Studente: Jacopo Maria Mengarelli

Matricola: 292728

Email: j.mengarelli@campus.uniurb.it

Corso di Programmazione e Modellazione a Oggetti

Progetto sessione invernale 2020/2021

Specifica del software

Il progetto consiste nella realizzazione di un client per la gestione di un filtro mail, sviluppato secondo l'architettura RESTful. L'API (server-side) gestisce un database locale, catalogando tutte le e-mail considerate "spam" o malicious in un file JSON.

L'API accetta richieste HTTP di tipo GET e POST, rispettivamente utilizzate per controllare se una mail (o più mail) è presente nella blacklist e per la fase di autenticazione, senza richiedere particolari requisiti.

I metodi PATCH e DELETE, utilizzati rispettivamente per aggiungere o rimuovere una mail, vengono, invece, protetti con la basic-authentication (over-https), in quanto unicamente l'amministratore dell'API ha diritto all'utilizzo.

Per una comoda gestione dell'API, si è quindi scelto di creare un client C#, che aiuti nell'ordinaria amministrazione, e che consenta di gestire i vari metodi anche da remoto, così da aggiungere e rimuovere le diverse e-mail all'interno della blacklist.

Studio del problema

La struttura generale dell'applicazione non è particolarmente complessa:

L'idea è quella di sfruttare il form di login per salvare le credenziali (e per garantire l'accesso ai metodi "privati" citati sopra). Una volta eseguito il log-in, se questo va a buon fine, sia ha la possibilità di accedere alle varie opzioni che l'applicazione offre.

La **prima criticità** viene riscontrata proprio qui, nel form di log-in: l'API gestisce più accessi concorrenziali, e diversi servizi posso avvalersi dello stesso **endpoint** per le diverse richieste. Potrebbe quindi accadere che la richiesta di log-in venga messa in coda ad altre operazioni, nel caso in cui l'API stia già gestendo altre richieste differenti, e potrebbe volerci alcuni secondi prima di verificare se le credenziali inviate siano valide o meno. Per questo motivo, la funzione di log-in, è stata implementata come **task asincrono**, che consente di non bloccare la **GUI** del programma e di non "**freezare il main thread**" per tutta la durata della verifica.

È stata inoltre implementata una **progress bar**, che viene visualizzata mentre il task invia le credenziali, quest'ultima mostra il caricamento delle stesse, e sparisce non appena il task ritorna il risultato del log-in (che questo sia positivo o meno).

Effettuato l'accesso, il client fa sparire il form di login, e richiama un form secondario denominato "MainForm". Il MF ha il compito di offrire un'interfaccia diretta ai metodi dell'API: a seconda di quale tabella viene mostrata (è necessario prendere visione della documentazione per capire cosa s'intende per tabella), si ha la possibilità di interagire con questi:

- È possibile inviare la richiesta di ricerca di una mail: il client deserializza, quindi la risposta in json dell'API e a seconda della risposta, riporta se l'e-mail è presente o meno.

- È possibile inviare una richiesta per mostrare tutta la black-list o esportarla in un file locale (**seconda criticità**). Anche in questo caso, si è optato per implementare questa funzione come **task asincrono**, in quanto la black-list potrebbe avere dimensioni particolarmente elevate (a seconda del numero di email presenti), e per gli stessi motivi del log-in form, l' "**async task**", sembrava l'implementazione più corretta per ovviare a questa criticità (windows a volte rileva il freeze dei form come deadlock dell'applicazione stessa, e incarica il sistema operativo, mediante i "demoni", di chiudere i processi "freezati"). L'API risponde con un file json, che dopo la deserializzazione, inserisce una una textbox bi-dimensionale tutte le mail presenti nel db locale dell'API (o scrive su file, in caso si scelga la funzione exporting).

- È possibile inviare una richiesta di aggiunta di una mail. Essendo che l'applicazione è accessibile SOLO agli amministratori della rete, il client consente di utilizzare anche il metodo "PATCH". Ogni richiesta di aggiunta, allega con sé le credenziali di accesso. L'API verifica se l'e-mail che si sta tentando di aggiungere non sia già presente nel db locale, e in caso di riscontro negativo, prosegue con l'aggiunta. Come per le altre richieste, anche qui, l'API risponde con un file JSON che verrà deserializzato ed interpretato, mostrando il responso positivo o meno. Per eseguire una richiesta

di tipo PATCH, è necessario creare una richiesta http con una struttura specifica (**terza criticità**). Vedremo nella sezione “Scelte architetturali” come questa è stata codificata.

