Missive - Application de messagerie sécurisée

Anthony Rodriguez - Technicien ES 2023/2024 - Supervisé par M. Francisco Garcia

Table des matières

1. Cahier des charges 4
1.1 Contexte 4
1.2 Description du projet 4
1.3 Analyse de l'existant 4
1.4 Plus value 5
1.5 Spécifications techniques 5
1.6 Technologies utilisées 6
1.7 Budget 6
2. Fonctionnement 8
2.1 Analyse fonctionnelle 8
2.2 Analyse organique 15
2.3 Intégration continue 37
2.4 Déploiement 37
2.5 Justifications techniques et réflexions 38
2.6 Références 40
3. Plan de tests 41
3.1 API 41
3.2 Serveur WebSocket 44
3.3 Client 47
4. Bilan personnel 52
4.1 Bilan personnel 52
5. Glossaire 55
6. Table des figures 59
7. Journal de bord 64
7.1 2024-03-27 64
7.2 2024-03-28 64
7.3 2024-03-29 65
7.4 2024-05-02 65
7.5 2024-05-03 65
7.6 2024-05-04 66
7.7 2024-05-05 67
7.8 2024-05-08 67
7.9 2024-05-09 68
7.10 2024-05-11 68
7.11 2024-05-12 68

7.12 2024-05-15 71
7.13 2024-05-16 71
7.14 2024-05-17 71
7.15 2024-05-18 73
7.16 2024-05-19 74
7.17 2024-05-21 75
7.18 2024-05-22 76
7.19 2024-05-23 78
7.20 2024-05-24 78
7.21 2024-05-25 78
7.22 2024-05-26 79
7.23 2024-05-28 81
7.24 2024-05-29 82
7.25 2024-05-30 82
7.26 2024-06-01 83
7.27 2024-06-02 83
7.28 2024-06-03 85
7.29 2024-06-05 86
7.30 2024-06-06 87
7.31 2024-06-07 87
7.32 2024-06-08 87
7.33 2024-06-09 88
7.34 2024-06-13 88
7.35 2024-06-14 88
7.36 2024-06-15 88
7.37 2024-06-16 89
7.38 2024-06-17 89
7.39 2024-06-21 90
7.40 2024-06-22 90
7.41 2024-06-23 90
7.42 2024-06-24 91
7.43 2024-06-27 91
7.44 2024-06-28 91
7.45 2024-06-30 92
7.46 2024-06-31 92
7.47 2024-07-03 93

1. Cahier des charges

1.1 Contexte

Durant la deuxième et dernière année du diplôme de technicien ES en développement d'applications, il est demandé aux étudiants de réaliser un travail de diplôme. Ce travail consiste en la réalisation d'un projet informatique, qui doit être réalisé seul, et qui doit être présenté à la fin de l'année. Ce projet doit être réalisé en 10 semaines, et doit être accompagné d'un journal de bord, qui décrit les différentes étapes de réalisation du projet, les choix techniques et les difficultés rencontrées, et d'une documentation utilisateur·rice, qui décrit le fonctionnement de l'application et les différentes fonctionnalités dans un langage accessible au type d'utilisateur·rice visé·e.

1.2 Description du projet

L'objectif principal de ce projet est de développer une application de messagerie sécurisée qui garantit la confidentialité des messages échangés entre les utilisateur·rice·s grâce à l'implémentation du protocole de chiffrement Signal. L'application sera conçue pour fonctionner sur la majorité des plateformes (iOS, Android, Windows, macOS, Linux). Elle permettra aux utilisateur·rice·s de communiquer en temps réel de manière sécurisée, de synchroniser leurs conversations entre plusieurs appareils, et d'être disponible sur les principaux app stores. De plus, l'application sera hébergée chez Infomaniak pour garantir la disponibilité et la sécurité des données des utilisateur·rice·s.

1.3 Analyse de l'existant

Les applications de chat étant nombreuses, j'ai décidé de me pencher sur deux de ces applications qui utilisent la sécurité comme un de leurs arguments principaux. Ces deux applications sont *Telegram* et *Signal*.

1.3.1 Telegram

Telegram est une application de messagerie sécurisée qui utilise le protocole MTProto, qui est un protocole de chiffrement de bout en bout. Cependant, cette application stocke les messages sur son serveur, ce qui peut potentiellement poser problème en ce qui concerne la confidentialité des messages.

De plus, Telegram n'est pas chiffré de bout en bout par défaut, et il est nécessaire de créer un chat secret pour pouvoir bénéficier de cette fonctionnalité. Cela peut poser problème, car il est possible que les utilisateur·rice·s ne soient pas au courant de cette fonctionnalité, et que leurs messages passent dont en clair sur les serveurs de Telegram.

1.3.2 Signal

Signal est l'application qui se rapproche le plus de la sécurité, car elle utilise le protocole Signal, qui est un protocole de chiffrement de bout en bout, et qui est open source.

Un de ses points forts est le fait que Signal ne stocke aucunement les messages de l'utilisateur·rice, et que ces derniers sont chiffrés de bout en bout par défaut. Le seul cas où le message va être stocké sur le serveur est de manière temporaire, en attente de réception par le destinataire dans le cas où ce dernier est hors-ligne.

Cependant, Signal n'est malheureusement pas complètement open-source (une partie des serveurs est propriétaire). Cela peut poser quelques soucis en ce qui concerne la confiance des utilisateurs, malgré la réputation très sûre et positive de l'organisation. De plus, Signal est hébergée par Amazon Web Services, qui est une entreprise américaine, et qui est donc soumise à la législation américaine, qui n'est pas forcément la plus respectueuse de la vie privée (pays membre du traité de *14 eyes* / UKUSA).

1.4 Plus value

Après avoir analysé les applications actuelles de messagerie sécurisée, il y a quelques points qui pourraient être améliorés, et qui pourraient apporter une plus-value à l'application que je souhaite réaliser :

- Confidentialité des messages : Les messages seront chiffrés de bout en bout par défaut, et ne seront jamais stockés sur le serveur (seulement de manière temporaire en cas de non-réception par le destinataire)
- Open-source : L'application sera complètement open-source, ce qui permettra de garantir la confiance des utilisateur·rice·s, et de permettre à des tiers de vérifier la sécurité de l'application et de l'améliorer
- Hébergement en Suisse : L'application sera hébergée chez Infomaniak, qui est une entreprise suisse, et qui est donc soumise à la législation suisse, qui est beaucoup plus respectueuse de la vie privée que la législation américaine

1.5 Spécifications techniques

1.5.1 Messagerie sécurisée

L'application doit permettre aux utilisateur·rice·s d'envoyer des messages chiffrés de bout en bout à d'autres personnes. Ce chiffrement sera assuré par le protocole Signal. Ce dernier a été retenu car il est tout d'abord open-source (il est très facile de trouver des implémentations dans différents langages), et la documentation du protocole est extrêmement bien fournie. Une explication de ce protocole est disponible sur la documentation de Signal.

Communication en temps réel

Elle fonctionnera avec la mise en place d'un serveur WebSocket, qui permettra l'échange des messages en temps réel, ainsi qu'une API plus classique, qui permettra de gérer les comptes utilisateur·rice·s, les connexions, ainsi que les messages stockés temporairement en attente de réception. Ils fonctionneront en parallèle : le serveur de WebSocket utilisera également l'API afin de pouvoir gérer les messages en attente de réception. Le serveur de WebSocket ainsi que l'API seront protégés par les mécanismes classiques au web (entête CORS afin d'éviter les CSRF / requêtes depuis des périphériques non autorisés, authentification avec jeton pour s'assurer de l'identité des utilisateurs, et chiffrement SSL avec HTTPS).

Le serveur ne stockera jamais d'autres informations que les informations de connexion, et les messages en attente de réception. Les messages seront supprimés dès qu'ils auront été reçus par le destinataire. Cela permettra de garantir la confidentialité des messages, même dans le cas où le serveur serait compromis.

1.5.2 Multiplateforme

L'application doit être compatible avec les principales plateformes mobiles (iOS / Android), ainsi que les différentes plateformes bureautiques (Windows macOS Linux). Le choix de Flutter permettra de garder une seule base qui permettra ensuite de build automatiquement l'application, notamment grâce à fastlane.

1.5.3 Hébergement

Les différents serveurs de l'application seront hébergés sur les serveurs / VPS de chez Infomaniak. Cette dernière étant basée sur la confidentialité des données, le choix d'un hébergeur suisse est donc crucial même pour ce projet, où les données ne seront conservées que sur le client.

