Relazione
Prenotazioni Mediche
Ingegneria del software

D. Quattrociocchi (2024239) - G.Sepe (1983219)
L. Mastromattei (1958925)

20 dicembre 2024

Indice

Capitolo 1

Descrizione generale

Si intende progettare una piattaforma digitale per la gestione delle prenotazioni mediche, che consenta ai pazienti, previa registrazione, di prenotare visite specialistiche, ai medici di vedere i loro impegni e agli amministrativi di gestire i vari appuntamenti.

Un paziente può registrarsi sulla piattaforma fornendo le seguenti informazioni personali: nome, cognome, codice fiscale, data di nascita, indirizzo email, numero di telefono e indirizzo completo, composto da via, numero civico, CAP, città e provincia. Ad un potenziale paziente è possibile utilizzare la piattaforma, anche senza registrazione, per effettuare una ricerca dei medici disponibili. Ma solamente una volta completata la registrazione, il paziente avrà la facoltà di prenotare una visita medica, e in seguito lasciare un feedback relativo alla propria esperienza esprimendo due valutazioni numeriche, da 0 a 5, rispettivamente sulla soddisfazione complessiva e sulla puntualità del servizio ricevuto. La piattaforma consente inoltre ai pazienti di accedere alla cronologia delle prenotazioni effettuate.

Per quanto riguarda i medici, questi una volta inseriti nel sistema possono visualizzare sia le visite già svolte sia quelle ancora da svolgere. Inoltre, possono consultare le statistiche relative alla puntualità delle loro visite e alla soddisfazione dei propri pazienti. Al termine di ogni visita, il medico è tenuto a registrare sulla piattaforma il completamento della prestazione.

L'ultima entità che abbiamo realizzato è l'amministrativo, responsabile della supervisione della piattaforma e del corretto funzionamento di quest'ultima. L'amministrativo ha accesso a funzioni avanzate che gli permette di inserire

un nuovo medico nel sistema e aggiungere periodi di indisponibilità o nuove specializzazioni acquisite ai medici già presenti. Inoltre agli amministratici è affidata la gestione delle richieste di prenotazione dei pazienti, che possono decidere se accettare o rifiutare; queste entreranno a far parte della cronologia delle prenotazioni gestite dall'amministrativo, che può essere poi da questo visualizzata.

