Videogame Store – Antonio Binanti

Il progetto "Videogame Store" è stato sviluppato per creare una piattaforma di e-commerce dedicata alla vendita di videogiochi. L'obiettivo principale era fornire un'interfaccia utente intuitiva e funzionale che permettesse agli utenti di navigare, cercare, acquistare e recensire videogiochi in modo semplice e veloce.

Caratteristiche Principali

Di seguito vengono descritte le caratteristiche principali del videogame store:

- **Catalogo di Videogiochi**: Un catalogo di videogiochi con ordine personalizzato in base alle preferenze passate dell'utente. Gli utenti possono sfogliare e filtrare i giochi in base alle loro titolo, prezzo e genere.
- **Sistema di Ricerca**: Funzionalità di ricerca che permette agli utenti di trovare videogiochi specifici utilizzando parole chiave, filtri e ordinamenti.
- **Carrello e Acquisti**: Un carrello virtuale dove gli utenti possono aggiungere i giochi desiderati e procedere con l'acquisto.
- **Recensioni e Valutazioni**: Gli utenti possono lasciare recensioni e valutazioni sui giochi acquistati, aiutando altri acquirenti a fare scelte informate.
- **Sistema di notifiche**: Il sistema di notifiche permette di avvisare l'utente ogni volta che viene inserito o modificato un nuovo videogioco nello store, mentre avviserà l'admin ogni volta che un utente effettuerà un'azione su un videogioco.

