# **NotifyMaps: Progetto Advanced Programming Languages**

**C# - Client dell’applicativo**

Per il client C# si è voluto utilizzare Windows Form, framework di sviluppo di applicazioni grafiche per il sistema operativo Microsoft Windows. Windows Form fornisce un'interfaccia di programmazione delle applicazioni che consente agli sviluppatori di creare applicazioni desktop con interfaccia grafica per Windows, grazie al supporto del concetto di eventi, fornisce un sistema di gestione degli eventi che permette di rispondere alle azioni dell'utente, inoltre include una vasta gamma di oggetti predefiniti come Button, TextBox, Label, etc. che ci hanno permesso di semplificare la creazione delle varie finestre.

L’intero applicativo viene suddiviso principalmente in due parti: una parte grafica presente nella cartella **UI** che contiene al suo interno tutti i componenti delle varie finestre e gli eventi ad essi associati e la cartella **Struttura** che contiene tutti gli oggetti e interfacce utilizzate nella parte di logica dell’applicativo.

L’utente una volta avviata l’applicazione si trova davanti a una prima schermata dove sarà possibile scegliere quale azione intraprendere, se registrarsi poiché si sta utilizzando per la prima volta l’applicazione o effettuare il login per poter accedere al lato manager dell’applicazione.

Una volta cliccata nella sezione di registrazione vi si troverà davanti a una nuova finestra così da poter inserire i seguenti dati: **nome,cognome,email,password,confermapassword** per poter effettuare tale azione, nella parte inferiore della finestra vi sarà un bottone che permetterà l’invio dei dati, una volta cliccato il pulsante verrà chiamato l’evento associato a tale pulsante, esso verificherà che nessuno dei campi sia vuoto, che la password inserita uguale a quella del campo presente nel “conferma password” e che sia inserita una mail valida per la registrazione, una volta effettuati tali controlli, se si avrà riscontro positivo si effettuerà si avrà un riscontro in una label e verrà effettuata una chiamata API di tipo post per procedere con la registrazione dell’utente, se tale chiamata api ha esito positivo l’utente mediante un **MessageBox** riceverà un riscontro a video consigliando successivamente il login, se la richiesta dovesse avere esito negativo apparirà l’errore relativo a schermo, se una delle condizioni riguardante i campi non dovesse essere rispettata, una label dedicata indicherà l’azione da intraprendere all’utente.

La logica utilizzata all’interno della finestra utilizza le varie classi definite nella cartella Struttura, dove viene definito una classe base **RichiestaRest** che contiene al suo interno dei metodi che verranno utilizzati in base al tipo di richiesta che si effettua GET/POST e al tipo di dato che ci aspettiamo come ritorno. La classe **RichiestaRest** si specializza in una **RichiestaLogin** che conterrà gli attributi di email e password, utilizzati per effettuare le operazioni di login, operazioni che vengono svolte nei vari metodi. A sua volta la **RichiestaLogin** si specializza in **RichiestaRegistrazione** per effettuare l’operazione di registrazione. Inoltre sarà presente un ulteriore oggetto **RichiestaRoute** che specializza **RichiestaRest** e che permette di effettuare tutte le operazione relative le tratte(iscrizione e eliminazione).Per fornire il dato in modo corretto per effettuare le richieste rest utilizzate all’interno dell’applicativo, si utilizza un metodo in particolare definito nella classe RichiestRest **GetOggettoSerializzato** che ci permette passando un tipo Object di avere indietro lo stesso oggetto serializzato usando la Libreria **Newtonsoft.Json**, una libreria JSON per la serializzazione e deserializzazione in .NET .

Un pattern sviluppato all’interno del nostro applicativo è quello del Circuit Breaker che permette di limitare le richieste al server se esso dovesse essere offline o non raggiungibile, esso sarebbe stato definito all’interno della classe **CircuitBreaker**, esso viene gestito come un singleton per permettere l’accesso a tale istanza all’interno di tutte le finestre e così mantenere lo stato del circuit breaker.

All’interno dell’applicazione vi sono presenti finestre per le seguenti operazioni: Login, Registrazione, Gestione delle tratte e aggiornamento, eliminazione dati utente.

La finestra più importante nonché cuore dell’applicativo client è **UiManagerWindow** ad essa si accede dopo aver fatto il login, e permette attraverso delle **ComboBox** di selezionare la provincia di partenza e destinazione così da poter inviare iscrizione della tratta al server mediante chiamata Rest e poter essere notificato in futuro mediante Client Telegram.