Capitolo 2

Analisi del Software

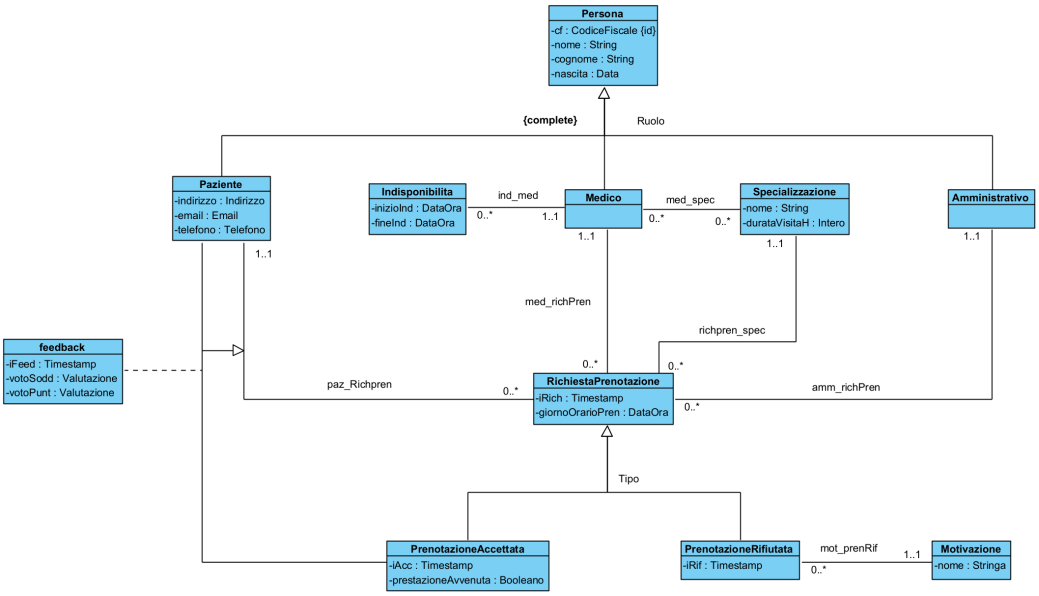
2.1 User Requirements

2.1.1 Lista dei Requisiti

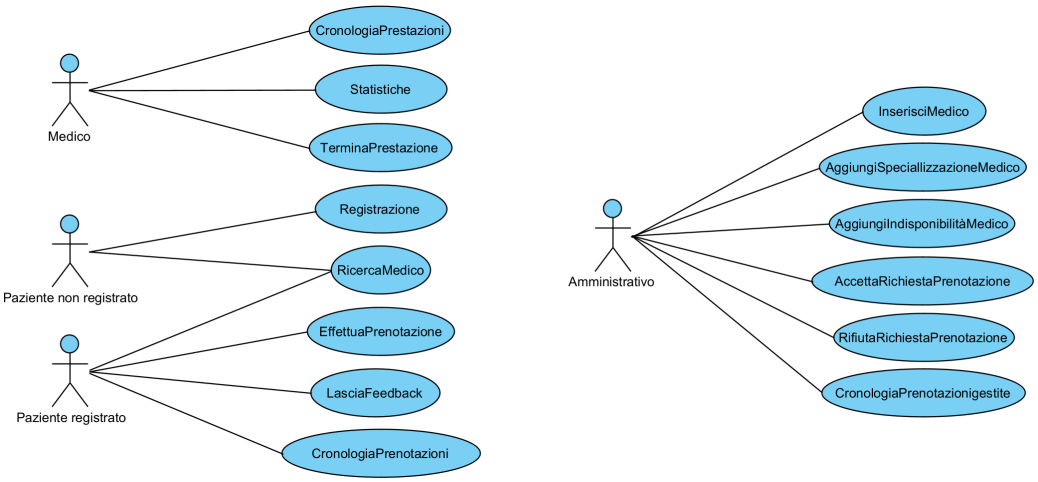
1. Paziente non registrato
    - 1.1 Ricerca medico
        - 1.1.1 Specializzazione
    - 1.2 Registrazione alla piattaforma
        - 1.2.1 Nome
        - 1.2.2 Cognome
        - 1.2.3 Data di nascita
        - 1.2.4 Codice Fiscale
        - 1.2.5 Email
        - 1.2.6 Recapito telefonico
        - 1.2.7 Indirizzo
            - 1.2.7.1 Via
            - 1.2.7.2 Civico
            - 1.2.7.3 CAP
            - 1.2.7.4 Città
            - 1.2.7.5 Provincia

2. Paziente
    - 2.1 Accesso cronologia prenotazioni
    - 2.2 Effettua prenotazione
        - 2.2.1 ID medico
        - 2.2.2 Specializzazione
        - 2.2.3 Data
    - 2.3 Lascia feedback
        - 2.3.1 ID prenotazione conclusa
        - 2.3.2 Voto soddisfazione
        - 2.3.3 Voto puntualità
    - 2.4 Ricerca medico
        - 2.4.1 Specializzazione
3. Medico
    - 3.1 Cronologia prestazioni
    - 3.2 Statistiche
    - 3.3 Termina prestazione
        - 3.3.1 ID prenotazione accettata
4. Amministrativo
    - 4.1 Inserisci medico
        - 4.1.1 Nome
        - 4.1.2 Cognome
        - 4.1.3 Data di nascita
        - 4.1.4 Codice Fiscale
    - 4.2 Aggiungi indisponibilità medico
        - 4.2.1 ID medico
        - 4.2.2 Data inizio
        - 4.2.3 Data fine
    - 4.3 Aggiungi specializzazione medico

2.1.2 UML diagram



2.1.3 Use-case diagram



2.2 System requirements

2.2.1 Requisiti funzionali

1. Paziente non registrato

   1.1 Ricerca medico

      1.1.1 Il sistema permette di ricercare i medici in base alla loro specializzazione, fornendo in output una lista di medici che la posseggono

   1.2 Registrazione alla piattaforma

      1.2.1 Il sistema consente la registrazione tramite l'inserimento dei propri dati personali al paziente non registrato

      1.2.2 Il nuovo utente deve essere inserito nel sistema sia come persona che come paziente

2. Paziente

   2.1 Accesso cronologia prenotazioni

      2.1.1 Il sistema permette all'utente di visualizzare la cronologia delle prenotazioni da lui effettuate che sono già state accettate

      2.1.2 Il paziente vedrà sia le prenotazioni che sono già state concluse sia quelle che ancora devono avvenire

   2.2 Effettuare prenotazioni

      2.2.1 Ogni prenotazione deve essere effettuata da un paziente registrato che fornisce al sistema l'ID del medico con cui vuole effettuare la prenotazione, la specializzazione e data desiderate

      2.2.2 Il sistema in automatico aggiunge alla richiesta di prenotazione l'ID del paziente che l'ha effettuata e quello dell'amministrativo che se ne occuperà

   2.3 Lascia feedback

      2.3.1 Il paziente una volta conclusa una visita può lasciare una valutazione da 0 a 5 sulla soddisfazione del cliente e sulla puntualità della visita; il sistema associa a questi due parametri l'ID del paziente e quello della prenotazione conclusa

2.4 Ricerca medico

2.4.1 Il sistema permette di ricercare i medici in base alla loro specializzazione, fornendo in output una lista di medici che la posseggono

3. Medico

3.1 Cronologia prestazioni

3.1.1 Il sistema permette ad ogni medico di visualizzare l'elenco sia delle prestazioni passato che di quelle ancora da svolgere

3.2 Statistiche

3.2.1 I medici possono visualizzare la media dei voti riguardanti la puntualità delle proprie visite e la soddisfazione dei pazienti, basate sui feedback di quest'ultimi

3.3 Termina prestazioni

3.3.1 Il sistema permette direttamente al medico di registrare il termine di ogni visita nel sistema non appena essa finisce

4. Amministrativo

4.1 Inserisci medico

4.1.1 Gli amministrativi possono inserire all'interno del sistema nuovi medici, fornendo i suoi dati personali

4.1.2 Anche in questo caso il nuovo utente deve essere inserito nel sistema sia come persona che come medico

4.2 Aggiunta indisponibilità medico

4.2.1 Il sistema permette ad un amministrativo di aggiungere periodi d'indisponibilità ai medici fornendo il loro ID, la data d'inizio e la data di fine di questa

4.2.2 Ciò fa si che il sistema non accetta visite per il medico specificato in un determinato giorno e ora, se questo cade in un periodo d'indisponibilità

4.3 Accetta prenotazione

4.3.1 Il sistema permette agli amministrativi di accettare le richieste di prenotazioni che gli sono state affidate

4.4 Rifiuta prenotazione

4.4.1 Il sistema permette agli amministrativi di rifiutare le richieste di prenotazioni che gli sono state affidate

4.4.2 In questo caso l'amministrativo aggiunte al rifiuto della richiesta di prenotazione anche una motivazione

4.5 Cronologia richieste prenotazioni

4.5.1 Un amministrativo grazie a questa funzione è in grado di controllare la lista di prenotazioni da lui accettate che sono presenti nel sistema

4.6 Aggiungi specializzazione medico

4.6.1 Il sistema permette agli amministrativi di aggiungere una specializzazione ad ogni medico specificando l'ID di questo e una specializzazione, che ancora non possiede

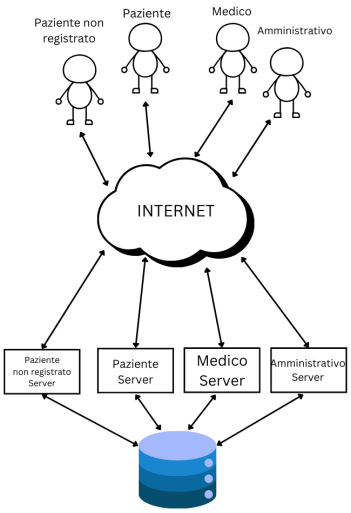
2.2.2 Requisiti non funzionali

1. Il tempo medio di sessione di un client, cioè il tempo medio trascorso fra la connessione di un client ad un server e la sua disconnessione, deve essere inferiore a 10 secondi

2. Il tempo medio di risposta del server ad una richiesta da parte di un client deve essere inferiore a 8 secondi

2.2.3 Architettura del sistema

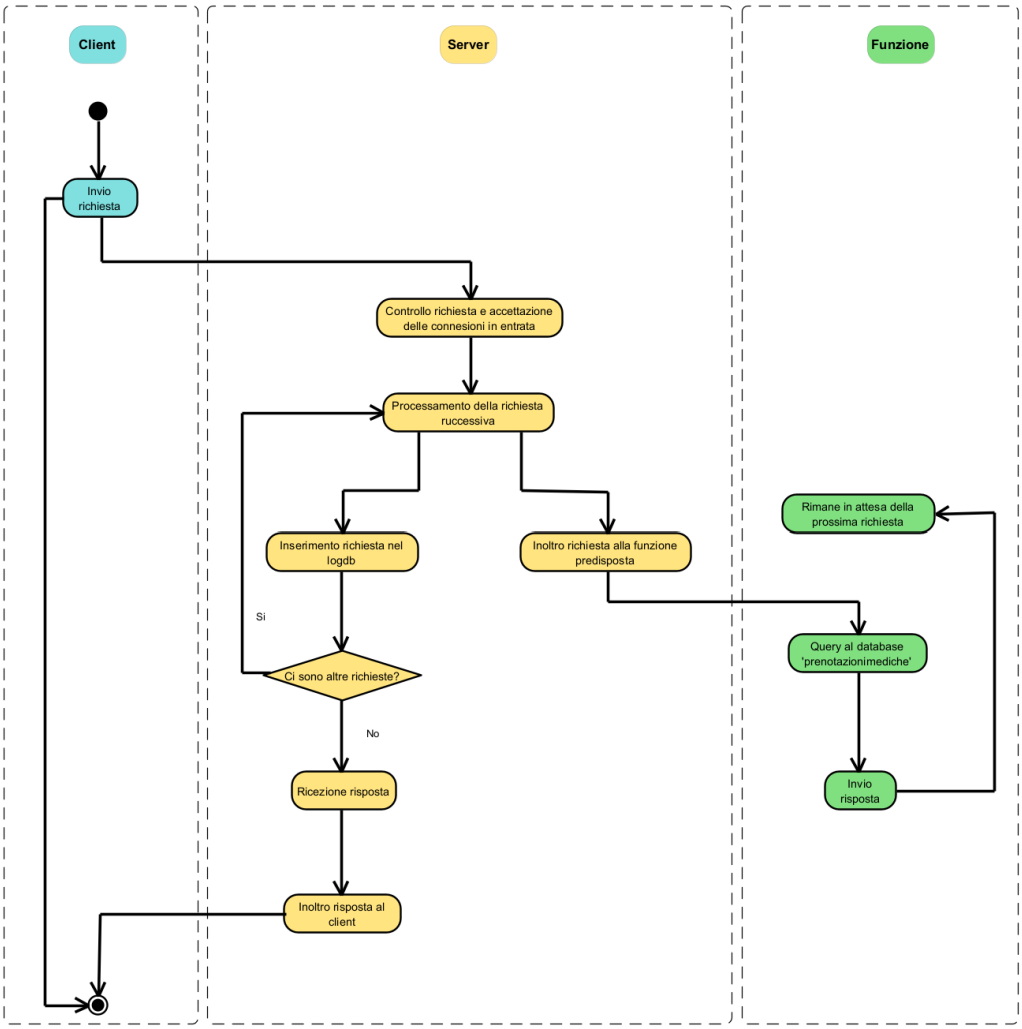
Il sistema è stato progettato seguendo un approccio basato sulla separazione delle richieste ricevute. L'intero sistema viene infatti decentralizzato su quattro server che gestiscono richieste specifiche. Un server gestisce i **Pazienti non registrati**, uno è accessibile dai **Pazienti registrati**, un altro gestisce i **Medici** ed infine l'ultimo è accessibile dagli **Amministrativi**.
Tutti i server sono eseguiti in parallelo in modo da restituire lo stato di una richiesta.



Ognuno dei quattro server viene a sua volta suddiviso in due componenti, un primo modulo denominato **Server**, addetto alla gestione delle richieste ricevute da ognuno degli utenti connessi al sistema e delle rispettive risposte, e un secondo modulo denominato **Handler**, addetto alla gestione dello smistamento delle richieste tra vari processi in esecuzione sul server stesso, inviando ogni richiesta tramite uno stream Redis al processo incaricato.
Ciascuno di tali processi viene denominato **Funzione**. Ogni Funzione è una macchina a stati finiti che rimane in attesa di ricevere una richiesta da parte dell'Handler associato, per poi processare tale richiesta interagendo con il database ed infine restituire l'esito all'Handler.

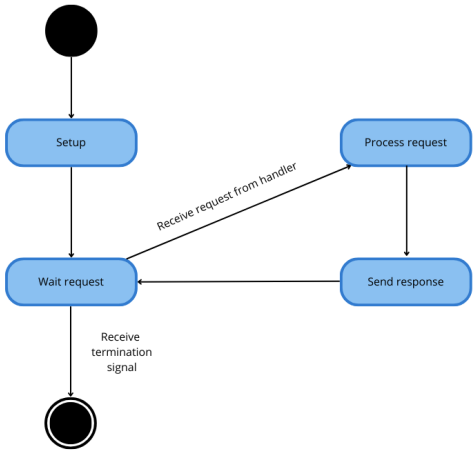
2.2.4 Activity diagram

L'activity diagram rappresentato di seguito illustra l'invio di una richiesta da un client, la sua elaborazione e l'invio della risposta da parte del server. Questo processo è comune a tutti i server e a tutte le loro funzioni, indipendentemente dal tipo di richiesta effettuata dal client.

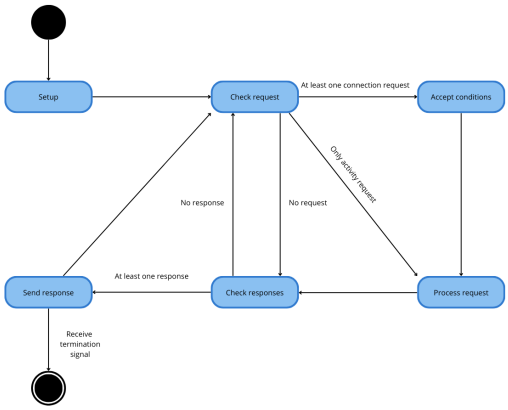


2.2.5 State diagram

Tutte le Funzioni in esecuzione sui server sono come macchine a stati finiti, che alternano lo stato di ricezione, di elaborazione e d'invio della risposta.

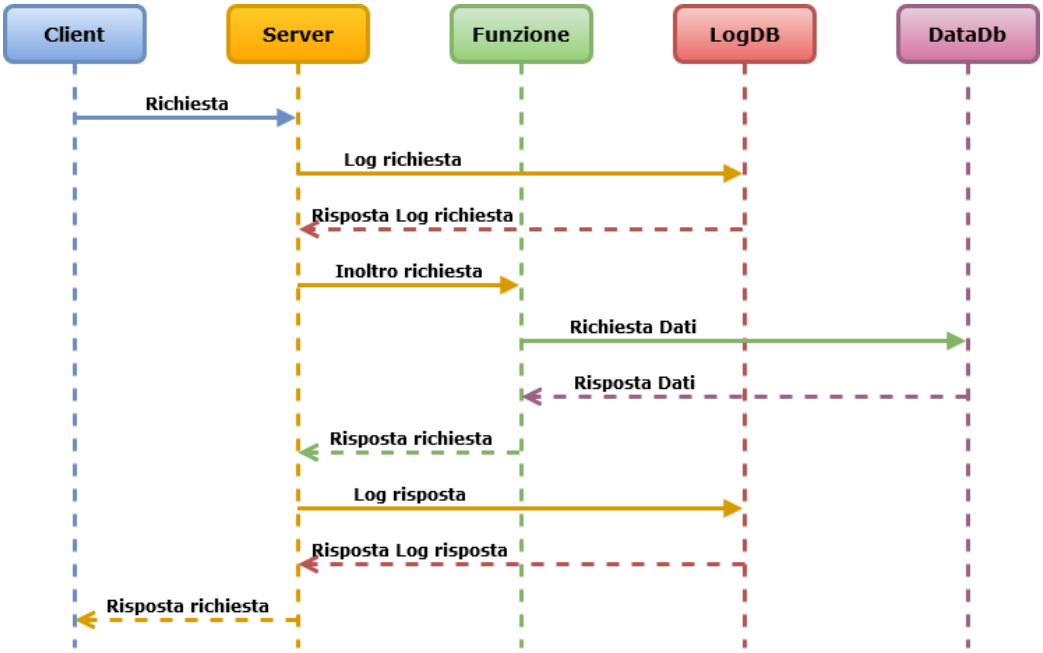


È importante notare come ogni Funzione rimane in esecuzione finché non viene ricevuto un segnale di terminazione da parte del server. Una volta ricevuto questo segnale ogni richiesta in esecuzione viene scartata senza comunicare il riscontro all'handler associato.
Anche ognuno dei 4 server può essere visto come una macchina a stati finiti. A differenza delle Funzioni, però, i server non rimangono in attesa perenne di una richiesta, bensì, nel caso in cui non ce ne siano, procedono alla fase di controllo per eventuali risposte ricevute da parte delle proprie Funzioni.



2.2.6 Message Sequence Chart

Il message sequence chart rappresentato di seguito illustra l'invio di una richiesta da un client, la sua elaborazione e l'invio della risposta da parte del server. Come per gli Activity diagram, questo processo è comune a tutti i server e a tutte le loro funzioni, indipendentemente dal tipo di richiesta effettuata dal client.



Capitolo 3

Implementazione del Software

3.1 Struttura del codice

Il codice del nostro progetto è strutturato in varie cartelle:

3.1.1 db_scripts

Nella cartella "db_scripts" troviamo gli script che permettono l'inizializzazione e la creazione del database. Grazie ad essi è possibile creare il database che colleziona le entità del sistema, "prenotazionimediche" e il database che permette il controllo e la gestione delle connessioni "logdb".

3.1.2 src

Qui abbiamo la maggior parte degli script *c++* che permettono la compilazione e l'esecuzione del progetto.

Nello specifico:

- nella cartella "**classes**" troviamo tutte le classi utili allo svolgimento delle funzioni, la cartella
- in "**utils**" troviamo la funzione che permette alle varie funzionalità di sistema di inviare lo stato della risposta ricevuta dal database al client, oltre alle costanti di sistema
- la cartella "**server**" definisce la struttura del server e dell'handler

- mentre nella cartella "**lib**" troviamo le librerie utilizzate per la connessione al database postgres e a redis.

La struttura di ogni classe definita all'interno del sistema, ha il seguente schema:

```
#include "classeesempio.h"

ClasseEsempio::ClasseEsempio(char *id, char *campo_n2){
    this->id = (char *)malloc(sizeof(char) * PARAMETERS_LEN);
    this->campo_n2 = (char *)malloc(sizeof(char) * PARAMETERS_LEN);

    strcpy(this->id, id);
    strcpy(this->campo_n2,campo_n2 );
}

ClasseEsempio::~ClasseEsempio(){
    free(this->id);
    free(this->campo_n2);
}

ClasseEsempio *ClasseEsempio::from_stream(redisReply *reply, int
    stream_num, int msg_num){
    char key[PARAMETERS_LEN];
    char value[PARAMETERS_LEN];

    char id[PARAMETERS_LEN];
    char campo_n2[PARAMETERS_LEN];

    char read_fields = 0b00;

    for (int field_num = 2; field_num < ReadStreamMsgNumVal(reply,
        stream_num, msg_num); field_num += 2){
        ReadStreamMsgVal(reply, stream_num, msg_num, field_num, key);
        ReadStreamMsgVal(reply, stream_num, msg_num, field_num + 1, value);

        if (!strcmp(key, "id")){
            strcpy(id, value);
            read_fields |= 0b01;
        }
        else if (!strcmp(key, "campo_n2")){
            strcpy(campo_n2, value);
            read_fields |= 0b10;
        }
        else
        {
        throw std::invalid_argument("Stream error: invalid fields");
        }
    }

    if (read_fields != 0b11){
        throw std::invalid_argument("Stream error: invalid fields");
    }

    return new Amministrativo(id_amministrativo, cf_amministrativo);}
```

Sempre nella directory scr abbiamo poi altre 4 cartelle che definiscono le varie entità che possono svolgere azioni all'interno del sistema, troviamo infatti "**paziente_non_registrato**", "**paziente**", "**medico**", "**amministrativo**". All'interno di ognuna di queste vi è una struttura ben definita che suddivide le funzioni e l'handler all'interno di due cartelle separate.

L'handler si occupa di istanziare un server dedicato e una volta che viene avviato direziona le richieste che gli giungono alle funzioni appartenenti alla propria entità. Le funzioni invece sono progettate per prendere in input gli stream dal flusso redis e utilizzarli o per fare query al database, o per creare oggetti, delle classi definite nella directory *classes*. Le funzioni sono di due tipi, ovvero: le INSERT che prendono input da stream redis e che eseguono query al db per inserire dati, e le SELECT che invece prendono parametri da stream redis per cercare dati all'interno del database.

Vediamo di seguito lo pseudocodice della Funzione "registrazione" utilizzata dai pazienti non registrati per appunto registrarsi al sistema di prenotazione:

```
INCLUDI "main.h"

funzione principale
    dichiara variabili di connessione Redis e risposte Redis
    dichiara variabile di risultato della query

    dichiara array per query, risposta, id del messaggio, prima chiave, id
        del client

    Inizializza connessione al database
    Inizializza connessione a Redis

    dichiara oggetti Persona e Paziente

    ciclo infinito
        leggi il comando Redis per il gruppo "main" e lo stream
            "paziente_non_registrato"

        verifica la risposta di Redis

        se non ci sono messaggi nello stream
            continua al prossimo ciclo

        leggi l ID del messaggio dallo stream

        leggi la prima chiave e l ID del client dallo stream

        se la prima chiave non e "client_id"
            invia la risposta con stato "BAD_REQUEST"
            continua al prossimo ciclo

        prova
            crea oggetti Persona e Paziente dallo stream
```

```
32          se si verifica un eccezione
33              invia la risposta con stato "BAD_REQUEST"
34              stampa l errore
35              continua al prossimo ciclo
36
37          formatta la query SQL per inserire i dati di Persona e Paziente nel
                database
38
39          esegui la query sul database
40
41          se la query non e andata a buon fine
42              invia la risposta con stato "DB_ERROR"
43              continua al prossimo ciclo
44
45          invia la risposta con stato "REQUEST_SUCCESS"
46
47      chiudi la connessione al database
48
49      ritorna 0
```