- Infine, è possibile inviare una richiesta di tipo “DELETE” per rimuovere una mail dal db locale. Come per il metodo di aggiunta, anche qui, la richiesta effettuata dal client allega con sé le credenziali di accesso, e sfrutta una struttura specifica per la creazione della richiesta http (**quarta criticità**). La risposta dell’API viene quindi deserializza, mostrando il risultato a schermo.

Tutti i form sono stati creati mediante l’utilizzo del **MetroFramework**, un framework che consente di visualizzare i form in maniera più nitida ed elegante, oltre ad offrire un set di caratterizzazioni aggiuntive (come il colore della barra dei form, o il colore della progress bar…).

Per la deserializzazione dei vari JSON, ci si è affidati, invece, alla libreria **Newtonsoft.Json**, che semplifica molto il lavoro di serializzazione e deserializzazione, grazie alla sua implementazione “a template”: è sufficiente specificare la classe necessaria alla deserializzazione (o serializzazione) e questa avviene.

Un primo approccio era stato anche implementato, mediante l’utilizzo della libreria “**JsonConverter**”, tuttavia l’approccio di quest’ultima risultava particolarmente laborioso, e avrebbe interferito con il concetto di “pulizia e manutenibilità del codice”.

Di seguito una testimonianza della prima implementazione della libreria **JsonConverter**, successivamente sostituita dal **Newtonsoft.Json**

```
    internal class ParseStringConverter : JsonConverter
    {
        0 references
        public override bool CanConvert(Type t) => t == typeof(long) || t == typeof(long?);

        0 references
        public override object ReadJson(JsonReader reader, Type t, object existingValue, JsonSerializer serializer)
        {
            if (reader.TokenType == JsonToken.Null) return null;
            var value = serializer.Deserialize<string>(reader);
            long l;
            if (Int64.TryParse(value, out l))
            {
                return l;
            }
            throw new Exception("Cannot unmarshal type long");
        }

        0 references
        public override void WriteJson(JsonWriter writer, object untypedValue, JsonSerializer serializer)
        {
            if (untypedValue == null)
            {
                serializer.Serialize(writer, null);
                return;
            }
            var value = (long)untypedValue;
            serializer.Serialize(writer, value.ToString());
            return;
        }

        public static readonly ParseStringConverter Singleton = new ParseStringConverter();
    }

}
0 references
public static class ObjectExtention
{
    1 reference
    public static void CustomDump<T>(this T data)
    {
        string json = JsonConvert.SerializeObject(data, Formatting.Indented);
        Console.WriteLine("Dumping object " + typeof(T).Name + " :");
        Console.WriteLine(json);
    }
```

Scelte architetturali

Poiché si parla di scelte architetturali, è bene citare l'utilizzo di **heroku** (per l'hosting gratuito del servizio API), e il rilascio di entrambi i sistemi (Client/server) sotto licenza **MIT** (nella documentazione è possibile trovare i link alle diverse repository, che indicano la motivazione dietro la scelta di tale licenza). Il client utilizza inoltre, diverse classi per gestire le diverse operazioni interne allo stesso, di seguito un breve riassunto sulle principali:

Server.cs

La classe Server, gestisce le richieste http per interfacciarsi con l'API:

- Dichiara 6 attributi di tipo stringa readonly, che contengono tutti gli "end point" dell'api, questa classe detiene ogni url dell'api e viene invocata ogni qual volta si necessita di un particolare end point, per una particolare operazione (ed esempio il metodo dedicato all'aggiunta di una mail, farà riferimento agli end point per l'aggiunta di email contenuti in questa classe).

- Definisce due metodi "MakeRequest" anch'essi statici e pubblici. La prima definizione di MR richiede: un oggetto di tipo User (user), il verbo http da utilizzare (http_verb) e l'end point alla quale bisogna eseguire la richiesta (EndPoint). Tale metodo è possibile utilizzarlo per le richieste di tipo "PATCH", "DELETE" e "POST", e viene sfruttato per il sistema di log-in e di aggiunta/rimozione di una e-mail.
```
public static string MakeRequest(string http_verb, string end_point)
```

- La seconda definizione di "MakeRequest" esegue invece l'overload della prima dichiarazione della funzione. Questa seconda implementazione consente di eseguire unicamente richieste di tipo GET, in quanto è necessario sfruttare un'altra metodologia per rapportarsi a tali richieste. Il codice interno delle due funzioni, infatti, è completamente differente, per quanto le due funzioni si chiamino allo stesso modo, e si è deciso di sfruttare l'overload in quanto, comunque, la funzione dei due metodi è simile: inviare richieste http. Questa seconda definizione richiede come parametro, unicamente l'end point alla quale ci si vuole riferire, e imposta "automaticamente" il verbo http su GET.
```
public static string MakeRequest(string end_point)
```

BlackList.cs

La classe BlackList, si occupa di gestire la blacklist principale, e offre un set di metodi differenti per gestire quest'ultima:

- Definisce 4 metodi: AddEmail, RemoveEmail, CheckEmail, GetList utilizzati rispettivamente per aggiungere, rimuovere, controllare una mail nel db remoto o nel caso del GetList per scaricare tutto il database. Tali metodi, sfruttano le funzioni dichiarate nella classe server, per eseguire le richieste http, e ritornano un json deserializzato.

