Messagyre

TM de Pietro Gravina

April 29, 2025

1 Introduction

1.1 Pourquoi ce sujet

J'ai choisi de réaliser ce projet car après plus d'un an au Gymnase de Renens, j'ai remarqué l'absence d'un moyen de communication plus informel, malgré la disponibilité de toute la suite Microsoft 365 et d'autres outils. Les e-mails sont trop formels et peu pratiques, tandis que le chat d'Outlook est peu connu et peu intuitif, tout comme celui de Teams.

Pour résoudre ce problème, j'ai conçu une application de messagerie dédiée exclusivement aux communications internes du gymnase : Messagyre. Son principe est similaire à celui de WhatsApp, à la différence que, plutôt que de nécessiter le numéro de téléphone du destinataire, Messagyre permet de communiquer simplement en connaissant son prénom (et, si nécessaire, son nom de famille), facilitant ainsi les échanges entre étudiants et enseignants.

Je considère ce projet particulièrement intéressant et ambitieux. Bien qu'une application de chat puisse sembler simple à développer au premier abord, il s'agit en réalité d'un projet complexe nécessitant des compétences avancées, tant en programmation qu'en design.

Bilan de vos connaissances, codebases avec lesquelles vous étiez familier.

Avant de commencer ce projet, je possédais déjà des connaissances avancées en programmation. Je programme depuis l'âge de huit ou neuf ans, ayant appris en autodidacte plusieurs langages, notamment Python, Lua, Java, JavaScript et C#.

Mes premières expériences en communication client-serveur consistaient en de simples échanges de données entre des pages HTML avec des scripts en JavaScript et un serveur basé sur Node.js. Cependant, afin de tester la faisabilité de ce projet et de proposer l'idée officiellement à la direction, j'ai développé un premier prototype en utilisant Unity comme client et un serveur en Node.js (JavaScript). Bien que le système fonctionnait, la principale difficulté était d'optimiser la communication entre ces deux langages, très différents dans leur approche.

Après quelques recherches, j'ai découvert qu'il était possible de développer le serveur directement en C#, en utilisant l'API ASP.NET de Microsoft. J'ai donc entièrement réécrit l'application sur Unity ainsi que le serveur, améliorant donc la stabilité et l'intégration du système. Pour le développement, j'ai utilisé Visual Studio 2022,

l'environnement de développement de Microsoft, pour le client ainsi que pour le serveur.

Grâce à cette expérience, j'ai pu consolider et approfondir mes compétences en programmation réseau, en gestion de serveurs et d'architecture logicielle, des aspects essentiels pour la réalisation d'une application de messagerie comme Messagyre.

1.2 Texte de ce qui a été fait dans un ordre chronologique. Bien -> on doit comprendre pourquoi vous avez obtenu ce produit final

Server:

1. Creazione del progetto del server dal modello "ASP.NET Core Web App" di Visual Studio 2022, rinominato come "MessagyreServer". Tutti gli asset necessari sono stati creati dallo stesso programma, col quale ho scritto tutto il codice di questo progetto.
2. Creazione della prima e più importante parte del codice del server: la gestione delle richieste. Le richieste al server vengono trasmesse tramite protocollo WebSocket, poi decifrate grazie alla libreria .NET C# System.Net.WebSockets di Microsoft. Gli script che si occupano di questo sono Program.cs (l'entry point del programma), Server.cs.
3. Creazione dello script Authenticator.cs, contenente la classe Authenticator, che gestisce come suggerisce il nome l'autenticazione delle connessioni.

Client:

1. Creazione del progetto del front end su Unity (2021.3.17f1), rinominato come "MessagyreClient".
2. Creazione di una classe MonoBehaviour "Router", che gestisce la comunicazione col server: la connessione, la disconnessione, gli errori, l'invio di richieste e la ricezione di messaggi.
3. Creazione di una classe

- *Tutoriel / Procédure d'installation*
- *Comment fonction la communication entre client et serveur ?*
- *Structure de la codebase*

2 Structure du serveur

Ogni componente del server è spiegato nelle sottosezioni a venire, che mostreranno un *"Extrait du code"*, cioè un illustrazione schematica del codice del componente. E possibile visitare il repository di GitHub del progetto del server per accedere al codice completo. Per comprendere facilmente il funzionamento dei vari componenti del server è disponibile anche un Diagramme UML.

2.1 User.cs

Quando il server riceve una richiesta, è spesso necessario poter riconoscere, distinguere e catalogare il mittente: questo viene rappresentato dalla classe `User`, una classe non statica (quindi istanziabile) contenente tutte le informazioni utili riguardanti l'utente connesso.

2.1.1 Extrait du code

```
using System.Net.WebSockets;

namespace MessagyreServer.Classes
{
    public class User
    {
        public Account? Account { get; private set; }
        public WebSocket Socket { get; }
        public bool IsAuthorized { get; private set; }
        public string RegistrationEmailAddress = "";
        public string RegistrationTempEmailAddress = "";
        public int RegistrationVerificationCode;
        public DateTime RegistrationCodeSentAt;

        public User(WebSocket Socket)
        {
            this.Socket = Socket;
        }

        public void Receive(Signal SignalToSend)
        {
            Socket.Send(SignalToSend.Pack());
        }

        public void Authenticate(Account? GivenAccount)
        {
            Account = GivenAccount;
            IsAuthorized = GivenAccount != null;
        }
    }
}
```

2.1.2 Variables principales

Une instance de la classe `User` contient trois informations principales : le compte (`Account`) lié à l'utilisateur (si authentifié), le `Socket` du client connecté (nécessaire pour pouvoir lui envoyer des messages), et un booléen

`IsAuthorized` qui indique si l'utilisateur s'est connecté à son compte ou non.

2.1.3 Methodes

Le premier "méthode" de ce script est un `Constructor`, il est appelé lorsque la classe est instanciée et stocke le `Socket` en mémoire.

Le second est uniquement pour la lisibilité, lorsqu'il est appelé, il envoie le message `Signal` au client.

Le dernier (`Authenticate()`), est appelé par la classe `Authenticator` lorsque l'utilisateur se connecte à son compte.

2.2 Signal.cs

Une autre classe non statique qui contient toutes les informations d'une requête/message. Elle est instanciée lorsque le serveur reçoit une requête ou lorsqu'il doit envoyer un message à un client spécifique.

2.2.1 Extrait du code

```
using Newtonsoft.Json;

namespace MessagyreServer.Classes
{
        public class Signal
        {
                [JsonIgnore] public User? Sender;

                public SignalType Type;
                public Dictionary<string, string> Data = new();

                public Signal(SignalType Type, User? Sender = null)
                {
                        this.Type = Type;
                        this.Sender = Sender;
                }

                public string Pack()
                {
                        return JsonConvert.SerializeObject(this);
                }
```

```
                public static Signal? Unpack(string Source, User Sender)
                {
                        Signal? Result;

                        try { Result = JsonConvert.DeserializeObject<Signal>(Source); }
                        catch { return null; }

                        if (Result != null) Result.Sender = Sender;

                        return Result;
                }

        }

        public enum SignalType
        {
                Login,
                Registration,
                Logout,
                Message,
                Search
        }
}
```

2.2.2 Variables principales

Les informations contenues dans cette classe sont réparties en trois variables :

1. `Sender` Indique l'utilisateur à partir duquel le serveur a reçu le message. `[JsonIgnore]` indique au Serializer de Newtonsoft.Json que la variable ne doit pas être incluse dans le contenu du Json lorsque l'instance de la classe sera stockée en mémoire non volatile. (voir Messages.cs)

2. `Type` Indique de quel type de `Signal` il s'agit, les options possibles se trouvent dans l' `enum` `SignalType` en bas du script.

3. `Data` Contient un "dictionnaire" (une liste où chaque élément a un nom) avec toutes les données du `Signal` , par exemple `Username` et `Password` dans le cas d'un `Signal` de `Type` `SignalType.Login` .

2.2.3 Methodes

La méthode `Pack` "emballe" le `Signal` , c'est-à-dire qu'elle convertit les trois variables en un objet *Json* via la méthode `SerializeObject` de `JsonConvert` , provenant de la bibliothèque `Newtonsoft.Json` . Cela est nécessaire pour pouvoir l'envoyer au client, car il n'est pas possible d'envoyer des instances de classes via *WebSocket*, mais uniquement des chaînes de caractères.

`Unpack` fait exactement l'inverse, en prenant une chaîne contenant l'objet *Json*, elle la "convertit" en une nouvelle instance de `Signal` . Cette action est entourée d'un `try catch` car si un client devait envoyer une requête avec des données corrompues, mal formatées ou s'il y avait un problème lors de la transmission, le *Json* ne serait pas valide et le serveur planterait.

2.3 Account.cs

Dernière classe non statique qui représente le compte d'un utilisateur.

2.3.1 Extrait du code

```
    using Newtonsoft.Json;

namespace MessagyreServer.Classes
{
    public class Account
    {
        public string Username { get; set; }
        public string Password { get; private set; }
        public string EmailAddress { get; set; }
        public DateTime CreationDate { get; }
        public DateTime? LastLogin { get; set; }
        public bool Banned { get; set; }

        public List<Signal> Inbox { get; set; } = new();


        // Constructors
        public Account(string username, string password, string emailAddress)
        {
            Username = username;
            Password = Hash(password);
            EmailAddress = emailAddress;
            CreationDate = DateTime.UtcNow;
        }

        [JsonConstructor]
        public Account(string username, string password, string emailAddress, DateTime creationDate, DateTime
        {
            Username = username;
            Password = password;
            EmailAddress = emailAddress;
```

```
            CreationDate = creationDate;
            LastLogin = lastLogin;
            Banned = banned;
        }



        // Private methods
        private static string Hash(string Password) => BCrypt.Net.BCrypt.HashPassword(Password);



        // Public methods
        public bool TryPassword(string Attempt) => BCrypt.Net.BCrypt.Verify(Attempt, Password);

        public static string GetUsernameFromEmail(string Address) => Address.Split('@')[0];

    }
}
```

2.3.2 Variables

Cette classe contient toutes les informations d'un compte :

1. `Username` : Nom d'utilisateur déduit de l'adresse email de l'utilisateur. (*"prénom.nom"* de l'adresse *"prénom.nom@eduvaud.ch"*)
2. `Password` : Mot de passe haché (chiffré avec l'algorithme *bcrypt*) utilisant `HashPassword` de la bibliothèque *BCrypt.Net*.
3. `EmailAddress` : Adresse email saisie par l'utilisateur lors de la création de son compte. Sauf exception, le domaine doit obligatoirement être *"@eduvaud.ch"*.
4. `CreationDate` : Date de création du compte.
5. `LastLogin` : Date de la dernière connexion à la plateforme.
6. `Banned` : Valeur booléenne indiquant si l'utilisateur peut accéder à l'application. Si *'true'*, la connexion est refusée.

2.3.3 Methodes

Le premier constructeur est appelé par l' `Authenticator` lorsque le compte est créé, tandis que le second est appelé par l' `AccountsManager` lorsqu'il est chargé en mémoire depuis la base de données.

La méthode `Hash()` est uniquement pour la lisibilité, elle retourne le mot de passe fourni après l'avoir chiffré. `TryPassword()` retourne si le mot de passe fourni comme paramètre `Attempt` est correct. La méthode `GetUsernameFromEma` extrait le nom d'utilisateur de l'adresse e-mail.

2.4 Program.cs (Entry point)

Point d'entrée du programme, démarrage de l'écoute, gestion et routage des requêtes vers les autres classes.

2.4.1 Démarrage du serveur

```
using System.Net.WebSockets;

WebApplicationBuilder Builder = WebApplication.CreateBuilder(args);
WebApplication App = Builder.Build();

[…]

App.UseWebSockets();
App.Use(RequestsHandling);
App.Run();
```

Dans le code ci-dessus, l'écoute des requêtes est lancée. Lorsqu'une requête est reçue, la méthode `RequestHandling` est appelée et les données de la requête sont transmises en tant qu'arguments de la fonction, comme le `Context` et `Next` .

2.4.2 Gestion des requêtes

```
async Task RequestsHandling(HttpContext Context, Func<Task> Next)
{
    if (Context.WebSockets.IsWebSocketRequest) await OnConnection(Context);
    else await Next();
}
```

`Context` est une instance de la classe `HttpContext` contenant toutes les informations du message *Http* reçu, y compris la propriété `IsWebSocketRequest` , qui détermine (intuitivement) si la requête reçue est une requête *WebSocket*. S'il s'agit d'un autre type de requête, alors celle-ci est ignorée et la requête suivante (`Next`) est exécutée.

2.4.3 Gestion des connexions

Une fois la requête reçue et déterminé qu'il s'agit d'une requête *WebSocket*, la fonction `OnConnection()` suivante est appelée :

```
async Task OnConnection(HttpContext HttpRequest)
{
    WebSocket Socket = await HttpRequest.WebSockets.AcceptWebSocketAsync();
    User User = Server.OnConnection(Socket);

    WebSocketReceiveResult? Result = null;
    byte[] Buffer = new byte[1024 * 4];

    do
    {
        try
        {
            Result = await Socket.ReceiveAsync(new ArraySegment<byte>(Buffer), CancellationToken.None);

            string Message = Encoding.UTF8.GetString(Buffer, 0, Result.Count);

            Server.OnSignal(User, Message);
        }
        catch (Exception Ex)
        {
            Log($"Error while receiving: {Ex.Message}");
            break;
        }
    }
    while (!Result.CloseStatus.HasValue && Running);


    Server.OnDisconnection(User);

    if (Socket.State != WebSocketState.Open &&
        Socket.State != WebSocketState.CloseSent &&
        Socket.State != WebSocketState.CloseReceived)
        return;

    await Socket.CloseAsync(WebSocketCloseStatus.NormalClosure, Result?.CloseStatusDescription, CancellationT
}
```

Cette fonction se charge d'accepter les connexions *WebSocket* des clients : Une fois la connexion acceptée et établie, le client est enregistré dans une nouvelle instance de la classe `User`, qui est ensuite ajoutée à une liste des utilisateurs connectés.

Ensuite, une boucle est ouverte et se répète tant que la connexion n'est pas fermée, durant laquelle une requête de l'utilisateur connecté est attendue, puis elle est stockée dans le `Buffer` et envoyée à la classe `Server`.

Pour éviter que d'éventuelles erreurs n'interrompent l'exécution du serveur, l'écoute est entourée d'un `try catch`. À la fin de la boucle, c'est-à-dire lorsque la connexion est fermée, la fonction `OnDisconnect()` de la classe `Server` est appelée pour gérer la déconnexion du client.

2.5 Server.cs

Gestion des utilisateurs connectés, du routage des messages vers les autres classes du serveur et des déconnexions.

```
namespace MessagyreServer
{
        public static class Server
        {
                public static List<User> ConnectedUsers = new();

                public static User OnConnection(WebSocket Socket)
                {
                        User NewUser = new(Socket);
                        ConnectedUsers.Add(NewUser);

                        return NewUser;
                }

                public static void OnSignal() { ... }

                public static void OnDisconnection(User DisconnectedUser)
                {
                        ConnectedUsers.Remove(DisconnectedUser);
                }
        }
}
```

Comme mentionné précédemment, la méthode `OnConnection()` est appelée par `Program` et se charge essentiellement d'instancier la classe `User` pour ensuite l'ajouter à la liste des utilisateurs connectés (`ConnectedUsers`).

La méthode `OnDisconnection()` est quant à elle appelée lorsque l'utilisateur se déconnecte, le retirant de la liste.

La méthode principale de cette classe est `OnSignal()` :

```
public static void OnSignal(User Sender, string JsonSignal)
{
        Signal? ReceivedSignal = Signal.Unpack(JsonSignal, Sender);
        if (ReceivedSignal == null || ReceivedSignal.Sender == null) return;

        // Routing
        switch (ReceivedSignal.Type)
        {
```

```
                case SignalType.Login:
                        Authenticator.OnLoginSignal(ReceivedSignal);
                        break;

                case SignalType.Registration:
                        Authenticator.OnRegistrationSignal(ReceivedSignal);
                        break;

                case SignalType.Logout:
                        Authenticator.OnLogoutSignal(ReceivedSignal);
                        break;

                case SignalType.Message:
                        Messages.OnMessageSignal(ReceivedSignal);
                        break;

                case SignalType.Search:
                        AccountsManager.OnSearchSignal(ReceivedSignal);
                        break;

        }
}
```

Cette méthode reçoit comme arguments une valeur `Sender`, c'est-à-dire l'expéditeur de la requête, et un `JsonSignal` contenant effectivement le contenu de la requête sous forme de *Json*.

Grâce à ces deux valeurs, une instance de la classe `Signal` est créée, puis elle est dirigée vers la classe appropriée, comme la classe `Authenticator` pour les demandes de connexion ou la classe `Messages` pour les demandes de messagerie.

2.6 Authenticator.cs

Il s'occupe de la gestion des accès à la plateforme et de la création des comptes.

```
using MailKit.Security;
using MessagyreServer.Classes;
using MimeKit;
using MailKit.Net.Smtp;

namespace MessagyreServer
{
        public class Authenticator
        {
                public static void OnLoginSignal()
```

```
                public static void OnRegistrationSignal()
        }
}
```

Les types de `Signal` gérés ici sont au nombre de deux :

Ceux de *login* et ceux de *inscription*. Il s'agit de fonctions très longues et une partie du code a été omise ; un lien vers le dépôt *GitHub* est disponible au début du chapitre *"Structure du serveur"*.

2.6.1 Login: Accès à un compte existant

Voici la méthode qui s'occupe de la gestion des `SignalType.Login` :

```
public static void OnLoginSignal(Signal LoginSignal)
{
    void RespondWith(string Message, string Field = "")
    {
        Signal ResponseSignal = new(SignalType.Login);
        ResponseSignal.Data.Add("Response", Message);
        ResponseSignal.Data.Add("Field", Field);
        LoginSignal.Sender?.Receive(ResponseSignal);
    }

    // Check if all data is there
    if (LoginSignal.Sender == null) return;
    if (!LoginSignal.Data.TryGetValue("Username", out string? Username)) return;
    if (!LoginSignal.Data.TryGetValue("Password", out string? Password)) return;

    // Check if the account exists
    if (!AccountsManager.GetAccount(Username, out Account? Account))
    {
        RespondWith("account_not_found", "username");
        return;
    }
    if (Account == null) return;

    // Check the password
    if (!Account.TryPassword(Password))
    {
        RespondWith("wrong_password", "password");
        return;
    }

    // Check if banned
    if (Account.Banned)
    {
```

```
        RespondWith("banned", "username");
        return;
    }

    // Complete the login
    LoginSignal.Sender.Authenticate(Account);
    RespondWith("success");

    // Sending the inbox content
    foreach (Signal InboxMessage in Account.Inbox) LoginSignal.Sender.Receive(InboxMessage);
}
```

Dans cette méthode, une instance de `Signal` est fournie en paramètre avec toutes les informations nécessaires à la connexion, le contenu du dictionnaire `Content` sera:

```
Dictionary<string, string> Content = {
        "Username" : "prénom.nom",
        "Password" : "$2a$12$A/4hYn5TE74CXQm5By0g7Ohx..."
}
```

La méthode `RespondWith()` est appelée pour envoyer une réponse au client : dans le cas où il y aurait un problème avec les données fournies pour l'accès, la méthode serait appelée avec une chaîne courte en *snake_case* indiquant le problème, ainsi qu'une autre chaîne représentant la donnée incorrecte. S'il n'y a aucun problème, la méthode est appelée avec la chaîne `"success"`.

La première chose vérifiée à la réception du signal est la présence des données : si l'expéditeur ou l'une des deux données est nulle, la requête est annulée.

Ensuite, on vérifie s'il existe effectivement un compte correspondant au nom d'utilisateur donné, si le mot de passe correspond, et si la personne n'est pas bannie de la plateforme.

Si ces contrôles sont concluants, la procédure de connexion à la plateforme est finalisée, donnant accès au client, et tous les éventuels messages envoyés à l'utilisateur pendant son absence sont transmis.

2.6.2 Registration: Creation d'un nouveau compte

En ce qui concerne les signaux de *inscription*, la gestion de ces requêtes est nettement plus complexe : lorsque le client envoie son adresse e-mail au serveur pour créer un nouveau compte, il est nécessaire de vérifier qu'il s'agit bien du véritable propriétaire de l'adresse, afin d'éviter la création de comptes au nom d'une autre personne. Ce contrôle supplémentaire rend le processus d'authentification beaucoup plus long et complexe.

```
public static void OnRegistrationSignal(Signal RegistrationSignal)
{
    var Sender = RegistrationSignal.Sender;
    var Data = RegistrationSignal.Data;
```

```
if (Sender == null || Data == null) return;


void RespondWith(string Message, string Field = "") { ...}

void SendVerificationEmail(string TargetAddress, int Code)
{
    // Configurations
    string SmtpServer = "smtp.gmail.com";
    int SmtpPort = 587;
    string SenderEmail = "messagyre@gmail.com";
    string SenderPassword = "ywgm otfr qgam pbmz";
    string EmailContentPath = Path.Combine(AppContext.BaseDirectory, "Assets", "VerificationCodeEmail.html

    // Creating the email
    var Email = new MimeMessage();
    Email.From.Add(new MailboxAddress("Messagyre", SenderEmail));
    Email.To.Add(new MailboxAddress("", TargetAddress));
    Email.Subject = "Verification Code";

    string EmailContent = string.Empty;
    if (File.Exists(EmailContentPath)) EmailContent = File.ReadAllText(EmailContentPath).Replace("{{CODE}}

    Email.Body = new TextPart("html") { Text = EmailContent };


    // Connecting to the SMTP server and sending the email
    using var TempClient = new SmtpClient();

    TempClient.Connect(SmtpServer, SmtpPort, SecureSocketOptions.StartTls);
    TempClient.Authenticate(SenderEmail, SenderPassword);
    TempClient.Send(Email);
    TempClient.Disconnect(true);

}



/* User sent e-mail address */
if (Data.TryGetValue("email_address", out string? Address))
{
    // Checking if the given address is valid
    if (!Address.Contains('@') || !Address.Contains('.'))
    {
        RespondWith("wrong_format", "email_address");
        return;
    }
```

```
    if (!Address.EndsWith("@eduvaud.ch"))
    {
        RespondWith("wrong_domain", "email_address");
        return;
    }
    if (AccountsManager.GetAccount(Account.GetUsernameFromEmail(Address), out var _))
    {
        RespondWith("already_exists", "email_address");
    }
    if ((DateTime.UtcNow - Sender.RegistrationCodeSentAt).TotalMinutes < 2)
    {
        RespondWith("wait", "email_address");
    }

    // Creating the verification code and storing it for the next step
    int VerificationCode = new Random().Next(100000, 1000000);
    Sender.RegistrationVerificationCode = VerificationCode;
    Sender.RegistrationTempEmailAddress = Address;
    Sender.RegistrationCodeSentAt = DateTime.UtcNow;

    // Proceeding
    RespondWith("success", "email_address");
    SendVerificationEmail(Address, VerificationCode);
    Log($"VerificationCode sent to {Address}: {VerificationCode}");
}

/* User sent verification code */
else if (Data.TryGetValue("verification_code", out string? Code))
{
    if (Code.Length != 6)
    {
        RespondWith("wrong_length", "verification_code");
        return;
    }
    if (Code != Sender.RegistrationVerificationCode.ToString())
    {
        RespondWith("wrong", "verification_code");
        return;
    }

    Sender.RegistrationEmailAddress = Sender.RegistrationTempEmailAddress;

    RespondWith("success", "verification_code");
}

/* User sent password */
else if (Data.TryGetValue("password", out string? Password))
```

```
    {
        if (Password.Length < 8)
        {
            RespondWith("too_short", "password");
            return;
        }

        string EmailAddress = Sender.RegistrationEmailAddress;
        string Username = Account.GetUsernameFromEmail(EmailAddress);

        Account NewAccount = new(Username, Password, EmailAddress);
        AccountsManager.AddAccount(NewAccount);
        Sender.Authenticate(NewAccount);

        RespondWith("success", "password");

        Log($"{Username} connected", true);
    }
}
```

Cette fonction est divisée en trois sections principales :

1. La gestion de l'adresse e-mail.

   - On vérifie d'abord si la valeur est effectivement fournie et n'est pas nulle ;
   - L'adresse doit contenir au moins un point (".") et une arobase ("@") ;
   - Le domaine de l'adresse doit être "@eduvaud.ch" ;
   - Il ne doit pas déjà exister un compte associé à cette adresse ;
   - L'utilisateur ne doit pas avoir déjà tenté de créer un compte moins de deux minutes avant la tentative actuelle.

Si toutes les conditions sont remplies, un e-mail est envoyé à l'adresse donnée, depuis l'adresse *"messagyre@gmail.com"* créée exclusivement pour le développement de cette application, à laquelle le programme accède automatiquement grâce aux identifiants écrits dans le fichier (`SmtpPort` , `SenderEmail` , `SenderPassword`). Le contenu de l'e-mail est chargé depuis le fichier *HTML* `VerificationCodeEmail.html` , situé dans le dossier *Assets* du répertoire du serveur. Pour envoyer l'e-mail, un client temporaire doit être créé pour se connecter au serveur *SMTP* (Simple Mail Transfer Protocol, serveur de gestion des e-mails).

2. La gestion du code de vérification.

   - Le code ne doit pas être plus court ou plus long que six chiffres ;
   - Le code doit correspondre à celui envoyé.

Si le code correspond, alors l'adresse est vérifiée et l'on peut passer à la dernière étape de l'inscription.

3. La création du mot de passe.

   - Il doit avoir une longueur minimale de 8 caractères.

Si le mot de passe respecte cette condition, le compte est créé avec succès, et l'utilisateur accède à l'application.

2.7 AccountsManager.cs

Si occupa della gestione degli account, del loro salvataggio in memoria, ma anche di fornire i risultati quando un utente cerca un altro account.

2.7.1 Extrait du code

```
using MySql.Data.MySqlClient;
using MessagyreServer.Classes;
using Newtonsoft.Json;
using System.Data;

namespace MessagyreServer
{
        public static class AccountsManager
        {
                private static readonly string ConnectionString ="Server=metro.proxy.rlwy.net;Port=17551;Datal

                public static bool GetAccount(string Username, out Account? Result) {...}

                public static void AddAccount(Account NewAccount) {...}

                public static void DeleteAccount(string Username) {...}

                public static List<string> SearchAccount(string Query) {...}

                public static void OnSearchSignal(Signal SearchSignal)
                {
                        if (SearchSignal.Sender == null) return;
                        if (!SearchSignal.Data.TryGetValue("Query", out string? Query)) return;

                        Signal ResponseSignal = new(SignalType.Search);
                        ResponseSignal.Data.Add("Result", JsonConvert.SerializeObject(SearchAccount(Query)));
```

```
                        SearchSignal.Sender?.Receive(ResponseSignal);
                }
        }
}
```

2.7.2 Variables

In questo codice l'unica variabile globale presente è `ConnectionString`. Si tratta di una stringa di testo contenente tutte le informazioni necessarie per la connessione al server SQL. In questo caso la `ConnectionString` permette al programma di connettersi al database *MySQL* in esecuzione sempre sulla piattaforma *Railway.app*. Le informazioni contenute nella stringa sono

1. `Server`: indica l'indirizzo del database SQL. Può essere un *IP*, un nome o *"localhost"* se è in esecuzione in locale. Per evitare spese inutili, per testare le funzionalità del server durante lo sviluppo, ho usato quest'ultima opzione.
2. `Port`: indica la porta sulla quale il server ascolta le richieste SQL.
3. `Database`: è il nome del database al quale il server si deve collegare.
4. `User` e `Password`: le credenziali necessarie per accedere al database e eventualmente modificarne proprietà e contenuti.

2.7.3 Obtention d'un compte

Questo metodo qui sotto viene chiamato principalmente dall `Authenticator` per ottenere un account oppure per verificarne l'esistenza. Perché funzioni, è necessario fornire un parametro `Username`, cioè il nome utente dell'account in questione; `Result` non è un parametro, come dimostrato dalla *keyword* `out`, bensì un modificatore di parametro, l'ho utilizzato in questo caso per poter restituire due valori da un singolo metodo, cosa non possibile cambiandone solo il tipo. Infatti questo metodo restituisce un valore booleano indicante l'esistenza dell'account ricercato, mentre `Result` ottiene come valore il "risultato" dell'operazione, ovvero l'account trovato.

```
public static bool GetAccount(string Username, out Account? Result)
{
        string Query = "SELECT
                Username,
                Password,
                EmailAddress,
                CreationDate,
                LastLogin,
                Banned
                FROM Account WHERE Username = @Username";
        Result = null;

        // Connecting to the SQL server
        MySqlConnection Connection = new(ConnectionString);
        Connection.Open();
```

```
        // Creating the command
        MySqlCommand Command = new(Query, Connection);
        Command.Parameters.AddWithValue("@Username", Username);

        // Sending the command and reading the response
        var Reader = Command.ExecuteReader();
        if (!Reader.Read()) return false; // No result

        Result = new Account(
        Reader.GetString("Username"),
        Reader.GetString("Password"),
        Reader.GetString("EmailAddress"),
        Reader.GetDateTime("CreationDate"),
        Reader.IsDBNull("LastLogin") ? null : Reader.GetDateTime("LastLogin"),
        Reader.GetBoolean("Banned")
        );

        return true;
}
```

Le azioni eseguite in questo metodo sono tipiche di una richiesta SQL con risultato:

- 1. Creazione della `Query`, cioè la richiesta da inviare al database;
- 2. Connessione al database;
- 3. Creazione di un istanza di `Command`, necessaria per la gestione della comunicazione col server SQL;
- 4. Invio del comando e l'attesa della risposta, entrambi gestiti da `Command.ExecuteReader()`.
- 5. Lettura della risposta del server, `Reader.Read()` restituisce *true* se il server ha risposto con almeno una riga di testo, *false* in caso contrario.
- 6. Instanziamento della classe `Account`, contenente le informazioni ottenute.
- 7. Restituzione di un valore `true`, indicante il successo dell'operazione.

Il comportamento del codice qui sopra è anche spiegato in inglese nei commenti (*"Connessione al server SQL"*, *"Creazione del comando"*, *"Invio del comando e attesa della risposta"*).

2.7.4 Creation d'un compte

Questo metodo invece viene chiamato quando è necessario creare un nuovo account. L'`Authenticator` ne fa uso quando un utente completa l'ultima tappa della *registration*. Per l'esecuzione di questo metodo è richiesto un solo parametro, un'istanza della classe `Account` contenente tutte le informazioni dell'account da aggiungere al database.

```
public static void AddAccount(Account NewAccount)
```

```
{
        if (GetAccount(NewAccount.Username, out var _)) return;

        string Query = "INSERT INTO Account (Username, Password, EmailAddress, CreationDate, LastLogin, Banned

        MySqlConnection Connection = new(ConnectionString);
        Connection.Open();
        MySqlCommand Command = new(Query, Connection);

        Command.Parameters.AddWithValue("@Username", NewAccount.Username);
        Command.Parameters.AddWithValue("@Password", NewAccount.Password);
        Command.Parameters.AddWithValue("@EmailAddress", NewAccount.EmailAddress);
        Command.Parameters.AddWithValue("@CreationDate", NewAccount.CreationDate);
        Command.Parameters.AddWithValue("@LastLogin", (object?) NewAccount.LastLogin ?? DBNull.Value);
        Command.Parameters.AddWithValue("@Banned", NewAccount.Banned);
        Command.ExecuteNonQuery();
}
```

La prima cosa controllata all'esecuzione del metodo, è l'esistenza di un altro account con lo stesso nome utente nel database. Questo controllo viene effettuato col fine di evitare conflitti con altri account omonimi. Il resto delle azioni sono simili al metodo precedente, con la differenza che questo non invia una richiesta con risultato, bensi un semplice comando. Invece di attendere una risposta dal server, come visto precedentemente con `Command.ExecuteReader()`, il metodo chiamato è `Command.ExecuteNonQuery()`, che invia unicamente il comando.

2.7.5 Suppression d'un compte

Metodo chiamato per eliminare un account. Anche in questo caso è richiesto un solo parametro, una stringa `Username` rappresentante il nome utente assegnato all'account da eliminare.

```
public static void DeleteAccount(string Username)
{
        string Query = "DELETE FROM Account WHERE Username = @Username";

        MySqlConnection Connection = new(ConnectionString);
        Connection.Open();
        MySqlCommand Command = new(Query, Connection);

        Command.Parameters.AddWithValue("@Username", Username);
        Command.ExecuteNonQuery();
}
```

La struttura è la medesima del metodo precedente, trattandosi di un comando senza risposta.

