Gymnase de Renens

Travail de Maturité 2024 - 2025

Messagyre

Le gymnase à portée de main

Répondant :
Joachim Hugonot

Auteur :
Pietro Gravina

Table des matières

1 Introduction

Le projet "Messagyre" est né de l'observation d'une série de difficultés rencontrées durant mon expérience au Gymnase de Renens. D'une part, il manquait un système informatique ou une application permettant aux élèves de gérer de manière complète leur année scolaire (devoirs, notes, communications avec les camarades et les professeurs), sans devoir recourir exclusivement à l'agenda papier, déjà dépassé dans de nombreux pays (comme l'Italie, dont je suis originaire). D'autre part, j'ai remarqué l'absence d'un outil de communication plus informel: bien que la suite Microsoft 365 et d'autres instruments soient disponibles, les e-mails sont trop formels et peu pratiques, tandis que les chats d'Outlook et de Teams[1] sont peu intuitifs et rarement utilisés.

Pour répondre à ces problématiques, j'ai choisi comme sujet de mon Travail de Maturité la conception, le développement et la réalisation d'une application réunissant dans une seule plateforme toutes les fonctions essentielles à la vie scolaire. "Messagyre" n'est en effet pas seulement une application de messagerie, mais comprend également une section pour la gestion des notes et une autre dédiée aux devoirs, permettant ainsi aux élèves d'avoir tout le nécessaire à portée de main.

Son système de communication est similaire à celui de WhatsApp[2], mais avec une différence fondamentale: il ne nécessite pas le numéro de téléphone du destinataire, car il permet de communiquer simplement en connaissant son prénom (et, si nécessaire, son nom de famille), rendant ainsi les échanges entre étudiants et enseignants beaucoup plus rapides et accessibles. Le nom "Messagyre" provient de la fusion des mots "Messages" et "Gyre" (ce dernier étant lui-même une contraction de Gymnase de Renens).

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.

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 C#[3], Python, Lua, Java, JavaScript[4] et Dart[5].

Mes premières expériences en communication client-serveur consistaient en de simples échanges de données entre des pages HTML avec des scripts en JavaScript[4] et un serveur basé sur Node.js[4]. 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[6] comme client et un serveur en Node.js (JavaScript[4]). 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#[3], en utilisant l'API ASP.NET Core[7] de Microsoft. J'ai donc entièrement réécrit l'application sur Unity[6] ainsi que le serveur, améliorant donc la stabilité et l'intégration du système. Pour le développement, j'ai utilisé Visual Studio 2022[6], l'environnement de développement de Microsoft, pour le client ainsi que pour le serveur.

Néanmoins, après avoir avancé avec Unity[6], j'ai constaté que cette plateforme n'était pas idéale pour développer une application mobile de messagerie, principalement à cause des contraintes liées à la gestion de l'interface utilisateur et à la lourdeur de l'environnement pour ce type d'application. J'ai alors décidé de migrer le client vers Flutter[8], un framework moderne spécialisé dans le développement d'interfaces mobiles performantes et adaptatives. Cette décision m'a permis d'améliorer considérablement l'expérience utilisateur tout en conservant ASP.NET Core[7] pour la partie serveur.

Dans ce document, les différentes phases du projet seront présentées en détail : du contexte initial et de l'analyse des applications existantes, au développement technique du client et du serveur, aux fonctionnalités avancées, aux résultats obtenus et aux difficultés rencontrées tout au long du projet. Enfin, je conclurai par une réflexion personnelle sur le travail accompli et les compétences acquises.

2 Contexte et analyse préliminaire

2.1 Panorama des applications de messagerie

Les applications de messagerie instantanée occupent aujourd'hui une place centrale dans la communication quotidienne, tant au niveau personnel que professionnel. Des services comme WhatsApp[2], Telegram[9] ou Signal[10] permettent des échanges rapides, souvent gratuits, et proposent diverses fonctionnalités telles que les appels vocaux, le partage de fichiers, et la création de groupes.

Ces applications visent à faciliter la communication en temps réel, en garantissant à la fois la simplicité d'utilisation et la sécurité des échanges. Cependant, elles présentent parfois des limites dans le cadre de communications institutionnelles ou spécifiques, comme celles d'un établissement scolaire, où la gestion des utilisateurs et la confidentialité doivent être adaptées, contrairement à Microsoft Teams[1] plus orienté vers les usages professionnels et collaboratifs.

2.2 Analyse de la concurrence

Les principales applications concurrentes sur le marché sont WhatsApp[2], Telegram[9], Signal[10] et Microsoft Teams[1]. WhatsApp[2] est très populaire, mais utilise le numéro de téléphone comme identifiant principal, ce qui peut poser des problèmes de confidentialité. Telegram[9] offre davantage de fonctionnalités avancées, comme des canaux publics et une meilleure gestion de la vie privée, mais nécessite un certain niveau de familiarité technique. Signal[10] se concentre sur la sécurité et la confidentialité, avec un chiffrement de bout en bout par défaut. Microsoft Teams[1], intégré à la suite Microsoft 365, permet la communication et la collaboration, mais reste plus formel et moins intuitif pour les échanges informels entre étudiants et enseignants.

Aucune de ces solutions ne répond parfaitement aux besoins spécifiques d'un environnement scolaire comme le Gymnase de Renens, notamment en termes de facilité d'accès basée uniquement sur le prénom et le nom, sans nécessiter de numéro de téléphone.

2.3 Technologies principales impliquées

Pour le développement de Messagyre, plusieurs technologies clés ont été utilisées tout au long du projet. Pour le client mobile multiplateforme, le framework de Google[11] Flutter[8] a été choisi, utilisant le langage Dart[5], permettant de créer une interface moderne, fluide et performante sur Android et iOS. Le serveur est développé avec le framework de Microsoft[12] ASP.NET Core[7] en C#[3], garantissant robustesse, performances et facilité de maintenance. Initialement, le backend était développé en Node.js[4] pour prototyper rapidement les fonctionnalités en temps réel.

La communication entre le client et le serveur utilise HTTP[13] pour les requêtes classiques (comme l'authentification et la récupération des données) et WebSocket[14] pour la messagerie instantanée en temps réel. Les données des utilisateurs et des messages sont stockées dans une base de données MySQL[15], initialement hébergée sur Railway[16], puis sur un VPS, c'est-à-dire un ordinateur « à louer » actif en permanence. Le programme backend était également initialement hébergé sur Railway, mais pour des raisons de coût (voir section Hébergement et Budget 5.2) il a été déplacé sur Fly.io[17], pour finalement être hébergé sur le même ordinateur que la base de données.

Pour la gestion locale des données sur l'appareil, comme l'historique des messages, Hive[18] est utilisé, une librairie Flutter qui permet de stocker des informations sur le disque dur en imitant une petite base de données. D'autres données plus sensibles, comme les tokens pour l'accès à l'application (voir section Développement du serveur 3.3), sont stockées dans Flutter Secure Storage, également une librairie Flutter capable de chiffrer et déchiffrer les données avant de les sauvegarder. Les notifications push sont gérées par Firebase[19], un ensemble de services web de Google[11]. L'envoi des emails de vérification utilisait initialement un serveur SMTP[20], mais à cause de certaines restrictions (voir section Hébergement et Budget 5.2) il utilise actuellement les API de Gmail[21], et les photos de profil sont hébergées par Cloudinary[22], un service tiers permettant de stocker des photos accessibles à tous gratuitement.

Pour la conception graphique des icônes et des éléments visuels de l'application, des outils comme Remove.bg[23] ont été utilisés pour supprimer l'arrière-plan, Pixlr E[24] pour modifier les couleurs et créer certaines icônes, Canva[25] et Adobe Express[26] pour des

graphismes plus complexes. Enfin, le versionnage du code et la gestion du projet ont été réalisés avec Git[27] puis publiés sur GitHub[28], tandis que le codage s'est déroulé principalement sur Visual Studio[6] pour le backend et Visual Studio Code[29] pour le frontend.

3 Développement du projet

3.1 Analyse des besoins et spécifications

L'application Messagyre a été conçue pour répondre à plusieurs besoins fondamentaux, dans le but de faciliter la communication interne au Gymnase de Renens tout en permettant une gestion complète des devoirs et des notes des élèves. L'application assure une authentification sécurisée des utilisateurs, permet l'envoi et la réception de messages texte en temps réel via WebSocket[14], et gère les profils utilisateurs, incluant la possibilité d'ajouter une photo de profil hébergée sur Cloudinary[22] et d'autres informations de contact telles que l'adresse e-mail, le numéro de téléphone ou les liens vers les réseaux sociaux. Messagyre sert également de véritable agenda scolaire, permettant l'ajout de devoirs et de notes, et offre une interface intuitive et accessible même aux utilisateurs moins expérimentés, garantissant une utilisation rapide, agréable et sécurisée, tout en protégeant la confidentialité des communications[30].

Le public cible comprend principalement les élèves, mais aussi les enseignants et le personnel du gymnase, avec une attention particulière portée à la facilité d'accès, sans nécessiter l'utilisation d'un numéro de téléphone. Des prototypes initiaux ont été réalisés pour définir la disposition des écrans principaux, tels que la page de connexion et d'inscription, la liste des conversations, la fenêtre de chat et le profil utilisateur, guidant ainsi le développement de l'interface.

Pour permettre le bon fonctionnement de l'application, un backend a été nécessaire, c'est-à-dire un système chargé de gérer les utilisateurs et leurs interactions, y compris toutes les opérations liées aux messages. Pour cela, ASP.NET Core[7], un framework de Microsoft pour le développement d'applications web en C#[3], a été choisi, langage avec lequel je possède une expérience solide. Le backend gère les utilisateurs connectés à la plateforme, leurs informations personnelles comme le nom d'utilisateur, l'adresse e-mail, le mot de passe, la photo

de profil et les détails du profil public, permet la création de nouveaux comptes et l'accès aux comptes existants, trie les messages et vérifie qu'ils sont correctement reçus par les destinataires. La communication entre le client et le serveur se fait via HTTP[13] pour les opérations classiques et WebSocket[14] pour la messagerie en temps réel.

Grâce à cette architecture, Messagyre combine sécurité, efficacité et simplicité d'utilisation, répondant de manière complète aux besoins de la vie scolaire numérique.

3.2 Développement du client (Flutter)

La structure de l'application mobile est basée sur le framework Flutter[8], choisi pour sa capacité à créer des interfaces réactives et multiplateformes à partir d'un seul code source en Dart[5]. Flutter permet également de visualiser en temps réel les modifications du code sans avoir à recompiler l'application entière, grâce à la fonction de hot reload, rendant le développement plus simple et rapide par rapport à d'autres systèmes, comme Unity utilisé précédemment.

Pour la persistance locale des données, telles que l'historique des messages et les paramètres de l'utilisateur, les bibliothèques Hive[18] et Flutter Secure Storage[31] sont utilisées, cette dernière servant également à stocker en toute sécurité les mots de passe. Les notifications push sont gérées via Firebase[19], par lequel le serveur les envoie directement au client, permettant aux utilisateurs de recevoir des mises à jour même lorsque l'application n'est pas active.

L'interface utilisateur a été conçue en suivant le style iOS avec Cupertino, une bibliothèque de composants intégrée à Flutter qui reproduit fidèlement l'apparence et le comportement natif des applications Apple. Ce choix a été fait pour sa modernité et sa cohérence visuelle, offrant une expérience fluide et agréable aux utilisateurs.

La première partie du développement de l'application a été l'implémentation de l'authentification : j'ai créé les pages dédiées à la connexion et à l'inscription, permettant à l'utilisateur de s'identifier une seule fois et d'accéder à l'application. Ensuite, je me suis concentré sur le développement d'une page listant les conversations, suivie d'une page pour chaque conversation. Pour permettre aux utilisateurs de rechercher d'autres utilisateurs avec lesquels échanger des messages, j'ai ajouté une page dédiée, par laquelle l'application envoie une requête HTTP[13] au serveur qui renvoie une liste de résultats. En cliquant sur l'un des résultats, une page "profil" permet de visualiser les informations de l'utilisateur, telles que son nom d'utilisateur, son surnom, son email scolaire, sa classe et tout

ce que cette personne a ajouté à son profil. Une fois la partie principale de "l'application de messagerie" terminée, je me suis concentré sur le développement d'une page pour les notes, avec une liste des branches et leurs moyennes, une sous-page pour visualiser les notes d'une branche spécifique et une autre sous-page pour ajouter une note, avec titre, valeur, branche, groupe d'évaluation et informations supplémentaires telles que la description et les photos du test ou du travail évalué. Enfin, la page pour les devoirs a été la dernière à être développée, avec un calendrier horizontal contenant tous les devoirs ajoutés par l'utilisateur via la sous-page correspondante, où l'on peut choisir la branche, une description, si c'est un devoir noté ou un test. La page des paramètres a été implémentée au fur et à mesure que l'application prenait forme, en ajoutant des options pour modifier son profil, se déconnecter, activer le mode sombre, changer l'arrière-plan des conversations, supprimer les données et personnaliser des composants de l'application tels que le calendrier et les branches.

3.3 Développement du serveur (ASP.NET Core)

Le serveur backend, c'est-à-dire le programme qui gère toutes les requêtes de l'application, comme l'envoi de messages, la recherche d'autres utilisateurs, etc., a été développé avec ASP.NET Core[7], un framework open source de Microsoft[12] pour le développement d'applications web modernes, multiplateformes et hautes performances. J'ai choisi cette technologie pour sa robustesse, ses hautes performances et ma familiarité avec le langage C#[3].

La communication en temps réel entre le client et le serveur est basée sur WebSocket[14], qui permet l'envoi et la réception instantanés des messages. Pour les opérations plus classiques, comme l'authentification, la création ou la modification d'un compte, le serveur expose des API REST[apirest]. Une API REST (Representational State Transfer) est un ensemble de règles permettant à deux systèmes de communiquer via des requêtes HTTP standard (GET, POST, PUT, DELETE), de manière simple et uniforme, rendant ainsi les interactions entre le client et le serveur plus claires et plus évolutives.

Le système d'authentification reçoit les identifiants saisis par l'utilisateur dans l'application, vérifie s'ils sont valides, et si c'est le cas, renvoie un jeton « JSON Web Token (JWT) »[13], c'est-à-dire un texte chiffré contenant diverses informations prouvant que l'utilisateur est autorisé à accéder à l'application, avec une certaine date d'expiration. En plus de ce jeton, un autre est également envoyé au client, moins complexe (il s'agit d'un simple UUID[32]), que j'ai appelé « refresh token », afin de valider et prolonger les sessions des utilisateurs sans devoir redemander leurs identifiants à chaque ouverture de l'application.

La base de données relationnelle MySQL[15] stocke des informations telles que la liste de tous les comptes utilisateurs et une liste de messages dans la « inbox », c'est-à-dire des messages qui n'ont pas encore été remis à leur destinataire, car celui-ci n'a pas été en ligne depuis leur envoi.

Chaque compte dans la base de données contient des informations telles que : le nom, l'adresse e-mail scolaire, le mot de passe chiffré (avec hachage[33]), la photo de profil, le jeton d'accès JWT[34], le refresh token, des informations publiques comme le pseudonyme et la biographie, ainsi que d'autres données nécessaires à certaines fonctionnalités comme les notifications push. Les messages, quant à eux, contiennent le contenu, l'heure d'envoi, l'état de lecture, etc.

La majorité des fonctionnalités sont gérées par le programme que j'ai écrit, sauf quelques exceptions : l'envoi des e-mails de vérification lors de la création d'un compte utilisateur a d'abord été géré par un serveur SMTP[20], puis par les API de Gmail[googleapi] ; les photos de profil sont stockées dans une base de données tierce, gérée par Cloudinary[22] ; la base de données était initialement hébergée sur Railway[16], mais a ensuite été transférée sur le serveur de mon professeur, voir la section Hébergement et budget 5.2 ; le programme backend lui-même, tout comme la base de données, a été déplacé plusieurs fois : il fonctionnait d'abord sur Railway[16], puis sur Fly.io[17], avant d'être finalement transféré sur le serveur de mon professeur.

3.4 Fonctionnalités avancées

Dans Messagyre, plusieurs fonctionnalités avancées ont été intégrées pour améliorer l'expérience utilisateur et la qualité globale de l'application. La gestion des photos de profil permet aux utilisateurs de télécharger une image personnelle qui est automatiquement mise à jour et affichée auprès des autres utilisateurs dès qu'elle est modifiée[22]. De plus, il est désormais possible de disposer d'un profil public consultable par les autres utilisateurs, contenant des informations de base telles que le nom, la photo et d'autres détails choisis par l'utilisateur, facilitant ainsi l'identification et l'interaction au sein de la plateforme.

Les notifications push sont entièrement opérationnelles : le serveur envoie aux utilisateurs une alerte immédiate lorsqu'un nouveau message ou une communication importante est reçue, garantissant la réception en temps réel même lorsque l'application n'est pas ouverte au premier plan[19]. En termes de performances, l'application gère efficacement les connexions WebSocket[14] et réduit le nombre d'appels HTTP inutiles[13], amélio-

rant ainsi la vitesse et réduisant la consommation de données, ce qui est particulièrement important sur les appareils mobiles avec des connexions instables ou limitées.

Concernant la sécurité, toutes les communications entre le client et le serveur sont chiffrées via HTTPS et des WebSockets sécurisés. Les données envoyées par les utilisateurs sont systématiquement validées côté serveur pour prévenir tout comportement inattendu ou malveillant. Des contre-mesures ont été mises en place pour se protéger contre les attaques courantes telles que les injections SQL, le Cross-Site Scripting (XSS) et les attaques de type Man-in-the-Middle (MITM), conformément aux recommandations de l'OWASP[35], qui définit les bonnes pratiques pour sécuriser les applications web contre les vulnérabilités et attaques informatiques.

Enfin, une page de débogage a été ajoutée dans les paramètres, permettant aux utilisateurs de vérifier d'éventuels problèmes avec l'application. Cette fonctionnalité est particulièrement utile pour détecter des erreurs qui ne se manifestent pas lors des tests sur émulateur ou sur les appareils de développement, facilitant ainsi le débogage et la résolution rapide des problèmes.

3.5 Tests et débogage

Au début de la phase de développement, les principales difficultés rencontrées concernaient la gestion en temps réel des messages, la synchronisation des horaires d'envoi et de réception, la stabilité des connexions et l'authentification via token (une nouveauté pour moi), résolues grâce à une meilleure gestion de l'état côté client et côté serveur.

Le projet a suivi plusieurs tests pour garantir la fiabilité ainsi qu'une bonne expérience utilisateur de l'application. Les tests les plus fréquents ont été des tests manuels où j'ai fait exécuter l'application sur l'émulateur Android officiel d'Android Studio[36] via le débogueur officiel de Flutter dans Visual Studio Code. Grâce à cela, j'ai pu développer une interface graphique propre et agréable à regarder, une accessibilité optimale en déplaçant les différents boutons pour qu'ils soient plus facilement actionnables par les doigts de l'utilisateur, le fonctionnement complet des fondations de l'application et l'exécution correcte des processus non visibles dans l'application elle-même (processus en arrière-plan). D'autres tests ont été réalisés en construisant l'application pour les plateformes spécifiques (Android et iPhone) en utilisant un téléphone physique ; grâce à ces tests, j'ai trouvé plusieurs « failles » qui n'apparaissaient pas sur l'émulateur, telles que des animations graphiques défaillantes, des champs de texte inutilisables, et autres. De plus, j'ai pu tester les notifications push, mais cela avec l'aide de la console Firebase, le service

utilisé par le serveur pour les envoyer.

Les tests mentionnés jusqu'ici concernaient uniquement l'application, la partie « client », mais le serveur a également nécessité plusieurs vérifications pour fonctionner correctement. Comme mentionné dans la section 3.3, le serveur a été déplacé entre plusieurs plateformes, chacune avec ses propres exigences. Pour vérifier que le serveur répondait correctement, j'ai utilisé le navigateur Chrome[37], en envoyant des requêtes HTTP[13] et HTTPS[30] au serveur via la barre de recherche du navigateur. Pour tous les autres tests, je me suis servi de l'application elle-même, car l'authentification nécessitait de nombreuses requêtes, ainsi qu'une connexion WebSocket, plus longue et complexe à réaliser via un navigateur.

La communication entre le serveur et la base de données MySQL[15] a également été testée, en effectuant plusieurs appels via l'application et un programme externe MySQL Workbench[38], et en vérifiant que les informations étaient correctement enregistrées et qu'aucun problème de données corrompues ne survenait lors de la lecture par le serveur.

4 Résultats obtenus

4.1 État actuel de l'application

À ce jour, l'application Messagyre fonctionne pleinement dans ses fonctionnalités principales. Les utilisateurs peuvent créer un compte via leur adresse e-mail scolaire, se connecter de manière sécurisée, envoyer et recevoir des messages en temps réel, modifier leur profil et consulter ceux des autres utilisateurs. Le système de messagerie repose sur WebSocket, garantissant une communication fluide et instantanée.

Certaines fonctionnalités secondaires sont encore en cours de développement ou prévues pour les prochaines versions, notamment :

- Implémentation des notifications push.
- Gestion complète des devoirs et des notes dans les pages « Devoirs » et « Notes » respectivement.
- Améliorations graphiques de l'interface utilisateur.
- Animations pour rendre l'utilisation de l'application plus agréable.
- Possibilité de créer et de rejoindre des groupes de discussion.
- Une page dédiée aux annonces et communications officielles (ou non) du gymnase.

4.2 Comparaison avec les objectifs initiaux

Le projet a atteint la majorité des objectifs fixés au départ :

- Développer un système de messagerie interne, sécurisé et facile à utiliser.
- Permettre une communication simple sans utiliser le numéro de téléphone.
- Intégrer des comptes scolaires avec vérification par e-mail.
- Implémenter une page pour la gestion des devoirs.

Dans l'ensemble, les résultats obtenus sont très satisfaisants, compte tenu de la complexité technique du projet, du temps disponible et du fait qu'il s'agit d'un développement réalisé entièrement de manière autonome.

4.3 Captures d'écran de l'application

Voici quelques captures d'écran de Messagyre dans son état actuel, accompagnées d'une brève description :

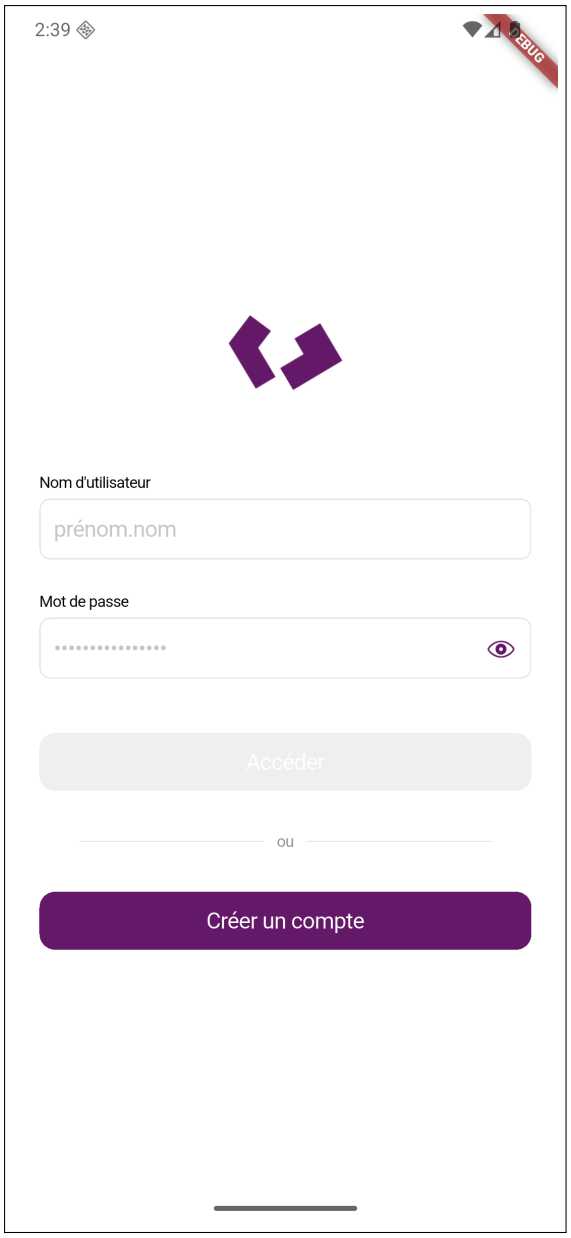

Figure 4.1: *
Page de connexion : avec les champs de saisie et le bouton de connexion.

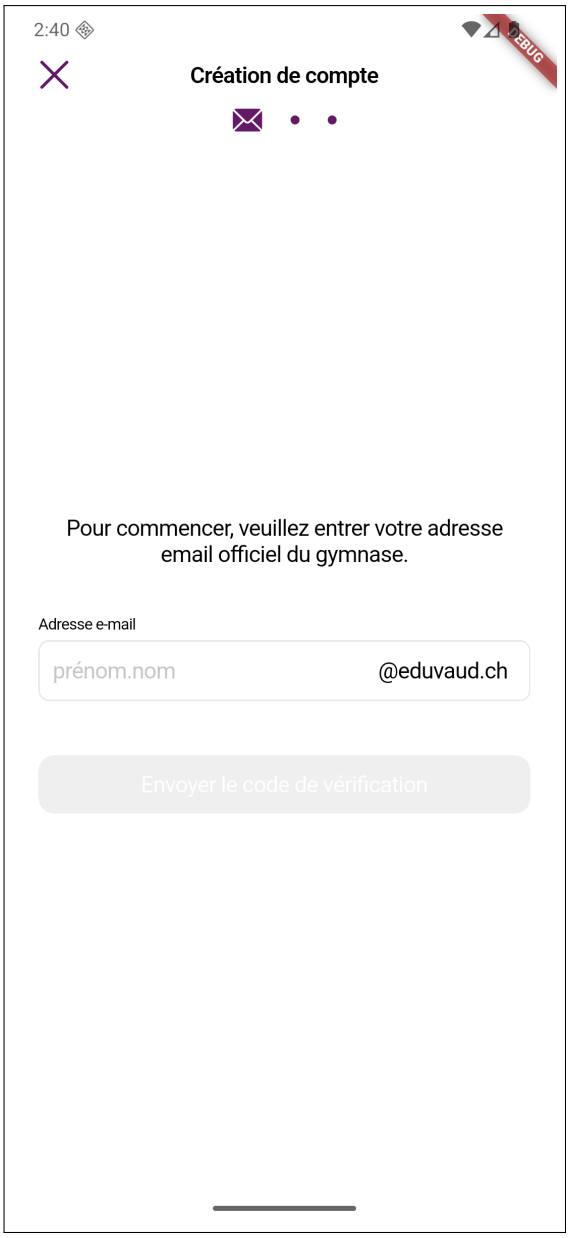

Figure 4.2: *
Page d'inscription : formulaire avec vérification de l'adresse e-mail.

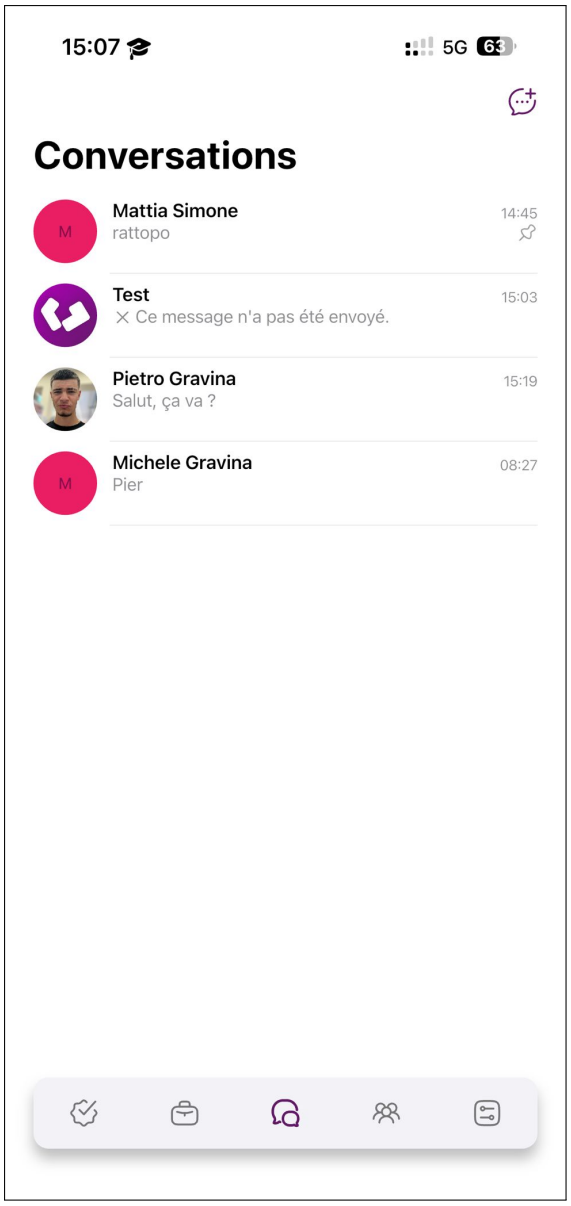

Figure 4.3: *
Liste des conversations : affichage des contacts et des derniers messages reçus.