Una volta iscritto a una tratta, essa verrà mostrata nella sezione centrale della pagina mediante una label e con accanto un pulsante per la disiscrizione quando non si vuole più ricevere notifiche sullo stato del traffico riguardante quella tratta.

Sempre nella stessa schermata sarà presente un pulsante per effettuare logout con relativa chiusura della finestra.

Un pulsante per aggiornare la lista delle tratte alla quale si è iscritti così da aggiornare la parte centrale della finestra e ricaricare le tratte nel caso vi siano state delle modifiche mediante l’utilizzo di un altro client. Questo meccanismo viene gestito mediante l’utilizzo di una **lista di tuple** dove vengono assegnati come elementi di una **tupla** la relativa label della tratta e relativo tasto per la disiscrizione da tale tratta.

Tutti i dati mostrati nella seguente finestra sono frutto di richieste rest, una relativa alla provincia che vengono caricate come elementi delle **ComboBox** e una che effettua una GET delle tratte assocciate all’email dell’utente.

**GO - Server Principale e Server Autenticazione**

La struttura lato server implementata mediante linguaggio go si basa su due server un server principale che si occupa di gestire tutte le chiamate API provenienti dall’esterno e un server che si occupa esclusivamente del lato utenti.

Il server principale gestisce non solo la comunicazione proveniente dall’esterno mediante chiamate Rest, gestisce la comunicazione con Kafka per la persistenza e inoltro dei dati dei vari topic, la comunicazione con un database MySQL per la memorizzazione dei dati e delle relazioni tra utenti e tratte, gestisce le chiamate effettuate alle API Google di Geocoding e Routes.

Una volta avviato il server, esso avvierà una goroutine per la creazione dei topic che saranno tutte le possibili combinazioni tra le province siciliane sia per le tratte di andata, sia per le tratte di ritorno da ogni provincia, se esse sono già presenti ignorerà tale creazione, proverà a connettersi al database MySQL se si effettua correttamente la connessione egli creerà il database Route e internamente inserirà una tabella per le province, una tabella per le route dove saranno presenti tutte le tratte dove è iscritto ogni singolo utente. Una volta completato ciò si aprirà un server http in modo tale da poter accettare richieste REST, grazie all’avvio della goroutine in parallelo verrà gestito l’invio dei messaggi ai topic KAFKA una prima volta all’avvio e poi periodicamente ogni ora.

All’interno del Server è stato implementato il pattern CircuitBreaker mediante l’utilizzo del package gobreaker, la possibilità di utilizzare tutti i meccanismi di Query di un Database di tipo mysql tramite il package go-sql-driver/mysql.

Ogni connessione a un servizio/server differente ha un CircuitBreaker dichiarato con regole di visibilità del package dove si trova, sono stati utilizzati gli stessi parametri di configurazione per tutti i circuit breaker.

All’interno del file di configurazione saranno presenti tutti gli indirizzi per la connessione con i vari servizi/server e le api google utilizzate.

La struttura delle cartelle del progetto rispecchia la struttura nonché i servizi con la quale ogni server si interfaccia. Nel server main si gestirà mediante il package database verranno gestite tutte le chiamate al database per le operazioni creazione, lettura, cancellazione, attivazione e disattivazione di una tratta.

Tramite i servizi google di Geocoding possiamo andare a definire una posizione mediante una città e/o una via in modo tale da ottenere indietro una posizione in latitudine e longitudine utilizzate per utilizzare i servizi google di Route e ottenere una tratta con i relativi dati ad essa associata.

Tramite Kafka si va a gestire la comunicazione tra i vari client mobile, poiché una volta creati i topic i messaggi verranno inviati ad essi tramite il server python che potra smistarli alle relative chat telegram.

Tramite l’apertura della porta 25536 accettiamo chiamate rest che permettono di ricevere dati dal client C# o mediante altre piattaforme (ex. Postman) che permettono l’inoltro al serverauth per quanto riguarda le operazioni di registrazione e autenticazione, mentre se vengono fatte richieste riguardanti gli endpoint:

* deletesRoute

Tramite la funzione **HandleDeleteRouteRequest** riusciamo a cancellare una specifica Tratta prendendo in considerazione la partenza e la destinazione e l’email dell’account alla quale essa è collegata.

* registerRoute

Tramite la funzione **HandleRegisterRouteRequest** riusciamo a registrarci a una specifica Tratta prendendo in considerazione la partenza e la destinazione e l’email dell’account alla quale essa è collegata.

* enableRoute

Tramite la funzione **HandleEnableRouteRequest** riusciamo ad attivare la ricezione di notifiche a tutte le tratte prendendo relative all’email dell’account alla quale esse sono collegate.

