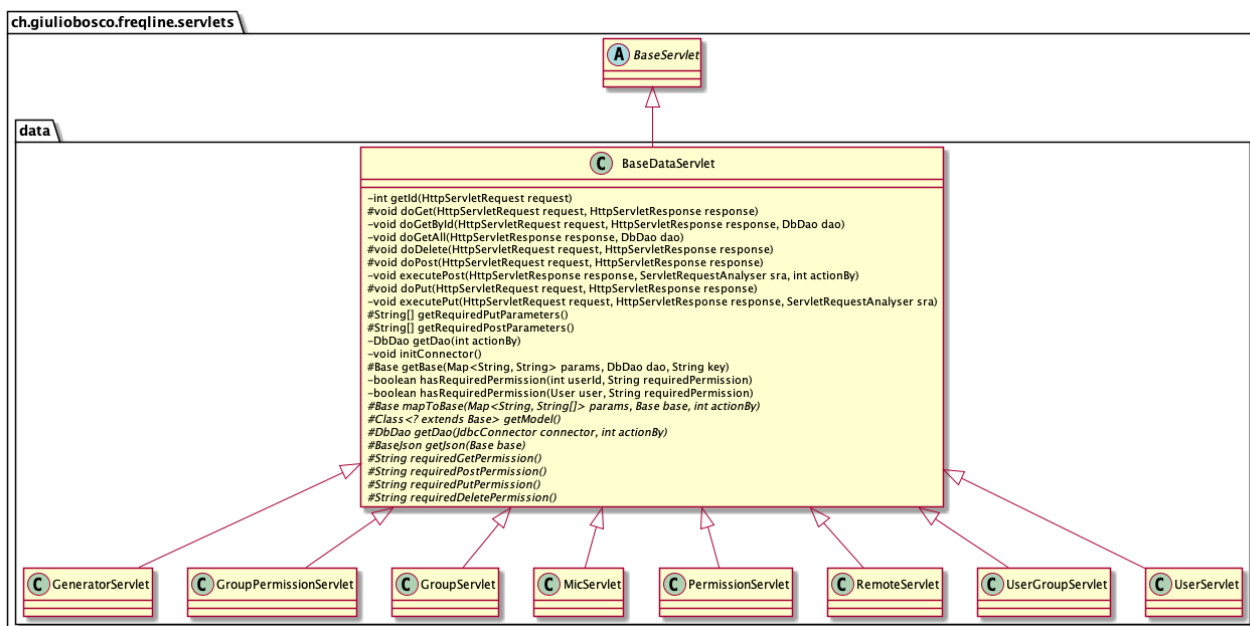


Figura 15 - Diagramma UML delle classi (servlets.data)

3.4.1.8 actions servlets (servlets.action)

Mentre la per quanto riguarda le Restful API che non riguardano i dati, ma le azioni come il login, oppure delle azioni specifiche cambiare la frequenza, accendere o spegnere il generatore, quelle di controllo dell'Arduino. Per ognuna di questa API vi è una servlet indipendente.

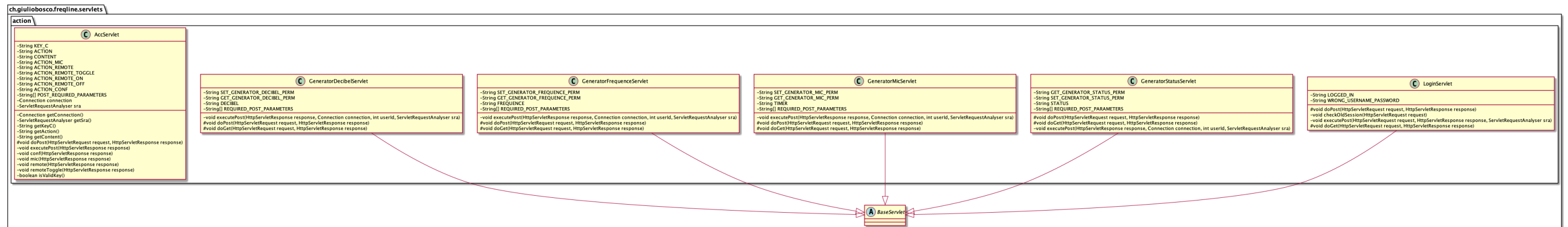


Figura 16 - Diagramma UML delle classi (servlets.action)

Le quali sono molto semplici, servono semplicemente per eseguire delle operazioni di base, come il login, oppure controllare se l'utente è collegato. Per eseguire queste operazioni vengono utilizzate le queries nel caso in cui le operazioni sono di base, mentre per quanto riguarda le operazioni più complicate vi possono essere dei piccoli moduli, così da separare ulteriormente il codice e renderlo più facile da leggere e mantenere.

3.4.1.9 authentication (auth)

Per utilizzare questo applicativo bisogna essere autenticati, quindi vi è un modulo nel quale vengono controllati gli accessi, tramite username e password (con controllo di permessi), mentre viene utilizzata una key, per gli arduino.

Per quanto riguarda la autenticazione tramite utente, vi sono due possibilità, una che controlla solamente username e password dell'utente siano corretti ed una che controlla se sono corretti e carica l'utente. Inoltre vi è un gestore delle sessioni per controllare velocemente che l'utente sia già autenticato ed abbia permessi corretti. Siccome la password dell'utente nel database viene salvata sotto forma di hash con anche il salt, cioè per migliorare la sicurezza viene presa la stringa della password, ci viene aggiunto dietro il salt (che sarebbe una stringa di caratteri casuali) e poi ne viene fatto un hash. Viene utilizzato il salt per rendere più complesso il reverse-engineering della password. Per gestire l'inserimento e l'aggiornamento di un utente sono state creati dei metodi apposta, così da mantenere l'integrità dei dati.

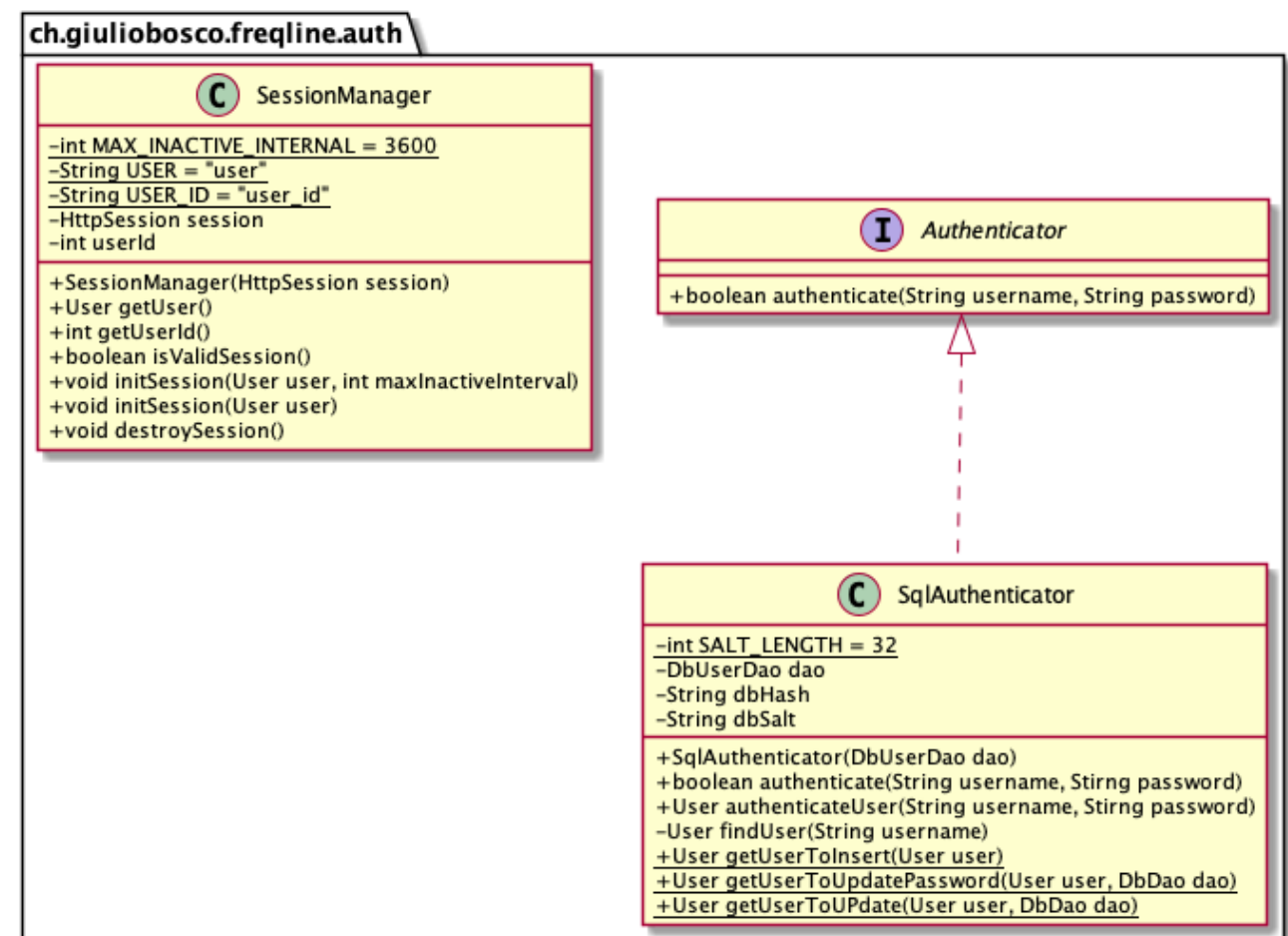


Figura 17 - Diagramma UML delle classi (auth)

3.4.1.10 arduino connection controll (acc)

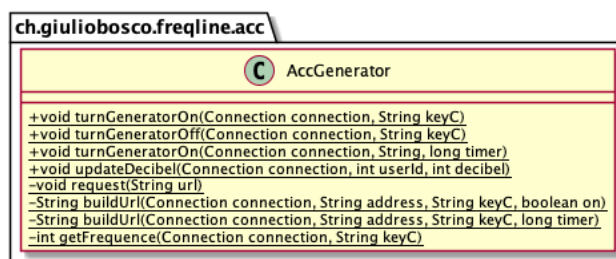


Figura 18 - Diagramma UML delle classi (acc)

Per quanto riguarda la comunicazione con l'Arduino, sono state create delle classi, le azioni da inviare al Arduino, sono state raccolte in una classe, così che venga controllato tutto in un punto singolo, l'aggiornamento dei dati nel database e l'esecuzione delle richieste. Per ogni richiesta che viene fatta, si necessita la connessione al database ed i parametri per configurare l'Arduino.

3.4.2 Applicativo (front-end)

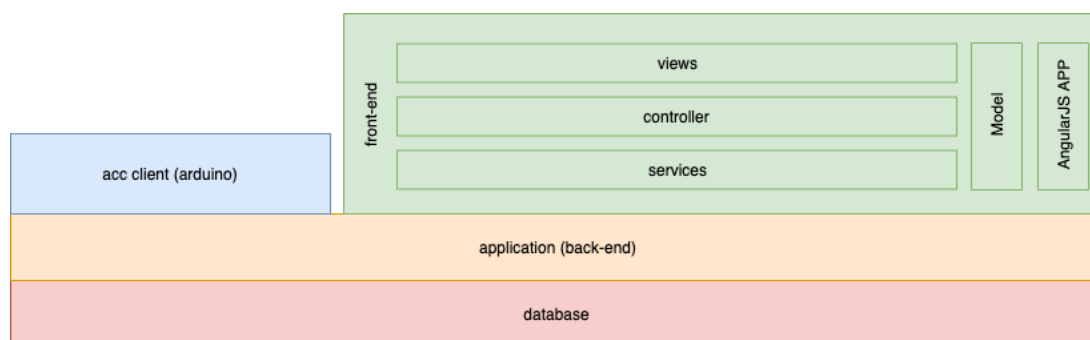


Figura 19 - Schema componenti applicativo (front-end)

Il lato front-end, dell'applicativo, verrà sviluppato basandosi su è anch'esso basato sui Modelli, per la rappresentazione dei dati. Utilizzando il modello MVC (verrà spiegato nell'implementazione). Buona parte dell'applicativo è eseguito sul client, quindi anche su esso vi deve essere una struttura ben precisa. Sfruttando le Restful API ed AJAX (Asynchronous JavaScript and XML), le varie grafiche vengono create con i vari dati.

3.4.3 Acc Client (Arduino)

Per quanto riguarda il controller, per il generatore di onde ad ultrasuoni, il telecomando ed il microfono, è un Arduino YÜN, che sarebbe un microcontrollore, composto da due componenti, una scheda ATmega32u4 ed un chip Athreos AR9331. Sulla scheda ATmega32u4 vi è un microsistema di controllo simile a quello di un Arduino UNO, quindi il collegamento hai pin ed alla porta seriale; in più vi è un collegamento con il chip, chiamato Bridge. Il chip ha un sistema operativo UNIX/Linux di base, minimale, OpenWRT, che è fatto apposta per circuiti embedded, quindi molto leggera e con il software di base. Il chip è collegato alla porta USB, posta sulla scheda, alla porta ethernet ed al chip Wi-Fi.

Il lato Arduino del chip Athreos viene programmato in python e può accedere al bridge verso il lato ATmega32u4, il quale viene programmato in una versione semplificata di C, adeguata all'Arduino (sviluppata apposta). Alla scheda ATmega32u4, vi sono collegati il generatore, il microfono ed il telecomando.

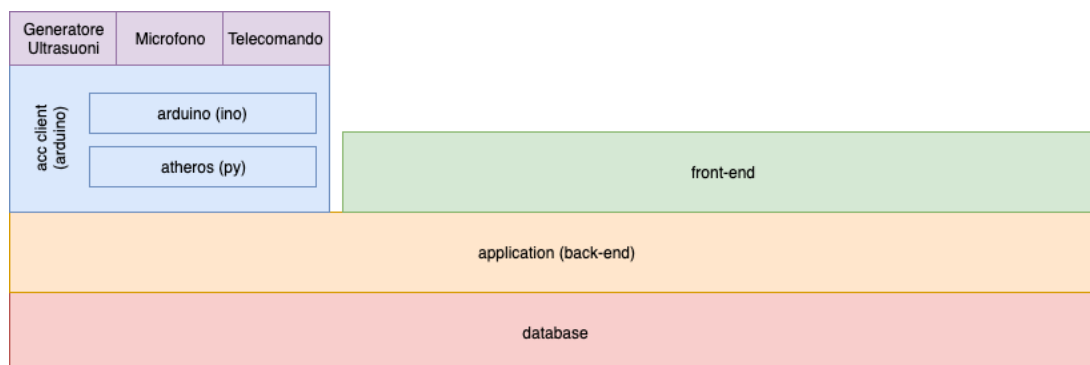


Figura 20 - Schema componenti arduino

La parte del programma scritta in python comprende un web-server personalizzato, che scrive i dati sul Bridge, verso la parte di programma scritta in C per la ATmega32u4, tramite il bridge e tramite delle thread dedicate legge i valori dal bridge, nel caso essi dovessero cambiare invia la modifica al server. Mentre la parte ATmega, continua ad aggiornare lo stato dei vari pin tramite i valori che legge dal Bridge.

Il circuito di base per generare la frequenza con l'Arduino UNO era già stato creato, era come il seguente:

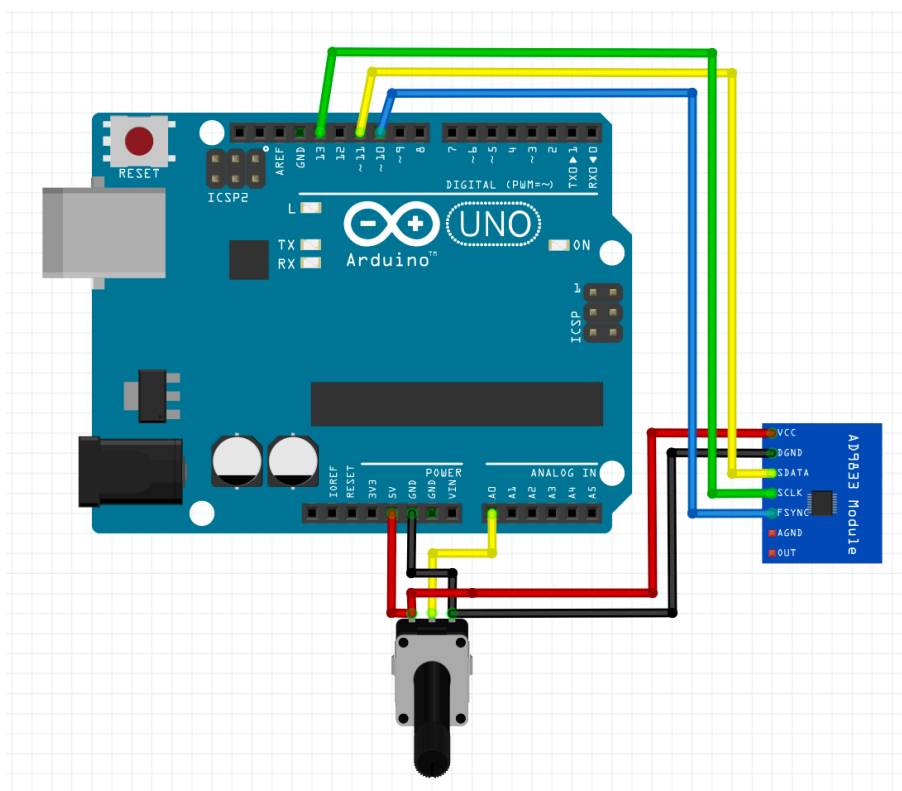


Figura 21 - Circuito Arduino Iniziale

Questo circuito prende il valore dal potenziometro e invia i dati al circuito AD9833 per la generazione dell'onda della frequenza.

4 Implementazione

1.1 Database

Il database è stato sviluppato con SQL, per un server MySQL.

Inizialmente volevo utilizzare MySQL, poi spostando il progetto sul Raspberry PI, mi sono reso conto che non è stato implementato MySQL vi è a disposizione mariadb, che è molto simile ma per esempio il driver è diverso. Alcune piccolezze per quanto riguarda il back-up dei database.

Quindi dopo aver sviluppato tutto ho dovuto cambiare il driver da MySQL a mariadb, nella classe JdbcConnector, in particolare, la linea di caricamento del driver alla linea 465:

```
//Class.forName("com.mysql.cj.jdbc.Driver");
Class.forName("org.mariadb.jdbc.Driver");
```

Poi modificare anche la linea 409 per la stringa di connessione:

```
//return "jdbc:mysql://" + this.host + ":" + this.port + "/" + this.database;
Return "jdbc:mariadb://" + this.host + ":" + this.port + "/" + this.database;
```

1.1.1 Tables

Ho trasformato il diagramma entità relazione in codice SQL, partendo dalle tabelle marginali per arrivare, per arrivare a quelle centrali del corpo. Così da poter inserire subito le foreign key.

Le varie tabelle hanno tutti degli attributi standard, che sono:

- **id**: Numero intero incrementale ed univoco che rappresenta una riga della tabella
- **created_by**: Id dell'utente che ha creato il record
- **created_date**: Timestamp del momento in cui è stato creato il record
- **updated_by**: Id dell'utente che ha eseguito l'ultima modifica sul record (quanto viene creato ha lo stesso valore di **created_by**)
- **updated_date**: Timestamp del momento in cui è stato modificato l'ultima volta il record (vale la stessa del **created_by**)
- **deleted_by**: Id dell'utente che ha eliminato il record
- **deleted_date**: Timestamp del momento in cui è stato eliminato il record

Il codice SQL per i seguenti punti è:

```
create table <table_name> (
  id INT          AUTO_INCREMENT PRIMARY KEY,
  created_by      INT NOT NULL,
  created_date    TIMESTAMP NOT NULL,
  updated_by      INT NOT NULL,
  updated_date    TIMESTAMP NOT NULL,
  deleted_by      INT,
  deleted_date    TIMESTAMP,

  <other_attributes>
);
```

La tabella **user** viene creata prima della tabella **generator**, quindi la foreign key **favorite_generator**, da errore quando viene creata, quindi viene creata la tabella con l'attributo, poi quando la tabella **generator** è stata creata, viene aggiunta la foreign key alla tabella **user**.

Tutte le tabelle hanno una nomenclatura standard, tutti i nomi delle tabelle e degli attributi sono minuscoli, in inglese, al singolare, se sono composti da due parole diverse lo spazio viene sostituito da un trattino underscore (_).

Come descritto nel capitolo della progettazione, vi sono anche le tabelle di audit, che sono state implementate, tutte nella stessa maniera, così da mantenere una certa ricorrenza nel database. Queste hanno, un id, una azione e tutti gli stessi attributi della tabella normale del database, ma gli attributi davanti al nome hanno audit_, così da poter distinguere bene gli elementi della tabella originale.

```
create table <table_name>_audit (
  id INT AUTO_INCREMENT PRIMARY KEY,
  action INT NOT NULL,

  audit_id INT NOT NULL
  audit_created_by INT NOT NULL,
  audit_created_date TIMESTAMP NOT NULL,
  audit_updated_by INT NOT NULL,
  audit_updated_date TIMESTAMP NOT NULL,
  audit_deleted_by INT,
  audit_deleted_date TIMESTAMP,

  audit_<other_attributes>,

  FOREIGN KEY (action) REFERENCES audit_action(id)
);
```

La seguente struttura è stata utilizzata per ogni singola tabella (eccetto la tabella **action**, che non possiede audit).

Per tutte le FOREIGN KEY bisogna inserire le operazioni da fare nel caso in cui, una delle chiavi alle quale ci si riferisce venisse aggiornata o eliminata, vi sono diverse possibilità:

- NO ACTION: non viene eseguita nessuna operazione, non viene eliminato nessun dato
- CASCADE: viene eliminato sia il dato padre che il dato figlio (bisogna utilizzarlo con molta prudenza, perché potrebbe scatenare un'eliminazione di dati a catena)
- SET NULL: viene settato a NULL il dato figlio (se il dato può essere nullo, altrimenti viene generato un errore)

Siccome in questo database quasi tutte le FOREIGN KEY non possono essere nulle e l'opzione CASCADE, è molto pericolosa ho optato per il NO ACTION, Anche perché così si ha un controllo incrociato nel caso in cui viene eliminato un dato.

L'elenco delle tabelle:

- audit_action
- generator
- generator_audit
- group
- group_audit
- group_permission
- group_permission_audit
- mic
- mic_audit
- permission
- permission_audit
- remote
- remote_audit
- user
- user_audit
- user_group
- user_group_audit

1.1.2 Triggers

I triggers, sono degli oggetti SQL, che vengono associati alle tabelle, ve ne possono essere di 6 tipi, due per ogni operazione CRUD (fatta eccezione per read – SELECT). Vengono 2 perché possono essere eseguiti prima della query oppure dopo. Tutti i trigger sottostanti vengono eseguiti autonomamente dal DBMS (DataBase Management System). I trigger possono disporre di due tipi di oggetti, NEW ed OLD, che rappresentano, rispettivamente la riga nuova, che verrà inserita (o che sostituisce quella esistente) oppure rappresenta la riga che viene sostituita (o eliminata). Per ogni tabella vi può essere solamente un trigger per ognuna di queste combinazioni.

- BEFORE INSERT: viene eseguito prima dell'inserimento e dispone dell'oggetto NEW
- AFTER INSERT: viene eseguito dopo l'inserimento e dispone dell'oggetto NEW
- BEFORE UPDATE: viene eseguito prima dell'inserimento e dispone sia dell'oggetto NEW che OLD
- AFTER UPDATE: viene eseguito dopo l'inserimento dei dati e dispone sia dell'oggetto NEW che OLD
- BEFORE DELETE: viene eseguito prima dell'eliminazione e dispone dell'oggetto OLD
- AFTER DELETE: viene eseguito dopo l'eliminazione e dispone dell'oggetto OLD

Siccome tutte le tabelle sono molto simili anche i triggers, saranno molto simili. Per ogni tabella vi è il sistema di audit, per di ogni operazione che viene fatta sul database bisogna eseguire l'audit, cioè aggiungere la modifica nelle tabelle apposite, quindi per ognuna di esse vi è una funzione trigger. Ogni tabella avrà tutti i trigger di tipo AFTER.

Il codice sottostante rappresenta un trigger AFTER INSERT per quanto riguarda la tabella permission:

```
DELIMITER //

CREATE TRIGGER `freqline`.`insert_permission_audit`
AFTER INSERT
ON `freqline`.`permission`
FOR EACH ROW
BEGIN
    DECLARE `v_action` INT;

    SELECT `id` FROM `freqline`.`audit_action` WHERE `audit_action`.`name` = 'created'
    INTO `v_action`;

    INSERT INTO `freqline`.`permission_audit` (`action`,
                                                `audit_id`,
                                                `audit_created_by`,
                                                `audit_created_date`,
                                                `audit_updated_by`,
                                                `audit_updated_date`,
                                                `audit_deleted_by`,
                                                `audit_deleted_date`,
                                                `audit_name`,
                                                `audit_string`,
                                                `audit_description`)

VALUES (`v_action`,
        NEW.`id`,
        NEW.`created_by`,
        NEW.`created_date`,
        NEW.`updated_by`,
        NEW.`updated_date`,
        NEW.`deleted_by`,
        NEW.`deleted_date`,
        NEW.`name`,
        NEW.`string`,
        NEW.`description`);

END;
//
```

Nella prima riga si può notare che vi è un DELIMITER, che significa: fin quando non vi è un altro carattere (o insieme di caratteri) esegui il seguente codice come blocco, altrimenti al primo carattere di fine istruzione (;) verrebbe eseguito il codice precedente. Questo potrebbe generare errori e soprattutto potrebbe corrompere parte dei dati. Dopo di che vi è la creazione del trigger insert_permission_audit (nome univoco per ogni database), sul database freqline che sarebbe quello dell'applicativo del progetto. Poi vi è il tipo di trigger AFTER UPDATE e la tabella su cui deve agire. Poi vi è l'inizio della funzione SQL, nella quale viene creata una

variabile nella quale viene messo, l'id dell'operazione eseguita (tramite una SELECT) per poi inserire nella tabella permission_audit l'azione eseguita ed i dati che vengono inseriti nella tabella originale.

Questa operazione è eseguita nella medesima maniera per le altre 2 operazioni mancanti nella medesima maniera, con la differenza nella dichiarazione dove vi sarà UPDATE o DELETE. Per il trigger DELETE nella funzione al posto di NEW vi sarà OLD, siccome eliminando il record non si aggiunge niente alla tabella.

In totale sono stati fatti 3 trigger su 8 tabelle, quindi un totale di 24 trigger; che sono:

- insert_generator_audit
- update_generator_audit
- delete_generator_audit
- insert_group_audit
- update_group_audit
- delete_group_audit
- insert_group_permission_audit
- update_group_permission_audit
- delete_group_permission_audit
- insert_mic_audit
- update_mic_audit
- delete_mic_audit
- insert_permission_audit
- update_permission_audit
- delete_permission_audit
- insert_remote_audit
- update_remote_audit
- delete_remote_audit
- insert_user_audit
- update_user_audit
- delete_user_audit
- insert_user_group_audit
- update_user_group_audit
- delete_user_group_audit

Per ognuno di essi ho testato che funzionasse manualmente. Per farlo mi sono loggato via CLI, sulla console di MySQL, dalla quale ho eseguito i test, cioè ho creato due record per ogni tabella, poi li ho modificati marginalmente tutti quanti, dopo averli modificati tutti quanti ho eliminato i valori.

Poi per ogni tabella ho controllato che corrispondessero le modifiche fatte nelle tabelle apposite.

4.1 Backend

Il codice back-end è stato sviluppato in Java, ho preso questa decisione siccome è il linguaggio di programmazione con il quale ho più esperienza e di conseguenza ho molto codice già scritto, quindi potrò riciclare codice vecchio oppure prenderne spunto.

Per facilitare la scrittura del codice alcune parti del codice le ho scritte basandomi su delle librerie, spesso creando delle interfacce per facilitare il loro utilizzo nelle parti ripetitive più complesse. La connessione al database avviene tramite il driver di connessione JDBC, per il quale ho creato un'interfaccia che mi crea la stringa di connessione ed apre la connessione, Siccome è un'operazione sempre uguale.

Per la gestione di oggetti e array JSON, ho utilizzato una libreria (JSON-io), alla quale ho collegato in parte un elemento che mi trasformasse i Modelli Java in JSON. La terza libreria che ho utilizzato è relativa al web server ed alle Servlet (gretty & javax.servlet).