- È presente un'ulteriore classe pubblica definita come: ListResult, la quale rappresenta la blacklist. Tale classe contiene un array di oggetti di tipo "ListEntry" (un ulteriore classe dichiarata all'interno della blacklist → ListResult), e 2 metodi di tipo statico, utilizzati per serializzare e scrivere il database su file (nel caso in cui sia selezionata l'opzione di exporting su file locale).

Client.cs

La classe client, viene utilizzata per le funzionalità base del client:

- Detiene un oggetto di tipo "user", utilizzato dal login per immagazzinare informazioni e per gestire quest'ultime in base al log-in effettuato.

- Contiene un metodo statico che verifica se la stringa ricevuta come parametro è una stringa di tipo email valida.

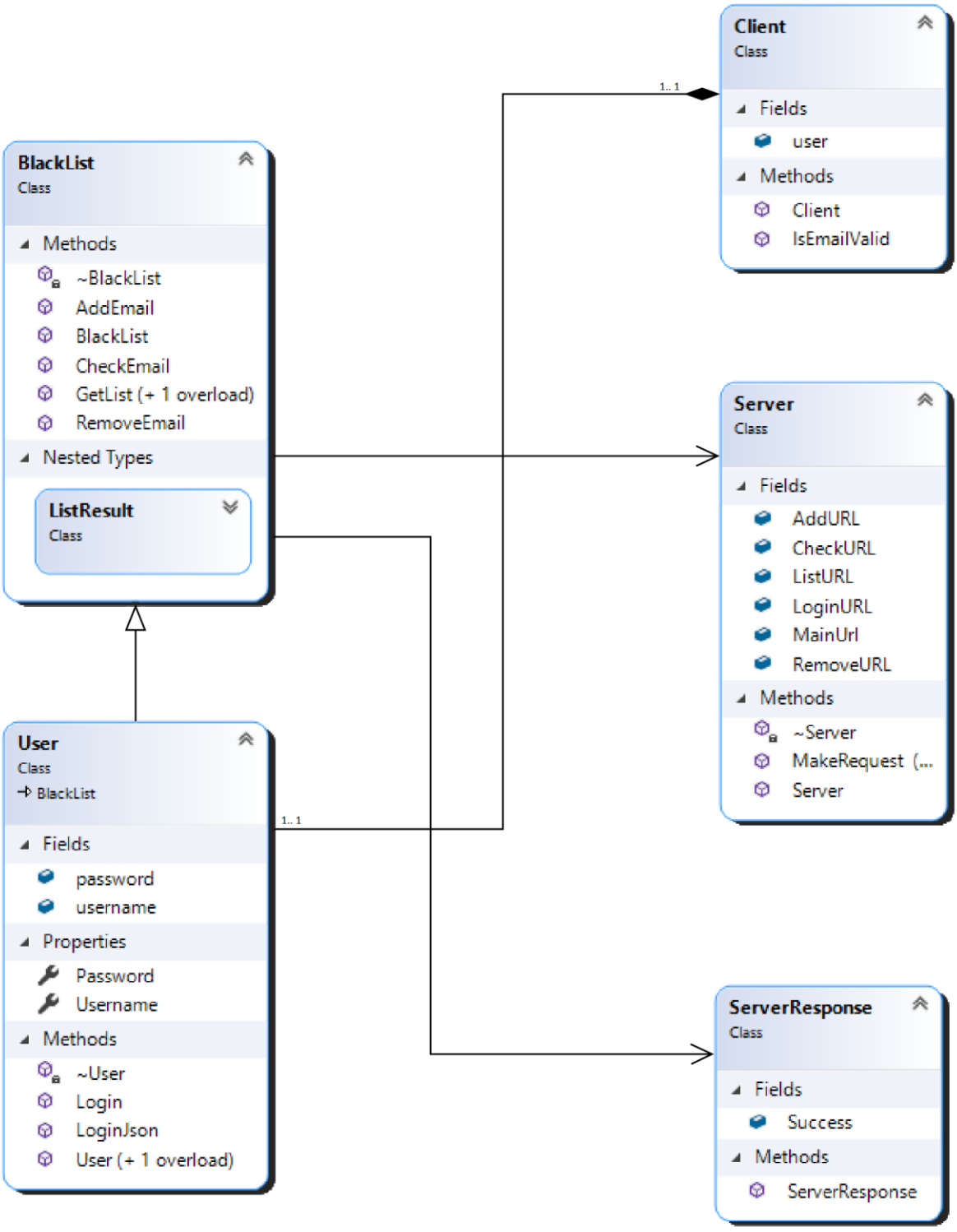
User.cs

La classe user, viene ereditata da blacklist, per un concetto di formalità sintattica, onde evitare la ripetizione della keyword "blacklist", in questo modo è possibile direttamente accedere alla blacklist, istanziando un oggetto di tipo client (che a sua volta contiene un oggetto di tipo user, che a sua volta contiene un membro di nome blacklist)

- Contiene due proprietà di tipo stringa: username e password

- Contiene un metodo "Login" utilizzato per scambiare le credenziali con il server API, e proseguire nell'autenticazione.

Di seguito un Diagramma UML delle classi, per presentare le principali relazioni fra le varie classi. Tale diagramma è stato ottenuto direttamente utilizzando il “Class Designer” direttamente fornito dall’ambiente di sviluppo (Visual Studio Community 2019), con l’aggiunta di frecce in post produzione.



Documentazione

Per la documentazione dell'API è possibile far riferimento al readme ufficiale della repository GitHub.

Non è invece presente alcuna documentazione ufficiale per quanto riguarda il client, poiché non sono richieste conoscenze approfondite per il suo utilizzo, tuttavia maggiori informazioni (con annessa una breve presentazione del client stesso) possono essere anch'esse trovate nel readme ufficiale della repository GitHub.

Di seguito invece, si possono trovare l'username e la password necessari per il login:

ID	oop
PASSWORD	oop

Un breve riassunto delle funzionalità generali dell'app di seguito:

- La tabella "**Check One Email**" è utilizzata per verificare se l'e-mail inserita è presente nella black-list locale all'API. È sufficiente inserire un indirizzo mail valido* nella textbox, e avviare il processo di ricerca premendo sul pulsante "Check", in qualche secondo l'applicazione dovrebbe rispondere se una mail è presente o meno.

- La tabella "**Check Blacklist**" è utilizzata per controllare tutto il database locale all'API. Facendo semplicemente click su "Load", il client avvia la richiesta di download dell'intera black-list, che verrà mostrata nella textbox sopra. Se fossero presenti molte e-mail, potrebbe essere necessario qualche secondo per la visualizzazione completa.

- La tabella "**Add email**" viene utilizzata per aggiungere e-mail alla black-list. È sufficiente inserire una mail valida* nella textbox, e poi premere il bottone "Add" poco sotto. Il client dovrebbe mostrare poco dopo il successo o meno dell'aggiunta. (il server è protetto contro l'aggiunta multipla della stessa mail: se una mail è già presente nel db locale all'API, lo stesso non consentirà la ri-aggiunta della mail duplicata, ma bensì mostrerà una label che indica che l'e-mail è già presente).

- La tabella "**Remove email**" è dedita alla rimozione di una mail dalla black-list. È sufficiente inserire la mail che si vuole eliminare nella textbox, e premere il pulsante "remove". Il client dovrebbe mostrare una label di successo o meno poco dopo. (la rimozione di una mail può ovviamente fallire se quest'ultima non è presente nel db locale dell'API).

- La tabella "**Settings**" è dedicata alle informazioni generali.
  *valida = le email inserite nelle textbox vengono controllate con la libreria **System.Net.Mail.MailAddress**, che si occupa di verificare se il campo contiene un indirizzo email nel formato valido.

Use Cases

Nello scenario plausibile, sono tre gli "attori" che posso utilizzare l'API:

- **Amministratore**: il quale ha accesso a tutti i servizi dell'API, mediante l'utilizzo del client.
- **Servizio mail**: il quale per ogni mail ricevuta, verifica che il mittente non sia presente nella blacklist.
- **Utente**: il quale può utilizzare richieste dirette all'API per controllare una mail specifica o l'intera blacklist. (Attualmente non è presente alcun client per questa categoria di utenti, per quanto l'endpoint dell'API risponda a qualsiasi richiesta GET senza richiedere particolari parametri. L'API risponde con "raw data", cioè un file json di tipo grezzo).

Uno schema dei servizi è presentato nel diagramma UML seguente: