Progetto LSO

Libreria

Ingenito Roberto - N86004077
Ingenito Simone - N86004063
Sequino Lorenzo - N86004367

Indice

Analisi dei requisiti

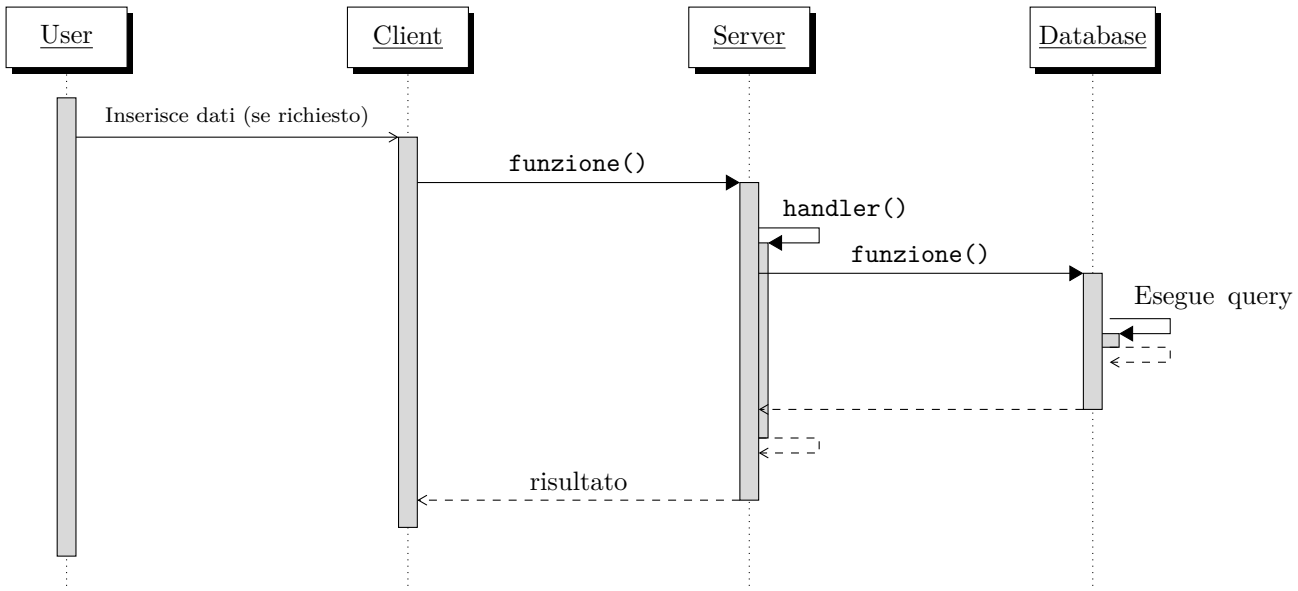
1.1 Obiettivo del progetto

Il progetto consiste nella simulazione di un sistema che modella una libreria per un numero non specificato di utenti. Gli utenti devono potersi registrare e accedere al server per interagire con la libreria, che contiene una lista di libri disponibili. Il sistema deve gestire il prestito dei libri, tenere traccia delle copie disponibili e delle copie prese in prestito, oltre a monitorare le scadenze di restituzione.

1.2 Funzionalità principali

- **Registrazione e Login degli Utenti**:
  Gli utenti devono potersi registrare e loggare al server per poter utilizzare il sistema.
- **Gestione dei Libri**:
  Il sistema mantiene una lista di libri disponibili nella libreria, con dettagli come il numero di copie disponibili e il numero di copie prese in prestito, oltre alla gestione di questa.
- **Ricerca dei Libri**:
  Gli utenti possono cercare i libri disponibili nella libreria.
  La ricerca può essere fatta per genere, nome o visualizzare la lista di quelli disponibili.
- **Prestito dei Libri**:
  Gli utenti possono prendere in prestito i libri se ci sono ancora copie disponibili.
  Gli utenti possono mettere nel carrello tutti i libri che intendono prendere in prestito e poi effettuare il check-out.
- **Notifiche**:
  Il libraio deve contattare i clienti che non hanno restituito i libri dopo la scadenza prevista tramite un messaggio.

Implementazione



Questo *sequence diagram* descrive la maggior parte delle funzionalità presenti nel progetto.
Viene fornito giusto un esempio per la registrazione nel paragrafo apposito.

2.1 Linguaggi utilizzati

Il progetto utilizza due linguaggi principali:

- **C**: Il linguaggio C è utilizzato per lo sviluppo dell'applicazione client-server. Il server e il client comunicano tramite socket, permettendo lo scambio di dati in rete.
- **SQL**: Il linguaggio SQL è utilizzato per la gestione del database PostgreSQL. Il server si connette al database per eseguire operazioni di lettura e scrittura, gestendo le informazioni richieste dai client.

2.2 Struttura del codice

Il progetto è organizzato in una struttura di cartelle che separa logicamente i vari componenti del progetto. Questa organizzazione aiuta a mantenere il codice pulito, modulare e facile da gestire.

Di seguito, è riportata la struttura gerarchica delle cartelle:

```
project_root
+-- client
|   +-- build
|   +-- include
|   +-- src
+-- config
+-- database
|   +-- include
|   +-- query
|   +-- src
+-- server
    +-- build
    +-- include
    +-- src
```

client / server

Le cartelle `client` e `server` hanno la stessa struttura:

- `build`: Contiene i file oggetto generati dalla compilazione del codice contenuto in `src`
- `include`: Contiene i file header
- `src`: Contiene il codice sorgente

La separazione tra `include` e `src` permette una chiara distinzione tra le interfacce e le implementazioni, migliorando la manutenibilità del codice.
La cartella build isola i file generati dalla compilazione, mantenendo pulito il repository.

config

Questa cartella possiede dei file di configurazione con implementazioni comuni sia per il *client* che per il *server*. Ad esempio la configurazione del database, le richieste che client e server possono (rispettivamente) fare e ricevere, oppure la configurazione degli indirizzi.

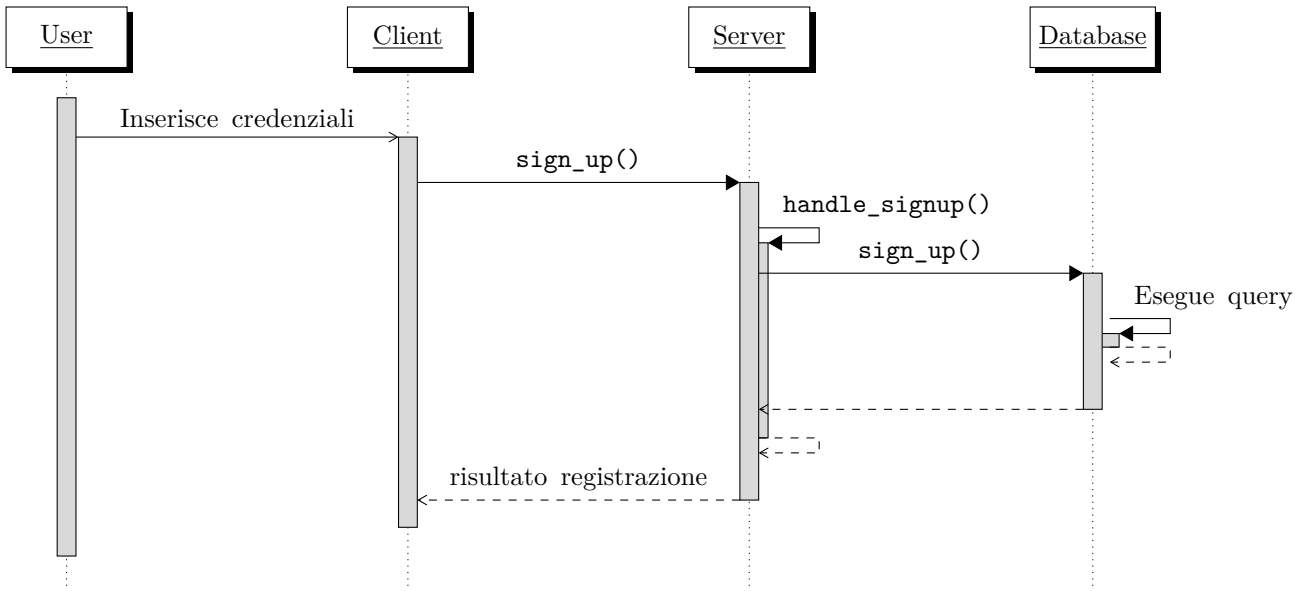
Centralizzare le configurazioni in una cartella dedicata facilita la gestione delle impostazioni del sistema e permette di modificare facilmente i parametri di configurazione senza dover cercare all'interno del codice sorgente.

database

- `include`: Contiene i file header per la gestione del database.
- `query`: Contiene i file SQL utilizzati per la gestione del database. (Creazione delle tabelle, trigger, funzioni, viste, …)
- `src`: Contiene il codice sorgente per l'interazione con il database.

La separazione dei file SQL e del codice sorgente relativo al database aiuta a mantenere organizzate le operazioni di gestione del database e facilita eventuali modifiche o aggiornamenti.

2.3 Registrazione e login



L'utente inserisce le credenziali per la registrazione (*sign_up*) o l'accesso (*sign_in*).
Il client invia tre messaggi al server:

1. tipo di richiesta: `SIGN_IN` o `SIGN_UP`
2. username
3. password

Il server elabora la richiesta eseguendo una query con i dati ricevuti e successivamente invia un messaggio al client per notificare l'esito dell'operazione, indicando se la richiesta è stata eseguita con successo o meno.

2.4 Gestione dei libri e del catalogo

Il client può inviare diverse richieste al server per ottenere informazioni sui libri presenti nel sistema. Il server risponde a queste richieste eseguendo specifiche funzioni che interrogano il database.
Di seguito, una panoramica delle funzionalità disponibili:

1. Visualizzare l'intero catalogo
2. Visualizzare solo i libri disponibili
3. Cercare libri per genere
4. Cercare libri per titolo

Tutte queste funzioni restituiscono i risultati in formato JSON. Invece di inviare una stringa per volta per poi dover scomporre ogni campo, tramite JSON si facilita l'integrazione lato client.

2.5 Prestito e restituzione libri

Il client può interagire con il server per gestire i prestiti dei libri. Il server offre diverse funzionalità per supportare queste operazioni:

1. Creare un nuovo prestito
2. Visualizzare i libri attualmente in prestito
3. Restituire un libro

2.6 Carrello e check-out

L'utente può aggiungere libri al carrello, che viene mantenuto localmente sul client stesso. Quando l'utente decide di procedere al checkout, i libri presenti nel carrello vengono inviati al server. Il server poi elabora la richiesta, gestendo il processo di prestito.

La funzione `handle_loan_requests` gestisce le richieste di prestito di libri da parte del client. Essa svolge le seguenti operazioni principali:

- Legge gli ISBN dei libri richiesti dal client e li memorizza in una lista fino a quando non riceve il messaggio "`STOP`".
- Blocca un mutex (semaforo) per garantire l'accesso esclusivo alla sezione critica del codice che gestisce la creazione dei prestiti, poi chiama la funzione `create_loans` per processare le richieste di prestito.
- Dopo aver elaborato le richieste, invia un codice di stato al client per indicare se la query è riuscita o meno.
- Sblocca il mutex al termine dell'operazione.

La richiesta di prestito di un libro è una sezione critica del sistema, poiché coinvolge la modifica dello stato, sul database, dei prestiti. In particolare, se un libro ha solo una copia disponibile e più utenti tentano di richiedere contemporaneamente il prestito di quel libro, il database potrebbe diventare inconsistente. Questo può succedere se entrambi i processi di richiesta vedono la copia come disponibile e procedono con l'assegnazione, portando a una situazione in cui il libro appare come prestato a più utenti contemporaneamente. L'uso di mutex garantisce che solo un processo possa effettuare la richiesta di prestito per quel libro alla volta, assicurando che il database rimanga coerente.

2.7 Gestione delle notifiche

Quando l'utente accede al sistema, il server verifica se ha prestiti scaduti che devono essere restituiti. Se l'utente ha prestiti scaduti, il server invia una notifica al client informandolo della situazione.

2.8 Funzioni speciali

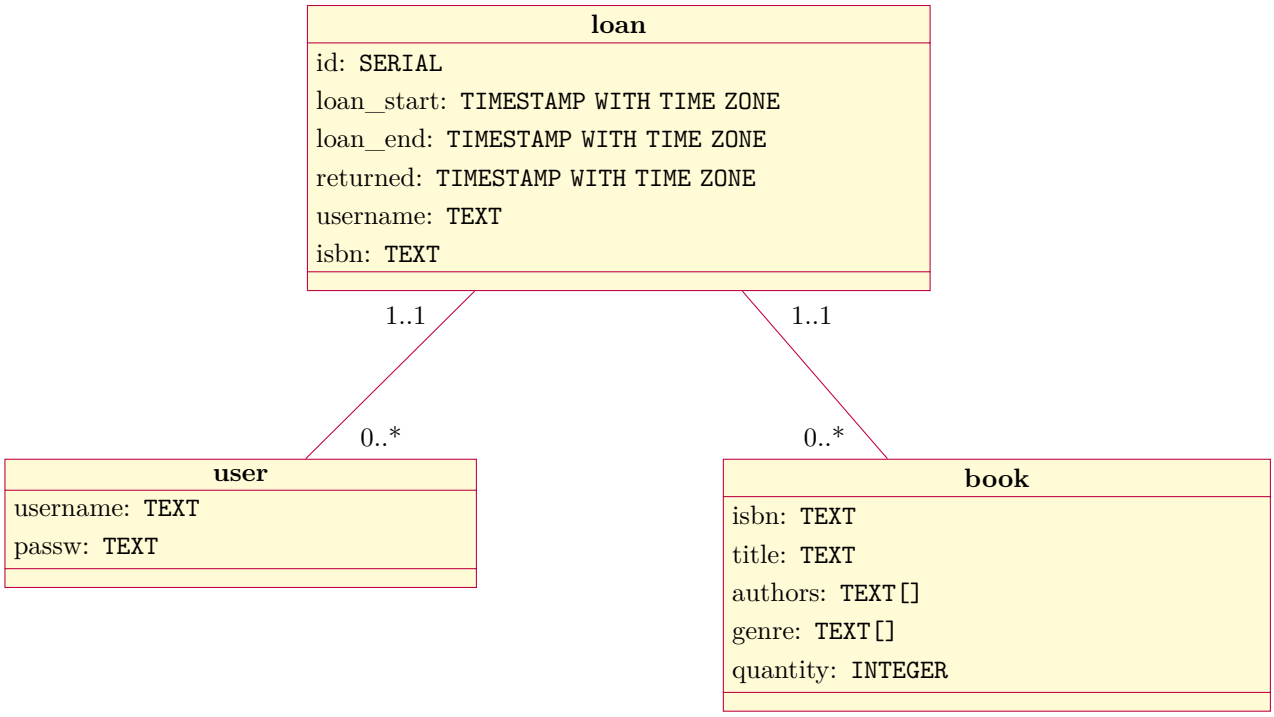
La funzione `send_string_segmented` invia una stringa al client tramite il socket, suddividendo la stringa in segmenti di lunghezza massima definita da `MAX_REQUEST_BUFFER_LENGTH`.
Iterativamente ogni segmento viene inviato al client in maniera ordinata.

La funzione `recv_and_compose_segmented_string`, riceve una stringa di lunghezza
massima `MAX_REQUEST_BUFFER_LENGTH`; esegue questo processo iterativamente, accodando il nuovo segmento al segmento precedente.
Al termine, la funzione restituisce l'intera stringa.

Queste funzioni sono necessarie per gestire l'invio e la ricezione di dati di lunghezza variabile poiché TCP non garantisce che tutti i dati vengano ricevuti in una singola operazione, specialmente se i dati sono di grandi dimensioni.

Database

3.1 Schema



3.2 Descrizione delle tabelle

La tabella `user` gestisce gli utenti del sistema.
Ogni record in questa tabella rappresenta un singolo utente, identificato univocamente dal campo `username` il quale funge anche da chiave primaria, garantendo cosi anche l'unicità.

La tabella `book` contiene le informazioni relative ai libri presenti nel catalogo della biblioteca. Ogni libro è identificato dal suo codice `ISBN`, che funge da chiave primaria. La tabella include campi anche per gli autori e i generi, entrambi memorizzati come array per gestire libri con molteplici autori o appartenenti a più generi. Il campo `quantity` tiene traccia del numero di copie disponibili per ciascun titolo, con un vincolo che assicura che questo valore non possa essere negativo.

La tabella `loan` gestisce le informazioni relative ai prestiti dei libri. Ogni prestito è identificato da un `ID`.

La tabella registra le date di inizio e fine prevista del prestito, nonché la data effettiva di restituzione (che può essere `null` per i prestiti in corso).

I campi `username` e `isbn` fungono da chiavi esterne. Questa struttura permette di tracciare efficacemente chi ha preso in prestito quale libro e quando.

3.3 Trigger

Il trigger `check_book_availability` verifica se un libro è disponibile prima di registrare un nuovo prestito. Quando viene eseguita, la funzione esamina la quantità di copie disponibili del libro: se il libro è disponibile inserisce il prestito; se il libro non è disponibile, viene sollevata un'eccezione, impedendo così l'inserimento del prestito e notificando l'utente che il libro non può essere prestato.

Il trigger `set_loan_dates` imposta la data di inizio del prestito alla data e ora corrente, e la data di fine a 30 giorni dalla data di inizio

3.4 View

Come si può intuire dal nome, `available_books` è una *view* che "aggiunge" alla tabella `book` il campo `available_quantity`, ovvero un intero che indica la quantità disponibile per quel libro.

Docker

4.1 Client

Il dockerfile del client utilizza l'immagine di base `gcc` per compilare e prepara l'ambiente per la compilazione del client:

1. installazione delle dipendenze di sistema, tra cui le librerie per `postgres` e `cJSON`
2. creazione delle cartelle di lavoro nella directory `/app`
3. imposta la directory di lavoro a `/app`
4. copia i file sorgente nelle cartelle precedentemente create
5. esegue il comando per compilare il client e successivamente il container entra in uno stato di sonno infinito mantenendo il container in esecuzione senza terminarlo, permettendo all'utente di accedere al container per ulteriori operazioni.

4.2 Server

Il dockerfile del server esegue le stesse operazioni specificate nel client fino al punto 4.
Dopodichè espone la porta 8080, utile per strumenti come Docker Compose, che configura il routing basandosi su questa informazione.

Quando il container viene avviato, non esegue alcun comando automaticamente (questo verrà specificato nel *docker-compose*).

4.3 Compose

Il file `docker-compose.yml` definisce una configurazione di `docker compose` per un'applicazione composta da tre servizi principali:

- Database
- Client
- Server

Database

Il database usa l'immagine di *postgres* per creare il container e specifica dei *volumes* tra cui:

- `db_data` per la persistenza dei dati
- monta la directory locale dove sono contenute le query che verranno eseguite alla creazione del database.

Il database si collega alla rete Docker "*library*" senza specifare un indirizzo IP poiché non è strettamente necessario per il suo funzionamento, dato che Docker ne gestisce automaticamente l'assegnazione.

Client

Il servizio *client* crea un'immagine *library_client* basato sul dockerfile presente nella cartella `./client`

Server

Il servizio *server* crea un'immagine *library_server* basato sul dockerfile presente nella cartella `./server` .

Al server viene assegnato un indirizzo IP specifico che servirà al client per stabilire la connessione tramite socket.

4.3.1 Network

La sezione `networks` definisce una rete che i servizi utilizzano per comunicare tra loro.

Specifica il tipo di rete a "*bridge*", un tipo di rete che collega i container tra loro e all'host Docker.