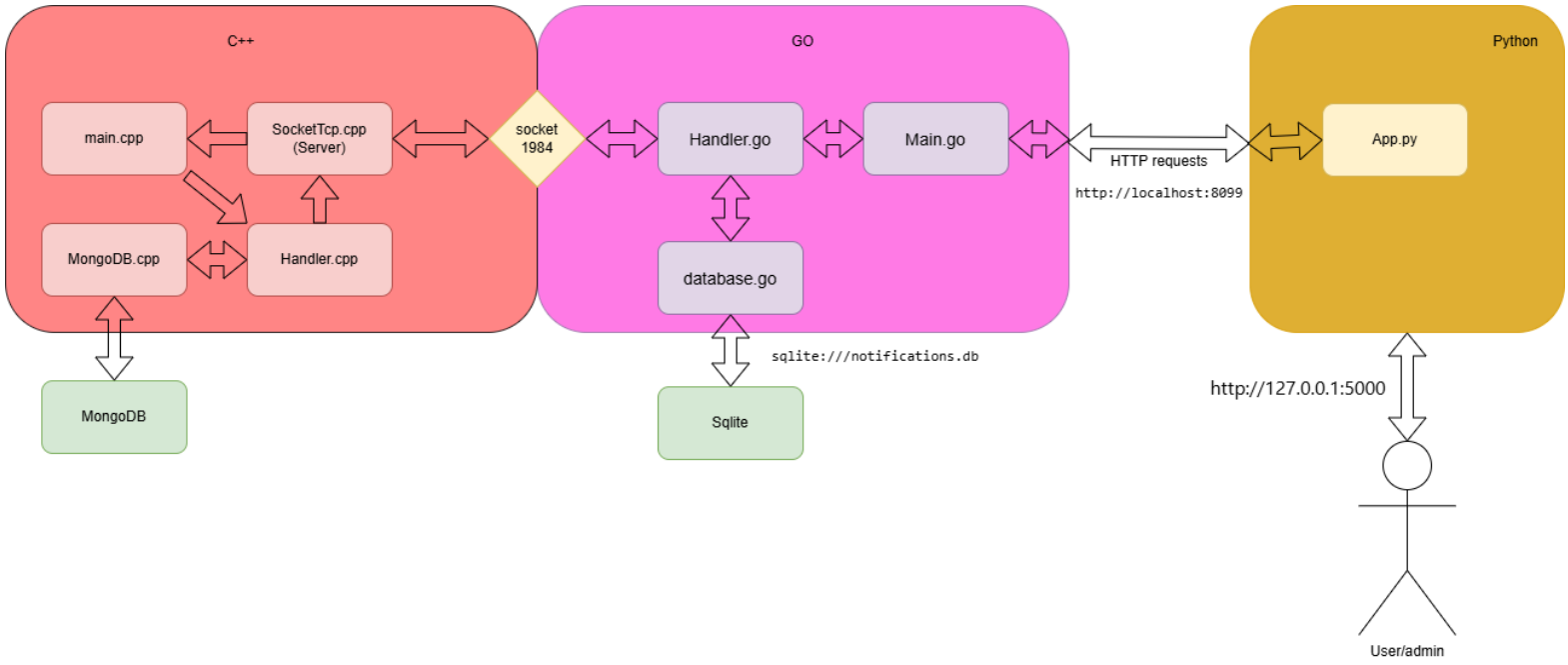
Architettura del Sistema

Il sistema è stato progettato utilizzando diversi moduli, ciascuno con funzionalità specifiche. Ecco una panoramica dell'architettura:

- **Backend (C++)**: E' stato utilizzato C++ per gestire la logica del negozio e la persistenza dei dati nel database. Per la persistenza dati viene utilizzato un database MongoDB. Implementa funzionalità come la gestione dei videogiochi, delle prenotazioni, delle recensioni e dell'ordinamento personalizzato. Inoltre effettua un sistema di autenticazione tramite jwt.
- **Middleware e gestore notifiche (Go)**: E' stato utilizzato Go per creare un servizio middleware tra frontend e backend. In particolare il modulo gestisce le richieste http provenienti dal frontend, le elabora e le inoltra al backent tramite una socket TCP, e lo stesso avviene per il verso opposto. Inoltre questo modulo gestisce il sistema di notifiche rivolto agli utenti e all'admin per informali su ciò che avviene all'interno del videogame store.

- **UI (Python)**: E’ stato utilizzato Python per implementare l'interfaccia utente. Nello specifico è stato utilizzato il framework per gli endpoint e il caricamento delle varie pagine.

Segue l’architettura generale dell’applicativo, in seguito verrano approfonditi i vari moduli.



Modulo Backend (C++)

Il modulo Backend è responsabile della gestione della logica del negozio di videogiochi e della persistenza dei dati nel database. È implementato in C++ e utilizza diverse librerie per gestire le operazioni di rete, l'interazione con MongoDB, l'autenticazione tramite JWT e altro ancora.

Descrizione dei Componenti Principali

1. **main.cpp**:

   Il file *main.cpp* costituisce il cuore del sistema, gestendo la configurazione, l'inizializzazione e l'esecuzione del server TCP. Esso carica le impostazioni dal file *config.ini* per stabilire la connessione con il database MongoDB e avviare il server sulla porta specificata. Il server accetta connessioni dai client in arrivo, gestendo ogni connessione in un thread separato per migliorare le prestazioni complessive. L'ascolto continuo delle connessioni e l'elaborazione delle richieste dei client avvengono in modo concorrente, garantendo una risposta tempestiva e efficiente.

2. **handler.h:**

   Il file *handler.h* contiene le funzioni necessarie per gestire una vasta gamma di operazioni richieste dai client, come il login, la registrazione, il recupero di informazioni sugli utenti e sui giochi, l'aggiunta e la gestione di recensioni, prenotazioni e acquisti. Le richieste degli utenti vengono autenticate utilizzando JSON Web Token (JWT), verificati ad ogni richiesta per garantire la sicurezza e l'autenticità delle operazioni. Le funzioni specifiche operano in modo coerente con il sistema di gestione delle sessioni utente, che prevede il rilascio e la gestione dei JWT validi per un massimo di due ore.

