Relazione progetto programmazione distribuita: "Warehouse Manager"

1 Introduzione

Warehouse Manager è una semplice applicazione web pensata ad essere usata per la gestione dei magazzini. Permette di seguire più magazzini contemporaneamente e visualizzarne lo stoccaggio, avvertendo l'utente ogni volta che un articolo sta per finire.

2 Architettura

Il software è stato realizzato con un'architettura a strati organizzato in tre moduli:

- model: modulo che fa da interfaccia fra gli strati più alti dell'applicazione e il database.
- auth: avvolge il modulo precedente per garantire un accesso sicuro ai dati da parte degli utenti.
- handlers: contiene i numerosi handler usati per implementare il website e costituisce la maggior parte della logica dell'applicazione.

Un'altra caratteristica dell'architettura software è che ha seguito per il più possibile una filosofia RESTful nell'implementazione dei servizi, andando a identificare ciascuna risorse/servizio con un URI e permettendo operazioni su di essa. L'applicazione è stata realizzata per essere usata in un'architettura client-server: gli eseguibili costituiranno solo la parte server dell'intero sistema in quanto quella client potrà essere gestita da qualunque browser moderno.

Per quanto riguarda il package model questa dichiara e implementa l'interfaccia WarehouseRepository per gestire in un unico luogo le operazioni di accesso al database. Lo struct GORMSQLiteWarehouseRepository, implementante WarehouseRepository, usa la libreria GORM per effettuare operazioni su un database SQLite. L'applicazione definisce 3 modelli per la memorizzazione dei dati nel repository:

- Warehouse: modella i magazzini. La chiave primaria è ID, con altri attributi aggiuntivi come nome, posizione e capacità.
- Item: modella gli articoli, e come warehouse ha chiave primaria ID. Altri campi sono nome, quantità, descrizione e categoria.

- WarehouseItem: modella la relazione di appartenenza tra articoli e magazzini. Ha due chiavi esterne: WarehouseID e ItemID. Questa relazione è importante perché permette di descrivere l'appartenenza di uno stesso articolo a più magazzini usando l'attributo quantità-articolo.

Il package Auth invece è usato per gestire gli account utente: lo struct più importante qui è il AuthenticationManager, che conserva una lista di utenti registrati e una lista di utenti attivi al momento. Viene anche usato un file json per persistere gli account anche in seguito alla terminazione dell'esecuzione. La lista degli utenti viene caricata a inizio esecuzione dal file json e usata per tener traccia degli utenti durante l'esecuzione. Se questa venisse modificata, ad esempio per aggiunta di un nuovo utente, si salvano le modifiche scrivendo nel file json. Viene anche assegnato ad ogni utente un database che useranno per conservare i dati: questo permette di avere un ulteriore livello di protezione per quanto riguarda la sicurezza degli dati. Gli utenti per poter accedere al repository assegnato dovranno effettuare il login. Il login consiste nel controllo del password crittografato per spostare l'istanza corrispondente dalla lista Users alla lista ActiveUsers. Una volta fatto ciò AuthenticationManager provvederà ad inoltrare le chiamate ai metodi al repository sottostante. Inoltre, dato che basta una sola istanza del manager per fornire le funzionalità richieste, ho pensato di usare il pattern singleton: il metodo LoadAuthManager è l'unico modo per ottenere oggetti tipo AuthenticationManager e resitituisce sempre la stessa istanza da una variabile globale. Un'altro pattern che ho usato è il dependency injector: Si passa a AuthenticationManager una funzione che poi sarà usata nel momento opportuno per creare una WarehouseRepository.

Infine abbiamo la parte del codice che implementa il business logic. Qui possiamo distinguere due tipi generali di handler: i handler che mostrano le pagine del web e i handler che gestiscono le richieste di operazioni. I primi si accedono con il metodo GET, mentre gli altri con il metodo POST (i metodi DELETE e PUT non sono supportati da HTML). Per i handler che ricevono solo richieste POST ho configurato il router gorilla/mux in modo che solo le richieste con metodi POST potevano arrivarci e ho fatto lo stesso per gli handler che ricevono solo metodi GET. Invece, per gli handler che possono ricevere sia richieste GET che POST ho usato uno switch per distinguere i due casi. Per poter visualizzare le pagine web stilizzate con CSS ho usato gorilla/mux per implementare un file server per servire i file statici. A proposito della stilizzazione con CSS degli file HTML ho usato il framework Simple.css che permette di avere una stilizzazione basilare della pagina web. Un pattern che ho fatto ampiamente uso nel modulo è il chain of responsability: ho aggiunto un middleware dedicato a redirigere le richieste in base allo stato della sessione di accesso per quasi tutti i handler. Per esempio SessionIsAbsentRedirectHandler si occupa di redirigere le richieste di client non autenticati alla pagina del login, mentre SessionIsPresentHandler fa il contrario, redirigendo le richieste di client autenticati per la pagina di login o di registrazione alla pagina home. Per quanto riguarda la gestione delle sessioni di accesso degli utenti, questa è fatta attraverso l'uso della variabile globale UserSessions. Questa è uno struct che contiene un sync.Mutex e una mappa da stringhe a UserSession. Il lock serve a sincronizzare gli accessi alla variabile dato l'esistenza di una goroutine parallela dedicata a eliminare le sessioni scadute. Ogni utente, quando fa il login, genera una nuova sessione utente