2.7.6 Recherche d'un compte

Metodo chiamato quando un utente, dalla barra di ricerca nella pagina *"Conversations"* dell'applicazione Messagyre, inserisce almeno due caratteri per cercare un altro utente. Questo metodo richiede come parametro la stringa immessa dall'utente nella sua barra di ricerca, da notare che questa stringa potrebbe non essere il nome utente completo, bensi una parte di almeno due caratteri di lunghezza.

```
public static List<string> SearchAccount(string Query)
{
        string SqlQuery = "SELECT Username FROM Account WHERE Username LIKE @Query";

        List<string> Results = new();

        MySqlConnection Connection = new(ConnectionString);
        Connection.Open();

        MySqlCommand Cmd = new(SqlQuery, Connection);
        Cmd.Parameters.AddWithValue("@Query", "%" + Query + "%");

        var Reader = Cmd.ExecuteReader();

        while (Reader.Read()) Results.Add(Reader.GetString("Username"));

        return Results;
}
```

La struttura è quella di una richiesta con risposta, con la differenza che in questo caso ci possono essere molteplici risultati. Verrà infatti restituita una lista `Results` con tutti i valori inviati dal database e letti da `Reader.Read()`, che legge la risposta riga per riga.

2.8 Messages.cs

Quando un utente invia un messaggio a un altro utente, questo messaggio viene inviato al server, che lo gestisce e lo reindirizza all'utente destinatario.

2.8.1 Extrait du code

```
using MessagyreServer.Classes;

public class Messages
{
        public static void OnMessageSignal(Signal MessageSignal) {...}

        public static void SendMessage(string SenderUsername, string RecipientUsername, string Content) {...}
```

```
}
```

2.8.2 Reception d'un message

Quando il server riceve un segnale con il valore della variabile `Type` pari a `SignalType.Message`, la classe `Server` chiama il metodo `OnMessageSignal()`, fornendo come parametro il segnale ricevuto.

```
public static void OnMessageSignal(Signal MessageSignal)
{
        // Checking if the sender is logged in
        if (!MessageSignal.Sender!.IsAuthorized) return;

        // Getting the data
        if (!MessageSignal.Data.TryGetValue("RecipientUsername", out string? RecipientUsername)) return;
        if (!MessageSignal.Data.TryGetValue("Content", out string? Content)) return;
        string SenderUsername = MessageSignal.Sender.Account!.Username;

        // Sending
        SendMessage(SenderUsername, RecipientUsername, Content);
}
```

Quando viene chiamato il metodo, viene innanzitutto controllato se l'utente dal quale client il server ha ricevuto il segnale sia *autentificato*, cioè abbia effettuato l'accesso alla piattaforma: in caso contrario, l'utente non dovrebbe essere in grado di inviare messaggi perché l'applicazione non dovrebbe permetterlo, quindi il segnale viene ignorato. In seguito vengono ottenuti i dati dal segnale: l'`Username` del destinatario e il contenuto (`Content`) del messaggio. Infine viene inviato il messaggio, chiamando il metodo `SendMessage`, trattato nella seguente sezione.

2.8.3 Envoi d'un message

Come visto precedentemente, per l'invio di un messaggio a partire dal server, viene chiamato il metodo `SendMessage`, che si occupa di recapitare il messaggio all'utente, che questo sia attualmente connesso alla piattaforma oppure no. I parametri richiesti sono il nome utente del mittente (`SenderUsername`), quello del destinatario (`RecipientUsername`) e il contenuto del messaggio (`Content`).

```
public static void SendMessage(string SenderUsername, string RecipientUsername, string Content)
{
        // Getting the recipient account
        if (!AccountsManager.GetAccount(RecipientUsername, out Account? RecipientAccount)) return;

        // Checking if the recipient is online
        User? Recipient = null;
        foreach (User ConnectedUser in Server.ConnectedUsers)
```

```
        {
                if (ConnectedUser.Account?.Username == RecipientUsername) Recipient = ConnectedUser;
        }

        // Creating the message signal
        Signal NewMessageSignal = new(SignalType.Message);
        NewMessageSignal.Data.Add("SenderUsername", SenderUsername);
        NewMessageSignal.Data.Add("Content", Content);
        NewMessageSignal.Data.Add("SentAt", DateTime.UtcNow.ToString("o"));

        // Sending to the online user or to their inbox
        if (Recipient != null)
        {
                Recipient.Receive(NewMessageSignal);
        }
        else
        {
                RecipientAccount!.Inbox.Add(NewMessageSignal);
        }
}
```

Come si puo capire dai commenti in inglese, viene dapprima cercato l'account `RecipientAccount` del destinatario. Se non viene trovato, l'invio del messaggio non viene effettuato. Poi viene controllato se l'utente destinatario del messaggio è attualmente online, controllando nella lista degli utenti connessi (`ConnectedUsers`) contenuta nella classe `Server` , per decidere il trattamento del messaggio: Se l'utente è online, il messaggio gli viene inviato direttamente, chiamando il metodo `Receive` della classe `User` . Altrimenti, se l'utente non compare nella lista degli utenti connessi, il messaggio viene aggiunto alla lista `Inbox` del suo account, cosi che alla prossima riconnessione dell'utente, il messaggio gli venga inviato.