Progetto Ingegneria del Software

Backend Gestione Biblioteca

Paolo Marchetti (1986485)
Davide Vittucci (1903954)
Lorenzo Zanda (2006432)

December 5, 2024

Contents

1 Descrizione generale

Si vuole rappresentare un **Sistema di Gestione Bibliotecario** progettato per ottimizzare l'interazione tra utenti, bibliotecari e fornitori.
Il sistema è ideato per consentire agli utenti di registrarsi, cercare e prendere in prestito libri, visualizzare lo stato dei prestiti attivi e passati, e gestire sanzioni derivanti da ritardi o danni sui libri presi in prestito. I bibliotecari, invece, potranno gestire il catalogo dei libri, supervisionare i prestiti e applicare o aggiornare le sanzioni, mentre i fornitori avranno la possibilità di rifornire il catalogo.

Gli **utenti** della piattaforma possono **registrarsi** inserendo i propri dati, quali nome, cognome, e-mail e username. Una volta registrati, potranno richiedere un **prestito** di libri disponibili e concluderlo con la **restituzione in sede** entro una data specifica. Il sistema, inoltre, offre funzionalità di **ricerca** per aiutare gli utenti a trovare i libri desiderati in base al **titolo**.
Un libro si divide nell'**edizione** e nell'effettivo **libro fisico**. Un'**edizione** è composta dal codice *ISBN*, dal titolo, dal numero di pagine, dal nome della casa editrice, autore e genere principale. Il **libro fisico** ha la propria edizione corrispondente e un codice identificativo. L'utente ha anche accesso a una sezione per visualizzare lo **storico dei prestiti** e lo **stato di quelli attivi**, oltre a una sezione dedicata alle sanzioni. La **gestione del profilo** consente agli utenti di aggiornare il proprio username.

Il sistema consente ai **bibliotecari** di essere registrati alla piattaforma tramite il loro nome, cognome, email e data di assunzione. Il bibliotecario svolge un **ruolo amministrativo** nella piattaforma, con accesso a strumenti per **gestire** il **catalogo** dei libri. Ciò include la possibilità di aggiungere nuove edizioni o rimuovere libri dal sistema, garantendo che l'inventario sia sempre aggiornato.
La gestione dei prestiti consente ai bibliotecari di **monitorare** l'intero **ciclo del prestito**, dall'inizio alla conclusione, accettando o rifiutando nuove richieste di prestito e controllando la restituzione dei libri.
In caso di ritardi o danni sui libri, il bibliotecario può applicare **sanzioni** agli utenti, specificando motivazioni e importi e **registrando le sanzioni** nei profili degli utenti.
La piattaforma include, inoltre, una funzionalità per la **richiesta di nuovi libri** ai **fornitori**.

Ai **fornitori** è consentito di **registrarsi** alla piattaforma, inserendo il loro nome e la loro email aziendale. Questi possono **rifornire il catalogo** di nuove copie di edizioni disponibili nella biblioteca a seconda delle richieste di un bibliotecario. Le operazioni di **restock** vengono registrate con dati quali titolo, edizione, quantità e data di approvvigionamento.

2 Analisi Software

2.1 Requisiti utente

2.1.1 Lista dei requisiti

1. **Utente**
   - 1.1 **Registrazione alla piattaforma**
     - 1.1.1 Username
     - 1.1.2 Nome
     - 1.1.3 Cognome
     - 1.1.4 E-mail
   - 1.2 **Effettuare richieste di prestito**
     - 1.2.1 Libro da prendere in prestito
   - 1.3 **Concludere prestiti**
     - 1.3.1 Prestito da concludere
     - 1.3.2 Libro preso in prestito
   - 1.4 **Ricerca libri**
     - 1.4.1 Titolo libro
   - 1.5 **Visualizzare prestiti attivi e cronologia**
   - 1.6 **Visualizzare sanzioni nel profilo utente**
   - 1.7 **Pagamento sanzione**
     - 1.7.1 La sanzione da pagare
   - 1.8 **Gestione profilo utente**
     - 1.8.1 Dati di modificare
2. **Bibliotecario**
   - 2.1 **Registrazione alla piattaforma**
     - 2.1.1 Nome
     - 2.1.2 Cognome
     - 2.1.3 E-mail
   - 2.2 **Gestire il catalogo dei libri**
     - 2.2.1 Nuove edizioni da aggiungere al catalogo
     - 2.2.2 Libri da rimuovere dal catalogo
   - 2.3 **Visualizzare richieste di prestito**
   - 2.4 **Gestione dei prestiti**
     - 2.4.1 Nuovo prestito
     - 2.4.2 Prestito concluso