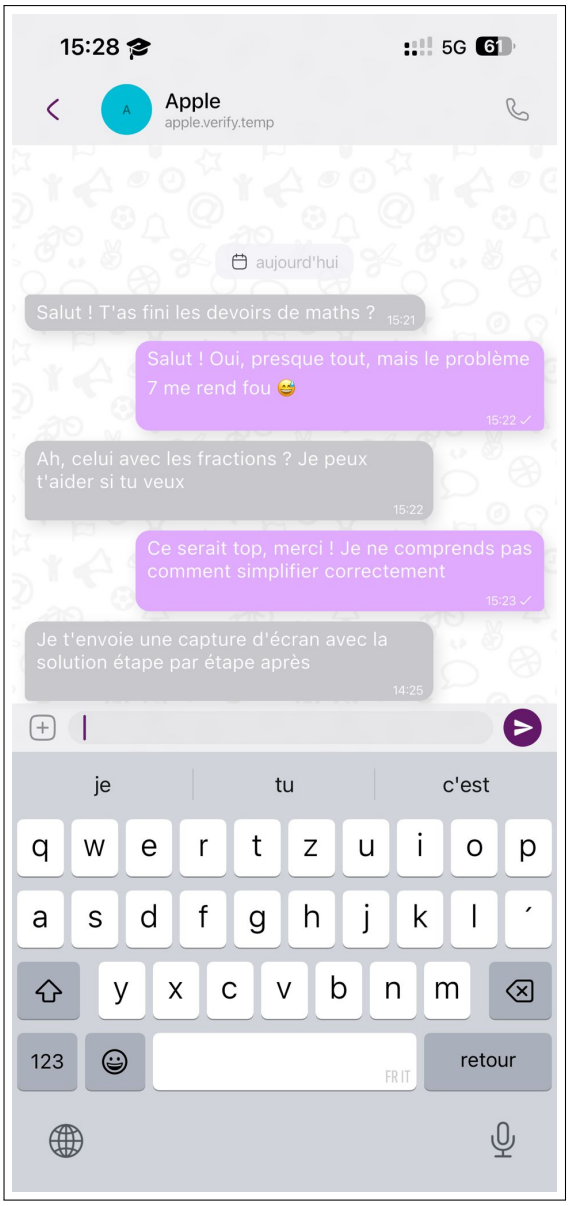

Figure 4.4: *
Fenêtre de discussion : interface avec les messages envoyés et reçus.

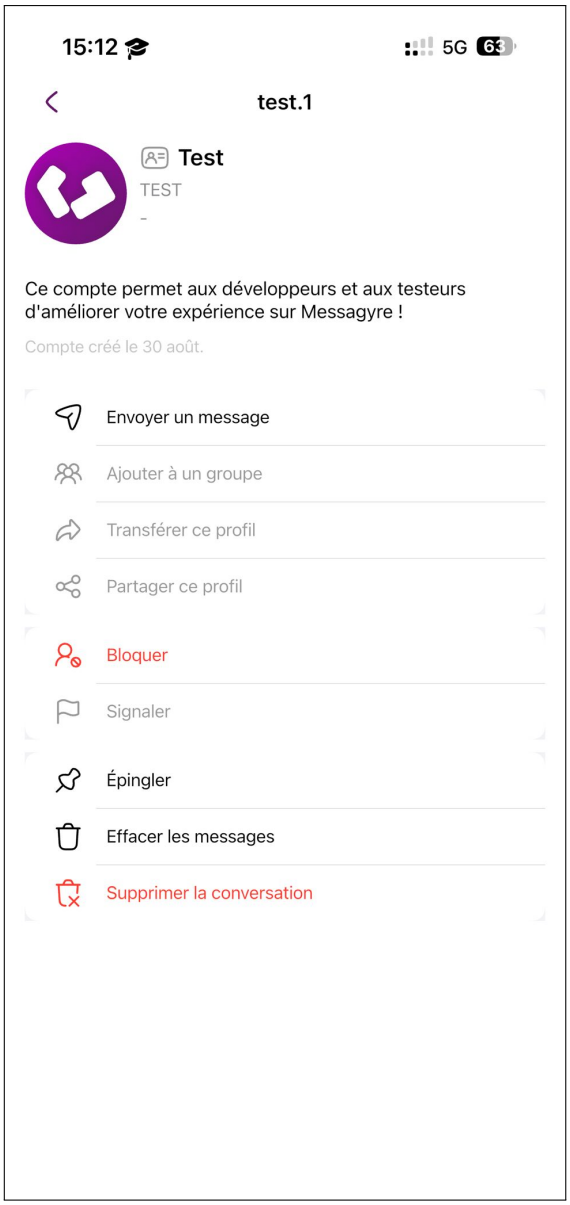

Figure 4.5: *
Profil utilisateur : aperçu d'un compte avec photo et informations visibles.

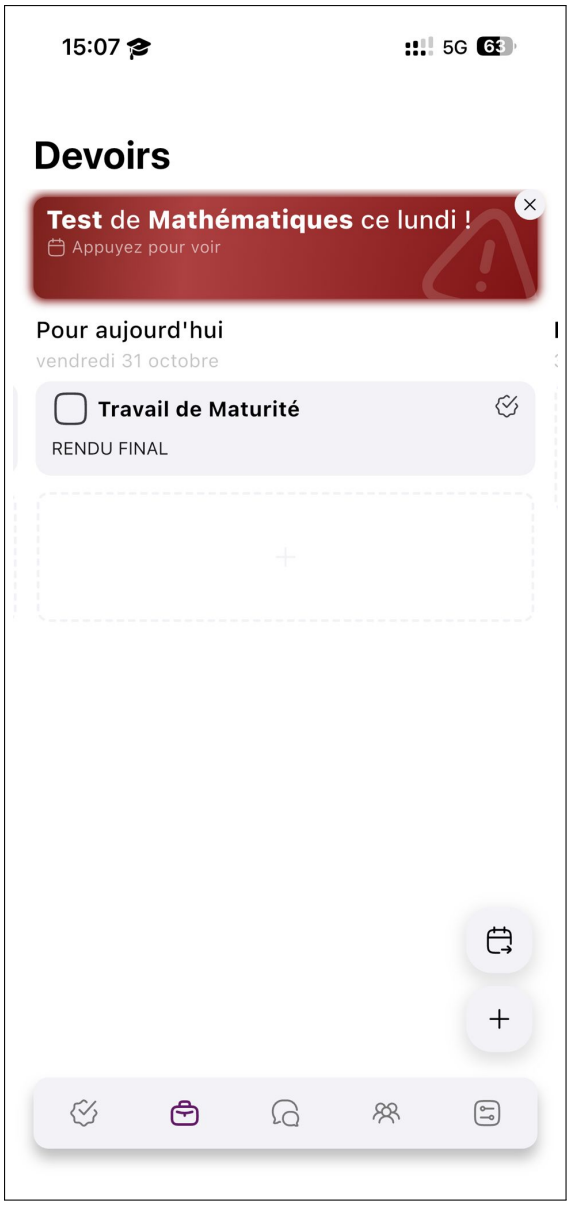

Figure 4.6: *
Page des devoirs : liste des prochains devoirs, ajout d'un nouveau et visualisation des détails.

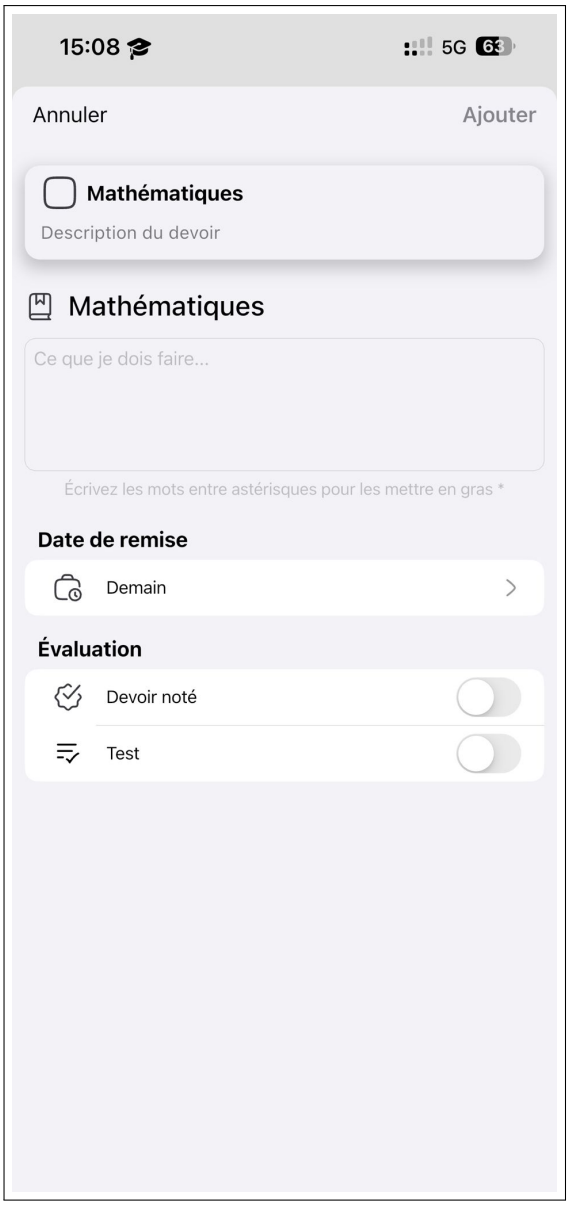

Figure 4.7: *
Ajout d'un devoir : page qui permet à l'utilisateur d'ajouter un devoir à la liste.

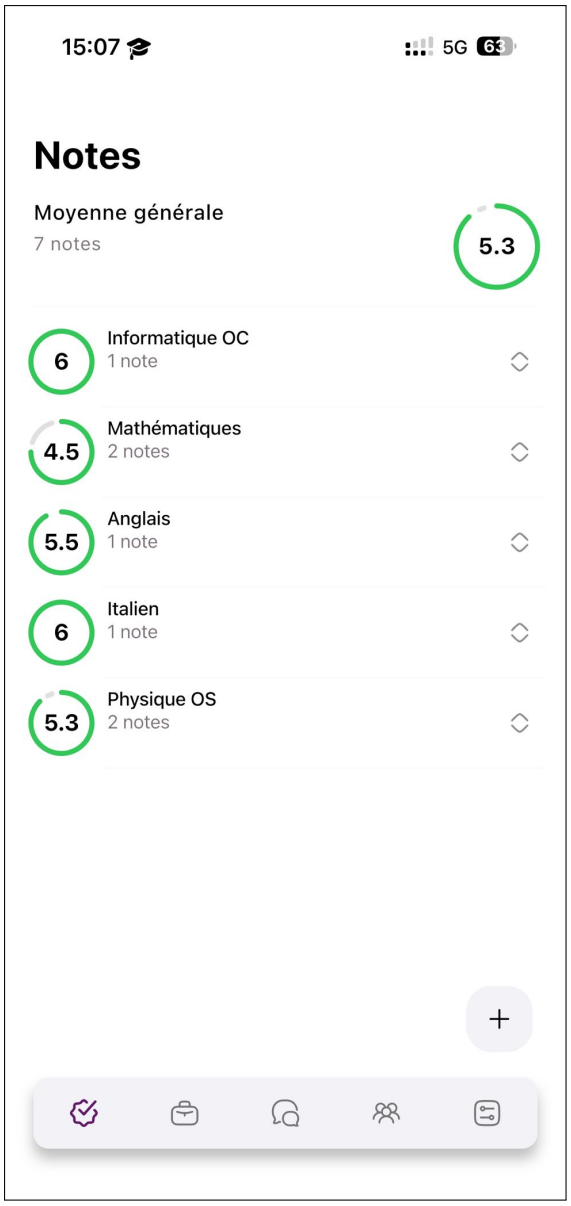

Figure 4.8: *
Page des notes : affichage des moyennes et des résultats obtenus dans chaque matière.

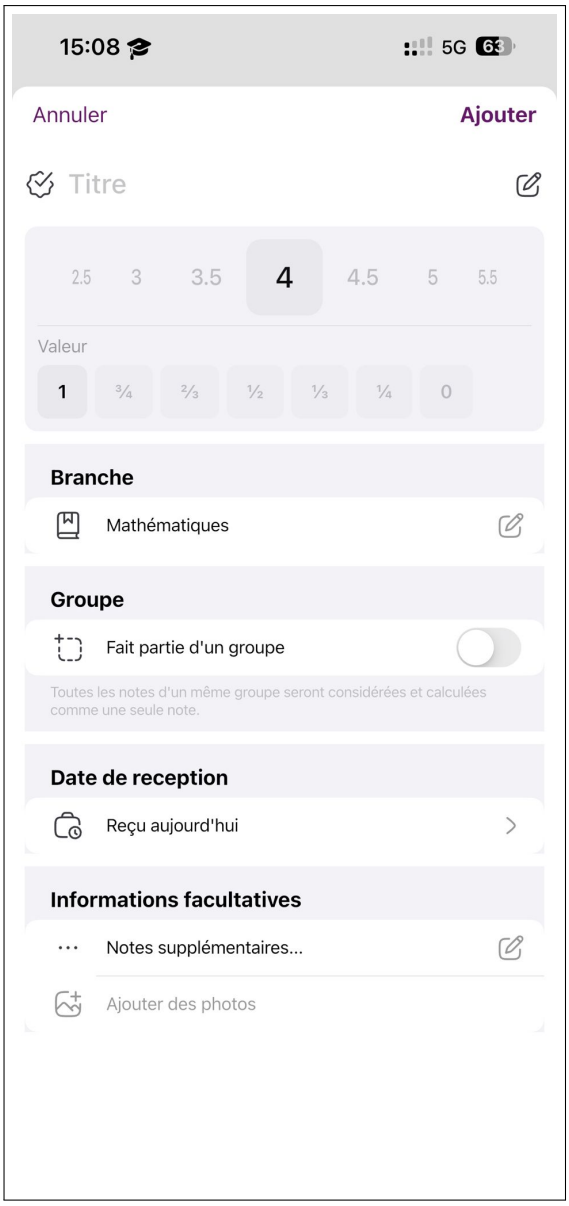

Figure 4.9: *
Ajout d'une note : formulaire permettant d'ajouter une nouvelle note avec matière et coefficient.

Figure 4.10: *
Page de recherche : permet de trouver d'autres utilisateurs ou conversations rapidement.

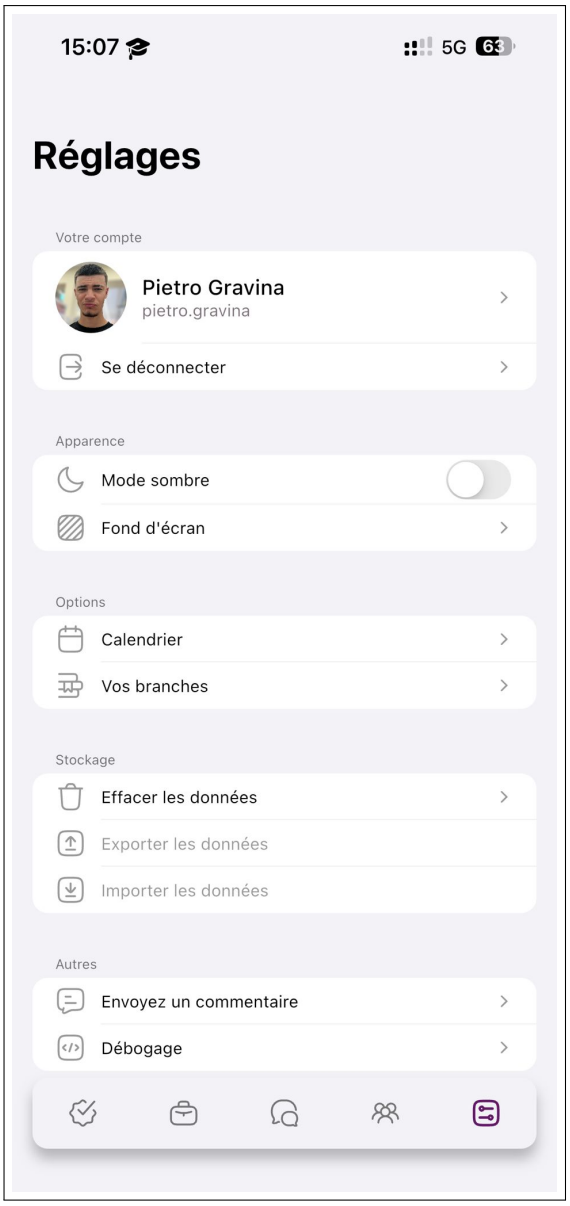

Figure 4.11: *
Page des paramètres : options de personnalisation du profil et de gestion de l'application.

5 Difficultés rencontrées

5.1 Difficultés techniques

Pendant le développement de Messagyre, j'ai dû faire face à de nombreuses difficultés techniques qui ont mis à l'épreuve mes connaissances et ma capacité d'adaptation.

L'une des premières étapes du projet a été l'utilisation d'Unity comme client. Ce choix initial s'est vite révélé peu adapté au développement d'une application mobile. L'interface utilisateur était plus complexe à construire et les intégrations avec HTTP et WebSocket étaient laborieuses. Chaque élément devait être programmé manuellement : animations, dimensions, positionnement des composants, logique dynamique et visuelle. Tout devait être écrit depuis zéro, sans composants natifs adaptés aux appareils mobiles. Cette gestion artisanale prenait environ 90% du temps de développement, ralentissant considérablement l'avancement du projet et rendant chaque modification fastidieuse. Le manque d'outils natifs pour les appareils mobiles a nui à l'expérience utilisateur, ce qui m'a poussé à migrer vers Flutter, bien plus adapté à une application moderne et fluide.

Flutter a également posé des difficultés, notamment au début. N'ayant jamais utilisé ce framework auparavant, j'ai eu du mal à comprendre le cycle de vie des widgets et la gestion de l'état. L'intégration de paquets externes pour le chargement d'images, les animations et les notifications m'a demandé de nombreuses tentatives et beaucoup de débogage.

L'un des défis les plus complexes a été la gestion de la communication en temps réel via WebSocket. En plus de devoir comprendre les différences avec HTTP, j'ai appris à gérer des connexions persistantes, des déconnexions inattendues et la synchronisation des messages entre le client et le serveur, tout en garantissant une bonne expérience utilisateur.

La mise en place d'un système d'authentification sécurisé basé sur JWT et RefreshToken a également été une nouveauté pour moi. J'ai étudié des aspects liés à la cryptographie

et à la sécurité des données pour assurer la fiabilité des sessions utilisateur. Le volet base de données n'a pas été plus simple : j'ai dû apprendre à configurer et utiliser MySQL, concevoir des tables avec des clés étrangères, optimiser les requêtes et gérer les erreurs de connexion, ce qui a demandé beaucoup de rigueur et de patience.

Le déploiement sur Railway m'a aussi posé des problèmes de compatibilité entre différentes versions de .NET. Adapter le projet à une version stable supportée, sans perdre en performance, a été un défi supplémentaire.

5.2 Hébergement et budget

Une difficulté importante a été de trouver une plateforme en ligne, un service externe, pour héberger le serveur ASP.NET Core. En effet, exécuter le programme que j'ai écrit pour le serveur de Messagyre nécessite certaines caractéristiques qui ne sont pas facilement accessibles, comme une adresse IP statique, indispensable pour être toujours visible par les appareils du monde entier et leur permettre de se connecter. Je cherchais une solution compatible avec la technologie .NET, mais aussi la plus économique possible, n'ayant pas une grande disponibilité financière. Après plusieurs recherches et essais, j'ai d'abord opté pour Railway, une plateforme moderne, facile à configurer et dotée d'une immense communauté, offrant un support complet et réactif.

Cependant, le coût initial était trop élevé pour moi : environ 3 francs par jour selon les estimations, ce qui représentait une dépense difficilement soutenable à long terme. J'avais envisagé de demander un financement à l'école, mais dans le contexte actuel, il est déjà difficile d'obtenir des fonds pour du matériel scolaire classique, alors un hébergement web non réutilisable paraissait encore moins justifiable.

J'ai donc décidé de calculer les dépenses annuelles et de chercher des alternatives. La solution retenue a été de passer à un modèle serverless, également proposé par Railway, qui permet un hébergement gratuit pendant la phase de développement, puis un forfait d'environ 5 francs par mois au moment de la publication. Ce compromis m'a permis de poursuivre le projet sans frais.

Plus tard, je me suis rendu compte que le plan gratuit de cette plateforme était trop limité, bloquant tout après quelques utilisations par mois. Cela m'a conduit à chercher une autre plateforme, et j'ai trouvé Fly.io.

Par rapport à Railway, celle-ci est beaucoup moins une « boîte noire », me permettant de mieux configurer l'environnement, mais elle a présenté certains problèmes que je n'avais

pas auparavant, comme le blocage des serveurs SMTP : un service que j'utilisais pour envoyer les e-mails de vérification aux nouveaux utilisateurs. J'ai donc dû passer par les API de Google, ce qui m'a offert moins de liberté de configuration.

Quelques mois après être passé sur cette nouvelle plateforme, je me suis rendu compte que j'utilisais beaucoup trop de services tiers : un pour stocker les photos de profil des utilisateurs, un pour envoyer les e-mails de vérification, un autre pour la base de données des utilisateurs, et ainsi de suite. Bien que l'application puisse fonctionner parfaitement avec tous ces services, le développement devenait plus complexe. Chacun de ces services proposait un plan gratuit limité, au-delà duquel il fallait souscrire à un abonnement mensuel ; additionnés, ces coûts auraient dépassé largement mon budget.

J'ai discuté de ce problème avec mon professeur d'Option Complémentaire d'Informatique, Micha Hersch, qui m'a gentiment proposé d'utiliser l'un de ses serveurs, qu'il employait déjà pour partager du matériel avec la classe d'OC — une proposition que je ne pouvais refuser, puisqu'elle résolvait bon nombre des problèmes rencontrés jusque-là. Je me suis donc organisé avec lui, j'ai modifié le programme et je l'ai transféré sur son serveur. Grâce à ce passage très rapide, les problèmes comme la lenteur d'accès à l'application, les serveurs qui ne répondaient pas, les déconnexions après peu de temps d'utilisation, et surtout le temps exagéré nécessaire pour appliquer des modifications au code du serveur, ont été résolus.

Actuellement, même si le professeur m'a permis d'utiliser son serveur jusqu'à la fin de l'année scolaire, ce changement me permet de ne supporter aucun coût pour faire fonctionner l'application, ce qui m'a permis — et me permettra encore — de continuer à développer le projet sans limites et sans devoir passer des heures à chercher le moyen le moins coûteux d'ajouter de nouvelles fonctionnalités à l'application.

5.3 Difficultés organisationnelles

Outre les aspects techniques, j'ai rencontré des difficultés organisationnelles. Réussir à concilier le développement du projet avec les cours, les devoirs et la vie personnelle n'a pas été simple. J'ai dû apprendre à mieux gérer mon temps, établir des plannings réalistes et éviter la procrastination.

Parfois, la motivation diminuait, surtout dans les moments où les résultats tardaient ou lorsque les problèmes semblaient insurmontables. Mais voir l'application fonctionner concrètement sur le téléphone de mes amis m'a redonné de l'énergie pour continuer.

5.4 Comment j'ai surmonté les difficultés

Pour surmonter ces obstacles techniques et organisationnels, je me suis appuyé principalement sur l'apprentissage autodidacte, avec l'aide précieuse des intelligences artificielles. J'ai consulté la documentation officielle, suivi des tutoriels, lu des discussions sur des forums comme Stack Overflow et GitHub, et expérimenté différentes approches jusqu'à trouver les bonnes solutions.

Les intelligences artificielles se sont révélées être des outils extrêmement efficaces : elles m'ont souvent proposé des solutions optimisées, ce qui m'a fait gagner un temps précieux. Elles m'ont aussi permis d'écrire du code plus propre et de découvrir des bibliothèques ou des paquets que je n'aurais probablement jamais trouvés seul, accélérant ainsi significativement le développement.

J'ai appris à diviser mon travail en objectifs concrets et atteignables, ce qui m'a permis de progresser étape par étape sans me décourager. Parler du projet avec des amis ou des enseignants m'a également aidé à clarifier mes idées et à trouver de nouvelles pistes.

Toutes ces expériences, bien que parfois éprouvantes, m'ont permis de grandir, aussi bien techniquement que dans la gestion de projet et l'autonomie.

6 Bilan personnel

Ce projet m'a permis de vivre une expérience de développement complète, avec toutes les phases que cela implique : conception, recherche, programmation, tests, déploiement et documentation. Sur le plan technique, j'ai énormément appris. J'ai approfondi mes compétences en C#, découvert ASP.NET Core, compris en profondeur le fonctionnement des WebSocket et de l'authentification via JWT. J'ai également acquis une grande maîtrise de Flutter et du développement mobile multiplateforme, ainsi que des bases de données MySQL.

Sur le plan personnel, ce travail m'a appris la persévérance, la rigueur et l'autonomie. J'ai dû me former seul sur de nombreuses technologies complexes, souvent sans aide extérieure directe. J'ai aussi pris conscience de l'importance de bien planifier son temps et de savoir demander de l'aide au bon moment. Ce projet m'a donné confiance en ma capacité à mener une idée de bout en bout, même face à des obstacles importants.

Si je devais refaire ce projet, je choisirais dès le départ des outils plus adaptés aux besoins d'une application mobile. J'éviterais Unity, dont la flexibilité est un atout dans le domaine du jeu vidéo, mais un inconvénient majeur dans ce type d'application. J'organiserais aussi mieux mon temps dès le début, avec une feuille de route plus précise et un système de gestion des tâches plus rigoureux.

Quant à l'avenir de Messagyre, je n'exclus pas de continuer à le développer et à l'améliorer. Plusieurs idées restent à concrétiser, comme l'ajout de notifications push, une version web de l'application, l'intégration de messages vocaux ou vidéos, ou encore une meilleure gestion des groupes et des discussions collectives. Le projet pourrait même, un jour, être proposé à d'autres écoles, au-delà du Gymnase de Renens, si son usage s'y prête.

En somme, cette expérience a été à la fois un défi technique et une aventure personnelle très enrichissante, qui m'a beaucoup apporté et dont je suis fier.

7 Conclusion

Le développement de Messagyre a représenté pour moi un parcours long, exigeant et formateur. De l'idée initiale à la réalisation de l'application, j'ai traversé de nombreuses étapes : changements de technologies, obstacles techniques, réécritures complètes et améliorations successives. Chaque choix, même ceux qui semblaient initialement erronés, a contribué à construire un projet solide et concret.

D'un point de vue technique, j'ai pu expérimenter de manière pratique de nombreuses compétences acquises au fil des années, en consolider de nouvelles et approfondir des technologies complexes comme Flutter, ASP.NET Core, JWT et WebSocket. Mais au-delà des aspects techniques, j'ai acquis une méthode de travail plus organisée et plus consciente, en affrontant des problèmes réels et en apprenant à chercher des solutions efficaces.

Messagyre n'est pas seulement une application de messagerie : c'est le résultat de centaines d'heures de travail, d'essais, d'erreurs et de corrections. C'est aussi la preuve qu'un projet ambitieux peut devenir réalité, même s'il est développé par une seule personne, avec passion et détermination.

Personnellement, je considère ce projet comme l'un des travaux les plus complets et significatifs que j'aie jamais réalisés. Il a une grande valeur, tant du point de vue professionnel — car il démontre mes compétences de développeur — que du point de vue personnel — car il m'a appris à croire en mes idées et à les porter jusqu'au bout avec détermination.

J'espère que Messagyre pourra continuer à évoluer au-delà de ce travail de maturité. Qu'il puisse être utile aux élèves et aux enseignants, et peut-être devenir un véritable outil de communication scolaire, simple, sécurisé et accessible à tous.

8 Annexes

8.1 Code source

Le code complet du projet est disponible sur GitHub aux adresses suivantes :

- Client Flutter : `https://github.com/Gravi32/MessagyreClient`
- Serveur ASP.NET Core : `https://github.com/Gravi32/MessagyreServer`

Pour faciliter l'accès, voici deux codes QR menant aux dépôts.