1.1.3 Gradle

Per la gestione delle librerie in Java, vi sono due opzioni, gestirle manualmente, che vuole dire scaricarle manualmente, aggiungerle alla path delle librerie Java per poter compilare i programmi. Che ha un grande svantaggio, se si cambia macchina di sviluppo, oppure si mette in produzione un programma, bisogna spostare anche tutte le librerie ed aggiungerle alla path. Nel caso in cui le librerie vengono scaricate nuovamente (e non spostate) bisogna controllare che siano le stesse versioni, altrimenti si rischia di compromettere il funzionamento del programma. Per ovviare a questo problema sono stati inventati dei software chiamati Package Manager, che fanno il lavoro di gestire le dipendenze (cioè le librerie che vengono integrate ed i framework). Java principalmente ha due Package Manager, che sono Maven e Gradle, la grande differenza fra i due è la tecnologia, Gradle è il successore di Maven ed ha anche un grande vantaggio, permette di automatizzare delle operazioni, siccome si basa sul linguaggio Groovy. Questi punti di forza di Gradle mi hanno spinto ad utilizzarlo, per esempio per quanto riguarda lo sviluppo con le servlet, permette di avviare un web server di sviluppo senza dover installare niente sulla propria macchina.

Grazie a questa funzione di automazione, è possibile utilizzare molto facilmente i framework di test. Che servono per scrivere gli Unit Test, che servono per testare ogni singolo metodo di ogni singola classe.

Nel mio caso è stato inizializzato, ma poi non utilizzato, siccome non sono capace di scrivere gli unit test per eseguire test sul database e siccome non avevo abbastanza tempo per imparare a crearli ho preferito eseguire i test manualmente, che sono comunque stati fatti. Molto semplicemente li ho fatti mettendo del codice di test nel metodo main delle diverse classi.

Per creare il progetto con Gradle, bisogna creare una cartella, nella quale eseguire il comando:

```
gradle init
```

Questo comando fa partire una configurazione guidata del progetto (tramite command line), nel quale chiede il tipo di progetto da inizializzare, scegliere `java-application`, poi selezionare `groovy` come DSL script, che sarebbe il sistema utilizzato per eseguire le operazioni di automazione. La terza scelta da fare è il framework per lo unit testing, nel mio caso ho scelto JUNIT (siccome è quello di default), anche se non lo ho utilizzato. Poi viene chiesto il nome del progetto, dove ho inserito `freqline-be` ed infine il package di base del progetto `ch.giuliobosco.freqline`.

Poi aggiungere le dipendenze di cui necessitiamo, editando il file: `build.gradle`, ed aggiungendo nel gruppo plugin, le seguenti due linee per aggiungere il web server di sviluppo:

```
plugins (
    ...
    id 'war'
    id 'org.gretty' version '2.2.0'
)
```

Poi aggiungere le librerie, nel gruppo delle dipendenze:

```
dependencies (
    ...
    providedCompile 'javax.servlet:javax.servlet-api:3.1.0'
    compile group: 'mysql', name: 'mysql-connector-java', version: '8.0.11'
    compile 'org.json:json:20171018'
)
```

Gradle inoltre crea diversi files e directories, la directory, gradle, che contiene la libreria dell'applicativo gradle, poi vi sono due script, uno scritto in bash (gradlew) da utilizzare sui sistemi UNIX ed uno scritto in batch (gradlew.bat). Questi script sono utilizzati per eseguire le operazioni sul progetto, come compilare, avviare il programma oppure come nel nostro caso avviare il web server. L'ultimo file che viene creato è settings.gradle, nel quale sono contenute le proprietà per configurare gradle (viene creato e configurato autonomamente)

Per avviare il web server con gradle bisogna eseguire il seguente comando:

```
./gradlew appRun
```

Questo comando scarica le dipendenze necessarie per compilare ed eseguire il progetto, avvia il web server che compila il progetto. È il web server che compila il progetto e lo ricompila quando i files vengono modificati, in maniera autonoma.

1.1.4 JDBC

JDBC è la libreria di default per le connessioni ai database di java, non è nativo ma è molto ben supportato. Ha la possibilità di connettersi con diversi driver, come MySQL, mariadb, SQLite e diversi altri. Per utilizzare JDBC, vi sono due punti su cui spesso vi sono errori, uno l'importazione corretta della libreria, che nel nostro caso gestita da Gradle, quindi il problema non si propone, mentre il secondo problema è la stringa di connessione, la quale deve essere scritta correttamente. Per evitare che possano sorgere problemi dopo, ho deciso di creare una classe che mi generi secondo le mie configurazioni una stringa di connessioni, con più opzioni di configurazione possibili, le quali possono sempre venir aggiunte.

Come scritto nel capitolo della progettazione, è buona abitudine non salvare i dati di accesso al database nel codice, siccome esso potrebbe essere spostato da una macchina ad un'altra e messo su dei repository che potrebbero diventare pubblici. Quindi nella classe JdbcConnector (che serve per generare le connessioni con JDBC), ho messo un costruttore per creare la connessione tramite un file di properties.

```
/**
 * Create the jdbc connector with properties file.
 *
 * @param propertiesFilePath Properties file path.
 * @throws IOException Error while reading the the properties file.
 */
public JdbcConnector(String propertiesFilePath) throws IOException {

    InputStream propertiesFile = new FileInputStream(propertiesFilePath);
    Properties properties = new Properties();
    properties.load(propertiesFile);

    this.username = properties.getProperty(S_USERNAME);
    this.password = properties.getProperty(S_PASSWORD);
    this.host = properties.getProperty(S_HOST);
    try {
        this.port = Integer.parseInt(properties.getProperty(S_PORT));
    } catch (NumberFormatException nfe) {
        this.port = PORT;
    }
    this.database = properties.getProperty(S_DATABASE);
    this.useSsl = Boolean.parseBoolean(properties.getProperty(S_USE_SSL));
    this.useUnicode = Boolean.parseBoolean(properties.getProperty(S_USE_UNICODE));
    this.jdbcTimezoneShift = Boolean.parseBoolean(properties.getProperty(S_JDBC_TIMEZONE_SHIFT));
    this.legacyDatetime = Boolean.parseBoolean(properties.getProperty(S_LEGACY_DATETIME));
    this.serverTimezone = properties.getProperty(S_SERVER_TIMEZONE);
    this.zeroDateTimeBehavior = properties.getProperty(S_ZERO_TIME_BEHAVIOIR);

    this.connection = null;
}
```

Nel codice soprastante si può notare che viene letto il file passato come parametro nel costruttore, viene trasformato in delle properties ed ogni property viene assegnata ad un attributo della classe, infine viene svuotata la connessione.

Poi viene ricreato l'oggetto con le properties (viene fatta questa operazione nel caso in cui non si utilizzasse il costruttore con files delle properties. Queste servono per aprire la connessione con il server.

```
/**
 * Prepare the properties for the connection to the database.
 * Properties with:
 * <ul>
 * <li>user</li>
 * <li>password</li>
 * <li>useSSL</li>
 * <li>useUnicode</li>
 * <li>useJDBCCompliantTimezoneShift</li>
 * <li>useLegacyCode</li>
 * <li>serverTimezone</li>
 * <li>zeroDateTimeBehavior</li>
 * </ul>
 *
 * @return Connection properties.
 */
private Properties getConnectionProperties() {
    Properties properties = new Properties();

    properties.setProperty("user", this.username);
    properties.setProperty("password", this.password);
    properties.setProperty("useSSL", String.valueOf(this.useSsl));
    properties.setProperty("useUnicode", String.valueOf(this.useUnicode));
    properties.setProperty("useJDBCCompliantTimezoneShift", String.valueOf(this.jdbcTimezoneShift));
    properties.setProperty("useLegacyDatetetimeCode", String.valueOf(this.legacyDatetetime));
    properties.setProperty("serverTimezone", this.serverTimezone);
    properties.setProperty("zeroDateTimeBehavior", this.zeroDateTimeBehavior);

    return properties;
}
```

In questo metodo vengono presi tutti i valori ed inseriti in delle properties. Poi viene creata la stringa di connessione al database. Utilizzando il tipo di driver, l'hostname, la porta ed il database.

```
/**
 * Prepare the connection string.
 * Create a string like <code>jdbc:mysql://host:port/database</code>
 *
 * @return Connection string.
 */
private String getConnectionString() {
    return "jdbc:mysql://" + this.host + ":" + this.port + "/" + this.database;
}
```

Per aprire la connessione viene utilizzato il DriverManager di JDBC, il quale richiede che prima venga caricato il driver desiderato, utilizzando il metodo Class.forName(), inizializzare la connessione con la stringa di connessione e le connection properties.

```
/**
 * Open the connection to the MySQL database.
 *
 * @throws SQLException Error while connecting to the database.
 * @throws ClassNotFoundException Mysql Driver class not found.
 */
public void openConnection() throws SQLException, ClassNotFoundException {
    Class.forName("com.mysql.cj.jdbc.Driver");

    this.connection = DriverManager.getConnection(
        getConnectionString(),
        getConnectionProperties()
    );
}
```

La classe JapiConnetor() è una sotto classe di JdbcConnector, che semplicemente inizializza la super classe con il percorso del file di properties per il progetto.

I test di questo codice sono stati fatti manualmente, quindi creando un metodo main, che contenesse inizializzazione della classe, ed eseguisse una query (SHOW TABLES) se la query conteneva nel ResultSet generato i nomi delle tabelle del database il test è passato, per eseguire il controllo ci si collega direttamente alla console di MySQL sul database in questione e si esegue la stessa query.

Mentre DaoQueryBuilder è una classe che dato un modello dei dati, ne genera le query per le operazioni CRUD, le query comprendono anche i parametri di base presenti su tutte le tabelle. Questo permette di non doversi preoccupare di scrivere le varie query e velocizzare la scrittura del codice.

Testare questa classe è un meccanismo più laborioso, bisogna prima aver implementato la classe Base e le sue sotto classi (del package model), le quali vengono passate a questa classe, che le analizza e crea le query. Quindi le query sono tutte molto simili, per quanto riguarda le query di inserimento: la query deve essere:

```
INSERT INTO database.tabella(
    id,
    created_by,
    created_date,
    updated_by,
    updated_date,
    deleted_by,
    deleted_date,
    altri_parametri
) VALUES (?, ?, ?, ?, ?, ?, ?, ?altri)
```

Al posto di altri_parametri devono esserci gli attributi della classe modello, con la nomenclatura snake_case, e vi devono essere i rispettivi punti interrogativi nella sezione dei valori. La query di modifica deve essere:

```
UPDATE database.table SET
    created_by=?,
    created_date=?,
    updated_by=?,
    updated_date=?,
    deleted_by=?,
    deleted_date=?,
    altri_parametri=?,
WHERE id=?
```

Anche in questo caso altri_parametri deve essere modificato con gli attributi del modello. Mentre per le altre query il controllo è più facile:

Delete:

```
DELETE FROM database.table WHERE id=?
```

Select by id:

```
SELECT * FROM database.table WHERE id=?
```

Select:

```
SELECT * FROM database.table
```

Questo controllo va fatto per tutte le classi modello esistenti nel progetto.

1.1.5 DAO

Il modello DAO, che sarebbe un modello per estrapolare i dati dal database in oggetti utilizzabili in Java. Il quale si basa sulle 4 operazioni CRUD.

Rappresentate nell'interfaccia IDao:

```
/**
 * Dao Structure.
 *
 * @author giuliobosco (giuliobva@gmail.com)
 * @version 1.0 (2019-09-20 - 2019-10-16)
 */
public interface IDao<Base> {

    /**
     * Get element by id.
     *
     * @param id Id of element.
     * @return Optional with element if exists.
     * @throws Exception Error while getting the element.
     */
    Optional<Base> getByID(int id) throws Exception;

    /**
     * Get all elements.
     *
     * @return Stream of elements.
     * @throws Exception Error while getting the elements.
     */
    Stream<Base> getAll() throws Exception;

    /**
     * Add element.
     *
     * @param base element to add.
     * @return True if correctly added.
     * @throws Exception Error while adding element.
     */
    boolean add(Base base) throws Exception;

    /**
     * Update element (by element id).
     *
     * @param base Element to update.
     * @return True if correctly updated.
     * @throws Exception Error while updating element.
     */
    boolean update(Base base) throws Exception;

    /**
     * Delete element (by element id).
     *
     * @param base Element to delete.
     * @return True if correctly deleted.
     * @throws Exception Error while deleting element.
     */
    boolean delete(Base base) throws Exception;
}
```

Si possono notare 5 metodi, che sono 2 per selezionare i dati (uno per elemento singolo tramite il suo id ed uno per selezionare tutti gli elementi), uno per inserire, uno per modificare ed uno per eliminare un elemento. Il metodo `getById(int id)`, che serve per selezionare un elemento ritorna un `Optional<>` che può contenere un valore, nel caso in cui nella tabella del database non vi è nessun elemento con quell'id è vuoto. Il metodo `getAll()`, ritorna uno `Stream<>` contenente tutti i record della tabella. Il metodo `insert(Base base)` ritorna un valore booleano, `true` se il record è stato inserito correttamente altrimenti `false`. Stesso vale per il metodo `update(Base base)` ritorna `true` se l'elemento è stato aggiornato correttamente oppure `delete(Base base)` se eliminato correttamente.

L'interfaccia viene implementata dalla classe Dao che utilizza il tipo di dato Base, che da cui discendono tutti i modelli. Così da non dover creare interamente una classe di astrazione DAO per ogni modello, ma poter

riciclare il codice. La quale implementa tutti i metodi, ma nessun tipo di logica, quindi il metodo `getById()` ritorna un optional vuoto (`Optional.empty()`), il metodo `getAll()` ritorna `null`, e gli altri metodi ritornano `false`. Questa classe è stata creata semplicemente perché si vuole mantenere la possibilità di astrazione del codice sia la possibilità di avere diversi tipi di modello DAO, per esempio uno per un esempio (`MockDao`) uno per MySQL (o MariaDB) ed altri ancora qual ora ve ne fosse la necessità.

Nel nostro caso è stato implementato per MySQL, nella classe astratta `DbDao` che estende `Dao`, la quale ha solamente 2 metodi astratti, `create()` e `fillStatement()`, il primo serve a creare il modello dal `ResultSet`, della query e l'oggetto base che serve per i dati di base come `Id` e dati di audit. Mentre il metodo `fillStatement()` serve per inserire i dati del modello nel prepared statement.