3. **MongoDB.h**:

   Nel file *MongoDB.h*, la classe singleton *MongoDB* facilita l'interazione con il database MongoDB, gestendo operazioni CRUD relative agli utenti, ai giochi, alle recensioni, alle prenotazioni e agli acquisti. Le funzioni di autenticazione come *signup* e *login* permettono agli utenti di registrarsi e accedere in modo sicuro. Il file utilizza la libreria *nlohmann::json* per manipolare dati in formato JSON, facilitando la trasmissione e la gestione dei dati tra il backend e il client.

4. **SocketTcp.h:**

   Infine, il componente *SocketTcp.h* implementa le funzionalità di connessione, ascolto e comunicazione tra il backend e il client GO tramite socket TCP tilizzando le librerie *winsock2.h* e *ws2tcpip.h* su piattaforma Windows. Consente al sistema di scambiare messaggi terminati dal carattere "|" per indicare la fine del messaggio, assicurando una comunicazione affidabile e robusta tra i componenti del sistema.

   La configurazione del socket avviene tramite la funzione *setupHints*, che imposta i parametri di connessione specificando l'uso del protocollo TCP e configurando l'indirizzo e la porta. Una volta configurato, il socket viene creato con la funzione *createSocket*, che gestisce anche il binding nel caso del server.

   La gestione delle connessioni in entrata è realizzata con la funzione *listenForConnections*, che mette il socket in ascolto per le connessioni dei client. Quando una nuova connessione viene accettata, la funzione *acceptConnection* crea un nuovo socket per la comunicazione con il client.

   Per la comunicazione vera e propria, SocketTcp include le funzioni *sendMessage* e *receiveMessage*. La prima invia messaggi ai client, aggiungendo un carattere terminatore per facilitare la gestione dei messaggi completi, mentre la seconda riceve messaggi dai client e gestisce la chiusura delle connessioni.

   Infine, la classe gestisce la connessione al database MongoDB tramite la funzione *connect_to_mongodb*, integrando le funzionalità di persistenza dei dati necessarie per il backend dell'applicazione. La funzione *cleanup* è utilizzata per liberare tutte le risorse allocate, chiudendo i socket e liberando la memoria allocata.

Questa struttura modulare permette al backend di essere scalabile, mantenibile e di gestire tutte le operazioni richieste dal videogame store.

Middleware e gestore notifiche (GO)

1. **Main.go:**
   Nella funzione *main*, attraverso *viper* viene letto un file di configurazione *config.in* , contenente le informazioni sulle porte del server e del socket TCP.
   Viene chiamata la funzione *setupDatabase* che configurerà un database Sqlite per la persistenza delle informazioni relative alle notifiche.
   Il routing delle richieste HTTP è gestito con *mux*. Viene creato un router e configurato con varie route per gestire funzionalità del Videogame store. Ogni route è associata a un handler specifico, definito in *Handler.go*, per eseguire operazioni come creazione utenti, login, recupero informazioni utenti, aggiunta e recupero giochi, gestione recensioni, prenotazioni, acquisti, e notifiche. Il server HTTP viene avviato sulla porta specificata nel file di configurazione usando http.ListenAndServe.

2. **Handler.go:**
   Il file *Handler.go* rappresenta il nucleo della gestione delle richieste HTTP nel sistema, fungendo da tramite tra il client che effettua le richieste tramite protocollo HTTP e il backend implementato in C++. Le funzioni definite in questo file si occupano di gestire varie operazioni come registrazione, accesso, aggiornamento e recupero di dati relativi agli utenti e ai giochi.
   Ogni funzione di handler decodifica i dati necessari dal corpo della richiesta o dai parametri, costruisce un messaggio adatto per le comunicazioni con il backend tramite socket TCP, e invia tale messaggio. Il backend processa la richiesta, esegue le operazioni necessarie e restituisce una risposta, che le funzioni di handler gestiscono per restituire al client http. Tali risposte sono in formato JSON e possono contenere i dati richiesti o un messaggio di errore in caso di problemi durante l'elaborazione della richiesta.
   Le funzioni di handler si preoccupano anche di gestire gli errori durante la decodifica dei dati della richiesta, la comunicazione con il backend e la gestione delle risposte. Utilizzano un meccanismo di risposta standardizzato che include l'encoding dei dati in formato JSON prima di restituire la risposta al client, garantendo coerenza e uniformità nelle interazioni con il frontend.
   Oltre alla gestione delle operazioni CRUD (Create, Read, Update, Delete) per gli utenti e i giochi, *Handler.go* include anche funzioni per la gestione delle notifiche agli utenti, aggiornando lo stato delle notifiche lette o non lette nel database Sqlite.
3. **Database.go:**

   Il file *database.go* gestisce il database delle notifiche per l'UI del videogame store, utilizzando SQLite come motore di database.