Figure 8.1: *
Dépôt GitHub du Client

Figure 8.2: *
Dépôt GitHub du Serveur

8.2 Diagrammes

Pour illustrer la structure et l'architecture du projet, un diagramme UML expliquant le fonctionnement du backend de l'application a été réalisé et est disponible au lien suivant :

https://lucid.app/lucidchart/b8dadd52-61fd-468b-8bd6-a81aa133eb02/edit?viewport_loc=-3637%2C-1769%2C9681%2C4831%2C0_0&invitationId=inv_fa22bca5-082d-4666-b827-15f4adaa3

Figure 8.3: * Diagramme UML du backend

8.3 Icônes de l'application

Voici les icônes de l'application utilisées dans Messagyre pendant la premiere partie du développement :

Figure 8.4: Icône violette (version simplifiée)

Figure 8.5: Icône complète de l'application

Après quelques mois, j'ai décidé de modifier cette icône afin d'obtenir un rendu plus moderne, esthétique et cohérent avec le style des applications contemporaines. Voici trois versions de la même icône, réalisées pour différents cas d'utilisation :

Figure 8.6: Icône violette arrondie pour fonds clairs

Figure 8.7: Icône blanche arrondie pour fonds sombres

Figure 8.8: Icône arrondie complète de l'application

J'ai choisi la couleur principale de l'application, la nuance de violet présente dans toutes les icônes ainsi que dans plusieurs éléments de l'interface, en m'inspirant de la couleur du logo officiel du Gymnase de Renens :

Figure 8.9: Logo officiel du Gymnase de Renens
`https://www.gyre.ch`

8.4 Structure du serveur

Chaque composant du serveur est expliqué dans les sous-sections à venir, qui présenteront un *"Extrait du code"*, c'est-à-dire une illustration schématique du code du composant. Il est possible de visiter le dépôt GitHub du projet du serveur pour accéder au code complet. Pour comprendre plus facilement le fonctionnement des différents composants du serveur, un Diagramme UML (voir la section Diagrammes 8.2) est également disponible.

Attention : Cette section a été la première rédigée pour ce travail. Le code présenté n'est donc ni complet ni entièrement à jour.

8.4.1 User.cs

Lorsqu'un serveur reçoit une requête, il est souvent nécessaire de pouvoir reconnaître, distinguer et cataloguer l'expéditeur : cela est représenté par la classe `User`, une classe non statique (donc instanciable) contenant toutes les informations utiles concernant l'utilisateur connecté.

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;
                }
```

```
        }
}
```

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.

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.

8.4.2 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.

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
        }
}
```

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`.

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.

8.4.3 Account.cs

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

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? lastLogin,
                bool banned)
        {
                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];

    }
}
```

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.

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 `GetUsernameFromEmail()` extrait le nom d'utilisateur de l'adresse e-mail.

8.4.4 Program.cs (Entry point)

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

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`.

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.

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,
                CancellationToken.None);
}
```

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.

8.4.5 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.

8.4.6 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"*.

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.

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}}", Code.ToString());

        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.

8.4.7 AccountsManager.cs

Elle s'occupe de la gestion des comptes, de leur sauvegarde en mémoire, mais aussi de fournir les résultats lorsqu'un utilisateur recherche un autre compte.

Extrait du code

Les informations sensibles ont été effacés de l'extrait du code et remplacées par "..."

```
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=...;Port=...;Database=...;User=...;Password=...;\r\n";

                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)) r

                        Signal ResponseSignal = new(SignalType.Search);
                        ResponseSignal.Data.Add("Result", JsonConvert.SerializeObject(Sea
                        SearchSignal.Sender?.Receive(ResponseSignal);
                }
        }
}
```

Variables

Dans ce code, la seule variable globale présente est `ConnectionString`. Il s'agit d'une chaîne de texte contenant toutes les informations nécessaires pour établir la connexion avec le serveur SQL. Dans ce cas, la `ConnectionString` permet au programme de se connecter à la base de données *MySQL* exécutée sur la plateforme *Railway.app*. Les informations contenues dans la chaîne sont:
1. `Server` : indique l'adresse du serveur SQL. Cela peut être une adresse *IP*, un nom de domaine, ou bien *"localhost"* si le serveur est exécuté localement. Afin d'éviter des frais inutiles pendant le développement, j'ai choisi cette dernière option.
2. `Port` : indique le port sur lequel le serveur écoute les requêtes SQL.
3. `Database` : correspond au nom de la base de données à laquelle le serveur doit se connecter.
4. `User` et `Password` : les identifiants nécessaires pour accéder à la base de données et éventuellement en modifier les propriétés et le contenu.

Obtention d'un compte

La méthode ci-dessous est principalement appelée par l'`Authenticator` afin d'obtenir un compte ou d'en vérifier l'existence. Pour fonctionner, elle nécessite un paramètre `Username`, c'est-à-dire le nom d'utilisateur du compte concerné. `Result` n'est pas un paramètre classique, comme l'indique le mot-clé *out*, mais un modificateur de paramètre. Je l'ai utilisé ici pour pouvoir retourner deux valeurs à partir d'une seule méthode, ce qui n'est pas possible en ne changeant que le type de retour. En effet, cette méthode renvoie une valeur booléenne indiquant l'existence du compte recherché, tandis que `Result` reçoit comme valeur le "résultat" de l'opération, c'est-à-dire le compte trouvé.

```
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;
}
```

Les actions effectuées dans cette méthode sont typiques d'une requête SQL avec résultat:

- 1. Création de la `Query`, c'est-à-dire la requête à envoyer à la base de données;

- 2. Connexion à la base de données;
- 3. Création d'une instance de `Command`, nécessaire pour gérer la communication avec le serveur SQL;
- 4. Envoi de la commande et attente de la réponse, tous deux gérés par `Command.ExecuteReader()`;
- 5. Lecture de la réponse du serveur: `Reader.Read()` renvoie *true* si le serveur a répondu avec au moins une ligne de texte, *false* sinon;
- 6. Instanciation de la classe `Account`, contenant les informations obtenues;
- 7. Renvoi d'une valeur `true`, indiquant le succès de l'opération.

Le comportement du code ci-dessus est également expliqué en anglais dans les commentaires (*"Connecting to the SQL server"*, *"Creating the command"*, *"Sending the command and reading the response"*).

Création d'un compte

Cette méthode est appelée lorsqu'il est nécessaire de créer un nouveau compte. L'`Authenticator` l'utilise lorsqu'un utilisateur termine la dernière étape de l'inscription. Pour exécuter cette méthode, un seul paramètre est requis: une instance de la classe `Account`, contenant toutes les informations du compte à ajouter à la base de données.

```
public static void AddAccount(Account NewAccount)
{
        if (GetAccount(NewAccount.Username, out var _)) return;

        string Query = "INSERT INTO Account (Username, Password, EmailAddress, CreationDa

        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 ?? D
```

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

La première chose vérifiée lors de l'exécution de la méthode est l'existence d'un autre compte portant le même nom d'utilisateur dans la base de données. Ce contrôle est effectué afin d'éviter tout conflit entre comptes homonymes. Le reste des actions est similaire à la méthode précédente, à la différence que celle-ci n'envoie pas une requête avec résultat, mais une simple commande. Au lieu d'attendre une réponse du serveur, comme vu précédemment avec `Command.ExecuteReader()`, la méthode utilisée est `Command.ExecuteNonQuery()`, qui se contente d'envoyer la commande.

Suppression d'un compte

Méthode appelée pour supprimer un compte. Elle requiert également un seul paramètre: une chaîne `Username` représentant le nom d'utilisateur du compte à supprimer.

```
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 structure est identique à celle de la méthode précédente, puisqu'il s'agit d'une commande sans réponse.

Recherche d'un compte

Méthode appelée lorsqu'un utilisateur, depuis la barre de recherche de la page *"Conversations"* de l'application Messagyre, saisit au moins deux caractères pour chercher un autre utilisateur. Cette méthode prend comme paramètre la chaîne saisie par l'utilisateur dans

sa barre de recherche. À noter que cette chaîne peut ne pas être le nom d'utilisateur complet, mais simplement une partie d'au moins deux caractères de long.

```
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 structure est celle d'une requête avec réponse, à la différence que dans ce cas il peut y avoir plusieurs résultats. La méthode renvoie donc une liste `Results` contenant toutes les valeurs renvoyées par la base de données et lues par `Reader.Read()`, qui lit la réponse ligne par ligne.

8.4.8 Messages.cs

Lorsqu'un utilisateur envoie un message à un autre, ce message est transmis au serveur, qui le traite et le redirige vers le destinataire.