Nel modello `DbDao`, vi sono alcuni metodi di aiuto, come:

- `getDate(ResultSet rs, String column)`: prende una colonna del result set e lo trasforma in data
- `getTimestamp(Date date)`: trasforma un timestamp in data
- `getBase(ResultSet rs, String column, DbDao dao)`: prende una colonna del result set, che contiene l'id di un oggetto. Quell'id viene inserito nel metodo `getById()` della connessione dao passata come parametro. Se il metodo ritorna un oggetto viene ritornato, altrimenti viene ritornato `null`.
- `createBase(ResultSet rs)`: viene creato un oggetto Base, con i dati di base di un modello (id e dati di audit), per poi richiamare il metodo `create()`

Poi vi è un metodo per ogni azione CRUD implementata, adibito a creare e preparare il `PreparedStatement`. Il metodo `getById()`:

```
/**
 * Get entry in database by ID.
 *
 * @param id Id of the entry.
 * @return Entry.
 * @throws Exception Error while executing the query.
 */
public Optional<Base> getById(int id) throws Exception {
    ResultSet resultSet = null;

    try {
        PreparedStatement statement = getByIdStatement(id);
        resultSet = statement.executeQuery();

        if (resultSet.next()) {
            return Optional.of(createBase(resultSet));
        } else {
            return Optional.empty();
        }
    } catch (SQLException sqle) {
        throw new DaoException(sqle.getMessage(), sqle);
    } finally {
        if (resultSet != null) {
            resultSet.close();
        }
    }
}
```

Il quale esegue il prepared statement, controlla che la query abbia ritornato almeno un valore, in tale caso trasforma il `ResultSet` in oggetto, lo inserisce in un `Optional<Base>`, altrimenti ritorna un `Optional` vuoto.

Il metodo getAll():

```
/**
 * Get all entries in database.
 *
 * @return All entries in database.
 * @throws Exception Error while executing the query.
 */
public Stream<Base> getAll() throws Exception {
    PreparedStatement statement = getAllStatement();
    ResultSet resultSet = statement.executeQuery();

    return StreamSupport.stream(new Spliterators.AbstractSpliterator<Base>(Long.MAX_VALUE,
    Spliterator.ORDERED) {
        @Override
        public boolean tryAdvance(Consumer<? super Base> action) {
            try {
                if (!resultSet.next()) {
                    return false;
                }

                action.accept(createBase(resultSet));
                return true;
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }, false).onClose(() -> mutedClose(getConnection(), statement, resultSet));
}
```

Esegue la query e ne ritorna il contenuto sottoforma di Stream di dati, lo stream viene creato eseguendo un ciclo per ogni record ritornato nella query, il quale viene trasformato in oggetto ed aggiunto allo Stream.

Il metodo add():

```
/**
 * Add entry to database.
 *
 * @param base Entry to add in the database.
 * @return True if added correctly.
 * @throws Exception Error while executing the query.
 */
public boolean add(Base base) throws Exception {
    if (this.actionBy != NULL_ACTION_BY) {
        base.setCreatedBy(this.actionBy);
        base.setCreatedDate(Base.NOW);
        base.setUpdatedBy(this.actionBy);
        base.setUpdatedDate(Base.NOW);
    }

    if (getById(base.getId()).isPresent()) {
        return false;
    }

    PreparedStatement statement = getAddStatement(base);

    int affectedRows = statement.executeUpdate();

    ResultSet rs = statement.getGeneratedKeys();
    if (rs.next()) {
        this.lastGeneratedKey = rs.getInt(1);
    }

    return affectedRows > 0;
}
```

Questo metodo, setta i dati di audit, tramite il `actionBy` e la data attuale, controlla che l'id inserito nell'oggetto non sia presente nel database, crea il prepared statement.

Esegue l'inserimento per poi scrivere l'id dell'elemento creato nell'attributo `lastGeneratedKey`, per poi ritornare true nel caso in cui l'elemento è stato inserito correttamente.

```
/**
 * Update entry in database.
 *
 * @param base Entry to update in the database.
 * @return True if updated correctly in database.
 * @throws Exception Error while executing the query.
 */
public boolean update(Base base) throws Exception {
    if (this.actionBy != NULL_ACTION_BY) {
        base.setUpdatedBy(this.actionBy);
        base.setUpdatedDate(Base.NOW);
    }

    try {
        PreparedStatement statement = getUpdateStatement(base);
        int affectedRows = statement.executeUpdate();

        ResultSet rs = statement.getGeneratedKeys();
        if (rs.next()) {
            this.lastGeneratedKey = rs.getInt(1);
        }

        return affectedRows > 0;
    } catch (SQLException sqle) {
        throw new DaoException(sqle.getMessage(), sqle);
    }
}
```

Il metodo per eseguire l'aggiornamento di un dato `update()`, aggiorna i dati di audit ne modello con la data attuale ed il `actionBy`, se è settato. Poi crea lo statement lo esegue ed inserisce l'id dell'elemento nell'attributo `lastGeneratedKey`.

Mentre il metodo di eliminazione prima setta il timestamp di eliminazione l'autore dell'azione e poi elimina l'elemento tramite il suo id.

```
/**
 * Delete entry in database.
 *
 * @param base Entry to delete in database.
 * @return True if deleted correctly in database.
 * @throws Exception Error while executing the query.
 */
public boolean delete(Base base) throws Exception {
    try {
        if (this.actionBy != NULL_ACTION_BY) {
            base.setDeletedBy(this.actionBy);
            base.setDeletedDate(Base.NOW);
        }

        if (this.update(base)) {
            PreparedStatement statement = getDeleteStatement(base);
            return statement.executeUpdate() > 0;
        }

        return false;
    } catch (SQLException sqle) {
        throw new DaoException(sqle.getMessage(), sqle);
    }
}
```

Come si può notare in questa classe vi è sempre l'oggetto generico Base, che permette di estendere questa classe per ogni modello discendente da Base. In questo caso è stato fatto per le classi:

- Permission
- Group
- PermissionGroup
- User
- UserGroup
- Generator
- Mic
- Remote

Per tutte le classi è stata utilizzata la stessa nomenclatura, Db<NomeClasse>Dao, come DbGroupDao, che rappresenta il modello DAO per la classe Group. La quale contiene i metodi astratti create() e fillStatement(), inoltre ovviamente ha i costruttori, per configurare la super classe, come segue:

```
/**
 * Group to MySQL in DAO pattern.
 *
 * @author giulio bosco (giulio bosco@gmail.com)
 * @version 1.1.1 (2019-09-27 - 2019-10-17)
 */
public class DbGroupDao extends DbDao {

    /**
     * Create the DbGroupDao with the connection to MySQL Database.
     * Use Group query builder.
     *
     * @param connection Connection to MySQL database.
     */
    public DbGroupDao(JdbcConnector connection) {
        super(connection, Group.class);
    }

    /**
     * Create the DbGroupDao with the connection to MySQL Database.
     * Use Group query builder.
     *
     * @param connector Connection to MySQL database.
     * @param actionBy Action By.
     */
    public DbGroupDao(JdbcConnector connector, int actionBy) {
        super(connector, Group.class, actionBy);
    }

    /**
     * Create the groups object from the result set and base entity.
     *
     * @param resultSet Result set of the query.
     * @param base Base.
     * @return Group object created from result set.
     * @throws Exception SQL Exception.
     */
    protected Base create(ResultSet resultSet, Base base) throws Exception {
        String name = resultSet.getString("name");
        Group parentGroup = null;

        Optional<Base> optionalBase = this.getById(resultSet.getInt("parent_group"));

        if (optionalBase.isPresent()) {
            parentGroup = (Group) optionalBase.get();
        }

        return new Group(base, name, parentGroup);
    }

    /**
     * Fill the prepared statement from the base (have to be a base with groups attributes).
     *
     * @param base Base element.
     * @param statement Statement to fill.
     * @throws Exception SQL Exception.
     */
    protected void fillStatement(Base base, PreparedStatement statement) throws Exception {
        Group group = (Group) base;
        Group parentGroup = group.getParentGroup();

        statement.setString(7, group.getName());
        if (parentGroup != null) {
            statement.setInt(8, parentGroup.getId());
        } else {
            statement.setObject(8, null);
        }
    }
}
```

Vi sono due costruttori, i quali ricevono entrambi la connessione al database, mentre uno dei due riceve anche l'autore delle azioni, così che venga inserito autonomamente nella query. Dopo di che vi è il metodo create() che prende i risultati della query ne estrapola i dati e ne crea gli oggetti, nel caso di questa classe prende il nome, lo inserisce in una variabile stringa, poi prende l'id del gruppo padre fa la query e lo assegna ad un'altra variabile. Se il gruppo padre esiste crea il gruppo con esso, altrimenti solamente con il nome.

Il secondo metodo per legare i parametri al prepared statement, prende il gruppo passato come argomento, inserisce al settimo index, il nome del gruppo ed all'ottavo il gruppo padre se esiste, altrimenti inserisce NULL. Viene usato il settimo index, perché in quelli precedenti vi sono l'id e gli attributi di audit. Mentre per l'ordine in cui vanno inseriti è lo stesso in cui sono creati nell'oggetto modello (model).

Anche nel caso di queste classi, di astrazione dei dati dal database alle classi modello, i test sono eseguiti manualmente, per eseguire i test, sarebbe opportuno eseguirli su tutti i modelli, per avere un controllo, che tutte le componenti sviluppate singolarmente per ogni modello siano corretti, mentre per quanto riguarda i test delle procedure che sono nella classe DbDao, basterebbe eseguire i test con una classe soltanto, siccome è stato scritto tutto il codice in maniera astratta. La classe che viene testata, deve avere almeno una foreign key, per permettere di controllare anche la parte di codice per l'estrazione di un elemento figlio (figlio inteso come relazione, quindi a livello database, entità padre - entità figlio).

Anche in questo caso il test viene fatto nel metodo main, che viene eseguito. Nel metodo main, istanziare una connessione al database, con la quale creare un modello DAO, per esempio per il model Group, che ha una stringa ed una foreign key (verso sé stessa, questo permette di testare tutte le funzioni con solamente una classe). Prima di tutto creare due oggetti Group, inserendovi un nome ed un parentGroup nullo (i nomi possibilmente diversi), poi aggiungerli al database con il metodo dao.add(group), poi controllare tramite la console di MySQL, se gli elementi sono stati aggiunti correttamente, una volta creati gli elementi selezionarne tramite DAO i due elementi, in oggetti separati, poi modificare il secondo ed inserire il primo come parentGroup, eseguendo l'update con il metodo adeguato. Dopo questo passo controllare ancora che le modifiche siano state eseguite correttamente, tramite la console di MySQL.

Per testare il metodo di selezione di tutti gli elementi selezionarli tutti e controllare che corrispondano tutti a quelli che sono stati precedentemente aggiornati. Infine eliminare prima uno poi l'altro gli elementi, con l'apposito metodo delete().

Questo procedimento è stato eseguito per tutte le classi presenti nel progetto, per essere sicuri di individuare sempre i problemi nella maniera meno dispendiosa di risorse, sono stati prima implementati i modelli DAO dei model che non richiedono delle foreign key su altre tabelle, dopo aver testato le classi sono stati creati gli altri modelli.

1.1.6 Queries:

Per le actions, sono state implementate delle query specifiche, in maniera che possano essere eseguite più velocemente e senza dispendi inutili di risorse, hanno tutte una struttura molto simile, una stringa costante contenente la query pronta per essere inserita in un prepared statement, che viene fatto in un metodo statico dedicato, che riceve come parametri la connessione al database, dalla quale viene preparato lo statement ed i vari parametri per la query.

Tutte le queries sono implementate con la medesima struttura, cambiano le query effettuate ed l'interpretazione dei dati.

```
/**
 * Id of user.
 *
 * @author giuliobosco (giuliobva@gmail.com)
 * @version 1.0.2 (2019-10-26 - 2019-11-12)
 */
public class UserIdQuery {
    // ----- Costants

    /**
     * Id of user query.
     */
    private static final String QUERY = "SELECT id FROM freqline.user WHERE username=?";

    /**
     * Username of user query index.
     */
    private static final int USERNAME_INDEX = 1;

    // ----- Static Components

    /**
     * Get id from username.
     *
     * @param connection Connection to MySQL database.
     * @param username Username of the user.
     * @return Id of the user, -1 if user does not exists.
     * @throws SQLException Error with mysql.
     */
    public static int getUserId(Connection connection, String username) throws SQLException {
        PreparedStatement statement = connection.prepareStatement(QUERY);
        statement.setString(USERNAME_INDEX, username);
        ResultSet resultSet = statement.executeQuery();

        if (!resultSet.next()) {
            return -1;
        }

        return resultSet.getInt(Base.ID);
    }
}
```

Il metodo statico riceve la connessione al database e lo username, prepara lo statement, lega il parametro username al suo index, esegue la query ed estrapola il valore dalla query. Nel caso in cui la query ha un esito negativo ed è un oggetto viene ritornato NULL, mentre se è un tipo di dato fondamentale viene ritornato un valore non valido, come potrebbe essere 0 o -1.

Per alcune query, simili, potrebbero esservi dei metodi statici privati, che eseguono le operazioni ripetitive, come per esempio preparare lo statement da una stringa SQL, fare il binding dei parametri (che devono essere dello stesso tipo e nello stesso ordine) ed eseguire la query, questo è possibile solamente siccome le query sono molto simili.

Sono state implementate le seguenti query, che possono essere eseguite semplicemente con un metodo:

- AccCheckQueries:
 - getJsonConf
 - isValidGeneratorKey
 - isValidMicKey
 - IsValidRemoteKey
- GeneratorQuery
 - getGeneratorStatus
 - getGeneratorFrequency
 - setStatus
 - getMicTimer
 - setFrequency
 - getIp
 - getKeyByUserId
 - setMicTimer
 - setDecibel
 - getDecibel
- UserIdQuery
 - getUserId
- PermissionUserQuery
 - getPermission

Alcune delle seguenti query, possono essere eseguite con parametri diversi, quindi sono state implementate più volte.

1.1.7 Models:

I model, che sono la rappresentazione delle tabelle, sotto-forma di oggetti Java, sono state implementate anch'esse sfruttando le proprietà dell'ereditarietà, sia per scrivere meno codice in queste classi in particolare, sia per poterle usare tutte quante facilmente all'interno dei metodi delle classi DAO.

La classe Base, dalla quale discendono tutti i modelli, contiene gli attributi che sono comuni in tutte le classi, come id e gli attributi di audit che sono: createdBy, createdAt, updatedAt, deletedBy e deletedDate. Tutti gli attributi citati sono privati, dispongono di un metodo getter, per richiedere il loro valore, mentre nel caso di alcuni di loro vi sono disponibili anche i metodi setter: setUpdatedBy(), setUpdatedDate(), setDeletedBy() e setDeletedDate(). I metodi che non sono disponibili, lo sono per aggiungere un livello minimo di sicurezza, qual'ora si lavorasse in più persone sul codice oppure il codice venga mantenuto o riutilizzato da un'altra persona, così da riuscire a mantenere una certa consistenza nei dati.