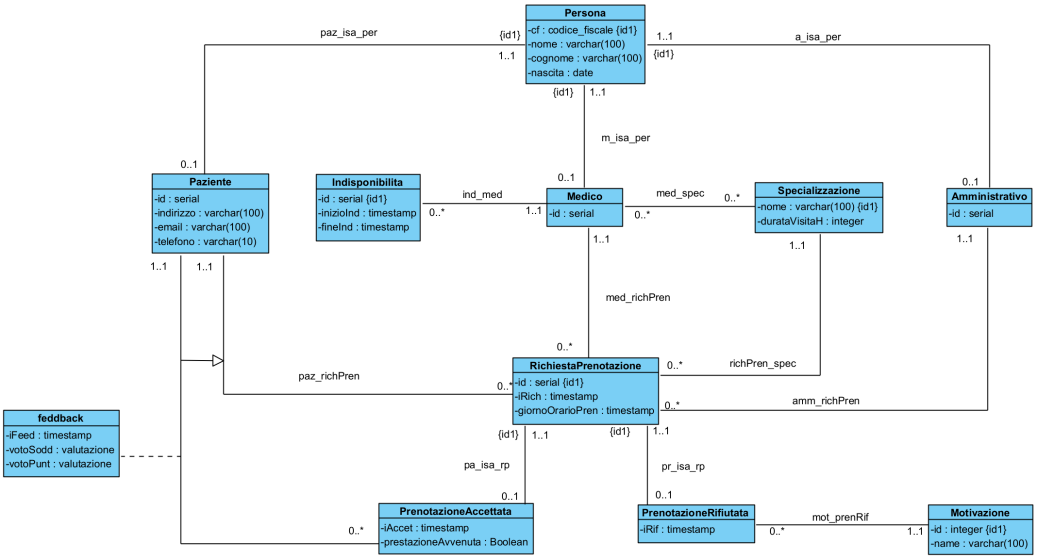
3.1.3 tests

Nella cartella tests troviamo il test generator del sistema, il quale funzioan nel segueente modo: innanzitutto configura l'indirizzo ip del server, e inizializza una lista per tenere traccia degli errori. Successivamente vengono generati valori casuali per un determinato parametro e viene generata una richiesta casuale, per ogni set di argomenti viene ritornata una stringa formattata di richiesta. Abbiamo poi la funzione **send_request** che invia una richiesta al server utilizzando una connessione al socket. Una volta avviato il test generator abbiamo l'inizializzazione dei generatori di parametri casuali, la creazione di variabili casuali, l'invio della richiesta al server, viene poi restituita la risposta.

3.2 Descrizione connessioni Redis

In questo progetto viene utilizzato redis per inviare richieste al server che leggerà da stream e registrerà i dati all'interno delle classi, inoltre vengono inviati anche stream redis in output come risposta da parte del server per indicare al client i risultati delle query.

3.3 Schema del database



3.4 Monitor funzionali

Nella realizzazione del database sono stati integrati vari monitor funzionali, realizzati mediante trigger. Questi operano come monitor attivi, prevenendo situazioni in cui i dati risultino inconsistenti tra loro, impedendo che vengano fatti INSERT, DELETE o UPDATE dal database che vadano a violare i vincoli esterni stabiliti durante il raffinamento dei requisiti, la creazione del diagramma UML e la sua ristrutturazione.

Di seguito l'implementazion di tre trigger, presi come esempio:

- Trigger per la disgiunzione fra le richieste di prenotazione accettate e quelle rifiutate

```
-- [V.richiestaprenotazione.disgiunzione]
CREATE OR REPLACE FUNCTION verifica_disgiunzione_richiesta()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_TABLE_NAME = 'prenotazioneaccettata' THEN
        IF EXISTS (
            SELECT 1 FROM prenotazionerifiutata WHERE richiesta_id =
                NEW.richiesta_id
        ) THEN
            RAISE EXCEPTION "Violazione del vincolo: La
                RichiestaPrenotazione e' gia' stata rifiutata";
        END IF;
    ELSIF TG_TABLE_NAME = 'prenotazionerifiutata' THEN
        IF EXISTS (
            SELECT 1 FROM prenotazioneaccettata WHERE richiesta_id =
                NEW.richiesta_id
        ) THEN
            RAISE EXCEPTION "Violazione del vincolo: La
                RichiestaPrenotazione e' gia' stata accettata";
        END IF;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_verifica_disgiunzione_accettata
BEFORE INSERT OR UPDATE ON prenotazioneaccettata
FOR EACH ROW
EXECUTE FUNCTION verifica_disgiunzione_richiesta();

CREATE TRIGGER trg_verifica_disgiunzione_rifiutata
BEFORE INSERT OR UPDATE ON prenotazionerifiutata
FOR EACH ROW
EXECUTE FUNCTION verifica_disgiunzione_richiesta();
```

- Trigger per la consistenza tra le richieste di prenotazione presenti nel sistema e quelle accettate o rifiutate dagli amministrativi

```
-- [V.prenotazioneaccettata/rifiutata.legale]
CREATE OR REPLACE FUNCTION controllo_richiesta_valida()
RETURNS TRIGGER AS $$
DECLARE
    richiesta RECORD;
BEGIN
    -- Recupera i dati della richiesta corrispondente
    SELECT irich, giornoorariopren
    INTO richiesta
    FROM richiestaprenotazione
    WHERE id = NEW.richiesta_id;

    -- Verifica che la richiesta esista
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Violazione del vincolo: La prenotazione deve
            riferirsi a una richiesta valida';
    END IF;

    -- Logica per prenotazioneaccettata
    IF TG_TABLE_NAME = 'prenotazioneaccettata' THEN
        IF NOT (NEW.iaccet > richiesta.irich AND NEW.iaccet <
            richiesta.giornoorariopren) THEN
            RAISE EXCEPTION 'Violazione del vincolo: iaccett deve
                essere maggiore di irich e minore di
                giornoorariopren';
        END IF;

    -- Logica per prenotazionerifiutata
    ELSIF TG_TABLE_NAME = 'prenotazionerifiutata' THEN
        IF NOT (NEW.irif > richiesta.irich AND NEW.irif <
            richiesta.giornoorariopren) THEN
            RAISE EXCEPTION 'Violazione del vincolo: irif deve essere
                maggiore di irich e minore di giornoorariopren';
        END IF;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER controllo_accettazione_valida
BEFORE INSERT OR UPDATE ON prenotazioneaccettata
FOR EACH ROW
EXECUTE FUNCTION controllo_richiesta_valida();

CREATE TRIGGER controllo_rifiuto_valido
BEFORE INSERT OR UPDATE ON prenotazionerifiutata
FOR EACH ROW
EXECUTE FUNCTION controllo_richiesta_valida();
```