1.5.4 Déploiement

Le déploiement de l'application sera complètement automatisé grâce à fastlane, un service acquéri en 2017 par Google. Il permet le build automatique, le déploiement en accès anticipé sur les différentes plateformes mobiles (TestFlight, Play Console), la soumission sur les app stores, et il est complètement intégrable avec GitLab Runner pour autant que le serveur soit sous macOS (afin de pouvoir build pour iOS).

1.5.5 Tests et validation

Une batterie de tests unitaires et E2E sera déployée grâce aux outils de test de Flutter pour le client, et un framework plus classique pour la partie serveur. Cela permettra de garantir le bon fonctionnement du projet, et de pouvoir détecter les éventuels bugs et problèmes de sécurité avant même de déployer l'application. Ces tests seront automatiquement exécutés avec la pipeline CI/CD, afin de garantir que chaque commit est fonctionnel.

1.5.6 Maintenance

La maintenance de l'application sera assurée grâce à des mises à jour automatiquement publiées sur les app stores grâce à fastlane. Cela permettra de garantir la sécurité des utilisateur·rice·s, et de corriger les éventuels bugs qui pourraient être découverts.

1.6 Technologies utilisées

1.6.1 Flutter

Le framework Flutter sera utilisé afin de réaliser le client. Ce dernier permettra de pouvoir réaliser une application multi-plateformes, qui fonctionne sur iOS, Android, Windows, macOS et Linux. L'utilisation de React Native a été considérée, mais Flutter a été retenu car la bibliothèque du protocole Signal ne fonctionnait pas (WebCrypto, l'API de chiffrement, n'est malheureusement pas disponible sur React Native, malgré les tentatives de polyfills et d'installation de différentes solutions alternatives).

1.6.2 Typescript

Le back-end de l'application sera réalisé en Typescript, couplé à Bun, nouveau runtime qui offre des performances nettement supérieures à Node.JS, et qui est disponible en version 1.x depuis maintenant quelques mois. Typescript a été retenu pour sa facilité d'utilisation, sa solidité, et sa compatibilité avec les différentes bibliothèques utilisées.

1.6.3 Docker

Docker sera utilisé afin de pouvoir déployer le back-end de manière simple et reproductible. Un docker-compose sera réalisé, qui permettra de déployer l'application en une seule commande, et également de la mettre à jour avec les différents tags et versions proposés.

1.7 Budget

1.7.1 Hébergement Jelastic Cloud Infomaniak

L'application mobile sera hébergée sur la plateforme Jelastic Cloud d'Infomaniak. Les coûts d'hébergement débutent à partir de 20 .- par mois, mais ils seront adaptables en fonction de la consommation de ressources et de la configuration sélectionnée.

Coût estimé : À partir de 20 .- par mois (ajustable selon la consommation)

1.7.2 Licence Développeur Apple

Une licence de développeur Apple, au coût de 99 $ par an, sera nécessaire pour accéder aux outils de développement, effectuer des tests sur des appareils réels, et soumettre l'application à l'examen d'Apple.

Coût estimé : 99 $ par an

1.7.3 Licence Développeur Android

Pour la publication de l'application sur le Google Play Store, l'obtention d'une licence de développeur Android est requise. Cette licence est un paiement unique de 25 $.

Coût estimé : 25 $ (unique)

1.7.4 Runner Gitlab macOS

Un runner Gitlab macOS sera utilisé pour automatiser la construction de l'application iOS. Les coûts associés commencent à partir de 15 .- par mois, mais peuvent varier en fonction des spécifications et de l'utilisation.

Coût estimé : À partir de 15 .- par mois (variable selon les spécifications)

2. Fonctionnement

Le fonctionnement de Missive repose sur l'efficacité du protocole Signal, qui est décrit sur cette page. Vous trouverez ci-dessous un schéma d'utilisation classique, de la création du compte utilisateur à l'envoi et la réception d'un message.

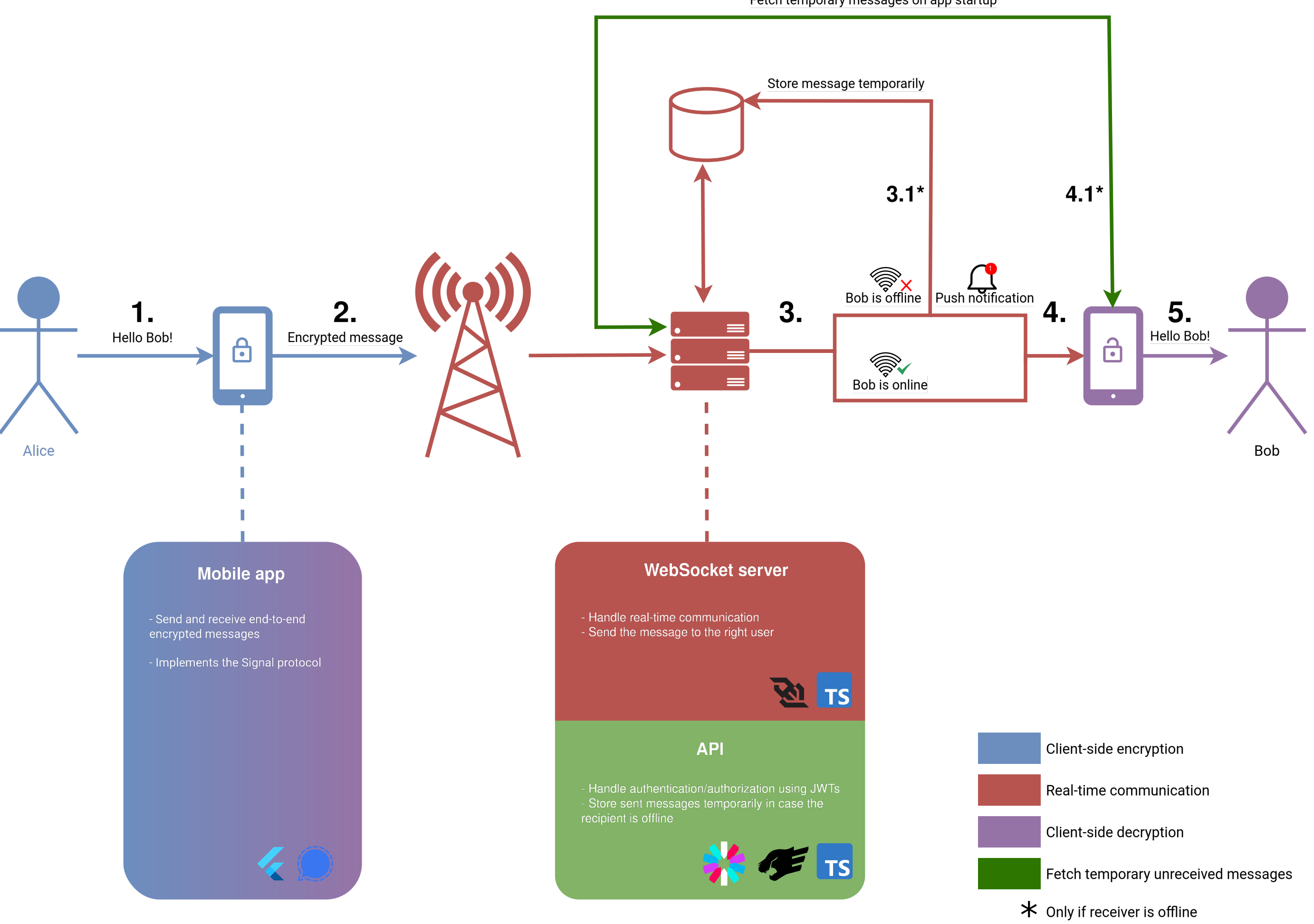


Figure 2 — Schéma de l'architecture de l'application

2.1 Analyse fonctionnelle

Pour mieux comprendre le fonctionnement global, nous allons étudier un cas d'utilisation typique, de la création du compte utilisateur à l'envoi et la réception d'un message.

2.1.1 Création du compte

La création du compte s'effectue de la manière suivante : l'utilisateur rentre ses informations, et les confirme. Un ID d'enregistrement et une paire de clés d'identités sont générées. Toutes ces informations sont ensuite envoyées à l'API (seulement la partie publique de la clé). L'API vérifie que les informations sont correctes, puis crée un compte utilisateur. Après la confirmation de la création du compte, le reste des clés nécessaires au fonctionnement du protocole sont ensuite générées, envoyées à l'API, et stockées en base de données.

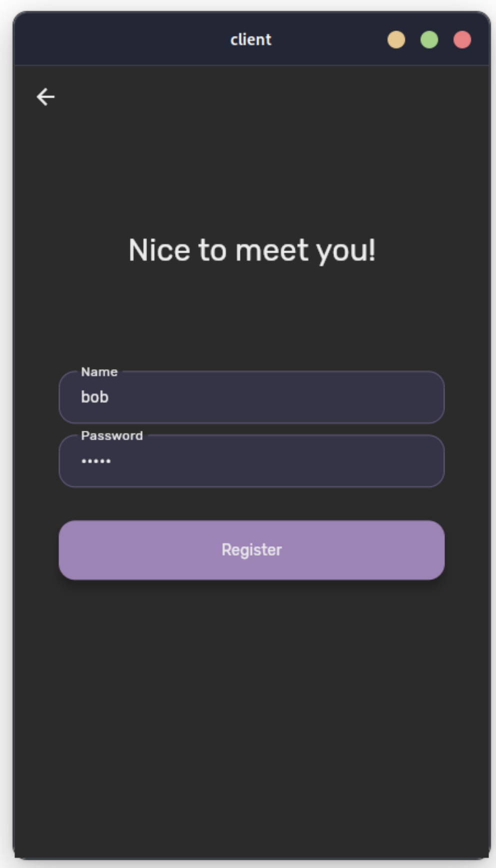


Figure 3 — Écran de création du compte

Un jeton d'accès et de rafraîchissement sont ensuite envoyés à l'utilisateur, qui lui permet de se connecter à l'application, ainsi que de rafraîchir son jeton d'accès, qui a une durée de vie de 15 minutes. Ce jeton de rafraîchissement est crucial car il est stocké en base de données avec l'ID de l'utilisateur, et peut donc être révoqué en cas de perte ou de vol du compte.

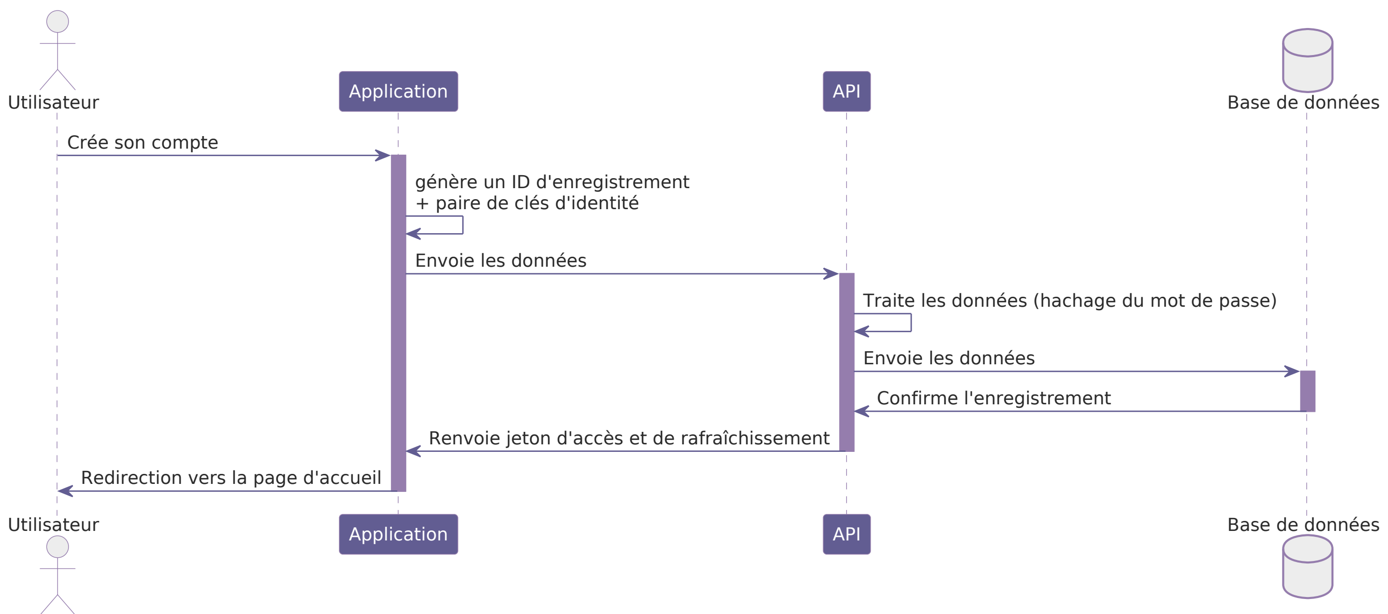


Figure 4 — Diagramme de séquence de la création d'un compte

Une fois le compte de l'utilisateur créé, il est redirigé sur la page d'accueil de l'application, d'où commence la prochaine étape, la génération des clés.

2.1.2 Génération des clés

Une fois que l'utilisateur arrive sur la page d'accueil, que ce soit après la création de son compte ou après s'être connecté, une vérification du statut de l'application est effectuée. Si ce compte vient d'être accédé pour la première fois sur ce périphérique, alors une génération des clés s'effectue. Ce processus permet de s'assurer que les différentes clés soient bien disponibles, et qu'il n'y ait aucun conflit.

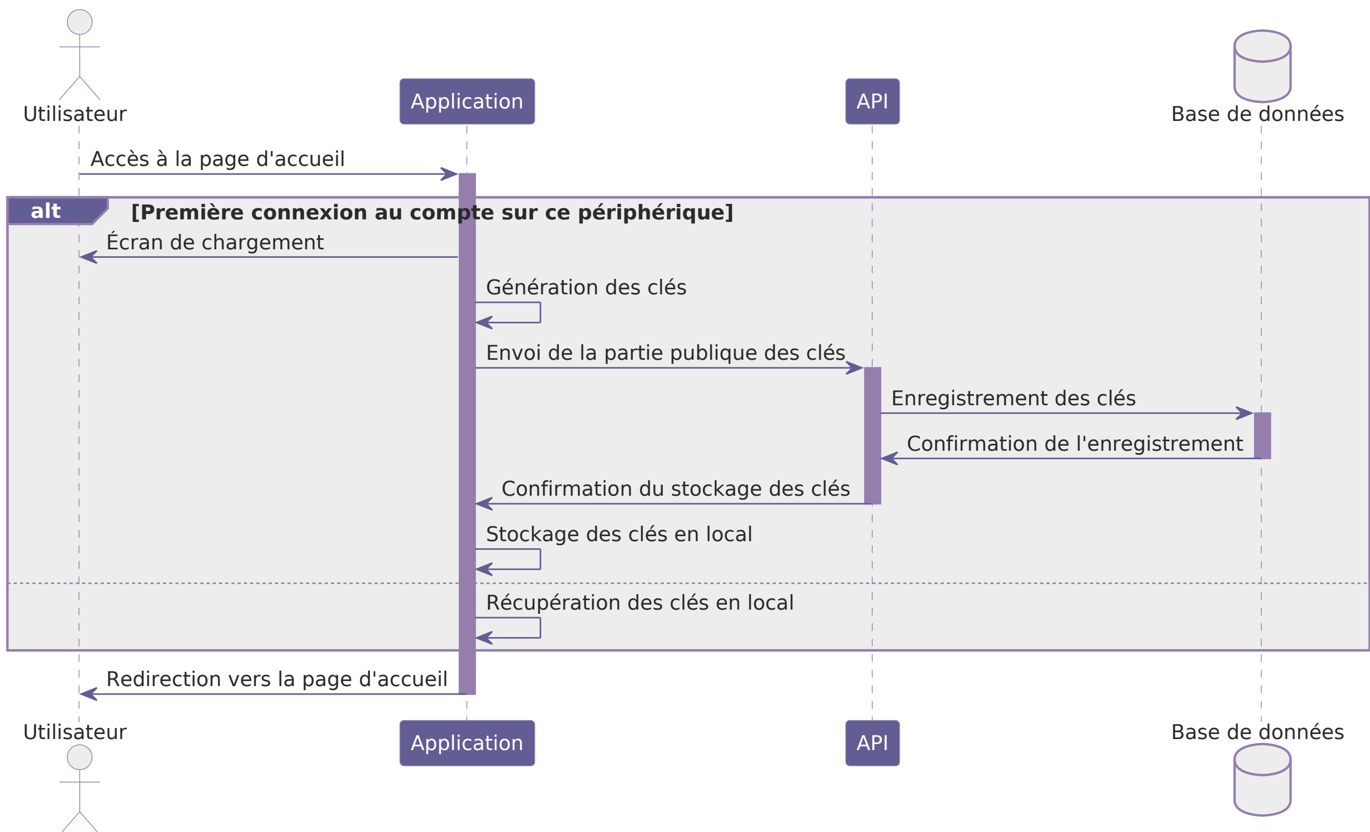


Figure 5 — Diagramme de séquence de la génération des clés

En résumé, les différentes clés privées et publiques sont générées, stockées dans le stockage sécurisé du système d'exploitation, puis envoyées à l'API pour être stockées en base de données. Il est nécessaire que tous les utilisateurs puissent avoir accès aux clés publiques d'un autre utilisateur afin d'établir une connexion et autorisation initiale. Une fois que les clés sont stockées, l'utilisateur peut commencer à envoyer et recevoir des messages.

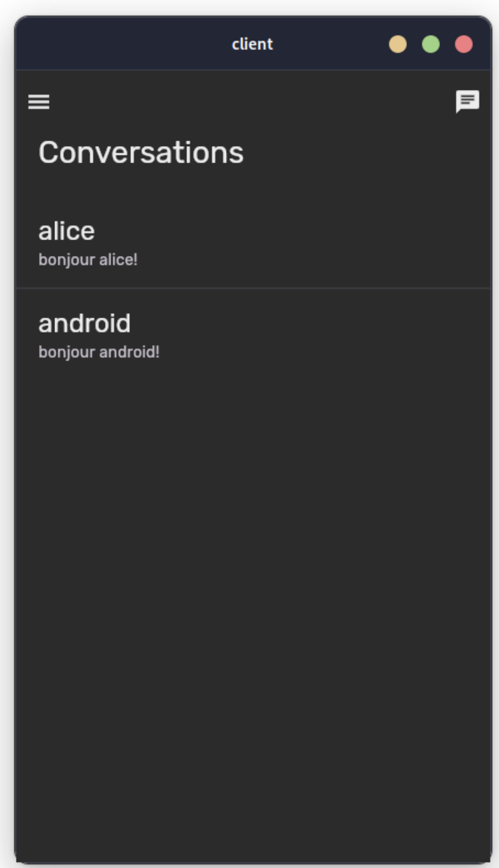


Figure 6 — Écran des conversations

2.1.3 Début de la conversation

Quand un utilisateur•trice souhaite envoyer un message à un•e autre utilisateur•trice pour la première fois, il/elle dispose d'un bouton en haut à droite de l'application pour commencer une conversation. Il/elle rentre le nom de l'utilisateur•trice destinataire, qui sera cherché•e en temps réel dans la base de données. Une fois l'utilisateur•trice trouvé•e, une conversation est créée, stockée en local dans une base de données et l'utilisateur•trice peut envoyer un message. Cette dernière viendra également se rajouter sur la page d'accueil afin que l'utilisateur•trice puisse y accéder facilement.

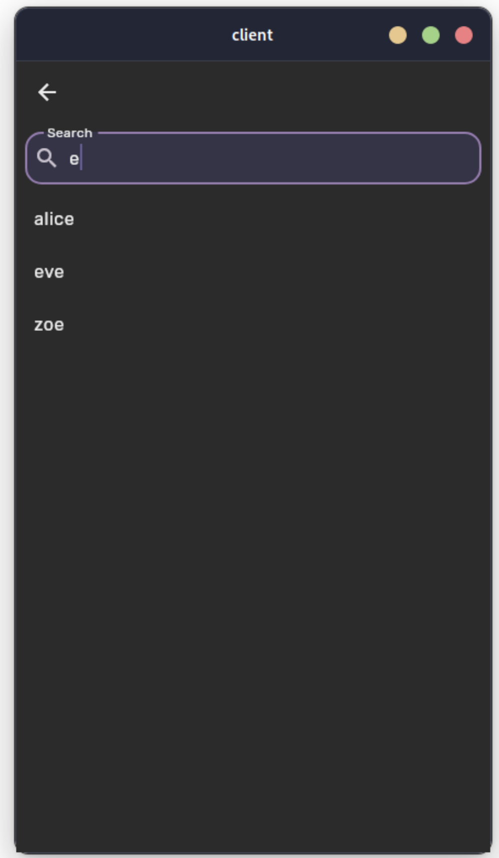


Figure 7 — Écran de recherche d'un utilisateur

2.1.4 Envoi du message

Lorsque l'utilisateur•trice envoie un message, ce dernier est chiffré en utilisant le protocole Signal, puis envoyé au serveur de WebSocket. Ce dernier vérifie que l'utilisateur est bien connecté, puis envoie le message à l'utilisateur destinataire. Si ce dernier est connecté, le message est directement envoyé à l'utilisateur•trice. Sinon, le message est stocké en base de données, et sera récupéré dès que l'utilisateur•trice se connectera.

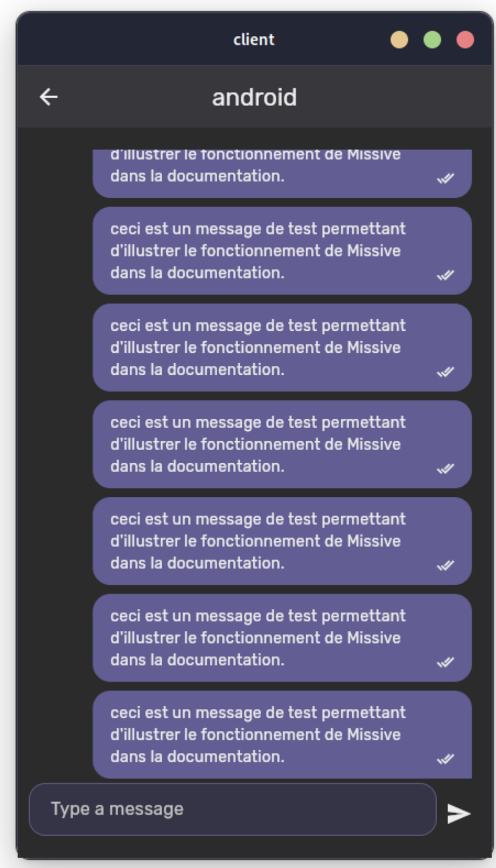


Figure 8 — Écran de conversation

À la réception de ce message par le destinataire (la réception étant soit en temps réel si l'utilisateur est sur l'application, soit en différé s'il n'est pas connecté), l'envoyeur est directement notifié, et le changement de statut est reflété sur l'application par des petites coches à droite du message. Une notification push est également envoyée au destinataire pour lui indiquer qu'un nouveau message est disponible.

2.1.5 Réception du message

Maintenant que l'expéditeur a envoyé son message, il est temps pour le destinataire de le récupérer. Ce dernier se connecte sur son application, ou l'ouvre simplement s'il est déjà connecté. Une vérification des messages en attente est effectuée à chaque démarrage de l'application. Si le destinataire a des messages en attente, ils sont récupérés depuis l'API, déchiffrés, et affichés à l'utilisateur. Cela permet de garantir que l'utilisateur ne rate aucun message, même s'il n'était pas connecté à l'application.

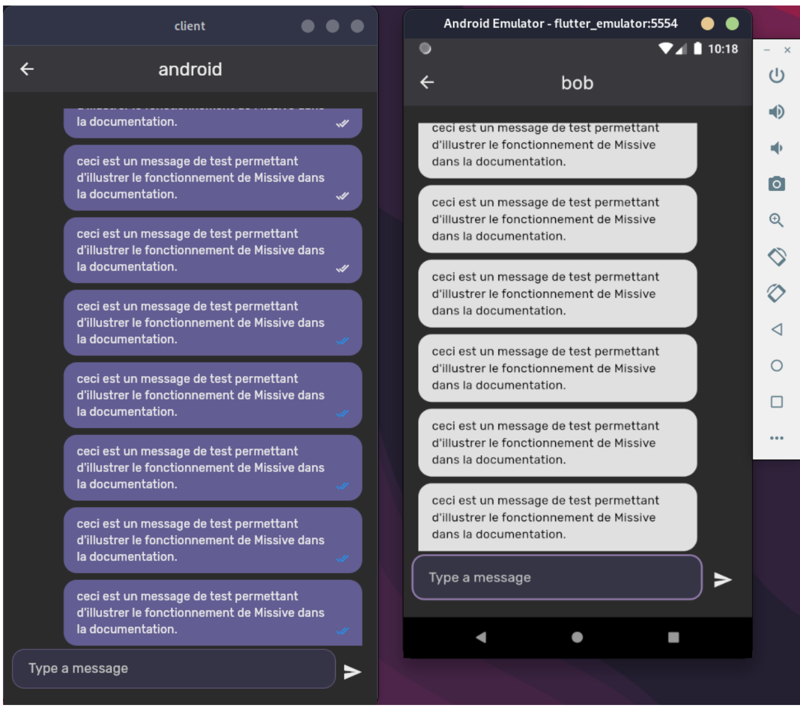


Figure 9 — Écrans de conversation - messages lus

Dans le cas où l'utilisateur était déjà présent sur l'application, le message lui est directement envoyé, et est traité en temps réel par l'application.

Quand l'utilisateur lit le message (le message est affiché à l'écran, dans l'écran de conversation), un message de confirmation est envoyé à l'expéditeur pour lui indiquer que le message a bien été lu. Cette confirmation lui est soit envoyée directement s'il est sur l'application, soit stockée en base de données s'il n'est pas connecté, et sera récupérée dès qu'il se connectera.

2.2 Analyse organique

L'application comporte trois parties distinctes : le client, qui est l'application mobile réalisée en Flutter, l'API, qui est une API REST en TypeScript, ainsi qu'un serveur de WebSocket, qui est lui aussi en TypeScript. Le client peut communiquer avec l'API pour la partie autorisation (gestion de la connexion à l'application, de l'authentification en 2 étapes...), ainsi que la réception des messages depuis le serveur si l'on était hors-ligne, et avec le serveur de WebSocket pour la partie communication en temps réel. Vous trouverez ci-dessous un schéma de l'architecture de l'application.

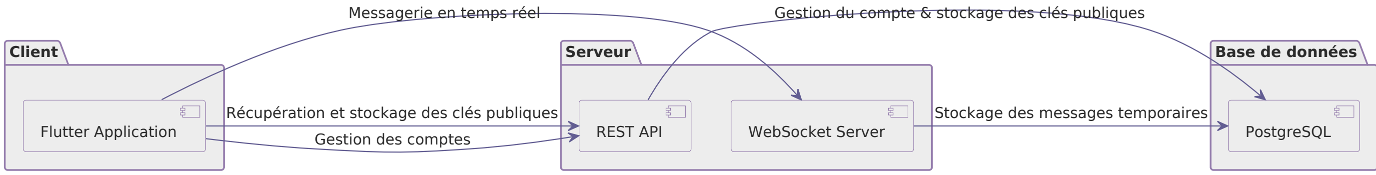


Figure 10 — Diagramme de composants de l'application

1. Le client envoie un message à un•e utilisateur•trice via son application. Ce message est chiffré au niveau de l'application grâce au protocole Signal.
2. Le message est envoyé grâce à une connexion WebSocket au serveur.
3. Le serveur vérifie l'authentification et les permissions de l'utilisateur•trice grâce au jeton d'accès, puis regarde si la•le destinataire est connecté•e.
   - Si la•le destinataire est connecté•e, le message est envoyé directement à l'utilisateur•trice.
   - Si la•le destinataire n'est pas connecté•e, le message est stocké dans la base de données, et sera envoyé dès que la•le destinataire se connectera. Une notification est également envoyée.
4. L'utilisateur•trice reçoit le message, qui est déchiffré au niveau de l'application grâce au protocole Signal.

a. Au démarrage de l'application, l'utilisateur•trice se connecte à l'API pour récupérer les messages en attente.

Le fonctionnement technique de ces différentes parties est détaillé ci-dessous.

2.2.1 Client

Le client est l'application mobile, réalisée en Flutter. Il est le point d'entrée de l'utilisateur·rice, et permet de communiquer avec l'API et le serveur de WebSocket. Il permet aussi de gérer le chiffrement, le déchiffrement, et la signature des messages. Flutter a été retenu pour sa rapidité de développement, son expérience développeur, sa facilité de déploiement, et sa capacité à gérer les différentes plateformes (iOS, Android, Web, Desktop). Il permet également de gérer les mises à jour de manière efficace, et de gérer les différentes versions de l'application.

Concepts Flutter

Avant de commencer à parler du fonctionnement de l'application, il est important de comprendre certains concepts de Flutter, qui est le framework utilisé pour le développement de l'application. Ces concepts vont beaucoup revenir par la suite.

PROVIDER

Une application front-end doit gérer un état global, qui permet de gérer les différentes parties de l'application. Par exemple, dans le cas de Missive, il faut un moyen de gérer les différentes clés, les messages, les utilisateurs, etc. Il faut également que l'application puisse réagir à certains de ces événements (ie. réception d'un message, envoi d'un message, etc.).

Pour palier à ce problème, Flutter propose un système de gestion de l'état global, appelé Provider. Ce dernier permet de gérer l'état global de l'application, et de le partager entre les différentes parties de l'application. Un Provider est une classe, qui étend la classe `ChangeNotifier`, et qui permet de notifier les différentes parties de l'application lorsqu'un changement d'état a lieu.

Pour Missive, trois Provider sont disponibles :

- `AuthProvider` : permet de gérer l'authentification de l'utilisateur·rice, ainsi que son compte utilisateur.
- `SignalProvider` : propose une interface plus haut niveau de mon implémentation du protocole Signal, et permet de chiffrer, déchiffrer, et signer les messages. Il interagit également avec le SecureStorage pour stocker les clés de manière sécurisée (sérialisation / désérialisation).
- `ChatProvider` : permet de gérer la communication en temps réel avec le serveur de WebSocket.

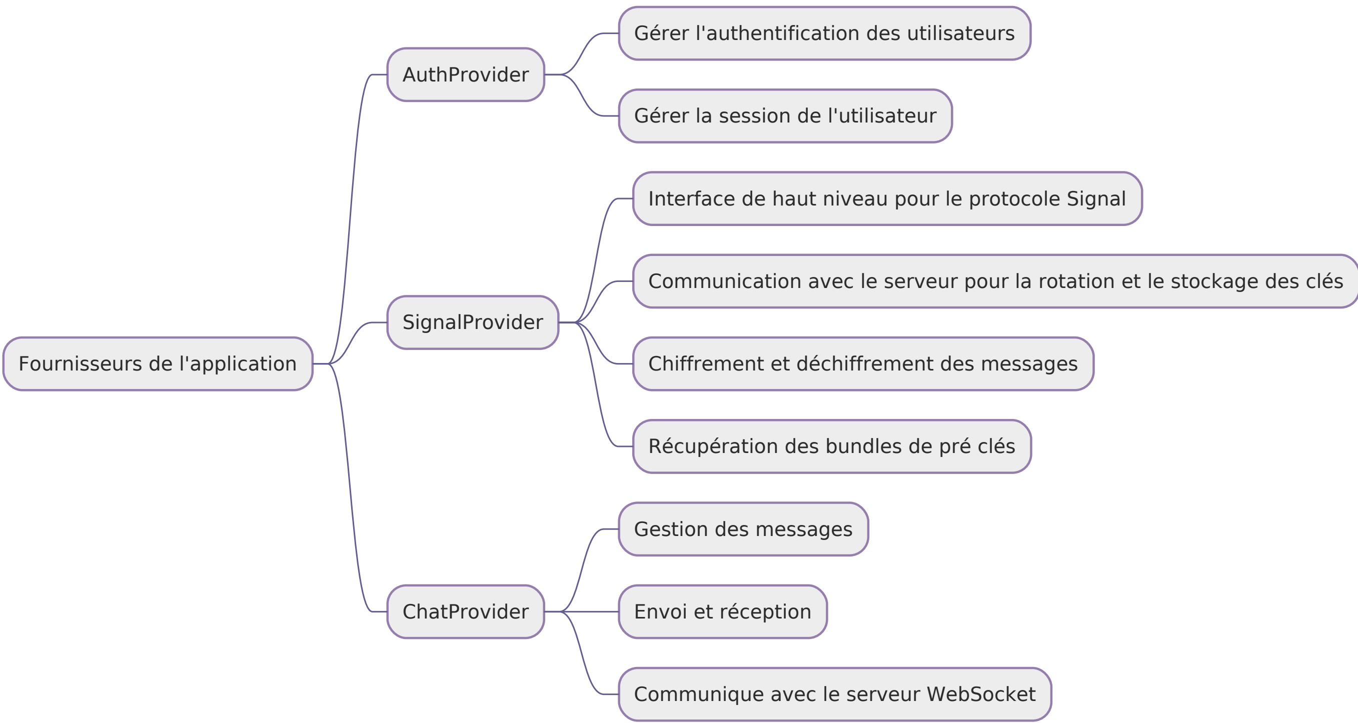


Figure 11 — Mindmap du fonctionnement des Provider

Provider a été retenu pour sa simplicité, et la possibilité de séparer clairement la logique de l'interface de manière efficace.

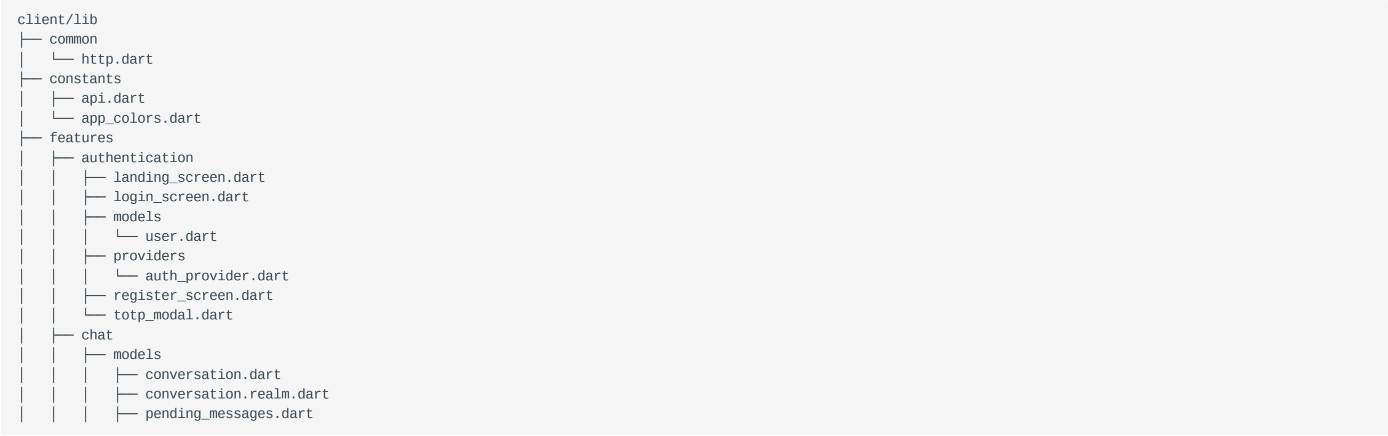
FLUTTER_SECURE_STORAGE

flutter_secure_storage est une bibliothèque Flutter qui permet de persister des données de manière sécurisée dans le stockage sécurisé du système d'exploitation (Keychain pour iOS, Keystore pour Android, libsecret pour les systèmes Linux). Elle est une partie extrêmement cruciale de Missive, car elle est utilisée dans les stores de mon implémentation du protocole Signal afin de récupérer les clés, sessions et autres données sensibles.

Les données sont stockées en base64 après avoir été sérialisées, car le stockage sécurisé ne permet que le stockage de chaînes de caractères. Il y a donc tout un travail de sérialisation/désérialisation à faire pour stocker des objets plus complexes, sur lequel nous allons revenir plus tard.

Il permet également de stocker les clés de plusieurs utilisateurs, car il préfixe chaque clé avec le nom de l'utilisateur grâce à `NamespacedSecureStorage`, une classe dérivée de `flutter_secure_storage`. Cela permet de gérer plusieurs comptes sur la même application, et de les isoler les uns des autres afin d'éviter les problèmes et les conflits.

Arborescence



```
client/lib
├── common
│   └── http.dart
├── constants
│   ├── api.dart
│   └── app_colors.dart
├── features
│   ├── authentication
│   │   ├── landing_screen.dart
│   │   ├── login_screen.dart
│   │   ├── models
│   │   │   └── user.dart
│   │   ├── providers
│   │   │   └── auth_provider.dart
│   │   ├── register_screen.dart
│   │   └── totp_modal.dart
│   ├── chat
│   │   ├── models
│   │   │   ├── conversation.dart
│   │   │   ├── conversation.realm.dart
│   │   │   ├── pending_messages.dart
```

```
│   │   │   └── pending_messages.realm.dart
│   │   ├── providers
│   │   │   └── chat_provider.dart
│   │   └── screens
│   │       ├── conversation_screen.dart
│   │       ├── conversations_screen.dart
│   │       ├── message_bubble.dart
│   │       └── user_search_screen.dart
│   └── encryption
│       ├── namespaced_secure_storage.dart
│       ├── providers
│       │   └── signal_provider.dart
│       ├── secure_storage_identity_key_store.dart
│       ├── secure_storage_pre_key_store.dart
│       ├── secure_storage_session_store.dart
│       └── secure_storage_signed_pre_key_store.dart
├── firebase_options.dart
└── main.dart
```

Figure 12 — Arborescence de l'application

Comme vous pouvez le voir, l'application est divisée en plusieurs parties, en utilisant l'approche *feature-first* : chaque fonctionnalité a son propre dossier, et est organisée de la même manière, avec les écrans séparés des providers. Cela permet de voir rapidement les différentes parties de l'application, de séparer au maximum la logique de l'interface, et m'a beaucoup aidé lors du développement.

Ces différentes parties sont ensuite importées dans le fichier `main.dart`, qui est le point d'entrée de l'application. Il contient le routeur, importe tous les providers afin d'y avoir accès dans toute l'application, et redirige les utilisateurs par rapport à leur état de connexion.

Protocole Signal

Le protocole Signal, qui a été retenu pour sa fiabilité, son excellente documentation en ligne ainsi que ses implémentations en un grand nombre de différents langages, est implémenté grâce à la bibliothèque `libsignal_protocol_dart`. Cette dernière nous permet d'avoir une implémentation personnalisée du protocole, dans le sens où elle nous permet de gérer le stockage des différentes clés et sessions de la manière que l'on souhaite.

Cette bibliothèque fonctionne en deux parties :

IMPLÉMENTATION DES STORES

Premièrement, il faut s'occuper d'implémenter les stores. Ces derniers sont les classes qui vont interagir avec le stockage de notre choix. Des classes abstraites sont fournies, `IdentityKeyStore`, `PreKeyStore`, `SessionStore`, `SignedPreKeyStore`, qui détaillent les différentes méthodes à implémenter afin que le protocole puisse fonctionner. Une fois ces dernières fonctionnelles, il est possible de les passer en paramètre à d'autres classes, sur lesquelles nous allons revenir plus tard.

Dans le cas de Missive, le stockage qui a été choisi est `flutter_secure_storage`, qui permet de stocker les clés de manière sécurisée dans le stockage sécurisé du système d'exploitation. Il est important de noter que les clés sont sérialisées en base64 avant d'être stockées, car le stockage sécurisé ne permet que le stockage de chaînes de caractères.

Prenons l'exemple de `PreKeyStore`, qui permet de gérer les pré-clés, à savoir les clés qui sont utilisées pour établir une session de chiffrement avec un autre utilisateur. Dans la version en Dart, il est nécessaire d'avoir une classe qui implémente les méthodes suivantes :

- `containsPreKey` : permet de vérifier si une pré-clé est déjà stockée
- `loadPreKey` : permet de charger une pré-clé stockée
- `removePreKey` : permet de supprimer une pré-clé stockée
- `storePreKey` : permet de stocker une pré-clé

Voici un exemple d'implémentation de `PreKeyStore`, plus précisément de la méthode loadPreKey :

```
class SecureStoragePreKeyStore implements PreKeyStore {
  final SecureStorage _secureStorage;

  SecureStoragePreKeyStore(SecureStorage secureStorage)
      : _secureStorage = secureStorage;


  /// Loads a serialized [PreKeyRecord] from [SecureStorage] and returns it as a [PreKeyRecord].
  /// Throws an [InvalidKeyIdException] if the pre key is not found.
```

```
  @override
  Future<PreKeyRecord> loadPreKey(int preKeyId) async {
    final preKeys = await _loadKeys();

    if (preKeys == null) {
      throw InvalidKeyIdException('There are no preKeys stored');
    }

    final preKey = preKeys[preKeyId.toString()];
    if (preKey == null) {
      throw InvalidKeyIdException('PreKey with id $preKeyId not found');
    }

    // Decode the base64 string and deserialize it as a PreKeyRecord
    return PreKeyRecord.fromBuffer(base64Decode(preKey));
  }

  /// Loads all pre keys from the secure storage
  /// Returns a [Map] of [PreKeyRecord]s
  Future<Map<String, dynamic>?> _loadKeys() async {
    final preKeysJson = await _secureStorage.read(key: 'preKeys');
    if (preKeysJson == null) return null;
    final preKeys = jsonDecode(preKeysJson);

    return preKeys;
  }

  // Autres méthodes (storePreKey, removePreKey, containsPreKey)...
}
```

Figure 13 — Exemple d'implémentation du store PreKeyStore

Comme vous pouvez le voir, la méthode loadPreKey s'occupe de désérialiser les clés depuis le stockage sécurisé, et de les renvoyer sous forme de PreKeyRecord, qui est un objet de la bibliothèque `libsignal_protocol_dart`.

Les différents stores sont disponibles dans le dossier `client/features/encryption`.

UTILISATION DES STORES

Une fois les stores implémentés, il est possible de les utiliser dans les différentes parties de l'application. `libsignal_protocol_dart` fournit des classes qui permettent d'être instanciées en utilisant les stores, ce qui assure une cohérence dans l'application, ainsi qu'une grande facilité d'utilisation une fois les stores fonctionnels.

Missive simplifie l'utilisation de ces classes en utilisant un Provider, `

SignalProvider`, qui permet de gérer le chiffrement et le déchiffrement de messages. Nous allons voir un exemple d'utilisation de `SignalProvider` pour chiffrer un message :

```
class SignalProvider extends ChangeNotifier {
    late SecureStorageIdentityKeyStore _identityKeyStore;
    late SecureStoragePreKeyStore _preKeyStore;
    late SecureStorageSignedPreKeyStore _signedPreKeyStore;
    late SecureStorageSessionStore _sessionStore;

    Future<void> initialize(
      {required bool installing,
      required String name,
      String? accessToken}) async {
        // Initialisation des stores (cette méthode doit être appelée avant d'utiliser les autres méthodes)
        ...
      }

    Future<CiphertextMessage> encrypt(
      {required String name, required String message}) async {
        final remoteAddress = SignalProtocolAddress(name, 1);
        final sessionCipher = SessionCipher(_sessionStore, _preKeyStore,
            _signedPreKeyStore, _identityKeyStore, remoteAddress);

        final cipherText = await sessionCipher.encrypt(utf8.encode(message));
        return cipherText;
    }
}
```

Figure 14 — Méthode encrypt de SignalProvider

Comme vous pouvez le voir, la méthode `encrypt` utilise les différents stores pour chiffrer un message, en instanciant un `SessionCipher`. Cette classe est la classe principale de la bibliothèque, et permet de chiffrer / déchiffrer les messages en utilisant les différentes clés stockées dans les stores. Elle s'occupe automatiquement d'utiliser les différentes méthodes des stores pour récupérer les clés nécessaires, et de les utiliser pour chiffrer le message.

SIGNALPROVIDER

Afin de simplifier tout ce processus, un Provider a été créé afin d'envelopper les méthodes et la logique plus bas niveau du protocole Signal. Ce dernier contient des méthodes permettant d'effectuer les opérations principales dont Missive a besoin (initialisation du protocole ainsi que des stores, récupération des pré-clés, chiffrement / déchiffrement des messages). Cela permet d'avoir une interface simple afin d'utiliser le protocole, et de séparer encore plus la logique de l'interface afin d'éviter d'avoir des composants trop longs et complexes.

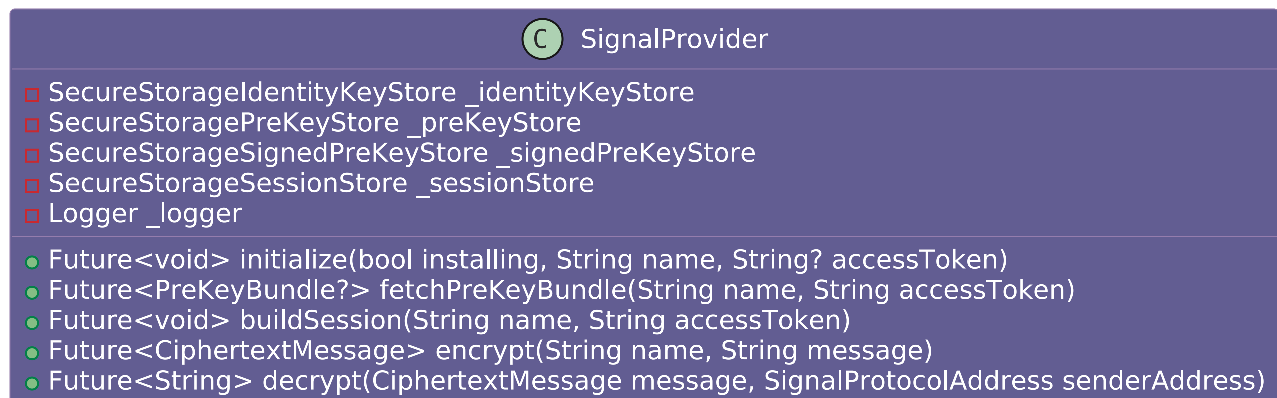


Figure 15 — Diagramme de classe de SignalProvider

INITIALISATION DU PROTOCOLE

À chaque fois que l'utilisateur lance l'application, le protocole est initialisé, en utilisant la fonction `initialize({required bool installing, required String name, String? accessToken})`. Le fonctionnement de cette dernière est le suivant :

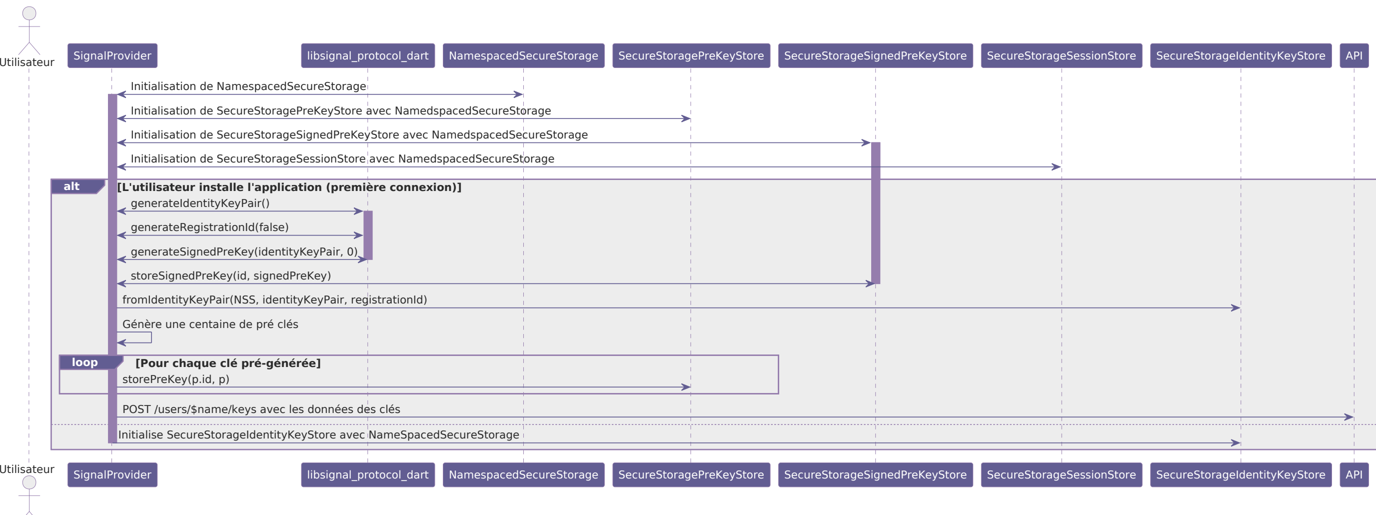


Figure 16 — Diagramme de séquence de SignalProvider.initialize()

Une fois le protocole initialisé, `SignalProvider` est prêt à être utilisé. Si ce dernier ne l'est pas, une erreur sera retournée afin d'éviter tout comportement non attendu.

ÉTABLISSEMENT D'UNE SESSION

Le protocole Signal requiert l'établissement d'une session entre deux utilisateur•trice•s afin de pouvoir envoyer et recevoir des messages chiffrés. Ce processus est abstrait dans la fonction `buildSession(name: 'username', accessToken: 'access-token')`. Cette dernière est appelée à chaque fois qu'une conversation est accédée, ou si un message est reçu sans qu'une session soit disponible (si c'est le premier message que l'on reçoit de cet•te utilisateur•trice). Voici comment elle fonctionne :