La classe Base dispone di diversi metodi costruttori, i quali permettono di creare l'oggetto da varie situazioni, per esempio creare l'oggetto nuovo, quindi semplicemente con il creatore dell'oggetto, per cui l'id rimane nullo (valore int non può essere nullo, ma zero in MySQL è nullo, per gli id), l'id dell'utente che ha creato l'oggetto, è quello indicato sopra, come per l'ultimo che ha eseguito un update al record, mentre la data è quella del momento in cui viene eseguita l'operazione.

Il secondo costruttore usato è quello per la creazione dell'oggetto con tutti i parametri, viene utilizzato durante la ricostruzione dell'oggetto presente nel database, prende i dati da esso e li utilizza per creare l'oggetto.

Il codice dei costruttori principali, quello con tutti i parametri e quello con la base dell'oggetto (usato dalle sottoclassi):

```
/**
 * Create the base entry with all parameters.
 */
@param id Id of the entry.
@param createdBy Id of the user who created the entry.
@param createdAt Date of the creation of the entry.
@param updatedAt Id of the user who did last update of the entry.
@param updatedDate Date of the last update of the entry.
@param deletedBy Id of the user who deleted the entry.
@param deletedDate Delete date of the entry.
*/
public Base(int id, int createdBy, Date createdAt, int updatedAt, Date updatedDate, int deletedBy, Date
deletedDate) {
    this.id = id;
    this.createdBy = createdBy;
    this.createdAt = createdAt;
    this.updatedBy = updatedAt;
    this.updatedDate = updatedDate;
    this.deletedBy = deletedBy;
    this.deletedDate = deletedDate;
}

/**
 * Create a copy of the base passed as argument.
 */
@param base Base to copy.
*/
public Base(Base base) {
    this(
        base.getId(),
        base.getCreatedBy(),
        base.getCreatedAt(),
        base.getUpdatedBy(),
        base.getUpdatedDate(),
        base.getDeletedBy(),
        base.getDeletedDate());
}
```

Tutti i model estendono la classe Base, per poi implementare i propri attributi specifici i quali vengono inseriti per coerenza nello stesso ordine in cui sono nel database, siccome si utilizza lo stesso ordine durante le operazioni al livello DAO. Anche questi parametri sono tutti con diritti di accesso private, per poter regolare gli accessi al livello di metodi getter e setters.

1.1.8 Servlets

Le servlets sono la parte del programma che espongono i dati e le azioni sul protocollo http, sono paradigma di programmazione tipiche del linguaggio Java, sono solitamente utilizzate nel campo delle applicazioni basate sul testo, per esempio per ritornare file JSON o XML, mentre per creare file HTML, si utilizzano le JavaServer Pages (JSP) che sono dei file, di testo, che possono contenere parti di codice Java, che vengono compilate (trasformate) in delle Servlet. L'utilizzo di esse si potrebbe paragonare a PHP, ma per convenzione in quelle pagine si mette solamente il codice strettamente necessario, il resto del codice viene messo in delle classi separate.

Siccome per questo progetto verranno solamente utilizzate delle servlet normali che ritornano file JSON, ho deciso di implementare una classe di Base con dei metodi di aiuto, come inviare la risposta al client, con un determinato codice http, quindi ho implementato nella classe un metodo che invia la risposta al client, mettendo il codice dello stato, la stringa dello stato e settando i parametri dell'header.

```
/**
 * Write status and content to client.
 *
 * @param response    Http Response.
 * @param statusCode  Http status code.
 * @param statusString Http status string.
 * @param jo          Json to write to client (http status code and string will be added).
 * @throws IOException Error while writing to client.
 */
private void writeStatus(HttpServletResponse response, int statusCode, String statusString, JSONObject jo)
throws IOException {
    response.setStatus(statusCode);
    response.setHeader("Access-Control-Allow-Credentials", "true");
    response.setHeader("Access-Control-Allow-Origin", "*");
    response.setHeader("Content-Type", "application/json");
    response.setHeader("Access-Control-Allow-Methods", "POST, GET, PUT, DELETE, OPTIONS");
    response.setHeader("Access-Control-Allow-Headers", "Content-Type");
    response.setHeader("Access-Control-Max-Age", "86400");
    jo.put(S_STATUS_STRING, statusString);
    jo.put(S_STATUS, statusCode);
    write(response, jo.toString());
}
```

Questo metodo viene utilizzato da dei metodi che preparano le risposte standard, come ok, created, notFound, notAllowed, unauthorized. Ai quali si passa un oggetto JSON oppure una stringa che viene inserita in un oggetto JSON. Codice del metodo ok:

```
/**
 * Return http status code 200.
 *
 * @param response Http response.
 * @param jo        JSON to return.
 * @throws IOException Error while returning http status code.
 */
protected void ok(HttpServletResponse response, JSONObject jo) throws IOException {
    writeStatus(response, response.SC_OK, S_200_OK, jo);
}
```

Il quale richiama il metodo writeStatus() passando come argomenti la risposta (per inviare al client la risposta), la stringa della risposta (ok), il codice http della risposta (200) e l'oggetto JSON.

Vi sono 2 metodi standard, utilizzati dopo aver analizzato la richiesta, che servono per ritornare una risposta nel caso in cui i parametri inviati con la richiesta non fossero validi o non fossero completi. Questi metodi ritornano il codice http 400 Bad Request, con all'interno l'elenco dei parametri mancanti o non validi. I metodi sono molto simili, creano l'oggetto json, inseriscono i parametri ed inviano la risposta 400.

```

/**
 * Write to client not valid parameters.
 *
 * @param response      Response to client.
 * @param notValidParameters Not valid parameters.
 * @throws IOException Error while writing to client.
 */
protected void notValidParameters(HttpServletResponse response, String[] notValidParameters) throws
IOException {
    JSONObject jo = new JSONObject();

    jo.put(S_NOT_VALID_PARAMETERS, StringArrayHelper.toJsonArray(notValidParameters));

    writeStatus(response, response.SC_BAD_REQUEST, S_400_BAD_REQUEST, jo);
}

```

Un altro metodo molto importante è `firstValue()`, serve per prendere il primo valore di un array, posto in una mappa composta da una stringa (key) ed un array di stringhe (valori). Nel caso di questo progetto viene inserito solamente un valore per ogni chiave, quindi sarà il valore sarà sempre e solamente nella prima cella dell'array. Per facilitare l'utilizzo dei parametri è stata creata questo metodo che prende il primo elemento dell'array. Nel caso il valore non fosse presente oppure la chiave non fosse valida viene ritornato un valore nullo.

```

/**
 * Get the first value in the array of the map.
 *
 * @param map Map.
 * @param key Key of the array
 * @return First value of array if exists.
 */
protected String firstValue(Map<String, String[]> map, String key) {
    String[] parameterValues = map.get(key);

    if (StringArrayHelper.arrayEmpty(parameterValues) || !StringHelper.is(parameterValues[0])) {
        return null;
    }

    return parameterValues[0];
}

```

1.1.8.1 Actions

Le actions sono delle servlet che contengono delle azioni cioè delle query molto specifiche, che potrebbero essere eseguite con una o più query sui dati (DAO tramite le Data Servlets), ma sarebbero molto poco performanti. Quindi creando delle query specifiche si velocizzano molti processi. Per esporre queste query, sono state create le actions, le quali sono molto diverse le une dalle altre, per esempio possono essere per il login, mentre altre sono fatte per la gestione specifica dei dati del generatore di frequenze, cambiare la frequenza, cambiare lo stato, cambiare il timer o i decibel del microfono.

```
/**
 * Do get, execute http get request.
 * @param req HTTP request.
 * @param resp HTTP response.
 * @throws ServletException Error with servlet.
 * @throws IOException I/O Exception.
 */
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    try {
        SessionManager sm = new SessionManager(req.getSession());

        if (sm.isValidSession()) {
            int startIndex = req.getRequestURI().lastIndexOf('/') + 1;
            String requestType = req.getRequestURI().substring(startIndex);

            JSONObject jo = new JSONObject();
            jo.put("isLoggedIn", true);

            if (requestType.equals("permissions")) {
                JdbcConnector connector = new JapiConnector();
                connector.openConnection();
                String[] perms = PermissionsUserQuery.getPermissions(connector.getConnection(), sm.getUserId());

                jo.put("permissions", perms);
            }

            ok(resp, jo);
        } else {
            unauthorized(req, resp);
        }
    } catch (Exception e) {
        internalServerError(resp, e.getMessage());
    }
}
```

Metodo per controllare se è stato effettuato il login e leggere i permessi nel caso in cui l'utente fosse loggato nell'applicativo. Il metodo controlla se l'utente è loggato, nel caso in cui non lo fosse invia una risposta negativa (unauthorized), mentre nel caso in cui lo è esegue la query sul database per controllare i permessi dell'utente e li sotto forma di un array json.

Per testare queste servlet, ho deciso di utilizzare il comando CURL, che permette di eseguire delle richieste http, via linea di comando, ho deciso di utilizzare questo programma siccome le richieste sono ripetitive e molto simili, l'altra opzione è utilizzare post man, un programma fatto apposta per testare le RestfulAPI. Il quale ha un funzionamento molto simile a quello di CURL ma con interfaccia grafica.

Per eseguire le richieste con CURL, bisogna specificare l'URL a cui fare la richiesta, il metodo i dati e nel caso in cui si vuole salvare la sessione, bisogna specificare anche la path di un file in cui salvare i cookies. Tutte le RestfulAPI sono state testate tramite questo sistema. Per esempio, l'API per il login è stata testata con i seguenti comandi, il primo per testare un'utente esistente che dovrebbe eseguire il login, mentre il secondo un utente che non esiste.

```
$ curl -X POST -c c -b c -d "username=admin&password=123qwe" http://localhost:8080/freqline-be/action/login
{"status":200,"statusString":"ok","message":"ok"}
$ curl -X POST -c c -b c -d "username=adm&password=pwd" http://localhost:8080/freqline-be/action/login
{"status":401,"statusString":"unauthorized","message":"wrong username or password"}
```

1.1.8.2 Data Servlets

Per quanto riguarda il codice delle servlets dei dati, riprende lo stile di quello dell'interfaccia dei dati con il database. Scritto in maniera astratta, per isolare il codice ricorsivo e scriverlo una volta sola. Creando dei metodi astratti da implementare nelle sottoclassi (una per ogni model), nei quali vengono configurati, i modelli da creare, l'accesso al database (DAO) da creare ed i permessi per le varie operazioni possibili, i metodi sono:

- mapToBase(): trasformare una mappa di parametri in un oggetto, da scrivere sul database
- getModel(): il model dell'oggetto da utilizzare
- getDao(): il modello di accesso al database DAO
- getJson(): la classe di conversione da model a json
- requiredGetPermission(): il permesso richiesto per eseguire l'operazione get
- requiredPostPermission(): il permesso richiesto per eseguire l'operazione post
- requiredPutPermission(): il permesso richiesto per eseguire l'operazione put
- requiredDeletePermission(): il permesso richiesto per eseguire l'operazione delete

La classe ha molti metodi corti, come il resto per progetto, per facilitare la lettura del codice. Diversi metodi sono stati creati per facilitare lo sviluppo delle sottoclassi, come getBase(), che serve per trasformare un parametro di un oggetto, nel suo oggetto.

```
/**
 * Get base from the id in the params at key.
 *
 * @param params Parameters.
 * @param dao Dao, for load the base.
 * @param key Key in parameters.
 * @return The base if exists, other wise null.
 * @throws Exception Error with MySQL.
 */
protected Base getBase(Map<String, String[]> params, DbDao dao, String key) throws Exception {
    Base base = null;

    String baseIdString = firstValue(params, key);

    if (StringHelper.is(baseIdString)) {
        try {
            int baseId = Integer.parseInt(baseIdString);
            Optional<Base> optional = dao.getById(baseId);

            if (optional.isPresent()) {
                base = optional.get();
            } else {
                return null;
            }
        } catch (NumberFormatException nfe) {
            return null;
        }
    }

    return base;
}
```

Mentre i metodi principali della classe sono quelli delle operazioni importanti che vengono fatte:

- doGet(): metodo eseguito alla richiesta http get
- doPost(): metodo eseguito alla richiesta http post
- doPut(): metodo eseguito alla richiesta http put
- delete(): metodo eseguito alla richiesta http delete

Questi metodi funzionano per tutte le sottoclassi, quindi non vi è la necessita di riscriverli per ogni classe che creiamo.

Metodo doGet:

```
/**
 * Do get.
 *
 * @param request Http request.
 * @param response Http response.
 * @throws ServletException Error in servlet.
 * @throws IOException      Input Output Error.
 */
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    SessionManager sm = new SessionManager(request.getSession());
    try {
        initConnector();
        boolean hasP = hasRequiredPermission(sm.getUserId(), requiredGetPermission());
        if (sm.isValidSession() && hasP) {
            DbDao dao = getDao(sm.getUserId());

            try {
                doGetById(request, response, dao);
            } catch (ServletNfe snfe) {
                doGetAll(response, dao);
            }

            closeConnector();
        } else {
            unauthorized(request, response);
        }
    } catch (Exception e) {
        internalServerError(response, e.getMessage());
    }
}
```

Metodo che controlla se l'utente è autenticato, nel caso lo fosse controlla se ha i permessi di eseguire la richiesta. Se li ha, a dipendenza della richiesta, esegue la richiesta come getById oppure come getAll, la prima seleziona l'elemento dal database, tramite l'id passato nella richiesta, mentre il secondo prende tutti i parametri nel database. Il seguente metodo doDelete:

```
/**
 * Do delete.
 *
 * @param request Http request.
 * @param response Http response.
 * @throws ServletException Error in servlet.
 * @throws IOException      Input Output Error.
 */
@Override
protected void doDelete(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    SessionManager sm = new SessionManager(request.getSession());
    try {
        initConnector();
        boolean hasP = hasRequiredPermission(sm.getUserId(), requiredDeletePermission());
        if (sm.isValidSession() && hasP) {
            DbDao dao = getDao(sm.getUserId());

            try {
                int id = getId(request);

                Optional<Base> optional = dao.getById(id);
                if (optional.isPresent()) {
                    if (dao.delete(optional.get())) {
                        ok(response, getOkResponse(id));
                    } else {
                        internalServerError(response, getNotCapableResponse());
                    }
                } else {
                    notFound(request, response);
                }
            } catch (ServletNfe snfe) {
                notFound(request, response);
            }

            closeConnector();
        } else {
            unauthorized(request, response);
        }
    } catch (Exception e) {
        internalServerError(response, e.getMessage());
    }
}
```

Controlla sempre che l'utente sia autenticato ed abbia i permessi per eseguire l'operazione, poi seleziona l'elemento tramite il suo id, nel caso esiste lo elimina. Il metodo doPost():

```
/**
 * Do post, create new element.
 *
 * @param request Client request.
 * @param response Client response.
 * @throws ServletException Error in the servlet.
 * @throws IOException      Error writing to client.
 */
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
    SessionManager sm = new SessionManager(request.getSession());
    try {
        initConnector();
        boolean hasP = hasRequiredPermission(sm.getUserId(), requiredPostPermission());
        if (sm.isValidSession() && hasP) {
            String[] requiredAttributes = getRequiredPostParameters();

            ServletRequestAnalyser sra = new ServletRequestAnalyser(
                requiredAttributes,
                request.getParameterMap(),
                new StringValidator());

            switch (sra.getStatus()) {
                case ServletRequestAnalyser.NOT_VALID_PARAMETERS:
                    notValidParameters(response, sra.getNotValidParameters());
                    break;
                case ServletRequestAnalyser.MISSING_PARAMETERS:
                    missingParameters(response, sra.getMissingParameters());
                    break;
                case ServletRequestAnalyser.OK:
                    executePost(response, sra, sm.getUserId());
                    break;
                default:
                    notAcceptable(request, response);
                    break;
            }
        } else {
            unauthorized(request, response);
        }
    } catch (Exception e) {
        internalServerError(response, e.getMessage());
    }
}
```

Controlla che l'utente sia autenticato ed abbia i permessi per eseguire la richiesta, poi viene controllato che abbia inserito tutti i parametri necessari e che siano validi, nel caso in cui tutti questi controlli vanno a buon fine, viene creato l'oggetto, tramite il modello DAO del model. In caso contrario viene ritornato al client un errore. Stesso concetto vale per il metodo doPut() che esegue però il metodo update della connessione al database DAO.