- Trigger che controlla che un paziente lasci feedback solo su appuntamenti medici conclusi e da lui prenotati

```
CREATE OR REPLACE FUNCTION controllo_feedback_valido()
RETURNS TRIGGER AS $$
DECLARE
    prenotazione RECORD;
    paziente_richiesta INT;
BEGIN
    -- Recupera i dati della prenotazione accettata
    SELECT rp.giornoorariopren, pa.prestazioneavvenuta, rp.paziente_id
    INTO prenotazione
    FROM prenotazioneaccettata pa, richiestaprenotazione rp
    WHERE pa.richiesta_id = rp.id and pa.richiesta_id =
        NEW.prenotazione_accettata_id;

    -- Verifica che la prenotazione esista
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Violazione del vincolo: Il feedback deve
            riferirsi a una prenotazione accettata valida.';
    END IF;

    -- Verifica che il paziente che sta lasciando il feedback sia lo
        stesso che ha fatto la prenotazione
    IF prenotazione.paziente_id != NEW.paziente_id THEN
        RAISE EXCEPTION 'Violazione del vincolo: Il paziente non
            corrisponde a quello che ha effettuato la prenotazione.';
    END IF;

    -- Verifica che la prestazione sia avvenuta
    IF prenotazione.prestazioneavvenuta = false THEN
        RAISE EXCEPTION 'Violazione del vincolo: La prestazione deve
            essere avvenuta prima di lasciare un feedback.';
    END IF;

    -- Verifica che ifeed sia maggiore di giornoorariopren
    IF NOT (NEW.ifeed > prenotazione.giornoorariopren) THEN
        RAISE EXCEPTION 'Violazione del vincolo: Il timestamp del
            feedback deve essere maggiore di giornoorariopren.';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER controllo_feedback_valido
BEFORE INSERT OR UPDATE ON feedback
FOR EACH ROW
EXECUTE FUNCTION controllo_feedback_valido();
```

Nota: tutti gli altri trigger presenti nel databese possono essere visualizzati all'interno del file `db_scripts/triggers.sql`.

3.5 Monitor non-funzionali

```
while(true) {
    query = "
        SELECT EXTRACT(
            EPOCH FROM AVG(idisconnection - iconnection)
        ) * 1000 as avg
        FROM Client
        WHERE idisconnection IS NOT NULL
    ";

    average = log_db.execQuery(query);

    if average <= MAX_CONNECTION_TIME_AVG:
        responseStatus = "SUCCESS";
    else:
        responseStatus = "ERROR";

    query = "
        INSERT INTO
        SessionStatistic(type, iend, value, response_status)
        VALUES ('Session', CURRENT_TIMESTAMP, average, responseStatus)
    ";

    log_db.execQuery(query);

    query = "
        SELECT EXTRACT(
            EPOCH FROM AVG(iresponse - irequest)
        ) * 1000 as avg
        FROM Communication
        WHERE iresponse IS NOT NULL
    ";

    average = log_db.checkQueryResult(query);


    if average <= MAX_RESPONSE_TIME_AVG:
        responseStatus = "SUCCESS";
    else:
        responseStatus = "ERROR";

    query = "
        INSERT INTO
        SessionStatistic(type, iend, value, response_status)
        VALUES ('Response', CURRENT_TIMESTAMP, average, responseStatus)
    ";

    log_db.execQuery(query);

    sleep(60sec);
}
```

3.6 Risultati Sperimentali

Il progetto, nel suo complesso, rappresenta una piattaforma di prenotazioni mediche conforme agli standard moderni. I risultati ottenuti attraverso il test generator dimostrano che la piattaforma è in grado di adempiere efficacemente alle sue finalità, gestendo in modo semplice ed efficiente le prenotazioni ospedaliere sia per i pazienti che per il personale amministrativo. Inoltre, i test hanno evidenziato la capacità della piattaforma di affrontare eventuali sovraccarichi di richieste, grazie alla gestione parallela delle operazioni e alla capacità di gestire eventuali errori che possono verificarsi durante le richieste.