# PhotoHub: Progetto Advanced Programming Languages

**C# - Client dell'applicativo**

Per il Client scritto in C# si è utilizzato il framework multipiattaforma .NET MAUI con la versione .NET 7. Tale framework permette di scrivere un unico codice che verrà poi compilato nativamente per i vari dispositivi con SO differente su cui eseguire l'applicativo come ad esempio Windows, MacOS, iOS, Android e altri.
Il tutto richiamando le funzioni di libreria presenti in Microsoft.MAUI.*.
Il framework permette di tradurre file in formato .xaml in oggetti delle classi MAUI che verranno sostituiti con gli oggetti nativi di ogni piattaforma.
L'entrypoint del programma è il metodo CreateMauiApp nel file MauiProgram.cs che configura l'app MAUI e la fa partire. Una classe singleton UserService mantiene le informazioni dell'utente e ne permette la memorizzazione come Preference di MAUI per successivi avvii dell'app.
In base alla presenza o meno delle credenziali verrà visualizzata la pagina di Login o una TabbedPage che contiene una pagina Post e una pagina Profilo.
Ogni pagina sfrutta il pattern Model-view-viewmodel (MVVM): ogni pagina ha la proprietà BindingContext settata a un oggetto della classe che rappresenta il componente viewmodel per quella pagina. La classe viewmodel implementa l'interfaccia INotifyPropertyChanged che richiede la presenza dell'evento PropertyChanged invocato dalla funzione NotifyPropertyChanged che permette di notificare il cambiamento del valore della proprietà agli oggetti grafici che hanno effettuato il binding sulla proprietà in questione, il tutto sulla base del pattern Observer.
Altra parte rilevante è la comunicazione col server scritto in C++. È stato creato l'oggetto singleton di tipo RestService che utilizza l'oggetto di tipo HttpClient per inviare gli HttpRequestMessage e ricevere gli HttpResponseMessage. Nel caso in cui sia presente il token viene inserito nella richiesta. Per ogni endpoint è presente un metodo async della classe Restservice che permette di fare la richiesta al server, attendere la risposta e, in caso di presenza di dati, restituire i models relativi ai dati richiesti. Anche nella richiesta vengono utilizzati dei models per passare i dati convertiti in Json, con il pacchetto Nuget Newtonsoft.Json, al server. Tali models sono delle classi C# con delle proprietà il cui nome corrisponde alla proprietà Json.

La pagina più complessa riguarda la PostsPage che presenta un elemento grafico dal nome CollectionView. Tale elemento ha la proprietà ItemsSource bindata con i Posts, di tipo PhotoInfoModel, della classe viewmodel associata a tale view.
Per poter rendere modificabile graficamente il pulsante del like e il numero di likes, nella classe PhotoInfoModel è necessario implementare l'interfaccia INotifyPropertyChanged così come per le viewmodel.

Rocco Mattia Di Mauro: scrittura delle pagine, dei file XAML, del singleton UserService e implementazione del pattern model-view-viewmodel (MVVM).
Alessandro Giuseppe Gravagno: scrittura della classe RestService, delle classi modello per la comunicazione con il server e utilizzo di Newtonsoft.Json.

**Python - Rete Neurale dell'applicativo e Client di analisi**

La parte di rete neurale è stata realizzata attraverso il linguaggio Python. Nello specifico, le librerie utilizzate a questo scopo sono state *torch* e *torchvision*: attraverso queste librerie, all'interno dell'applicativo, si è riuscito ad implementare ed utilizzare la rete neurale AlexNET. In particolare, ogni immagine ricevuta, prima di essere analizzata dalla rete, viene sottoposta ad un primo processamento che si occupa di normalizzare l'immagine in modo da renderla "gestibile" dalla rete neurale. Dopodichè, l'immagine normalizzata viene sottoposta alla rete che restituirà, leggendo dal file "imagenet_classes.txt" (file fornito dagli sviluppatori della rete), i tag (e annesse percentuali di certezza) relativi all'immagine: importante osservare come i tag che vengano restituiti debbano avere una percentuale di certezza maggiore o uguale ad una certa soglia, in modo tale da restituire all'utente, per quanto possibile, dei tag che rispecchino l'immagine da lui caricata. Restituiamo un massimo di tre tag (sia percentuale di certezza che numero massimo di tag restituiti sono definiti come parametri di configurazione, passati al tagger in fase di istanziazione della classe).

Per quanto riguarda la ricezione dell'immagine, essa viene effettuata attraverso il meccanismo delle socket; nello specifico, per ogni nuova connessione in entrata, viene lanciato un thread che si occupa di ricevere l'immagine, elaborarla e trasmettere la relativa risposta (i tag) sempre tramite socket.

Da notare come in fase di ricezione, prima di scrivere l'immagine su RAM, essa venga decodificata: quest'operazione è dovuta al fatto che il server riceve dal client l'immagine codificata in base64 e ciò viene fatto per bypassare una problematica della libreria RestBed usata per l'implementazione del server in C++. Usando, infatti, tale libreria non è possibile leggere in modo opportuno la sequenza di byte che rappresenta l'immagine; di conseguenza, si è pensato di sfruttare un meccanismo di codifica/decodifica in base64 per superare tale problematica.

Per quanto riguarda la trasmissione dei tag al server, essi vengono posti in una stringa opportunamente formattata (es. "cane - animale - marrone") che verrà trasmessa sempre tramite socket al server.

Osserviamo inoltre, che ogni volta che la rete effettua un'inferenza, i tag con le relative percentuali vengono salvati all'interno di un database MongoDB (la cui connessione viene fatta tramite un'apposita classe MongoDB, costruita a partire dalla libreria pymongo): ciò è fatto allo scopo di permettere, in un secondo momento, all'amministratore dell'applicazione di effettuare un'analisi (tramite analyzer.py) sui tag restituiti dalla rete e sulla loro precisione.

Infine, per quanto riguarda il client usato dall'amministratore dell'applicazione, esso è stato costruito utilizzando la libreria grafica Tkinter: nello specifico l'utente ha a disposizione tre pulsanti con il quale, rispettivamente, accedere al database, ottenere i dati che superano una certa soglia che viene specificata dall'utente, tramite TextBox, ed, infine, un pulsante per plottarli come bar-chart. Inoltre, all'utente, viene notificata l'esito di ognuna di queste operazioni tramite delle opportune label (di color verde se tutto è andato bene, rosso altrimenti).

Rocco Mattia Di Mauro: inizializzazione della rete neurale e comunicazione col database.
Alessandro Giuseppe Gravagno: scrittura del client di analisi e comunicazione tramite socket.

**C++ - Server**

Il server è stato realizzato usando il linguaggio C++ (classe Server). Nello specifico, la libreria utilizzata per fare ciò è RestBED: abbiamo specificato sia la porta su cui si pone in

ascolto il server (1984) che gli endpoint (con annessi metodi HTTP) a cui esso risponde. Molto utile è stata la possibilità data da questa libreria di specificare il numero di thread da utilizzare in modo da gestire più connessioni contemporaneamente (abbiamo usato la funzione std::thread::hardware_concurrency() che restituisce il massimo numero di thread concorrenti supportati dalla piattaforma).
Il server, inoltre, prima di mettersi in ascolto di nuove richiesta, crea una connessione sia col database MongoDB (modellato dalla classe MongoDB) che col bucket S3 Minio (modellato dalla classe MinIOUploader).
Per quanto riguarda gli endpoint, ne abbiamo uno per ogni funzione dell'applicazione:

- ***/signup (POST):*** endpoint usato per la registrazione di un utente nella piattaforma: egli manda, attraverso una richiesta POST, username e password con il quale vuole registrarsi e il server verifica se tali dati sono validi o no (nello specifico, controlla se è già presente un utente con lo stesso username). In caso di successo, viene restituito all'utente un token (JWT) con il quale potrà autenticare entro un certo tempo di validità tutte le azioni successive.
- ***/login (POST):*** endpoint usato per l'accesso dell'utente nella piattaforma: egli manda, tramite richiesta POST, username e password al server, che verificherà se è presente nel database un utente con queste credenziali. In caso di esito positivo, viene restituito all'utente un token con il quale potrà autenticare tutte le azioni successive.
- ***/upload (POST):*** endpoint usato per il caricamento di un nuovo post nella piattaforma: l'utente manda, attraverso una richiesta POST, l'immagine che vuole caricare con allegata una descrizione. Da notare è che l'immagine che arriva non è in formato binario bensì è stata codificata su base64: ciò è stato fatto per ovviare ad una problematica della libreria RestBED, che non permette una lettura pulita di una sequenza di byte, portando a possibili letture troncate della stessa.
  Prima che venga eseguita una qualsiasi istruzione, viene verificata la validità temporale del token trasmesso (contenuto nel campo Authorization dell'header della richiesta), recuperando dal body del token, in caso di esito positivo, l'username dell'utente che sta effettuando l'upload. Verificato ciò, l'immagine viene trasmessa attraverso il meccanismo delle socket alla rete neurale, che si occuperà di individuare i relativi tag e di trasmetterli al server sempre tramite socket. In particolare, i tag vengono restituiti sotto forma di una stringa opportunamente formattata (es. "cane - animale - marrone"), che verrà successivamente analizzata per ottenere la lista di tag da caricare sul database.
  Dopodiché l'immagine, decodificata e scritta su RAM, viene caricata sul bucket S3 Minio, in modo da essere salvata definitivamente (il nome con cui viene caricata è ottenuto calcolando l'hash della combinazione di username più immagine codificata, così da evitare possibili ambiguità di più immagini caricate dallo stesso utente o, viceversa, di una stessa immagine caricata da utenti diversi).
  Infine, viene caricato il post sul database, trasmettendo username dell'utente, la descrizione che ha fornito, la lista dei tag ottenuti e l'indirizzo del bucket dal quale ottenere l'immagine caricata.
- ***/get (GET):*** endpoint usato per l'ottenimento dei post: l'utente trasmette una richiesta GET con la quale richiede al server di restituire dei post sulla base dei parametri query che vengono inviati. Tali parametri vengono utilizzati per costruire una query con il quale specificare quali post desidera visualizzare l'utente: si possono ottenere post con uno specifico tag oppure, tutti i post che sono stati caricati dagli altri utenti

(escludendo, quindi, quelli caricati dall'utente che ha fatto la richiesta) o combinare le due cose. Il database, per ogni richiesta get, restituirà un massimo di dieci risultati. Viene, inoltre trasmesso anche un parametro skip con il quale specificare quale "pagina" del database debba essere restituita.
Anche in questo caso, ovviamente, avviene la verifica del token trasmesso prima dell'esecuzione di qualsiasi altra istruzione.

- ***/like (PUT):*** endpoint usato per l'aggiunta/rimozione del like ad un post: l'utente manda, attraverso una richiesta PUT, l'id del post ed un campo like, che indica la sua volontà di mettere (1) o rimuovere (0) il like. Tali valori, insieme allo username ottenuto dal body del token trasmesso nella richiesta (e, come sempre, verificato prima di eseguire ogni altra istruzione), vengono trasmessi al database che, sulla base del valore del campo like, inserirà o rimuoverà l'utente dalla lista likes del documento mongo del post.

La classe SocketTCP modella, come suggerisce il nome, la comunicazione attraverso le socket che il server ha con la rete neurale: vengono implementate sia le funzioni di connessione, trasmissione e ricezione dati, sollevando un'eccezione user-defined (SocketException) in caso di problemi. Il costruttore inizializza una socket, settandone la famiglia di appartenenza (AF_INET), il protocollo di trasporto (TCP) e l'host (indirizzo e porta) con cui comunicare. Il distruttore chiude la socket.

La classe MongoDB modella la connessione con tale database, specificando le operazioni di lettura e scrittura che il server potrà fare con esso. È stato utilizzato il pattern Singleton in quanto la comunicazione col database e, quindi l'istanza della classe MongoDB che la modella, viene creata in fase di avvio del server e successivamente riutilizzata per le varie operazioni: il costruttore privato crea un'istanza di mongocxx::instance, oggetto che deve 'vivere' per tutta la durata della connessione, mentre il metodo statico getInstance() restituisce o crea, qualora non lo fosse già stato, l'istanza unica della classe MongoDB.
Poi abbiamo la funzione di connessione col database (connectDB) che avvia la connessione col server mongo in ascolto sull'host e sulla porta specificati, autenticandosi con le credenziali passate e specificando il database da utilizzare (tutti questi parametri vengono passati nella funzione main, dove essi vengono letti da un file di configurazione config.ini). Questa funzione si occupa, inoltre, di inizializzare due variabili collection che rappresentano le due collezioni (users e photos) che verranno interrogate in tutte le successive operazioni.
Le funzioni di signup e login gestiscono, rispettivamente, l'inserimento dell'utente nel database e la verifica della sua presenza in esso. In particolare, qualora il valore del campo username sia già presente nella collezione (signup) oppure non venga trovato alcun documento con quel username e password, vengono sollevate, rispettivamente, le eccezioni user-defined SignupException e LoginException.
La funzione di upload si occupa del caricamento di un post nella collezione photos; per ogni documento vengono salvati username dell'utente, descrizione dell'immagine, link di minio dal quale ottenerla, l'array dei tag ottenuti e l'array, inizialmente vuoto, dei like al post. In caso di errore nel caricamento viene lanciata l'eccezione user-defined UploadException.
La funzione getImages viene usata per ottenere, sotto forma di stringa opportunamente formattata, i post che corrispondono ai parametri query passati e che si trovano nella "pagina" della collezione individuata dal parametro skip, per un massimo di dieci post per pagina.

La funzione likeImage permette di aggiungere o rimuovere, sulla base del valore del parametro booleano like, l'utente in questione (parametro username) dal campo likes del documento individuato dal parametro idImage. Ritorna true se riesce a modificare il documento, altrimenti restituirà false.

La classe MinIOUploader modella la connessione con il bucket S3 Minio: anche in questo caso viene usato il pattern Singleton, in quanto la connessione con il bucket, e le relative operazioni di inizializzazione, vengono effettuate in fase di avvio del server; per cui, è in quella fase che viene creata l'istanza che modella la connessione, la quale verrà riutilizzata (attraverso il metodo statico getInstance) per le successive operazioni. Creata l'istanza, effettuo la connessione col server su cui si trova il bucket attraverso la funzione connectToBucket, alla quale vengono passati host e port del server, le credenziali di accesso e il nome del bucket da creare e/o utilizzare. In particolare, dentro questa funzione vengono invocate le funzioni di createBucket, usata per la creazione del bucket, e setBucketPolicy, funzione usata per settare la policy del bucket in modo da poter accedere senza autorizzazione alcuna ai dati (le immagini in questo caso) in esso salvate. Nello specifico la funzione createBucket controlla, innanzitutto, se il bucket "bucketName" è già stato creato o meno, e qualora non lo fosse, lo crea.
Infine, la funzione putImage si occupa di caricare nel bucket l'immagine individuata dal parametro filename, salvandola con il nome specificato dal parametro key.
In caso di errori, nelle varie funzioni, verrà sollevata l'eccezione user-defined MinioException.

Rocco Mattia Di Mauro: scrittura endpoint /get e /like e gestione bucket minio.
Alessandro Giuseppe Gravagno: implementazione server, socket e mongodb (mongocxx).