Per quanto riguarda le sottoclassi viene mostrata solamente quella del Group, siccome rappresenta sia le l'utilizzo di oggetti esterni sia l'utilizzo di componenti interni, come stringhe. La classe è GroupServlet, che implementa i metodi astratti, come segue:

- getDao()

```
return new DbGroupDao(connector, actionBy);
```

- getJson()

```
return new GroupJson(base);
```

- getModel()

```
return Group.class;
```

- **mapToBase():** questo metodo controlla che sia istanziato l'accesso al database, nel caso non lo fosse ritorna valore nullo. Poi prende il valore del nome dai parametri ed il numero del gruppo padre, che viene subito trasformato in oggetto Group. Se il nome non è settato viene ritornato valore nullo. Mentre se l'oggetto base esiste viene creato un oggetto Group con quest'ultimo, altrimenti viene creato con l'autore dell'operazione.

```
if (dao == null) {
    return null;
}

String name = firstValue(params, "name");
Group parentGroup = (Group) getBase(params, dao, "parent_group");

if (!StringHelper.is(name)) {
    return null;
}

if (base != null) {
    return new Group(base, name, parentGroup);
}

return new Group(actionBy, name, parentGroup);
```

- **getPath()**

```
return "data/group";
```

- **requiredGetPermission()**

```
return "user";
```

- **requiredPostPermission()**

```
return "db";
```

- **requiredPutPermission()**

```
return "db";
```

- **requiredDeletePermission()**

```
return "db";
```

Anche queste query, sono state testate con il programma da linea di comando CURL, con il quale ho testato tutte le operazioni di tutte le classi. Sotto viene mostrato come eseguire i test per le classi, utilizzando sempre come esempio l'oggetto Group.

Testare il metodo **getAll()**, quindi metodo http GET, inserire il metodo nella richiesta, poi il file nel quale sono salvati i cookies, quindi la sessione ed infine l'URL a cui eseguire la richiesta (per gli altri oggetti sostituire la parola **group**, con il nome del model desiderato).

```
$ curl -X GET -c c -b c http://localhost:8080/freqline-be/data/group
```

Per quanto riguarda il metodo **getById**, si utilizza sempre il metodo GET, ma si aggiunge alla fine un id, valgono le stesse regole che per il metodo **getAll()**:

```
$ curl -X GET -c c -b c http://localhost:8080/freqline-be/data/group/1
```

Per quanto riguarda il metodo add() si utilizza il metodo http POST, con i parametri inviati come dati. Si inserisce il metodo POST nella richiesta e si utilizza sempre il file per i cookies. Nel primo esempio si crea un gruppo senza parent group, mentre nel secondo con il primo come parent group (si assume che il database sia stato appena inizializzato).

```
$ curl -X POST -c c -b c -d "name=Gruppo 1" http://localhost:8080/freqline-be/data/group
$ curl -X POST -c c -b c -d "name=Gruppo 2&parentGroup=1" http://localhost:8080/freqline-be/data/group
```

Per eseguire un aggiornamento si fa una richiesta simile a quella di inserimento ma con il metodo PUT ed un id come parametro.

```
$ curl -X PUT -c c -b c -d "id=1&name=Gruppo 1.1" http://localhost:8080/freqline-be/data/group
```

Mentre per il metodo DELETE, si fa una richiesta simile al getByld ma utilizzando il metodo DELETE:

```
$ curl -X DELETE -c c -b c http://localhost:8080/freqline-be/data/group/2
```


4.2 Frontend

Il front-end è sviluppato con il *web framework* javascript AngularJS, cioè un applicativo che serve semplificare lo sviluppo delle applicazioni web, alleggerire il lavoro degli sviluppatori e mantenere un codice pulito e ben leggibile. AngularJS è un framework sviluppato e mantenuto da Google, il quale può essere utilizzato con due architetture di pattern per lo sviluppo web, cioè MVC (Model View Controller) e MVVM (Model View ViewModel). MVC a 3 elementi fondamentali che sono:

- Model: i modelli, quindi i dati
- View: Le view, le interfacce grafiche che vengono mostrate agli utenti
- Controller: Le azioni che vengono fatte, come caricare le view e modificare i dati

Mentre la struttura MVVM, è una architettura sviluppata sugli eventi, quindi tutte le azioni devono essere scatenate da un evento. Anche in questa struttura vi sono 4: elementi:

- Model: i modelli dei dati
- View: Le view, le interfacce grafiche che vengono mostrate agli utenti
- ViewModel: Un'astrazione dei dati da inserire nelle view
- Binder: Si occupa di legare i comandi agli eventi che avvengono nella view

Per questo progetto verrà utilizzato il pattern MVC. Siccome AngularJS è un framework javascript, viene scritto in dei file JS, utilizzando le strutture tipiche del linguaggio.

La WebAPP in AngularJS, deve essere inizializzata, la quale può essere utilizzata in modalità pagina singola ed in modalità pagine multiple, il punto di forza di utilizzare un framework per utilizzare diverse pagine, è che non è necessario ricaricare più volte le pagine per navigare fra esse, ma la base dell'applicativo viene caricata all'inizio, poi mano a mano che si necessitano pagine diverse vengono caricate solamente quelle, per esempio il footer, la barra di ricerca, vengono caricate solamente una volta, così come gli script (JS) e i fogli di stile (CSS). Questo permette una maggiore velocità di caricamento ed una minore latenza durante la navigazione.

Il segue codice serve per inizializzare l'applicazione ed il *router*, cioè il sistema che per ogni indirizzo scritto nella barra di ricerca, carica la view ed il controller giusto.

Nel codice sono configurate tutte le possibili view, per ognuna di esse, bisogna specificare a quale indirizzo corrisponde, quale file HTML caricare ed eventualmente si può inserire direttamente nella configurazione, altrimenti si può inserire nella view, nel caso la view dovesse caricare altri componenti, i quali hanno dei controller specifici.

In questo caso la webapp verrà chiamata **FreqlineAPP**, e dovrà caricare i seguenti 2 moduli **ngRoute**, che serve per analizzare la richiesta e reindirizzare sulla giusta view ed il modulo **ngSanitize** che serve per inserire tramite javascript del codice html, che venga interpretato e non scritto come testo.

Nella configurazione vengono configurate 7 views, la pagina index, quella degli utenti, la pagina di dettaglio degli utenti, che necessita un id, una pagina con lo stato del generatore di frequenze e due di errore, che corrispondono rispettivamente all'errore HTTP 401 e 404 che sarebbero UNAUTHORIZED (operazione non autorizzata) e NOT FOUND (elemento non trovato). Inoltre, viene configurato un redirect, che viene eseguito per ogni richiesta sconosciuta, alla pagina dell'errore 404.

```
var app = angular.module('FreqlineAPP', ['ngRoute', 'ngSanitize']);

app.config(function ($routeProvider) {
  $routeProvider.when('/', {
    templateUrl: 'views/index.html'
  });

  $routeProvider.when('/users', {
    templateUrl: 'views/users.html'
  });

  $routeProvider.when('/user/:id', {
    templateUrl: 'views/user.html',
  });

  $routeProvider.when('/login', {
    templateUrl: 'views/login.html'
  });

  $routeProvider.when('/status', {
    templateUrl: 'views/status.html',
    controller: 'StatusController'
  });
});
```

```
$routeProvider.when('/401', {
  templateUrl: 'views/errors/401.html',
  controller: 'Error401Controller'
});

$routeProvider.when('/404', {
  templateUrl: 'views/errors/404.html',
  controller: 'Error404Controller'
});

$routeProvider.otherwise({
  redirectTo: '/404'
});
}).run(function($rootScope, $route) {
  $rootScope.$route = $route;
});
```

Per richiedere i dati all'back-end vengono utilizzati i services, cioè degli elementi che si occupano di fare le richieste AJAX, che con i browser moderni ha un problema, cioè mantiene in cache i dati, quindi le pagine non vengono aggiornate. Per ovviare a questo problema, si invia come parametro HTTP, lo UNIX time, che sarebbe il numero di secondi passati dal primo gennaio 1970.

Per ogni service vi possono essere molteplici azioni, come per esempio nel service di login, vi è un'azione per eseguire il login ed una per controllare se il login è stato fatto.

Ogni service è però legato ad un indirizzo del back-end. Poi per ognuna delle azioni si associa un metodo specifico, come GET, POST, PUT e DELETE. Tutte queste richieste sono fatte in maniera asincrona, questo vuol dire che il caricamento della pagina continua indifferentemente dalla risposta del server, durante le richieste dei dati. Quando poi il server risponde i dati vengono automaticamente inseriti nelle view, questo grazie al framework AngularJS. Questa tecnica viene utilizzata per migliorare l'esperienza utente, quindi iniziare a caricare le componenti grafiche della pagina e secondariamente inserirne i dati. All'utente non sembra che vi sia un tempo di caricamento così lungo, siccome vede che la pagina inizia a prendere forma.

```
app.factory('LoginService', ['$http', function($http) {
  var service = {};
  var address = "localhost";
  var port = 8080;
  var baseApi = "/freqline-be"
  var urlBase = "http://" + address + ":" + port + baseApi;
  var baseApi = baseApi;
  urlBase += "/action/login";
  $http.defaults.withCredentials = true;

  service.login = function (username, password) {
    let url = urlBase + "?t=" + new Date().getTime() + "&username=" + username + "&password=" +
password;
    let data = "username=" + username + "&password=" + password;
    return $http.post(url, data, {}).then(function (response){
      return response.data;
    },function (error){
      return error;
    });
  };

  service.isLoggedIn = function() {
    let url = urlBase + "?t=" + new Date().getTime();
    let data = "";

    return $http.get(url, data, config).then(function (response) {
      return {response: response.data.isLoggedIn};
    }, function(error) {
      return {response: false};
    });
  };

  return service;
}]);
```

I controller servono per inserire i dati nelle view e gestire le azioni dell'utente sulla view. Quindi essi prima richiedono i dati dai services, sempre in modo asincrono; quando li ricevono li inseriscono nelle view, quando

poi viene eseguita un'azione sulla view. (che è controllata dal controller, come un click su un bottone), il controller esegue un'operazione, che può essere inviare i dati nuovi al server oppure li aggiorna. In questo controller si vede solamente il lato delle operazioni eseguite dal controller. Esso tramite il Model di login, che contiene username e password esegue la richiesta ai service, nel caso in cui la risposta del service è positiva cambia la view in quella dello stato del generatore di frequenze.

```
app.controller('LoginController', ['$scope', '$location', 'LoginService', function
($scope, $location, loginService) {
    $scope.login = function(login) {
        loginService.login(login.username, login.password).then(function (data) {
            if (data.message === "ok") {
                $location.path('/status');
            } else {
                $scope.message = data.message;
            }
        })
    }
}]);
```

Le view sono semplicemente i file html, che interpretati dal browser creano le pagine web che l'utente finale vede. Se il controller non è selezionato direttamente nella configurazione, ne carica uno (come nel caso seguente). Il codice della view sottostante rappresenta un form di login, il quale contiene un titolo della pagina, un sotto titolo 2 campi di input uno text per lo username ed uno per la password di tipo password, così che rimanga nascosta. Infine, vi è un bottone per eseguire il login di tipo submit. Ed un campo per scrivere gli errori di login. Il tutto in un form.

```
<div ng-controller="LoginController" class="full-size">
  <form novalidate class="css-form">
    <h3>Freqline</h3>
    <p>Gestore di generatore di frequenze</p>
    <p>&nbsp;</p>
    <div class="user-input-wrp">
      <br/>
      <input type="text" class="inputText" ng-model="login.username" required>
      <span class="floating-label">username</span>
    </div>
    <div class="user-input-wrp">
      <br/>
      <input type="password" class="inputText" ng-model="login.password" required>
      <span class="floating-label">password</span>
    </div>
    <label style="color:red; padding: 10px 10px 0 10px">
      {{message}}
    </label>
    <br>
    <input type="submit" ng-click="login(login)" value="Login" />
    <p style="padding: 15px 10px;">&copy; 2019 Giulio Bosco</p>
  </form>
</div>
```

I model, che sarebbero i dati servono per comunicare fra controller, views, e services. Nel caso di AngularJS e più in generale di JavaScript, che è un linguaggio non tipizzato (che vuol dire che non bisogna dichiarare i tipi di dati), i model possono contenere qualunque cosa. Quindi sono molto variabili.

In questo applicativo lo stile è fatto con CSS (Cascading Style Sheets). Utilizzando poche regole per mantenere la grafica pulita e semplice, seguendo la filosofia *less is more* cioè rendere le interfacce più semplici ed intuitive possibile.

4.3 Arduino

4.3.1 Arduino – python

Il codice di questa parte di progetto è scritto in python, inizialmente volevo scriverlo utilizzando il framework flask, per avere un web server funzionante e facile da utilizzare. Questo framework va inizializzato con il gestore di pacchetti per python, che è PIP. Per installare ed usare PIP è necessario diverso spazio sulla memoria, di cui l'Arduino YUN dispone, quindi ho cercato di utilizzare una memoria micro sd per estendere lo spazio disponibile, siccome ho avuto diversi problemi ed ho perso più di 2 ore, ho cambiato strategia, utilizzando il web server di base per python ed adattandolo alle mie esigenze. Creando gli elementi di cui necessitavo, principalmente i metodi per la gestione delle richieste ACC.

Per mantenere controllati i dati del bridge, utilizzo 2 thread, questo crea un problema, siccome il Bridge è solamente uno e può essere istanziato solamente una volta, ho creato un oggetto sincrono, condiviso fra tutte le thread, in maniera da essere in grado di poter accedere allo stesso oggetto da diverse thread in parallelo.

Codice sincrono:

```
import sys
sys.path.insert(0, '/usr/lib/python2.7/bridge')

from threading import Lock

from bridgeclient import BridgeClient

class BridgeGlobal:
    def __init__(self):
        """
        Create global bridge, initialize the bridge client and the lock for synchronization.
        """
        # initialize bridge
        self.bridge = BridgeClient()
        # initialize synchronization lock
        self.lock = Lock()

    def get(self, key):
        """
        Get from the bridge with synchronization.
        :param key: Key of the value (memory address).
        :return: Value of the key.
        """
        # acquire the synchronization lock
        self.lock.acquire()
        # get the value
        value = self.bridge.get(key)
        # release the synchronization lock
        self.lock.release()
        # return the value
        return value

    def put(self, key, value):
        """
        Put the value in the bridge with synchronization.
        :param key: Key of the value (memory address).
        :param value: Value to set.
        """
        # acquire the synchronization lock
        self.lock.acquire()
        # set the value
        self.bridge.put(key, value)
        # release the synchronization lock
        self.lock.release()
```

Nel quale si notano un costruttore che inizializza l'oggetto bloccante (per sincronizzare l'utilizzo dei metodi) ed il Bridge, verso il lato ATmega dell'Arduino, poi vi sono due metodi uno per richiedere un valore (get), che legge dal bridge ed uno per scrivere sul Bridge (put).

Le due thread sono simili, vengono istanziate, assieme ad un oggetto di tipo event, che serve semplicemente per interrompere il ciclo della thread, per il quale esistono 2 metodi, uno per controllare se la thread è stata interrotta ed uno per interrompere la thread. Il ciclo di vita delle thread è nel metodo run, che richiama altri metodi di aiuto, per spezzettare il codice (rendendolo più leggibile).

```
import threading
from time import sleep
from urllib2 import urlopen

class RemoteThread(threading.Thread):
    def __init__(self, bridge, host):
        super(RemoteThread, self).__init__()
        self._stop_event = threading.Event()
        self.bridge = bridge
        self.host = host
        self.status = 1

    def is_interrupted(self):
        return self._stop_event.is_set()

    def interrupt(self):
        self._stop_event.set()

    def is_status_changed(self):
        status = self.bridge.get("r")
        if not status == self.status:
            self.status = status
            return True
        return False

    def execute_http_req(self):
        request = "http://" + self.host + "/acc?key_c=AAAAAA&action=remote=content=t"
        return urlopen(request).read()

    def run(self):
        while not self.is_interrupted():
            if self.is_status_changed():
                response = self.execute_http_req()
                print(response)
                sleep(0.1)
```

Mentre il web-server estende il web server di base di python (funzionante solamente per la versione di python2, che è quasi deprecata, ma sull'Arduino YUN è supportata per il momento questa in maniera completa, mentre la versione 3 non ancora in maniera stabile). Il quale ha il metodo do_GET riscritto, per eseguire il codice che ci interessa.

```
def do_GET(self):
    """
    Do at get http request.
    """
    if "/acc?" in self.path:
        # if ACC request execute self.acc()
        self.acc()
    elif ("/alive" == self.path) or ("/" == self.path):
        # if alive request execute self.alive()
        self.alive()
    else:
        # otherwise 404 file not found
        # send response 404
        self.send_response(404, "page not found")
        # end the http header
        self.end_headers()
        # write return response not found
        self.wfile.write(ResponseRender(self.server.key_manager, self.server.bridge).not_found(self.path))
```

Sono possibili solamente due richieste per questo web server, una per i comandi ACC, quindi di comunicazione fra arduino e server, mentre l'altra serve semplicemente per controllare che l'ACC client sia attivo.

4.3.2 Arduino – c, ino

Il codice da eseguire sull'ATmega32u4 era già parzialmente scritto, il quale ha dei metodi scritti dall'ex allievo della sezione degli elettronici Nicola Finke, per gestire la frequenza, mentre io ho aggiunto la parte di codice di comunicazione con l'Arduino la OpenWRT.

La frequenza è gestita con la libreria SPI, che funziona in maniera diversa sull'Arduino YUN rispetto alla versione UNO, quindi ho dovuto cercare di capire come modificare il circuito.

L'implementazione della libreria SPI su arduino YUN, ha una mappatura dei pin diversa, al posto di utilizzare i pin 10, 11, 12, 13 utilizza i pin ISCP.

SPI – Arduino UNO	SPI – Arduino YUN
Pin 10	Pin fisico inesistente, si può utilizzare punto di saldatura RX
Pin 11	ISCP-4
Pin 12	ISCP-1
Pin 13	ISCP-3

Per quanto riguarda il pin 10, che non ha una rispettiva mappatura sull'Arduino YUN, si può utilizzare il punto di saldatura RX, posto sull'Arduino vicino al led corrispondente.

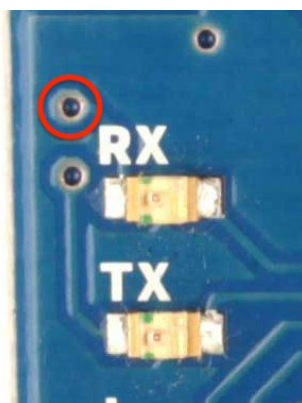


Figura 22 - Collegamento RX Arduino

Il circuito è stato modificato come segue, per essere compatibile con i nuovi componenti, Arduino YUN, microfono e telecomando. Durante lo sviluppo del progetto non è stato trovato nessun telecomando che potesse arrivare in tempo utile, quindi è stato simulato da un cavo con una resistenza (essa serve per evitare di creare corto circuiti).

Nello schema si può notare che il generatore di frequenze (AD9833) è collegato a dei pin diversi dal montaggio precedente, sui pin ICSP 3 (al posto del pin 13), ICSP 4 (al posto del pin 11), e sulla pista RX (al posto del pin 10). Il telecomando simulato si collega al pin 3, impostato come OUTPUT, lasciato sempre HIGH, così da permettere di avere un pin a 5V, quindi utilizzato come alimentazione ed inoltre collegato al pin 5 come INPUT digitale. Anche il microfono utilizza lo stesso stratagemma per il pin 5V (sul pin 7 in modalità OUTPUT HIGH fisso), mentre l'uscita analogica entra sul pin A1.

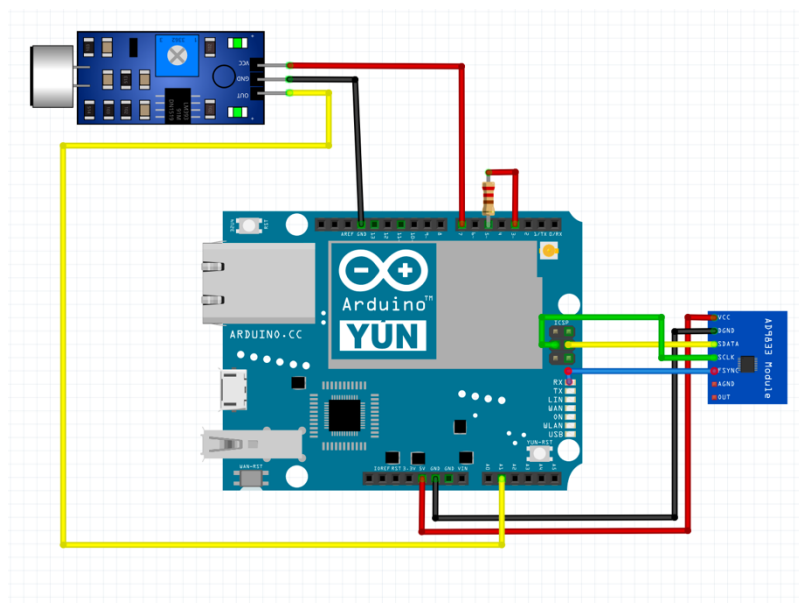


Figura 23 - Circuito Arduino Finale

Per adattare il circuito sono stato aiutato dal docente Thomas Bartesaghi per effettuare la saldatura sulla pista RX. Dopo aver fatto la saldatura ho collegato il modulo AD9833 per la generazione delle onde ad ultrasuoni ed il potenziometro, prendendo il vecchio programma, caricandolo sull'Arduino YUN, esso dovrebbe funzionare come funzionava sull'Arduino UNO. Per testare questa componente basta semplicemente collegare il circuito ad una presa, tramite l'adattatore e l'Arduino YUN tramite la porta Micro USB ad un trasformatore, (necessario per alimentare il chip arduino). Poi ruotando il potenziometro si dovrebbe sentire un fischio emesso dall'altoparlante del generatore.

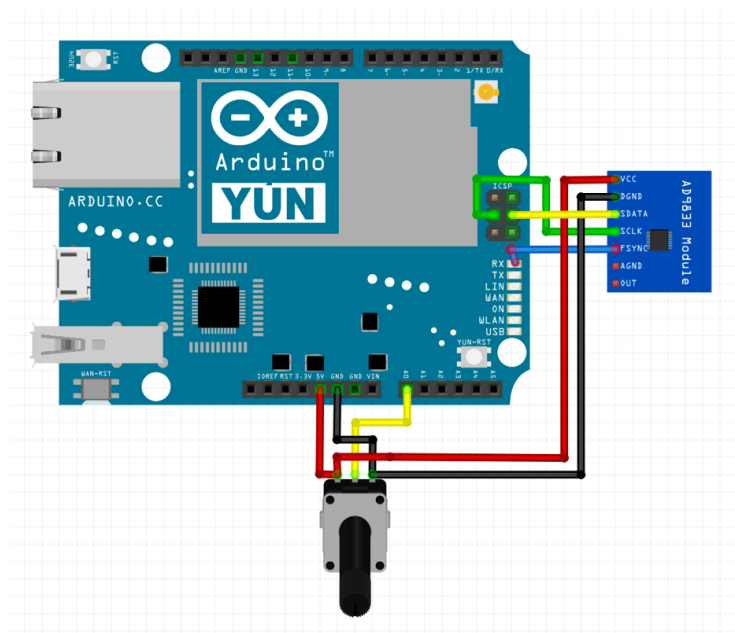


Figura 24 - Circuito Arduino di Test

Effettuando il test, il circuito non produceva nessuna onda sonora. Il problema rimane probabilmente nell'utilizzo della libreria SPI.

4.4 Installazione Raspberry

Il database, il back-end ed il front-end per poter essere messi in un apparecchio di piccole dimensioni e rimanere indipendenti vengono messi su un Raspberry PI, che funge da server. Questo Raspberry PI è stato installato, utilizzando la distribuzione di default Raspbian, che è una distribuzione Linux basata su Debian. La quale è stata scaricata dal sito ufficiale del progetto Raspberry.

Primo passo scrivere l'immagine sulla micro sd, con il comando:

```
sudo dd bs=1m if=path_of_your_image.img of=/dev/rdiskN conv=sync
```

Questo processo potrebbe durare diverso tempo, alla fine inserire la micro sd nel Raspberry, collegarlo ad uno schermo, ad una tastiera, ad una rete via cavo ed all'alimentazione. Quando sullo schermo compare la finestra di login (da CLI), inserire come nome utente `pi`, mentre come password `raspberry`, è molto consigliato abilitare ssh, così da potersi collegare in remote, con il comando:

```
sudo systemctl enable ssh
sudo systemctl start ssh
```

Poi come primo passo installare `mariadb`, siccome `mysql` non è disponibile per Raspbian, poi installare java e gradle per poter avviare la nostra applicazione:

```
sudo apt update
sudo apt install mariadb-server-10.5
sudo mysql_secure_installation
sudo apt install openjdk-8-jdk
```

Poi testare che java sia installato correttamente:

```
$ java -version
openjdk version "1.8.0_212"
OpenJDK Runtime Environment (build 1.8.0_212-8u212-b01-1+rp1-b01)
OpenJDK Client VM (build 25.212-b01, mixed mode)
$ javac -version
javac 1.8.0_212
```

Dopo di che scaricare gradle, ricordarsi di inserire il proxy (in questo caso sostituire con il proprio username: `user.name` e la propria password `pwd`), dopo di che estrarre il pacchetto ed aggiungere gradle alla variabile `path`, per poterlo eseguire come comando.

```
curl -proxy http://user.name:pwd@10.20.0.1:8080 -O /tmp
https://services.gradle.org/distributions/gradle-5.2.1-bin.zip
sudo unzip -d /opt/gradle /tmp/gradle-*.zip
echo "export GRADLE_HOME=/opt/gradle/gradle-5.2.1" >> /etc/profile.d/gradle.sh
echo "export PATH=${GRADLE_HOME}/bin:${PATH}" >> /etc/profile.d/gradle.sh
sudo chmod +x /etc/profile.d/gradle.sh
source /etc/profile.d/gradle.sh
gradle -v
```


Poi creare il database per il progetto utilizzando lo script SQL sql/db.sql, con il comando:

```
mysql -u root -p < sql/db.sql
```

Poi creare un nuovo utente con tutti i permessi sul database:

```
mysql -u root -p  
CREATE USER 'freqline'@'localhost' IDENTIFIED BY '1234qwer';  
GRANT ALL PRIVILEGES ON freqline.* TO 'freqline'@'localhost';
```

Dopo di che provare ad avviare la web-app, con il comando:

```
./gradlew appRun
```

Per testare il programma collegarsi con un browser all'indirizzo:

```
http://ip:8080/freqline/
```

Si dovrebbe vedere la pagina.

5 Test

Diversi dei test che sono stati eseguiti, sono stati fatti a livello di singole componenti del codice, i quali sono descritti più in dettaglio nel capitolo dell'implementazione del codice che viene testato. Purtroppo per mancanza di conoscenze (come eseguire unit testing in presenza di un database) non ho eseguito un vero e proprio unit testing, ma ho eseguito dei test manuali scrivendo del codice di test controllato manualmente nei metodi main delle classi, spesso hanno anche subito delle variazioni o sono stati modificati per eseguire i test di diverse parti delle classi. Non tutti i test sono stati documentati, per dimenticanza.