Extrait du code

```
using MessagyreServer.Classes;

public class Messages
```

```
{
        public static void OnMessageSignal(Signal MessageSignal) {...}

        public static void SendMessage(string SenderUsername, string RecipientUsername, s
}
```

Réception d'un message

Lorsque le serveur reçoit un signal dont la variable `Type` vaut `SignalType.Message`, la classe `Server` appelle la méthode `OnMessageSignal()`, en lui fournissant comme paramètre le signal reçu.

```
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? RecipientUse
        if (!MessageSignal.Data.TryGetValue("Content", out string? Content)) return;
        string SenderUsername = MessageSignal.Sender.Account!.Username;

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

Lorsque la méthode est appelée, elle vérifie d'abord si l'utilisateur dont le client a envoyé le signal est *authentifié*, c'est-à-dire connecté à la plateforme. Si ce n'est pas le cas, l'utilisateur ne devrait pas pouvoir envoyer de messages, donc le signal est ignoré. Ensuite, les données sont extraites du signal : le `Username` du destinataire et le contenu (`Content`) du message. Enfin, le message est envoyé en appelant la méthode `SendMessage`, présentée dans la section suivante.

Envoi d'un message

Comme vu précédemment, pour l'envoi d'un message depuis le serveur, la méthode `SendMessage` est appelée. Elle se charge de remettre le message à l'utilisateur, qu'il soit

actuellement connecté à la plateforme ou non. Les paramètres requis sont le nom d'utilisateur de l'expéditeur (`SenderUsername`), celui du destinataire (`RecipientUsername`) et le contenu du message (`Content`).

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

        // Checking if the recipient is online
        User? Recipient = null;
        foreach (User ConnectedUser in Server.ConnectedUsers)
        {
                if (ConnectedUser.Account?.Username == RecipientUsername) Recipient = Cor
        }

        // 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);
        }
}
```

Comme on peut le comprendre à partir des commentaires en anglais, le compte `RecipientAccount` du destinataire est d'abord recherché. Si aucun compte n'est trouvé, l'envoi du message n'est pas effectué. Ensuite, le serveur vérifie si l'utilisateur destinataire est actuellement en ligne, en parcourant la liste des utilisateurs connectés (`ConnectedUsers`) contenue dans la classe `Server` , afin de déterminer le traitement à effectuer : Si l'utilisateur est en ligne, le message lui est directement envoyé via la méthode `Receive` de la classe `User` . Sinon,

s'il n'apparaît pas dans la liste des utilisateurs connectés, le message est ajouté à la liste `Inbox` de son compte, afin qu'il lui soit transmis lors de sa prochaine reconnexion.

9 Sources et bibliographie

Pour mener à bien le développement de Messagyre, je me suis appuyé sur diverses ressources disponibles en ligne. Voici les principales :

[1] Microsoft Corporation. *Microsoft Teams*. Solution de communication et collaboration utilisée pour comparaison. URL: `https://www.microsoft.com/en/microsoft-teams/group-chat-software`.

[2] WhatsApp LLC. *WhatsApp*. Application de messagerie instantanée populaire utilisée pour comparaison. URL: `https://www.whatsapp.com`.

[3] *C Sharp*. Langage de programmation utilisé pour le serveur. URL: `https://learn.microsoft.com/en-us/dotnet/csharp/`.

[4] *Node.js*. Utilisé pour prototyper rapidement le backend initial. URL: `https://nodejs.org/`.

[5] *Dart*. Langage de programmation utilisé avec Flutter. URL: `https://dart.dev/`.

[6] *Visual Studio*. IDE principal pour le développement du serveur. URL: `https://visualstudio.microsoft.com/`.

[7] *ASP.NET Core*. Framework serveur robuste utilisé pour le backend. URL: `https://dotnet.microsoft.com/apps/aspnet`.

[8] *Flutter*. Framework utilisé pour le développement du client mobile multiplateforme. URL: `https://flutter.dev/`.

[9] Telegram FZ-LLC. *Telegram*. Application de messagerie avec fonctionnalités avancées. URL: `https://telegram.org`.

[10] Signal Foundation. *Signal*. Application axée sur la sécurité et la confidentialité. URL: `https://signal.org`.

[11] Google LLC. *Google*. Moteur de recherche. URL: `https://www.google.com/`.

[12] *Microsoft ASP.NET Core*. Framework utilisé pour le développement du serveur backend. URL: `https://learn.microsoft.com/aspnet/core`.

[13] *HTTP*. Protocole utilisé pour les requêtes classiques. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP.

[14] *WebSocket*. Protocole utilisé pour la messagerie instantanée en temps réel. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.

[15] *MySQL*. Base de données relationnelle pour stocker utilisateurs et messages. URL: https://www.mysql.com/.

[16] *Railway*. Plateforme cloud pour héberger la base de données. URL: https://railway.app/.

[17] *Fly.io*. Plateforme cloud utilisée pour héberger le serveur. URL: https://fly.io/.

[18] *Hive*. Gestion locale des données sur l'appareil (historique messages). URL: https://docs.hivedb.dev/.

[19] *Firebase*. Service pour gérer les notifications push. URL: https://firebase.google.com/.

[20] *Simple Mail Transfer Protocol (SMTP)*. Utilisé pour l'envoi d'emails via un serveur de messagerie. URL: https://datatracker.ietf.org/doc/html/rfc5321.

[21] *Gmail API*. Utilisé pour envoyer les emails de vérification. URL: https://developers.google.com/gmail/api.

[22] *Cloudinary*. Hébergement des photos de profil. URL: https://cloudinary.com/.

[23] *Remove.bg*. Outil utilisé pour créer et nettoyer les icônes. URL: https://www.remove.bg/.

[24] *Pixlr E*. Éditeur en ligne pour retouches d'images. URL: https://pixlr.com/e/.

[25] *Canva*. Outil pour créer les éléments graphiques de l'application. URL: https://www.canva.com/.

[26] *Adobe Express*. Outil pour créer et retoucher les éléments graphiques de l'application. URL: https://www.adobe.com/fr/express/.

[27] *Git*. Gestion de versions du code. URL: https://git-scm.com/.

[28] *GitHub*. Plateforme pour le versionnage et la gestion du projet. URL: https://github.com/.

[29] *Visual Studio Code*. Éditeur utilisé pour le développement du client et scripts. URL: https://code.visualstudio.com/.

[30] *HTTPS*. Protocole sécurisé pour la transmission des données sur Internet. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

[31] *Flutter Secure Storage*. Librairie officielle de Flutter pour stocker et chiffrer des données. URL: https://pub.dev/packages/flutter_secure_storage.

[32] *Universally Unique Identifier (UUID)*. Utilisé pour générer des identifiants uniques dans l'application. URL: `https://datatracker.ietf.org/doc/html/rfc4122`.

[33] *bcrypt*. Algorithme utilisé pour le hachage sécurisé des mots de passe. URL: `https://en.wikipedia.org/wiki/Bcrypt`.

[34] *JSON Web Token (JWT)*. Utilisé pour l'authentification et la gestion des sessions utilisateurs. URL: `https://datatracker.ietf.org/doc/html/rfc7519`.

[35] OWASP Foundation. *OWASP Top Ten*. Ressource de référence pour la sécurité des applications web. URL: `https://owasp.org/www-project-top-ten/`.

[36] *Android Studio*. IDE pour le développement d'applications pour Android utilisé pour tester Messagyre. URL: `https://developer.android.com/studio?hl=fr`.

[37] *Google Chrome*. Moteur de recherche utilisé pour l'envoi de requêtes HTTP et HTTPS. URL: `https://www.google.com/intl/fr/chrome/`.

[38] *MySQL Workbench 8.0 CE*. Logiciel officiel pour gérer la base de données. URL: `https://www.mysql.com/products/workbench/`.

9.1 Documentations officielles consultées

- *Flutter* — `https://docs.flutter.dev`
- *Flutter API* — `https://api.flutter.dev`
- *Dart* — `https://dart.dev/guides`
- *ASP.NET Core* — `https://learn.microsoft.com/fr-fr/aspnet/core`
- *C#* (syntaxe et curiosités) — `https://learn.microsoft.com/en-us/dotnet/csharp/`
- *MySQL* — `https://dev.mysql.com/doc/`
- *JWT (JSON Web Tokens)* — `https://jwt.io/introduction`
- *Railway* — `https://docs.railway.app`
- *Fly.io* — `https://fly.io/docs/` : plateforme utilisée pour déployer le serveur ASP.NET.
- *pub.dev* — `https://pub.dev` : plateforme utilisée pour trouver et installer des paquets et extensions Flutter.
- *Gmail API* — `https://developers.google.com/gmail/api` : utilisée pour l'envoi d'emails de vérification.

- *Bcrypt* — `https://github.com/BCryptNet/bcrypt.net` : utilisée pour le hachage sécurisé des mots de passe.
- *UUID* — `https://www.uuidgenerator.net/` : utilisée pour générer des identifiants uniques.

9.2 Tutoriels, articles et forums

- Stack Overflow — `https://stackoverflow.com`
- GitHub Discussions — `https://github.com`
- Flutter Community — `https://fluttercommunity.dev`
- Railway Community Forum — `https://community.railway.app`
- TeX Stack Exchange — `https://tex.stackexchange.com`
- Reddit (r/FlutterDev, r/dotnet, r/webdev) — `https://www.reddit.com`
- Wikipédia — pages sur *Base64* et *JSON Web Token* pour la compréhension des concepts fondamentaux.
- Chaine YouTube officielle de Flutter `https://www.youtube.com/@flutterdev`
- Installation des drivers Android et utilisation d'ADB — `https://androidmtk.com`

9.3 Outils d'assistance

- ChatGPT (OpenAI) — pour générer du code, trouver des solutions optimisées et explorer des alternatives techniques.
- Google Gemini — pour la recherche de documentation complémentaire et l'exploration d'approches différentes.
- Claude (Anthropic) — pour obtenir des résumés techniques et des explications de concepts complexes.
- GitHub Copilot — utilisé via Visual Studio Code pour l'auto-complétion de code, facilitant et accélérant le développement.

9.4 Outils de développement

- Visual Studio 2022 — environnement de développement principal pour ASP.NET Core et le serveur.
- Visual Studio Code — utilisé pour éditer des fichiers Dart, JSON, SQL et pour les tests rapides côté client.
- Android Studio — utilisé pour installer les SDK Android et tester l'application Flutter via son émulateur intégré.
- Apple XCode — utilisé pour construire l'application pour les dispositifs Apple (iOS).

9.5 Autres inspirations et connaissances utiles

- Cours de mathématiques sur la cryptographie (Gymnase de Renens) — a contribué à mieux comprendre les principes de sécurité appliqués à l'application (notamment l'utilisation des tokens, des mots de passe hachés et de la confidentialité des échanges).
- Patternico — utilisé pour générer les motifs de fond dans les fenêtres de discussion.
- Google Maps — utilisé pour dessiner l'icône de l'application à partir de la vue aérienne du bâtiment du gymnase.