La funzione *setupDatabase* configura il database, aprendo una connessione a `notifications.db` e creando due tabelle se non esistono già: *notifications* per memorizzare le notifiche e *notification_reads* per tracciare lo stato di lettura delle notifiche da parte degli utenti.

Per aggiungere notifiche, ci sono due funzioni principali. *addNotification* inserisce una nuova notifica per tutti gli utenti, marcandola come non letta nella tabella *notification_reads*. *addNotificationToAdmi*` funziona in modo simile, ma aggiunge la notifica solo per l'utente "admin".

Le notifiche possono essere marcate come lette con la funzione *markNotificationAsRead*, che aggiorna lo stato di una notifica specifica per un utente. Le funzioni *getUnreadNotifications* e *getAllNotifications* recuperano rispettivamente tutte le notifiche non lette e tutte le notifiche per un utente specifico.

La gestione di recupero degli utenti è facilitata dalla funzione `GetAllUsers`, che comunica con il backend per ottenere la lista degli utenti e decodifica la risposta JSON.

UI (Python)

Il moduo Python utilizza il framework Flask per creare un'applicazione web dedicata al Videogame store. Ogni parte del codice definisce funzioni che corrispondono a diverse pagine dell'applicazione.

Per quanto riguarda la gestione dell'autenticazione degli utenti: la funzione *login()* gestisce il processo di login verificando le credenziali con il backend tramite una richiesta POST. Se l'autenticazione ha successo, l'utente viene riceve un JWT tokene viene reindirizzato alla homepage (*home()*), altrimenti vengono gestiti vari tipi di errore di connessione o di richiesta.

La pagina principale dell'applicazione (*home()*) è accessibile tramite la radice del sito (/). Se l'utente non è loggato, viene reindirizzato alla pagina di *login*. Una volta autenticato, l'applicazione recupera le informazioni dell'utente e i suoi giochi preferiti tramite richieste al backend. È possibile applicare filtri di ricerca per titolo, genere, prezzo e ordinamento dei giochi. Inoltre, vengono gestite le notifiche non lette dell'utente, visualizzandole se presenti.

La pagina di dettaglio di un gioco (*game(gameTitle)*) mostra le informazioni dettagliate di un gioco specifico, recuperate dal backend tramite una richiesta GET. Gli utenti amministratori hanno la possibilità di aggiornare o eliminare giochi utilizzando le funzioni *update_game_form(gameTitle)* e *delete_game(gameTitle)*.

La funzione `create_game()` gestisce la creazione di nuovi giochi tramite una richiesta POST al backend, verificando i dati inviati dal form e gestendo eventuali errori di connessione o validazione.

L'applicazione gestisce anche il carrello degli acquisti (*cart()*) e il processo di checkout (*checkout()*). Nel carrello, gli utenti possono visualizzare i giochi selezionati per l'acquisto e rimuoverli se necessario. Durante il checkout, vengono processati gli acquisti, eliminando gli articoli dal carrello dopo aver completato con successo la transazione.

Le funzioni *notifications()* e *mark_as_read(notification_id)* gestiscono le notifiche degli utenti. La prima mostra tutte le notifiche dell'utente, mentre la seconda marca come letta una notifica specifica tramite una richiesta POST al backend.

Infine, l'applicazione fornisce anche la funzionalità di registrazione degli utenti (*signup()*), consentendo loro di creare nuovi account nel sistema.