5.1 Protocollo di test

In questo progetto sono stati eseguiti la maggior parte dei test, durante lo sviluppo del software, testando gli elementi singolarmente, sullo stile dello Unit Testing, quindi eseguire i test di ogni metodo, verificare il suo comportamento sia in caso di operazioni "giuste" (cioè operazioni che devono dare esiti positivi) sia per operazioni sbagliate (che devono dare esiti negativi). Questi test non sono stati eseguiti con degli Unit Test, siccome non sono capace di utilizzare questo approccio con un database, ma solamente con dei metodi procedurali. Per questo motivo ho deciso di eseguire i test manualmente, nel caso dei programmi scritti in Java, ho utilizzato il metodo main delle classi, nel quale ho scritto del codice, che ho eseguito manualmente e sempre manualmente ho controllato il risultato del test. Nel caso delle servlet i test sono stati eseguiti con il comando CURL, un comando che permette di fare richieste HTTP, configurando tutti i possibili parametri del protocollo.

Per quanto riguarda le interfacce sono state testate manualmente, provando ad utilizzare l'applicativo come farebbe un utente normale.

Il codice dell'Arduino, sia la parte scritta in python che quella scritta in c (ino), è stata testata anche quella manualmente, eseguendo il codice. Meno approfonditamente siccome ero in ritardo rispetto al diagramma di GANTT.

Tutti questi test possono essere trovati nei capitoli dell'implementazione delle varie componenti. Dopo averle sviluppate sono subito state testate. Mentre in questo capitolo si trovano i test generali sul sistema.

Test Case:	TC-001	Nome:	Gestione generatore di frequenze ad ultrasuoni tramite pagina WEB
Riferimento:	REQ-01		
Descrizione:	Testare che la pagina web per gestire il generatore di frequenze ad ultrasuoni funzioni correttamente con tutte le sue funzionalità.		
Prerequisiti:	L'applicativo WEB ed il generatore devono essere accesi.		
Procedura:	<ol style="list-style-type: none"> 1. Eseguire il login sull'applicativo con un utente di livello user o superiore. 2. Regolare la frequenza con un valore fra 0 e 25'000 MHz 3. Accendere il generatore 4. Spegnerne il generatore 		
Risultati attesi:	Il generatore dovrebbe accendersi con la frequenza selezionata e poi spegnersi.		

Test Case:	TC-002	Nome:	Test del funzionamento del telecomando del generatore.
Riferimento:	REQ-02		
Descrizione:	Testare che il telecomando per gestire il generatore sia funzionante.		
Prerequisiti:	L'applicativo WEB ed il generatore devono essere accesi.		
Procedura:	<ol style="list-style-type: none"> 1. Premere il bottone sul telecomando 		
Risultati attesi:	Il generatore dovrebbe accendersi e poi spegnersi.		

Test Case:	TC-003	Nome:	Test del funzionamento del microfono del generatore.
-------------------	--------	--------------	--

Riferimento:	REQ-03	
Descrizione:	Testare che il microfono per gestire il generatore sia funzionante.	
Prerequisiti:	L'applicativo WEB ed il generatore devono essere accesi.	
Procedura:	<ol style="list-style-type: none"> 1. Fare un rumore forte 2. Aspettare che il generatore si spegna autonomamente, in base al timer impostato tramite l'applicativo WEB. 	
Risultati attesi:	Il generatore dovrebbe accendersi e poi spegnersi.	

Test Case:	TC-004	Nome:	Test del funzionamento della pagina di amministrazione utenti.
Riferimento:	REQ-04		
Descrizione:	Testare che la pagina di amministrazione degli utenti funzioni correttamente.		
Prerequisiti:	L'applicativo WEB ed il generatore devono essere accesi.		
Procedura:	<ol style="list-style-type: none"> 1. Eseguire il login nell'applicativo con un utente di livello amministratore (admin). 2. Andare nella pagina della gestione degli utenti. 3. Creare un nuovo utente di livello user 4. Cambiare il livello di permessi dell'utente ad amministratore (admin) 5. Eseguire il login con l'utente creato 6. Cambiare il livello di permessi dell'utente usato precedentemente ad user 		
Risultati attesi:	Tutte le operazioni indicate dovrebbero riuscire senza particolari problemi.		

5.2 Risultati test

I test elencati nel precedente capitolo, sono stati eseguiti, ed hanno avuto i seguenti risultati:

Test	Risultato	Note
TC-001	NON passato	Il generatore non funziona, quindi non testabile
TC-002	NON passato	Il generatore non funziona, quindi non testabile
TC-003	NON passato	Il generatore non funziona, quindi non testabile
TC-004	Passato	

I primi 3 test effettuati sull'applicativo non sono passati siccome il generatore di frequenze non funziona. Quindi non possono essere proprio eseguiti i test.

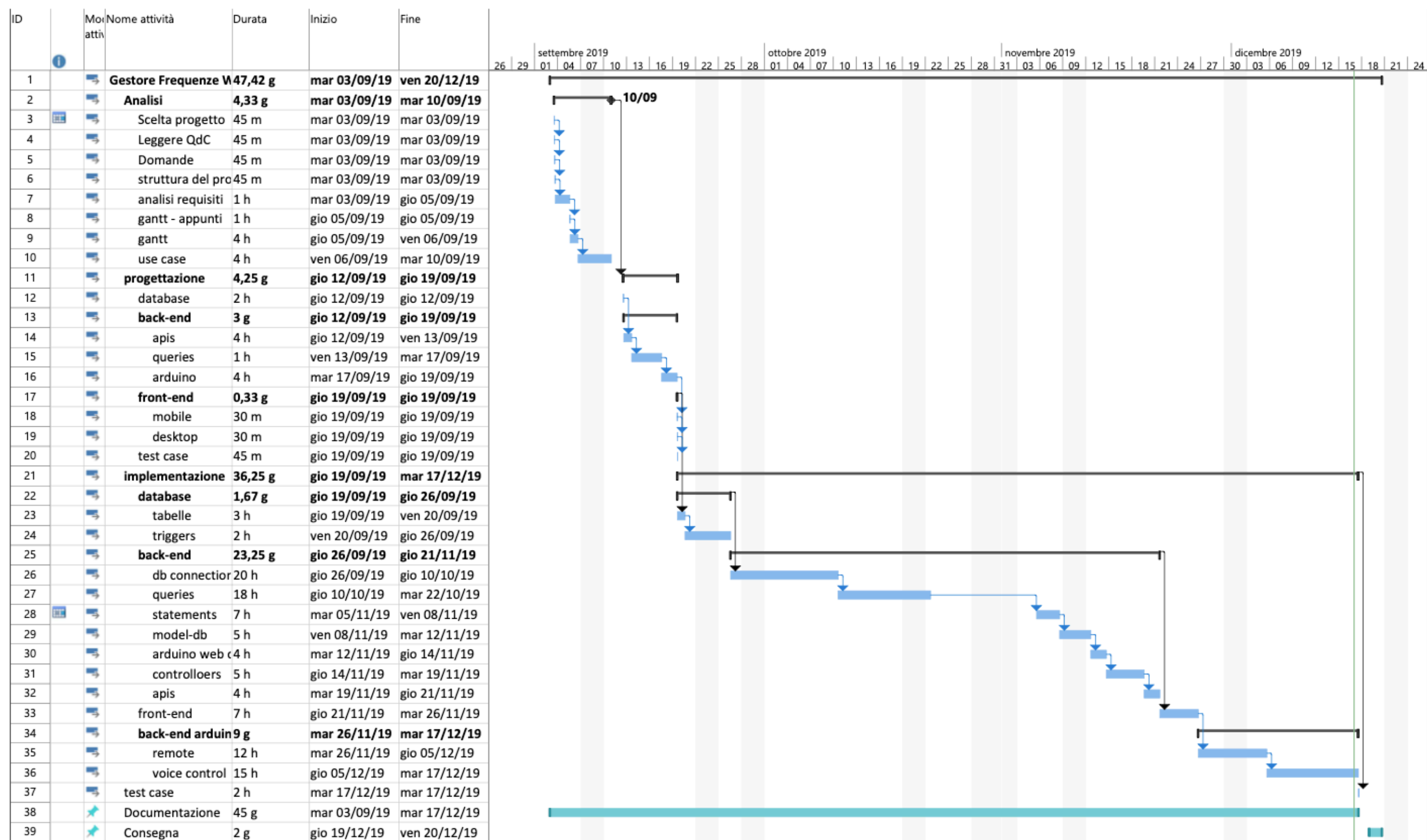
5.3 Mancanze/limitazioni conosciute

Questo progetto attualmente ha due mancanze, la prima più importante il generatore non funziona, il microcontrollore non riesce a comunicare con il circuito che genera le frequenze, quindi il prodotto è incompleto. Mentre per quanto riguarda la seconda mancanza è la password temporanea, che non è stata implementata.

Per quanto riguarda la prima mancanza si potrebbe risolvere in tre modi, il primo sarebbe cercare di capire come risolvere il problema della comunicazione con il generatore di frequenze mentre il secondo dovrebbe prendere in considerazione l'utilizzo dell'Arduino UNO e richiederebbe di trovare un modo per comunicare fra l'Arduino UNO ed il Raspberry, sul quale si potrebbe installare direttamente il codice installato su OpenWRT dell'Arduino YUN, facendo un adattamento della classe BridgeGlobal, in maniera che funzioni con il nuovo modo di comunicare, poi nel database modificare l'indirizzo del generatore, del microfono e del telecomando a localhost. La terza opzione sarebbe riuscire a comandare il modulo AD9833 direttamente tramite il Raspberry PI, ed utilizzare parte della seconda opzione per quanto riguarda la classe BridgeGlobal.

Mentre per quanto riguarda il problema del programma

6 Consuntivo



L'immagine soprastante rappresenta il diagramma di GANTT consuntivo, nel quale si possono notare alcune differenze, non erano stati calcolati i giorni di vacanza dei morti ed un giorno nel quale sono stato alla giornata informativa per il servizio militare. Per quanto riguarda le altre differenze, sono date dal fatto che alcune attività sono state svolte più velocemente mentre le altre hanno impiegato più tempo. Le attività sono state descritte troppo nel dettaglio, siccome con il corso dello sviluppo del progetto mi sono reso conto che alcune sarebbe stato più efficace farle in un altro ordine, come per esempio fare l'attività 30 prima della 27.

7 Conclusioni

La mia soluzione implica l'utilizzo di diversi componenti, diverse, basate su diverse tecnologie, le quali devono lavorare in maniera sincrona per permettere il corretto funzionamento del progetto. Purtroppo, due elementi non sono riusciti a collegarli fra loro, l'Arduino YUN ed il generatore di onde AD9833.

7.1 Sviluppi futuri

Come già messo in evidenza nel capitolo 5.3 si potrebbe cercare di trovare una soluzione al problema della comunicazione fra l'Arduino YUN ed il modulo AD9833.

7.2 Considerazioni personali

Da questo progetto ho imparato che prima di iniziare a sviluppare subito tutto l'applicativo che è stato progettato bisogna prima testare che i componenti fisici funzionino e bisogna iniziare dalla parte dell'applicativo a più basso livello, inteso come livello di programmazione. Per esempio, Arduino si programma ad un livello molto basso, vicino all'hardware, mentre per esempio le parti in Java e Python sono più distanti, quindi è probabile che generino meno problemi.

Per questo tipo di progetto, bisogna assicurarsi che le parti fondamentali del programma vengano soddisfatte, quindi farle per prime.

8 Glossario

API	Application Programming Interface: sistema per interfacciamento fra programmi diversi
back-end	Lato non visibile di un applicativo (dati su server)
CRUD	Create Read Update Delete: Operazioni generiche eseguibili su dei dati
CSS	Cascading Style Sheet: linguaggio per stile della formattazione (pagine web)
CURL	Comando per richieste http di UNIX
DAO	Data Access Object: pattern per accesso a dati
DBMS	DataBase Management System: gestore di database
front-end	Lato visibile di un applicativo (pagina web)
HTML	HyperText Markup Language: linguaggio formattazione (pagine web)
HTTP	HyperText Transfer Protocol: protocollo utilizzato per browser internet
Java	Linguaggio di programmazione (lato server)
JavaScript	Linguaggio di programmazione (lato client)
jdbc	Java DataBase Connectivity: Una API per la connessione al database tramite Java
JSON	JavaScript Object Notation: standard per formato di file e scambio di dati
Log	Traccia di tutte le operazioni fatte
MVC	Model View Controller: pattern per sviluppo web
MVVM	Model View ViewModel: pattern per sviluppo web
MySQL	Implementazione server per database (DBMS)
proxy	Sistema per centralizzare le richieste da inoltrare in internet (con cache)
RESTful	Representational State Transfer: architettura per trasmissione dati per servizi web
SQL	Simple Query Language: linguaggio per manipolazioni dati

9 Bibliografia

9.1 Sitografia

<https://github.com/giuliobosco/JavaSAMT> JavaSAMT 7.11.2019

<https://medium.com/bedefended/implement-secure-cors-tomcat-f813e308b67c> implementation secure CORS 22.11.2019

<https://restapitutorial.com/> RESTful API tutorial 22.11.2019

<https://stackoverflow.com/questions/5991194/gradle-proxy-configuration> java -Gradle proxy 28.11.2019

<https://www.raspberrypi.org/downloads/raspbian/> Download Raspbian 28.11.2019

<https://en.wikipedia.org/wiki/Model-view-controller> Model View Controller Wikipedia 29.11.2019

<https://en.wikipedia.org/wiki/Model-view-viewmodel> Model View ViewModel Wikipedia 29.11.2019

<https://stackoverflow.com/questions/6348499/making-a-post-call-instead-of-get-using-urllib2> python making post request using urllib2 6.12.2019

<https://www.programiz.com/python-programming/global-local-nonlocal-variables> python global variable 6.12.2019

<https://www.pythoncentral.io/pythons-time-sleep-pause-wait-sleep-stop-your-code/> python sleep 6.12.2019

<https://forum.arduino.cc/index.php?topic=535387.0> Change or define new SPI pins 12.12.2019

<https://gist.github.com/2e1hmk/d102c36f259060f51d95aa5dbc4c2372> AD9833 Example 13.12.2019

<https://www.instructables.com/id/Step-by-Step-Guide-to-the-Arduino-Leonardo/> Step by Step Guide to the Arduino Leonardo, 14.12.2019

<https://www.arduino.cc/en/reference/SPI> Arduino - SPI, 14.12.2019

10 Allegati

- a. Diari
- b. QdC
- c. Schema delle componenti
- d. Codice (in formato digitale)
- e. Diagramma unico delle classi (in formato digitale)