2.1.2 Diagramma UML

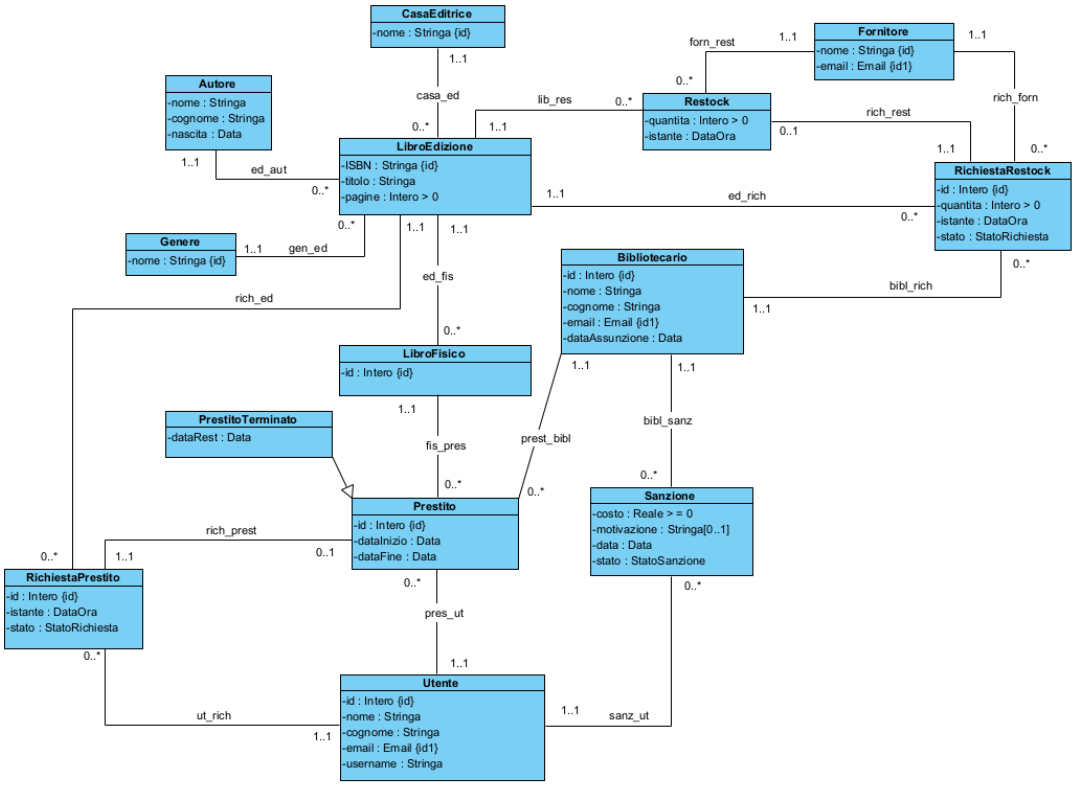

Figure 1: Diagramma UML del Sistema

2.1.3 Diagramma Use Case

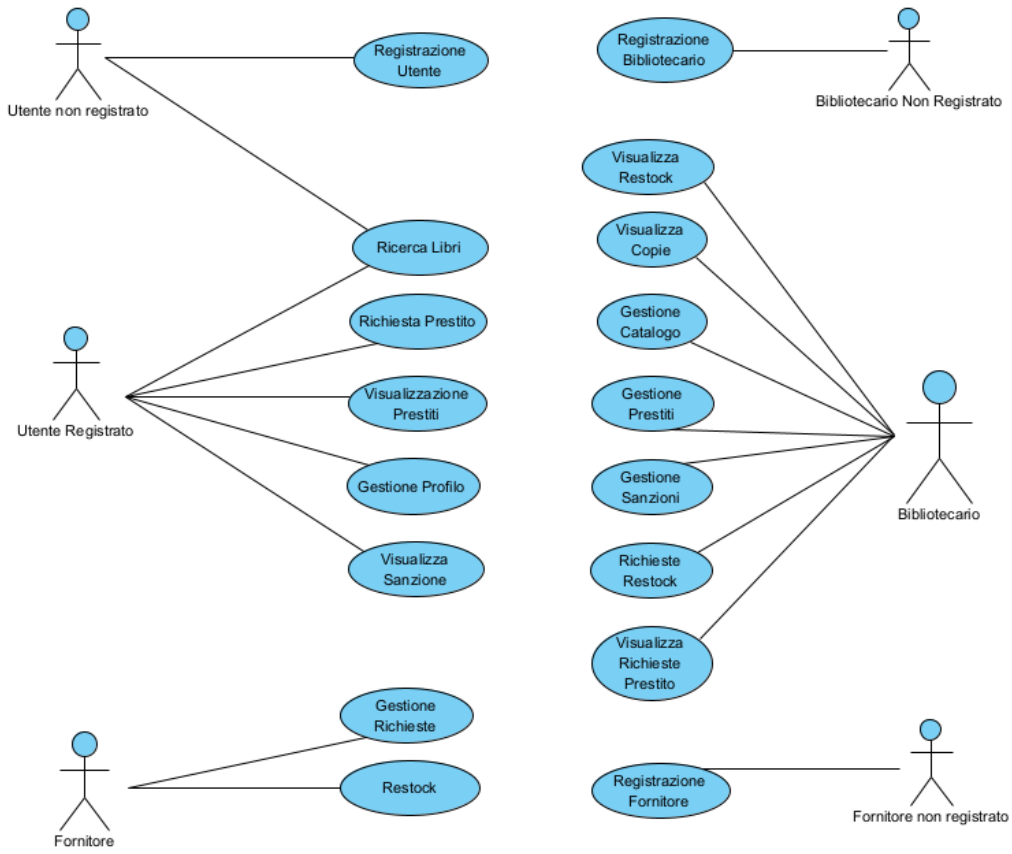

Figure 2: Diagramma Use Case del Sistema

2.2 Requisiti sistema

2.2.1 Requisiti funzionali e non funzionali

1. **Utente**

   1.1 **Effettuare richieste di prestito**

   1.1.1 Gli utenti possono richiedere il prestito di libri disponibili all'interno del catalogo.

   1.1.2 Ogni prestito viene associato all'utente richiedente tramite un ID univoco.

   1.1.3 La durata predefinita del prestito è di 30 giorni.

   1.2 **Ricerca libri**

   1.2.1 Gli utenti devono poter cercare un libro tramite il titolo.

   1.3 **Concludere prestito**

   1.3.1 Gli utenti devono restituire i libri entro la data di scadenza per evitare sanzioni.

   1.3.2 La restituzione è effettuata presso la biblioteca.

   1.3.3 Eventuali danni riscontrati nel libro o ritardi nella restituzione verranno sanzionati dal bibliotecario con una sanzione.