* disableRoute

Tramite la funzione **HandleDisableRouteRequest** riusciamo a disattivare la ricezione di notifiche a tutte le tratte relative all’email dell’account alla quale esse sono collegate.

* getRoute

Tramite la funzione **HandleGetRoute** riusciamo a disattivare la ricezione di notifiche a tutte le tratte relative all’email dell’account alla quale esse sono collegate.

* getprovince

Tramite la funzione **HandleGetProvince** riusciamo a ottenere una lista contenente tutte le province presente nel database.

* authentication

Tramite la funzione **HandleAuthRequest** effettuiamo una basic authentication cosi da inoltrare i dati al server di autenticazione che si occuperà di confermare o negare il login dell’utente

* register

Tramite la funzione **HandleRegisterRequest** effettuiamo un inoltro dei dati al server di autenticazione che si occuperà di registrare o meno in caso di errore l’utente.

* Gettg

Tramite la funzione **HandleGetTg** andiamo a selezionare tutte le route che hanno le notifiche abilitate e la inoltro al server auth successivamente tale lista contenente le email verrà filtrata in modo tali da avere email univoche e otterremo il corrispondente id\_tg da esse.

* getuserdata

Tramite la funzione **HandleGetUserData** andiamo a utilizzare tale funzione per il login mediante il cliente telegram poiché ci permette di aggiornare il campo id\_tg una volta che viene effettuato il login per la prima volta dall’app dopo aver effettuato la registrazione sul client C# e ritorna una lista delle tratte a cui è iscritto.

* updateuserdata

Tramite la funzione **HandleUpdateUserData** andiamo a utilizzare tale funzione per aggiornare i dati mediante l’inoltro dei dati provenienti dal client C#

* deleteuser

Tramite la funzione **HandleDeleteUserData** andiamo a utilizzare tale funzione per eliminare i dati delle route e dell’utente mediante l’inoltro della richiesta proveniente dal client C#

Le precedenti funzioni descritte che riguardano l’utente vengono dopo essere state ricevute dal server principale inoltrate al server Auth alla porta 8081.

I relativi endpoint ci permettono di ricevere i dati e aggiornare/aggiungere/eliminare i dati attualmente presenti sul database di autenticazione.

All’interno del Server auth è stato implementato il pattern CircuitBreaker mediante l’utilizzo del package gobreaker per la connessione con il database.

**Python - Server Manager** Notification **e Server Command Telegram**

* **Server Command Telegram**

Il bot gestisce un processo di autenticazione tramite una conversazione di login, connettendosi a un server di notifiche per verificare le credenziali dell'utente. Una volta effettuato il login l’utente potrà ricevere le informazioni sul traffico a cui ha effettuato l’iscrizione sul client desktop.

L’utente mediante il comando di start avvia la conversazione col bot, il quale replica con un messaggio di benvenuto e con un pulsante di login.

L’utente una volta premuti il pulsante di login entra in una conversazione col bot dove vengono chiesti in sequenza E-mail e Password per effettuare l’accesso al servizio. Durante tale conversazione è possibile interrompere l’inserimento delle credenziali digitando un comando specifico.

Una volta effettuato l’accesso sarà possibile interrompere il servizio con il comando di logout.

* **Connessione al Notification server**

Quando l’utente inserisce le credenziali di accesso o vuole eseguire il logout viene stabilita una connessione a un server Flask tramite richieste POST. Nelle richieste vengono incapsulati l’e-mail, la password e l’ID della chat di Telegram in modo da poter contattare l’utente per restituire i messaggi del servizio e comunicazioni di vario tipo, come la notifica di server momentaneamente non disponibile.

Una volta ricevuta una risposta dal server, questa, viene analizzata e restituita al bot in modo da poter rispondere all’utente con l’eventuale messaggio ricevuto dal server (autorizzato, logout effettuato, utente non trovato, ...).

* **Server Flask - Manager Notification**

Il server Flask implementa delle route per gestire le richieste di autenticazione e logout da parte degli utenti tramite richieste POST.

Durante l’avvio il server aspetta che il server principale si avvia con successo per effettuare una richiesta di tipo GET per recuperare eventuali utenti che precedentemente avevano fatto l’accesso, questo meccanismo viene utilizzato per notificare gli utenti di ripetere l’operazione di login dal momento che il server ha avuto un crash inaspettato.

Una volta che il server è avviato e riceve una richiesta di login, questo si occupa di inoltrare i valori ottenuti dal bot al server principale. Se la richiesta è andata a buon fine, quindi il server principale mi restituisce un valore che indica che l’utente è stato trovato nel database utenti, il server manda un’altra richiesta POST al server principale, dove manda solo l’e-mail dell’utente, per recuperare i vari percorsi a cui è iscritto.

Se anche questa richiesta va a buon fine viene effettuata un’iscrizione al topic di kafka che da quel momento in poi controlla periodicamente il topic. Quando il consumer kafka nota che è stato inserito un messaggio nel topic questo viene preso e scritto nella chat del bot.

Questo procedimento continua fino a quando non viene richiamata la route di logout. Questa come il login effettua una richiesta POST al server principale, dove viene passata l’e-mail, per notificare che l’utente ha chiesto di interrompere il servizio. A questo punto il server flask interrompe il polling del consumer kafka disiscrivendo l’utente dal topic.

* **Circuit Breaker**

Per tutte le richieste verso i server, sia dal bot al server flask e dal server flask al server principale, viene utilizzato il meccanismo di circuit breaker per individuare malfunzionamenti e disabilitare momentaneamente l’accesso ad un servizio non funzionante.

# Scelte implementative

**Python**

* **ConversationHandler**: il Conversation Handler è stato introdotto per semplificare la gestione delle conversazioni con gli utenti tramite Telegram, fornendo un'implementazione basata su una macchina a stati. Questo strumento offre una serie di funzioni di gestione (handler) progettate per gestire l'input dell'utente e gestire lo stato evolutivo della conversazione. Per associare gli handler al Conversation Handler, viene utilizzata la programmazione funzionale, consentendo così la chiamata selettiva delle funzioni quando richiesto.
* **Async e await**: l’utilizzo di Async e await è stato utilizzato per la facilità di gestione della programmazione asincrona per gestire le richieste concorrenti nel bot.
* **Decorator**: il decorator è stato utilizzato perché consente di separare il codice che implementa un aspetto specifico della logica da quella della funzione principale. Questo favorisce una struttura di codice più chiara e modulare. Inoltre, viene definito una volta e riutilizzato su più funzioni. Ciò riduce la duplicazione del codice e semplifica la manutenzione.
* **Flask**: viene utilizzato perché è semplice implementare e gestire le API rest.
* **Thread Lock**: viene utilizzato poichè fornisce un meccanismo di sincronizzazione per l'accesso concorrente alle risorse condivise in ambienti multithreading. La sua utilità principale è: evitare conflitti durante l'accesso concorrente, garantendo che solo un thread alla volta abbia accesso alle sezioni critiche del codice.

**C#**

* **Windows Form**: framework di sviluppo di applicazioni grafiche per il sistema operativo Microsoft Windows. Windows Form fornisce un'interfaccia di programmazione delle applicazioni che consente agli sviluppatori di creare applicazioni desktop con interfaccia grafica per Windows, grazie al supporto del concetto di eventi, fornisce un sistema di gestione degli eventi che permette di rispondere alle azioni dell'utente, inoltre include una vasta gamma di oggetti predefiniti come Button, TextBox, Label, etc. che ci hanno permesso di semplificare la creazione delle varie finestre.
* la Libreria **Newtonsoft.Json**, una libreria JSON per la serializzazione e deserializzazione in .NET.

**GO**

* **Routing delle Richieste**: Il ServeMux consente di associare diverse funzioni di gestione a diversi percorsi. Questo significa che si può definire come il server risponderà a diverse richieste a seconda del percorso specificato nella richiesta.
* **Gestione Concorrente:** http.ListenAndServe gestisce la concorrenza automaticamente, avviando una nuova goroutine per ogni richiesta. Quando si utilizza un ServeMux, puoi garantire che ogni richiesta venga instradata alla giusta funzione di gestione in modo sicuro e concorrente.
* **Organizzazione in oggetti**:

Utilizzare strutture permette di organizzare il codice in modo logico e coeso. Puoi raggruppare variabili e funzioni associate in una struttura, migliorando la leggibilità del codice.

**Generali**

* **Kafka**: abbiamo utilizzato Kafka come sistema di messaggistica per l'invio e la ricezione di messaggi tra diverse componenti del sistema. La messaggistica asincrona consente la comunicazione tra componenti senza richiedere una sincronizzazione immediata. Inoltre, i messaggi possono essere archiviati per un periodo definito, garantendo la durabilità e la possibilità di recuperarli in una fase successiva.
* **Circuit breaker**: Il circuit breaker è stato implementato poiché aiuta a gestire fallimenti e errori; infatti, quando un servizio esterno raggiunge un certo numero di fallimenti consecutivi, il circuit breaker interviene per prevenire ulteriori chiamate e permette al sistema di gestire il problema.