associato ad un token. Il token viene memorizzato attraverso un cookie lato client in modo che l'utente non avrà bisogno di identificarsi ogni volta. Il token sarà valido finché la corrispondente sessione è presente in userSessions e non è scaduta. Logout permette di eliminare il cookie e la sessione corrispondente in maniera precoce. Come per i password anche i token si ottengono applicando una funzione hash: nel caso dei token questo è utile perché rende molto difficile indovinarle in maniera casuale. Un'altra funzionalità importante del sito è quello di poter mostrare comunicazioni ed errori all'utente attraverso messaggi flash: questi vengono generati in seguito ad azioni dei handler e passati attraverso cookie ai handler GET che poi li mostrano nella pagina web integrandoli nei template. Per fare tutto ciò è molto importante che i cookie non vengano persi o ignorati dal browser. Infatti, se non si setta il campo Path quando si ridirige una richiesta alla URL di destinazione o quando si cerca di eliminare un cookie, questo risulta in errori come la scomparsa degli messaggi o la loro ritenzione prolungata. Path è un campo nel cookie che indica dove esso sarà accessibile nel server.

3 Tecnologie e strumenti

Nel progetto ho usato una serie di strumenti, ecco la lista completa:

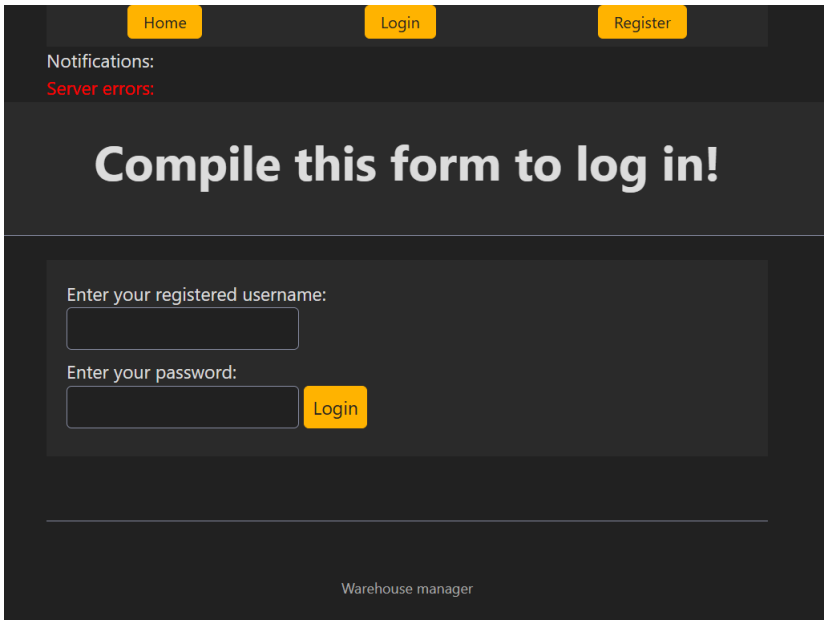
- SQLite: è una libreria C che implementa un database engine per SQL che è piccola e veloce, usata qui per conservare i dati applicativi.
- GORM: libreria ORM (Object-Relational Mapping) per golang, che permette di effettuare operazioni su un database sottostante attraverso costrutti in linguaggio go. Usato per gestire SQLite.
- crypto: pacchetto golang usato per operazioni crittografiche. L'ho impiegato per codificare i password degli utenti con sha256.
- encoding: pacchetto per conversione dati json, hex e base64 per i cookie.
- gorilla/mux: gorilla è una raccolta di strumenti per lo sviluppo web. Nel mio caso ho usato gorilla/mux per il routing delle richieste http.
- html/template: per generazione dinamica delle pagine html.
- net/http: fornisce implementazioni client http e server http. Usato qui per implementare i handler.
- sync: per sincronizzazione delle goroutine.
- testing, net/http/httptest e regexp: per gli unit test di auth e model, e integration test per il comportamento routing del prodotto finale.
- simplecss: per stilizzazione pagine web.

4 Argomenti coperti

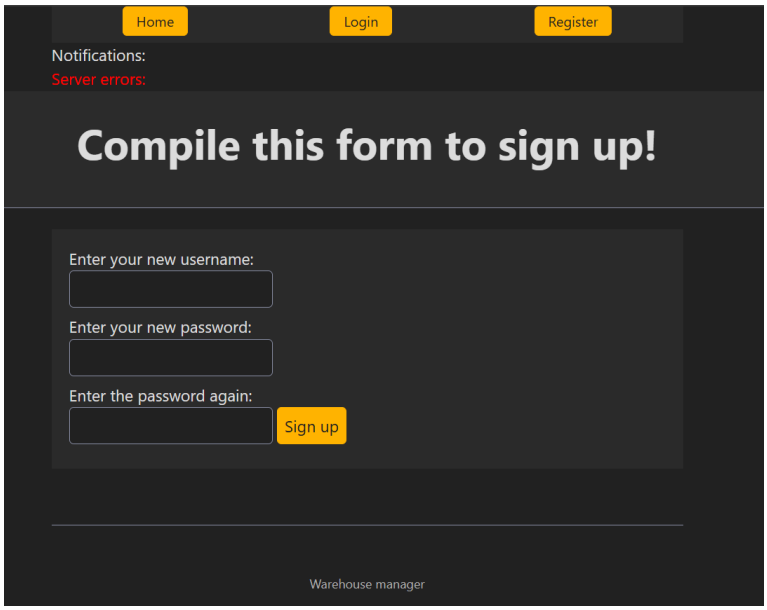
Il progetto copre soprattutto argomenti relativi alla sezione "web development" del corso anche se alcune volte fa riferimenti ad altri punti come nel caso del pattern dependency injection o il marshalling e unmarshalling dei dati in formato json.

5 Caso d'uso

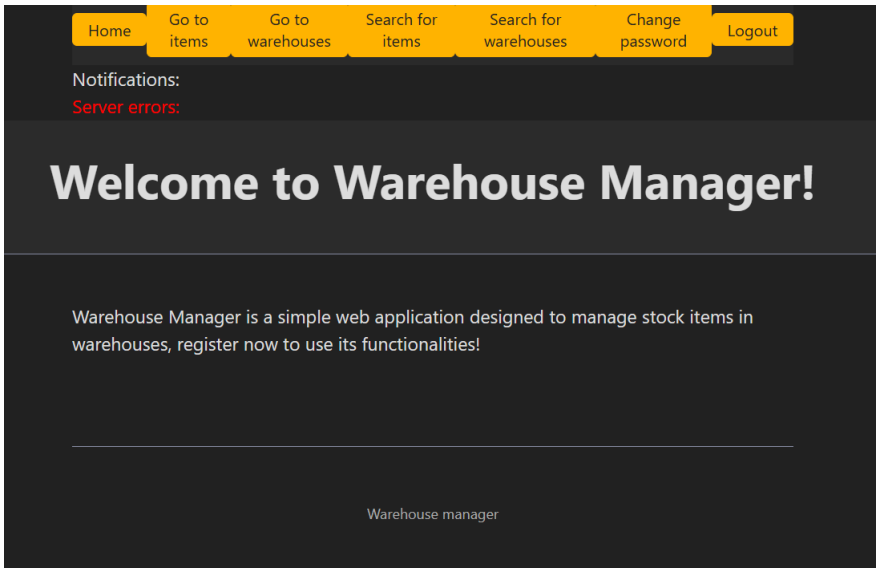
Pagina login del website:



Se si cerca di accedere a qualsiasi altra pagina che non è tra quelle mostrate in navbar, si è ridiretti alla pagina del login.



Prima di poter effettuare il login dobbiamo però registrarci, e questo si fa attraverso la finestra register.



Dopo aver terminato la registrazione saremo ridiretti alla pagina del login dove potremo accedere usando le credenziali appena create. Una volta fatto il login si è liberi di navigare nel sito e usare tutte le sue funzionalità.

6 Per eseguire

Per eseguire il software è importante:

- avere una cartella "data" nella root directory del progetto per contenere i file db e json prodotti durante l'esecuzione, in quanto golang non riesce a crearlo da solo.
- assicurarsi che la cartella static contenga styling.css per poterlo servire a richiesta e mostrare pagine più carine

Per eseguire i test:

- come prima abbiamo bisogno di una cartella data nelle directory per poterle eseguire senza errori (per testare warehouseRepository non c'è ne bisogno).

Se si usa windows per eseguire l'applicazione è possibile incontrare errori dati da incompatibilità del compilatore C, nel mio caso è stato "cgo cc1.exe: sorry, unimplemented: 64-bit mode not compiled in". La soluzione è installare la suite di compilatori per microsoft windows TMD-GCC e configurare la variabile d'ambiente PATH in modo che sia l'unico usato o comunque possa avere precedenza.