   1.4 **Visualizza sanzioni**

   1.4.1 L'utente può visualizzare nel proprio profilo i dati della Sanzione (codice, motivazione, importo).

   1.5 **Pagamento sanzione**

   1.5.1 La sanzione è pagata presso la biblioteca.

   1.6 **Visualizzare prestiti attivi e cronologia**

   1.6.1 Gli utenti possono visualizzare i libri presi in prestito e la loro cronologia di lettura.

   1.6.2 La visualizzazione include la data di scadenza dei prestiti attivi e lo storico dei libri già restituiti.

   1.7 **Gestire profilo utente**

   1.7.1 Gli utenti possono aggiornare l'username.

2. **Bibliotecario**

   2.1 **Gestire il catalogo dei libri**

   2.1.1 I bibliotecari possono gestire il catalogo dei libri disponibili

   2.1.2 Possono aggiungere nuove edizioni al catalogo, possono rimuovere libri fisici dal catalogo.

   2.2 **Gestione sanzioni**

   2.2.1 I bibliotecari devono poter gestire e aggiornare le sanzioni applicate agli utenti per ritardi o danni ai libri.

2.2.2 I bibliotecari possono applicare nuove sanzioni, annullare sanzioni erronee.

2.2.3 Il sistema registra la sanzione nel profilo utente.

2.3 **Visualizzare richieste di prestito**

2.4 **Gestione completa dei prestiti**

2.4.1 I bibliotecari devono poter supervisionare l'intero processo di prestito dall'inizio alla chiusura dei prestiti.

2.4.2 I bibliotecari possono registrare nuovi prestiti e registrare la restituzione dei libri.

2.4.3 I prestiti attivi e la cronologia dei prestiti vengono tracciati nel profilo di ogni utente.

2.5 **Richiesta di libri al fornitore**

2.5.1 I bibliotecari devono poter inviare richieste di riassortimento di libri ai fornitori.

2.5.2 Ogni richiesta deve includere l'edizione del libro e la quantità richiesta.

2.5.3 Le richieste di acquisto devono essere tracciate nel sistema con una notifica che conferma l'invio e lo stato (in attesa, completato, rifiutato).

2.6 **Visualizzare restock effettuati dai fornitori**

2.7 **Visualizzare le copie disponibili per le varie edizioni**

2.7.1 I bibliotecari possono visualizzare la quantità di copie disponibili (non attualmente occupate in prestiti) per le edizioni presenti nel catalogo.

3. **Fornitore**

3.1 **Restock**

3.1.1 I fornitori possono aggiungere scorte per i libri e registrare nuovi restock.

3.1.2 Ogni nuova scorta di un titolo è registrata come restock, con dati quali edizione, quantità e data di approvvigionamento.

3.2 **Gestione richieste**

3.2.1 I fornitori possono vedere la lista delle richieste a loro carico.

4. **Requisiti non funzionali**

4.1 Il tempo medio di sessione di un client deve essere inferiore o uguale al tempo massimo di sessione.

4.2 Il tempo medio di risposta del server ad una richiesta da parte di un client deve essere inferiore o uguale al tempo massimo di risposta.

2.2.2 Architettura del sistema

Il **sistema** è progettato adottando un approccio che prevede la **separazione delle richieste** in base alla loro natura. In particolare, l'architettura del sistema è **decentralizzata** e si basa su **tre server distinti**: un **server** dedicato agli **utenti**, uno per i **bibliotecari** e uno per i **fornitori**. Ogni server opera in **parallelo** agli altri e si occupa esclusivamente della gestione delle richieste specifiche per la categoria di clienti a cui è destinato.

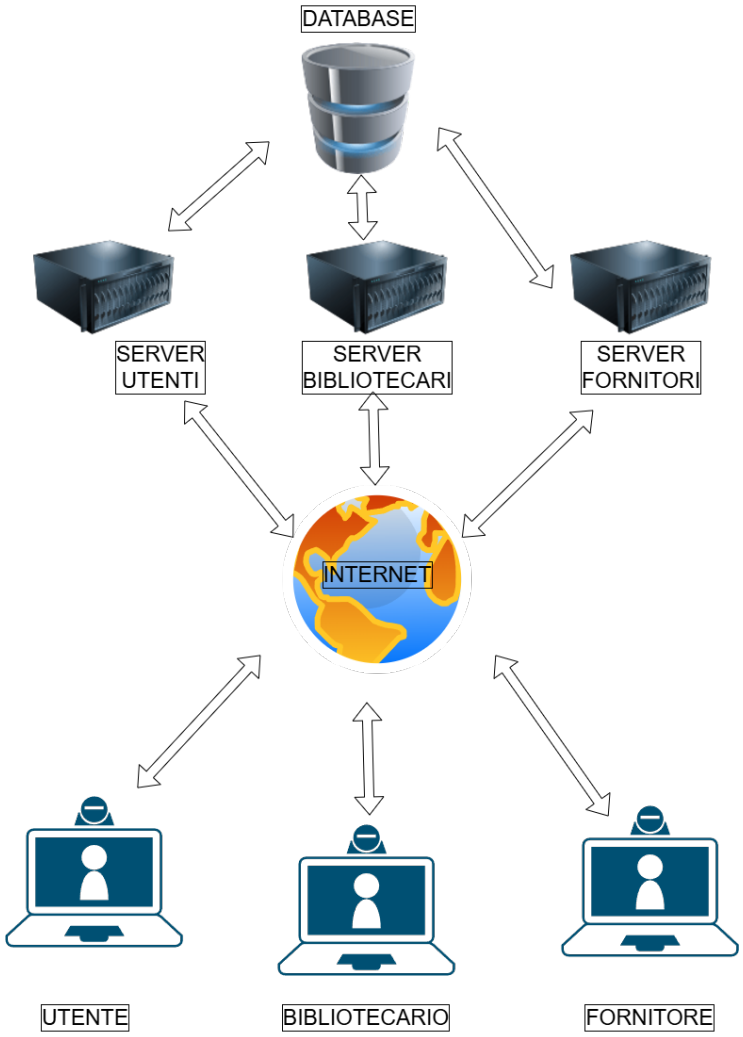

Figure 3: Architettura fisica del sistema

Ogni server è ulteriormente suddiviso in **due moduli principali**.

1. Il **primo**, denominato ***server***, si occupa della **gestione delle comunicazioni** con i clienti collegati, ricevendo e inviando richieste.

2. Il **secondo**, chiamato ***handler***, ha il compito di **smistare le richieste** verso vari processi in esecuzione sul server, utilizzando uno stream Redis per inviarle al processo responsabile.

Questi **processi** specifici, definiti ***functions***, sono implementati come **macchine a stati finiti**.
Le **functions** attendono richieste dall'**handler** associato, le elaborano interfacciandosi con il **database** per ottenere o aggiornare i dati necessari, e infine restituiscono il **risultato** all'**handler**, che lo invia al mittente. Questa struttura consente una gestione **organizzata** ed **efficiente** delle richieste, assicurando che ciascun componente del sistema sia specializzato nelle **operazioni che gli competono**.

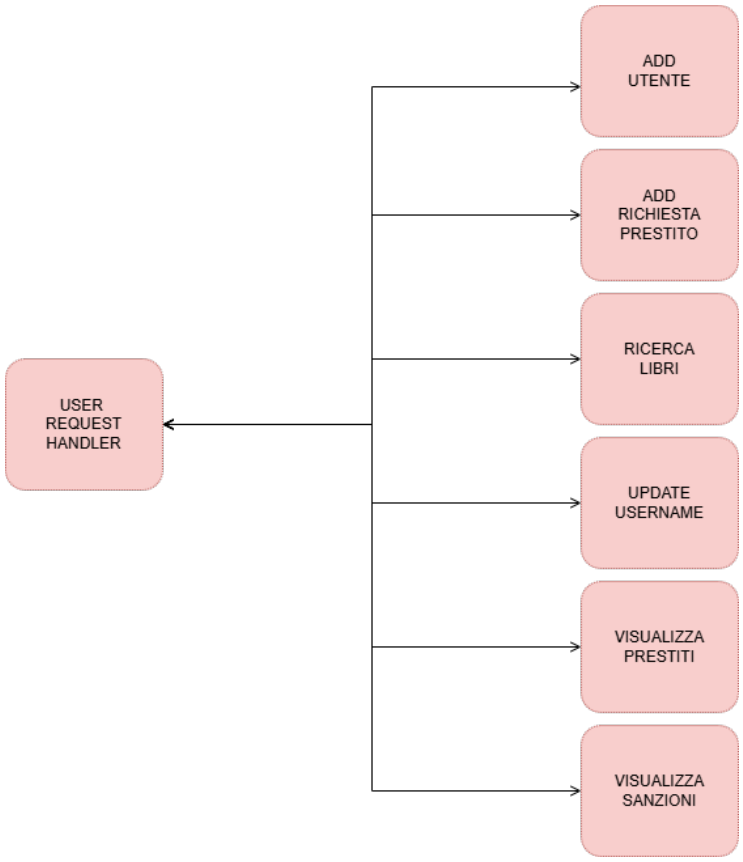

Figure 4: L'utente e le sue funzioni

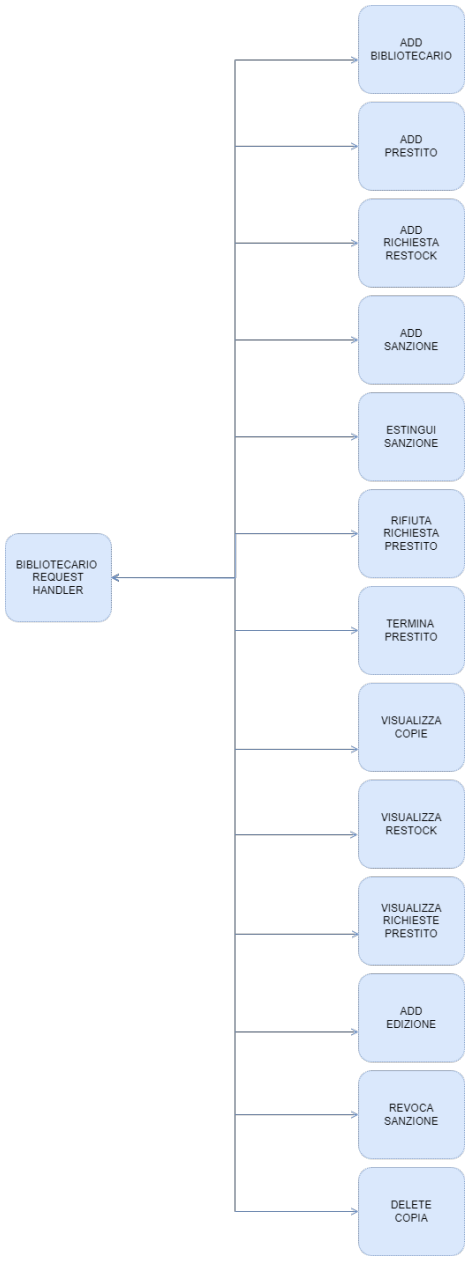

Figure 5: Il bibliotecario e le sue funzioni

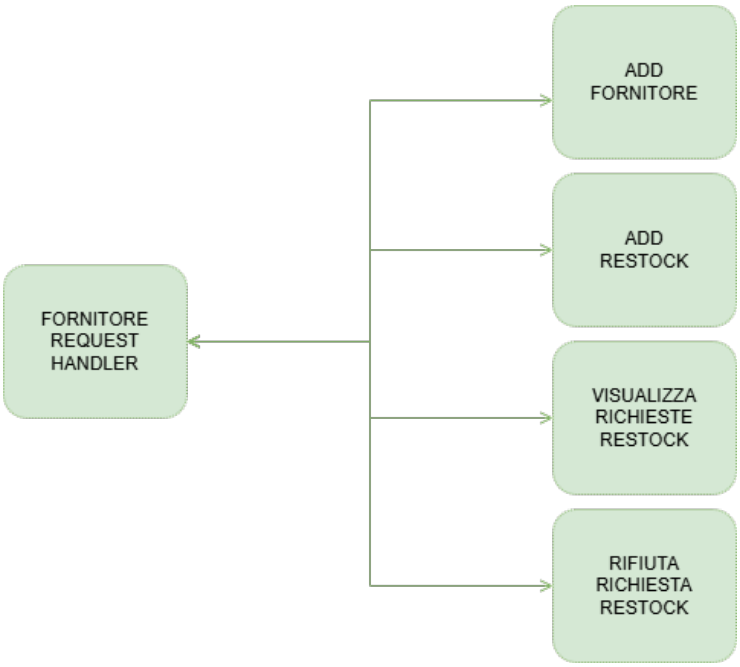

Figure 6: Il fornitore e le sue funzioni

2.2.3 Activity Diagram

Di seguito è illustrato l'**activity diagram** che rappresenta il **flusso** relativo all'**invio di una richiesta** da parte di un client, alla sua **elaborazione**, e alla **restituzione della risposta** da parte del server. Questo processo è uniforme per tutti i server e per le loro **functions**, indipendentemente dalla natura della richiesta effettuata dal client.
Si presuppone che il client abbia già stabilito una connessione con il server e che la **function** designata sia operativa e pronta a ricevere ed elaborare la richiesta.

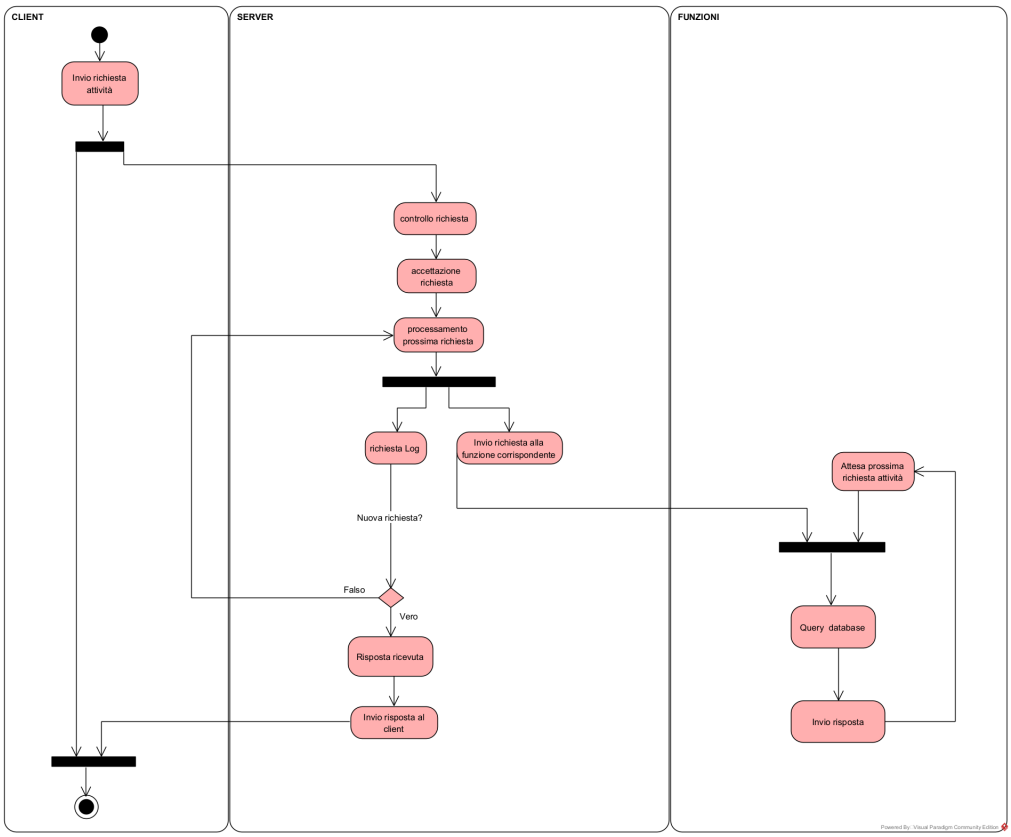

Figure 7: Activity Diagram

2.2.4 State Diagram

Ogni **function** in esecuzione su uno dei tre server si comporta come una **macchina a stati finiti**. Questa macchina alterna tra **tre stati principali**: **ricezione** della richiesta, **elaborazione** della richiesta, e **invio** della risposta. Questo comportamento è uniforme per tutte le **functions**, senza differenze operative tra loro.

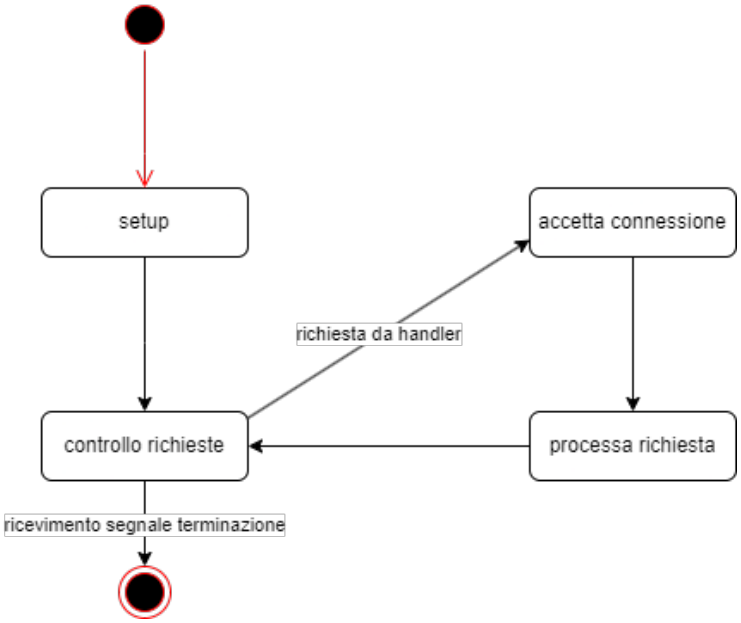

Figure 8: Functions state diagram

Analogamente, ciascuno dei tre server può essere rappresentato nella stessa maniera. A differenza delle **functions** però, in **assenza di richieste** in ingresso, un server passa direttamente alla **fase di controllo** per verificare l'eventuale presenza di risposte da parte delle **functions** a lui collegate.

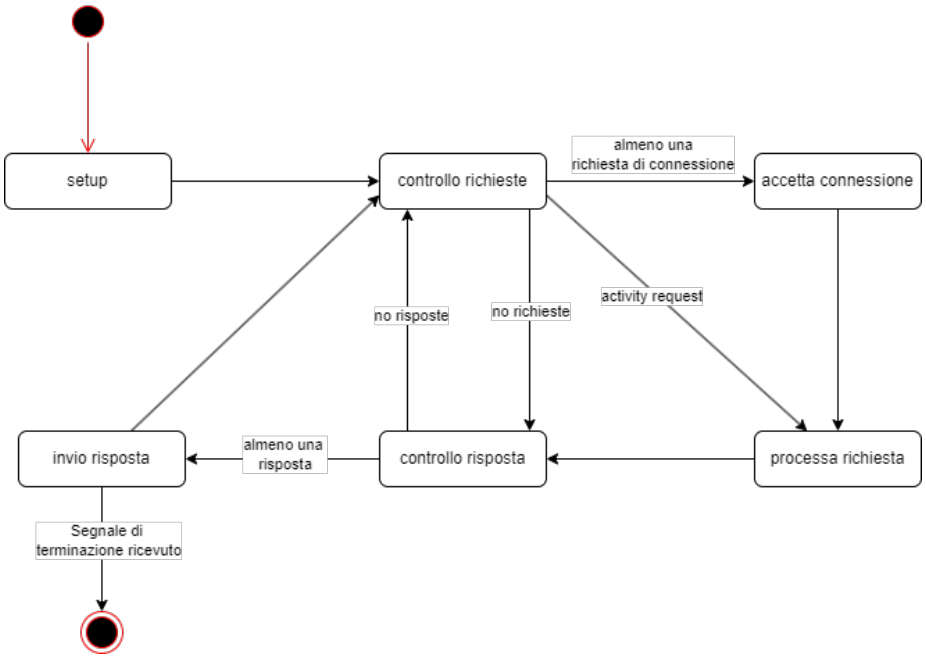

Figure 9: Server state diagram

2.2.5 Message Sequence Chart

Di seguito è presentato il message sequence chart che descrive il flusso di comunicazione per l'invio di una richiesta da parte di un client, la sua elaborazione e la successiva restituzione della risposta da parte del server. Questo schema rappresenta un processo standard condiviso da tutti i server e dalle rispettive **functions**, indipendentemente dalla tipologia di richiesta inviata dal client.

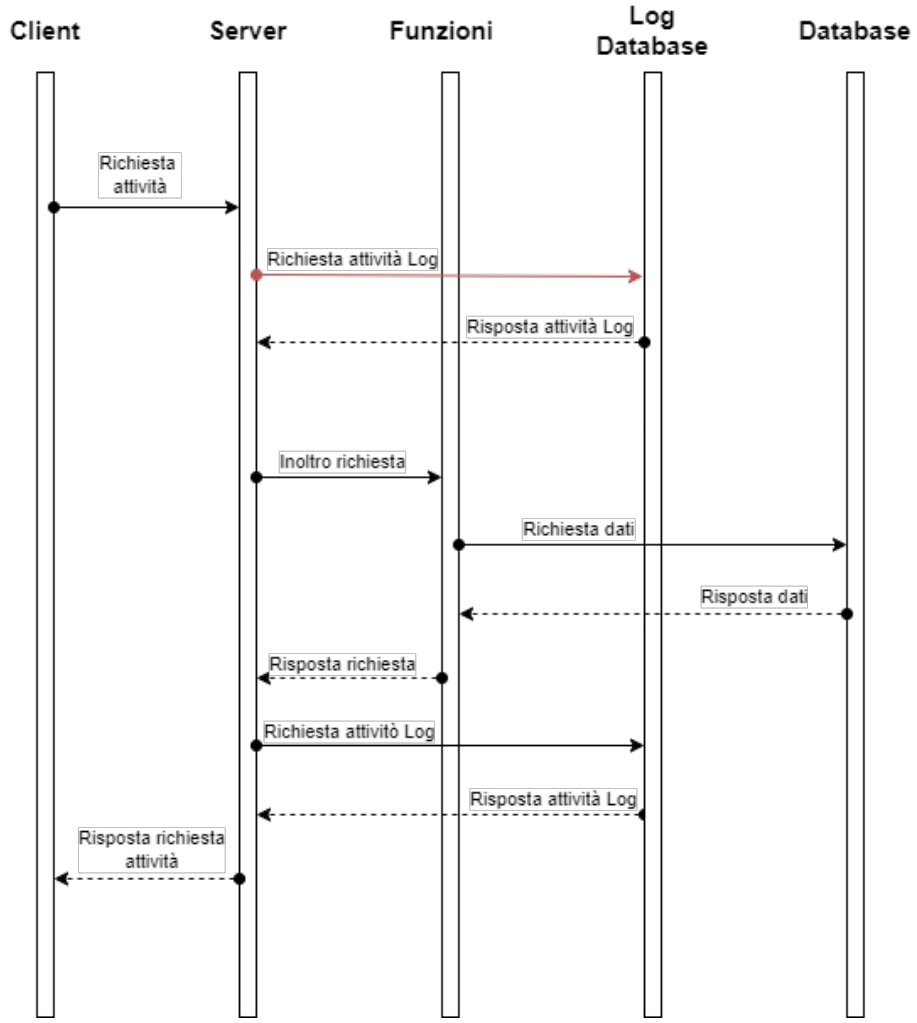

Figure 10: Server State diagram

3 Implementazione del software

Il codice è stato organizzato in **tre sezioni principali**: una per l'**utente**, una per il **bibliotecario** e una per il **fornitore**. Ogni sezione include il codice relativo al proprio handler e alle sue funzioni specifiche. Ciascuno dei tre server è stato realizzato utilizzando una singola classe **Server**. Questa classe si occupa di interfacciarsi con i client esterni e con il rispettivo gestore.

Ogni **handler** contiene una **lista di richieste** che è in grado di gestire, ognuna delle quali è associata a una **funzione** specifica del server. Nel caso in cui un server riceva una **richiesta non riconosciuta** dal proprio handler, viene restituito un messaggio di "Richiesta non valida". Inoltre, tutte le richieste e le risposte che attraversano un gestore vengono registrate nel **database dei log**.

Le **funzioni**, invece, sono implementate singolarmente in **file separati**. Questa scelta deriva dalla natura specifica di ciascuna funzione: pur seguendo il modello di una macchina a stati, le operazioni eseguite nei vari stati differiscono leggermente tra una funzione e l'altra, rendendo **impraticabile** una soluzione completamente generica.

Di seguito vengono presentati gli pseudo-codici relativi ai processi di **server**, **funzioni** e del **handler**.

Pseudo-codice del Server

Algorithm 1: Pseudo-codice del Server

```
while sigterm not received do
    requests ← checkRequests();
    acceptIncomingConnections();
    if requests exist then
        foreach request in requests do
            logRequest(request);
            isValidRequest ← sendRequestToHandler(request);
            if not isValidRequest then
                sendToClient(request.clientID, "BAD_REQUEST");
            end
        end
    end
    responses ← checkResponses();
    if responses exist then
        foreach response in responses do
            sendToClient(response.clientID, response.data);
        end
    end
end
```

Pseudo-codice dell'Handler

Algorithm 2: Pseudo-codice dell'Handler

```
request ← receiveRequest();
request_type ← getRequestType();
if request_type is not valid then
    return false;
end
else
    sendToFunctions(request);
    return true;
end
```

Pseudo-codice delle Functions

Algorithm 3: Pseudo-codice delle Functions

```
con2DB();
con2Redis();
while sigterm not received do
    request ← waitForIncomingRequest();
    if request is not valid then
        send("BAD_REQUEST");
    end
    else
        query ← convertRequest();
        result ← queryDB(query);
        if result is not valid then
            send("DB_ERROR");
        end
        else
            response ← formatResponse(result);
            send(response);
        end
    end
end
```

3.1 Schema Database

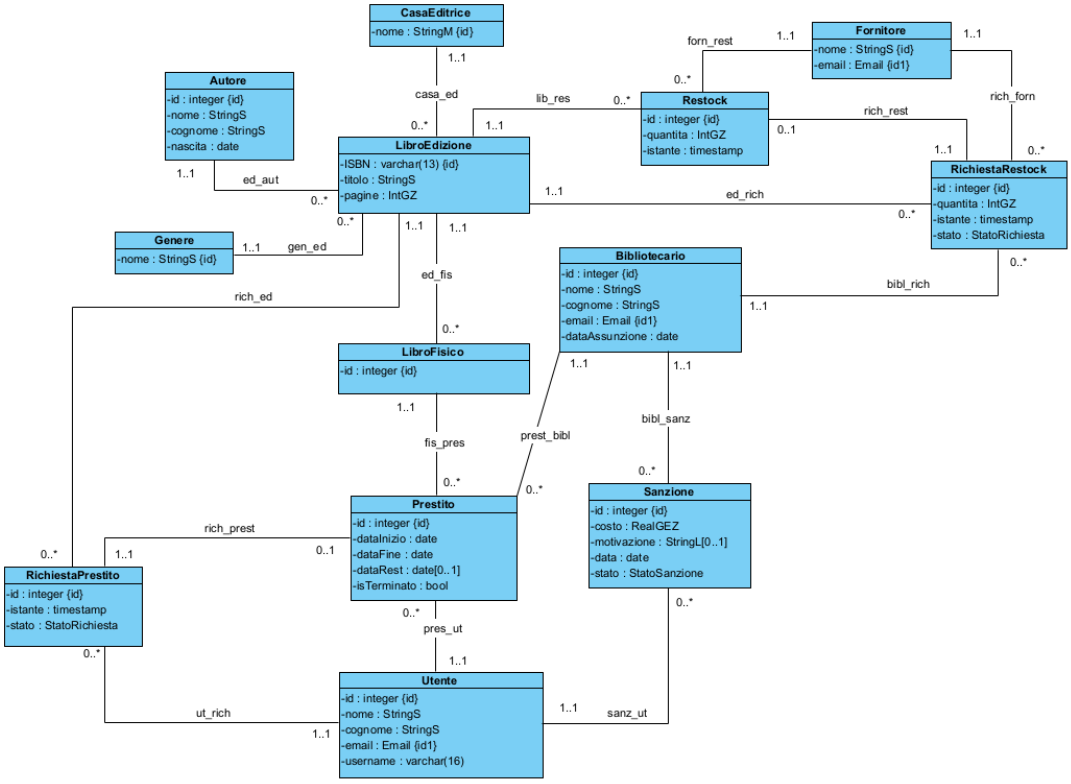

Figure 11: Diagramma UML del Sistema Ristrutturato

3.2 Redis

Ogni **handler** crea **due stream redis** per ciascuna **funzione** sotto il suo controllo. Ad esempio, per la funzione responsabile dell'aggiunta di nuove edizioni, `add-edizione` nel programma, il gestore di richieste del Bibliotecario istanzia uno stream per la comunicazione dal gestore verso la funzione e uno dalla funzione verso il gestore.
Per **monitorare** i clienti che interagiscono con il sistema, il server assegna un **ID univoco** a ogni client al momento della connessione. Questo ID viene utilizzato come prima chiave in tutti i messaggi scambiati tra i **gestori** e le **funzioni**, consentendo ai gestori di **identificare correttamente** il destinatario della risposta.

3.3 Monitor funzionali

All'interno del sistema sono stati inseriti dei monitor funzionali, implementati tramite dei **trigger**. Tali trigger fungono da monitor attivi e impediscono che i dati all'interno del database siano inconsistenti tra loro.
Di seguito vengono illustrati quattro dei trigger implementati ma ulteriori trigger possono essere trovati nei file:

`src/services/database/db-scripts/biblioteca/trigger.sql`
`src/services/database/db-scripts/logdb/trigger-log.sql`

```sql
-- Trigger sulla tabella Prestito: Impedisci la restituzione di un libro che non è stato preso in prestito dall'utente

CREATE OR REPLACE FUNCTION check_prestito_validity() RETURNS TRIGGER AS $$
BEGIN
    -- Verifica che l'utente stia restituendo un libro che ha effettivamente preso in prestito
    IF NOT EXISTS (
        SELECT 1
        FROM Prestito
        WHERE NEW.utente = Prestito.utente
        AND NEW.libro = Prestito.libro
        AND Prestito.isTerminato = FALSE
    ) THEN
        RAISE EXCEPTION 'L''utente non ha preso in prestito questo libro';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_check_prestito_validity
BEFORE UPDATE ON Prestito
FOR EACH ROW
WHEN (NEW.dataRest IS NOT NULL)
EXECUTE FUNCTION check_prestito_validity();



-- Trigger sulla tabella Prestito: Verifica che l'istante della richiesta sia minore della dataInizio del prestito

CREATE OR REPLACE FUNCTION verifica_istante_richiesta_prestito()
RETURNS TRIGGER AS $$
BEGIN
    IF NOT EXISTS (
        SELECT 1
        FROM RichiestaPrestito
        WHERE id = NEW.richiesta
          AND istante::DATE <= NEW.dataInizio
    ) THEN
        RAISE EXCEPTION 'La data di inizio del prestito deve essere successiva all''istante della richiesta.';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_verifica_istante_richiesta_prestito
BEFORE INSERT OR UPDATE ON Prestito
FOR EACH ROW
EXECUTE FUNCTION verifica_istante_richiesta_prestito();

```

```sql
-- Trigger sulla tabella Prestito: Controlla se esiste un altro prestito attivo per lo stesso libro nello stesso periodo

CREATE OR REPLACE FUNCTION verifica_prestito_unico()
RETURNS TRIGGER AS $$
BEGIN
    IF EXISTS (
        SELECT 1
        FROM Prestito
        WHERE libro = NEW.libro
          AND isTerminato = FALSE
          AND (
              (NEW.dataInizio, NEW.dataFine) OVERLAPS (dataInizio, dataFine)
          )
    ) THEN
        RAISE EXCEPTION 'Il libro è già in prestito in questo periodo.';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_verifica_prestito_unico
BEFORE INSERT ON Prestito
FOR EACH ROW
EXECUTE FUNCTION verifica_prestito_unico();



-- Trigger sulla tabella Restock: Verifica che i dati del restock corrispondano a quelli della richiesta

CREATE OR REPLACE FUNCTION verifica_coerenza_restock()
RETURNS TRIGGER AS $$
BEGIN
    IF NOT EXISTS (
        SELECT 1
        FROM RichiestaRestock
        WHERE id = NEW.richiesta
          AND quantita = NEW.quantita
          AND fornitore = NEW.fornitore
          AND edizione = NEW.edizione
    ) THEN
        RAISE EXCEPTION 'I dati del restock non corrispondono alla richiesta associata.';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_verifica_coerenza_restock
BEFORE INSERT OR UPDATE ON Restock
FOR EACH ROW
EXECUTE FUNCTION verifica_coerenza_restock();

```

3.4 Monitor non funzionali

```
#include "main.h"

// Requisiti non funzionali
// 1. Il tempo medio di sessione di un client deve essere inferiore o uguale al tempo massimo di
sessione.
// 2. Il tempo medio di risposta del server ad una richiesta da parte di un client deve essere inferiore
o uguale al tempo massimo di risposta.

int main()
{

    Con2DB log_db = Con2DB(POSTGRESQL_SERVER, POSTGRESQL_PORT, POSTGRESQL_USER, POSTGRESQL_PASSWORD,
POSTGRESQL_DBNAME);

    PGresult *query_res;
    char query[QUERY_SIZE];
    char response_status[8];

    while (true)
    {

        // Tempo medio di sessione
        sprintf(query, "SELECT EXTRACT(EPOCH FROM AVG(disconnTime - connTime)) * 1000 as avg FROM Client
WHERE disconnTime IS NOT NULL");
        query_res = log_db.execQuery(query, true);

        if ((PQresultStatus(query_res) != PGRES_COMMAND_OK && PQresultStatus(query_res) !=
PGRES_TUPLES_OK) || PQntuples(query_res) <= 0)
        {
            printf("Errore database\n");
            continue;
        }

        char *avg = PQgetvalue(query_res, 0, PQfnumber(query_res, "avg"));

        if (strlen(avg) == 0)
            sprintf(avg, "0");

        sprintf(response_status, atof(avg) <= MAX_CONNECTION_TIME_AVG ? "SUCCESS" : "ERROR");

        sprintf(query, "INSERT INTO SessionStats(sessionType, endTime, value, responseStatus) VALUES
('SESSION', CURRENT_TIMESTAMP, %s, \'%s\')", avg, response_status);

        query_res = log_db.execQuery(query, false);

        if (PQresultStatus(query_res) != PGRES_COMMAND_OK && PQresultStatus(query_res) !=
PGRES_TUPLES_OK)
        {
            printf("Errore database\n");
            continue;
        }
```

```
        // Tempo medio di risposta
        sprintf(query, "SELECT EXTRACT(EPOCH FROM AVG(responseTime - requestTime)) * 1000 as avg FROM
Requests WHERE responseTime IS NOT NULL");
        query_res = log_db.execQuery(query, true);

        if ((PQresultStatus(query_res) != PGRES_COMMAND_OK && PQresultStatus(query_res) !=
PGRES_TUPLES_OK) || PQntuples(query_res) <= 0)
        {
            printf("Errore database\n");
            continue;
        }

        avg = PQgetvalue(query_res, 0, PQfnumber(query_res, "avg"));

        if (strlen(avg) == 0)
            sprintf(avg, "0");

        sprintf(response_status, atof(avg) <= MAX_RESPONSE_TIME_AVG ? "SUCCESS" : "ERROR");

        sprintf(query, "INSERT INTO SessionStats(sessionType, endTime, value, responseStatus) VALUES
('RESPONSE', CURRENT_TIMESTAMP, %s, \'%s\')", avg, response_status);

        query_res = log_db.execQuery(query, false);

        if (PQresultStatus(query_res) != PGRES_COMMAND_OK && PQresultStatus(query_res) !=
PGRES_TUPLES_OK)
        {
            printf("Errore database\n");
            continue;
        }

        micro_sleep(60000000);
    }

    log_db.endDBConnection();
}
```

4 Risultati sperimentali

Il sistema permette la gestione di un servizio bibliotecario secondo gli standard odierni e i risultati ottenuti tramite il generatore di test dimostrano che il sistema è stabile, ben funzionante e permette di gestire correttamente eventuali errori negli input.