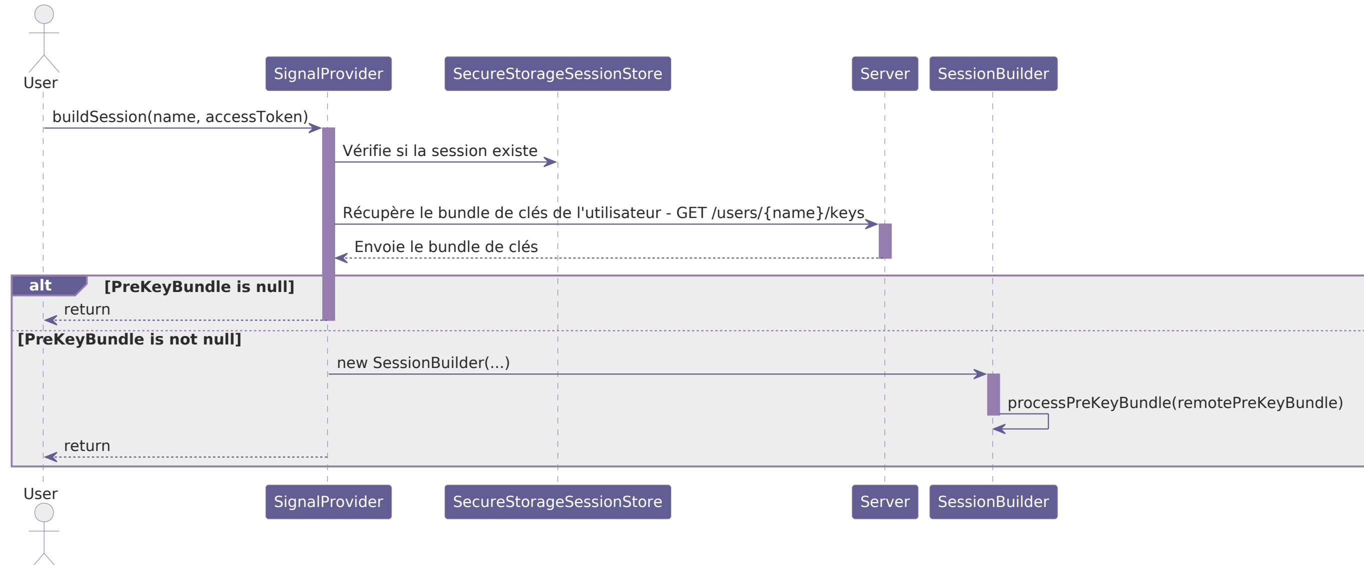


Figure 17 — Diagramme de séquence de l'établissement d'une session

Une fois toutes ces étapes effectuées, le SessionBuilder se chargera automatiquement de créer la session avec nos implémentations des différents stores (ceci est également la raison de pourquoi il est si important de bien les implémenter), du moment que nos méthodes font ce qu'elles sont sensées faire.

CHIFFREMENT D'UN MESSAGE

Une fois la session établie, nous pouvons enfin commencer à envoyer des messages. `SignalProvider` possède une fonction, `encrypt({required String name, required String message})` qui permet de chiffrer simplement un message, et de le préparer à l'envoi par le serveur WebSocket. Voici son implémentation :

```
Future<CiphertextMessage> encrypt(
    {required String name, required String message}) async {
  final remoteAddress = SignalProtocolAddress(name, 1);
  final sessionCipher = SessionCipher(_sessionStore, _preKeyStore,
      _signedPreKeyStore, _identityKeyStore, remoteAddress);

  final cipherText = await sessionCipher.encrypt(utf8.encode(message));
  return cipherText;
}
```

Figure 18 — Implémentation de la méthode encrypt

Il fonctionne de manière similaire au buildSession avec le SessionCipher. Il suffit de lui donner tous les stores, et il va aller chercher les bonnes données automatiquement avec les fonctions que nous avons implémenté au plus bas niveau.

DÉCHIFFREMENT D'UN MESSAGE

Une fois le message envoyé (le fonctionnement de la communication en temps réel est expliquée plus bas), Il sera nécéssaire de le déchiffrer de l'autre côté une fois reçu. Encore une fois, `SignalProvider` contient une fonction pour déchiffrer un `CiphertextMessage`, qui est la classe prodiguée par l'implémentation de `libsignal_protocol_dart`. Voici à quoi elle ressemble :

```
 Future<String> decrypt(
     CiphertextMessage message, SignalProtocolAddress senderAddress) async {
   final sessionCipher = SessionCipher(_sessionStore, _preKeyStore,
       _signedPreKeyStore, _identityKeyStore, senderAddress);
   Uint8List plainText = Uint8List(0);

   if (message is PreKeySignalMessage) {
     plainText = await sessionCipher.decrypt(message);
   }
   if (message is SignalMessage) {
     plainText = await sessionCipher.decryptFromSignal(message);
   }
   return utf8.decode(plainText);
 }
}
```

Figure 19 — Implémentation de la méthode decrypt

On peut voir deux types de messages dans cette méthode, `PreKeySignalMessage` et `SignalMessage`. Les deux doivent être déchiffrés différement, car un `PreKeySignalMessage` est envoyé en tant que message initial d'une conversation. Il a donc besoin d'une pré clé afin d'établir la chaîne initiale. Ensuite, les messages seront uniquement des `SignalMessage`, mais ce cas doit être traité à part car il nécéssite une logique et des données différentes.

Authentification

L'authentification de Missive est gérée par le provider `AuthProvider`. Ce dernier permet de connecter l'utilisateur, en interagissant avec l'API afin de récupérer les jetons d'accès et de rafraîchissement. Il permet aussi de gérer la création de compte, et la déconnexion.

AUTHPROVIDER

La connexion et la création de compte s'effectue en envoyant une requête à l'API (`POST /users` pour la création de compte, `POST /tokens` pour la connexion). Si la connexion est réussie, les jetons d'accès et de rafraîchissement sont stockés dans le stockage sécurisé, et l'utilisateur est redirigé vers la page des conversations. Tout ce comportement est enveloppé par la fonction `login(String name, String password)`. Cette dernière fonctionne de cette manière :

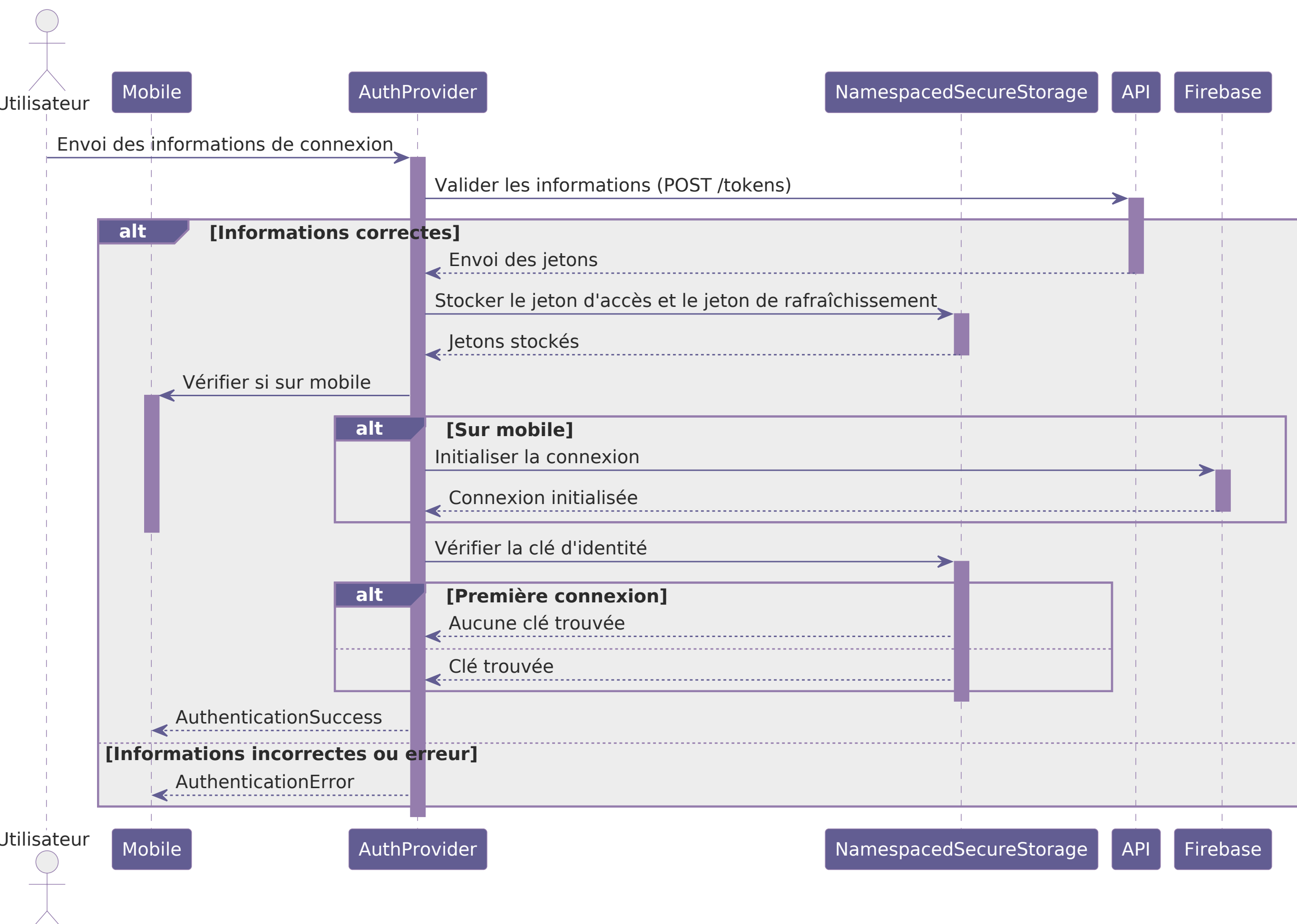


Figure 20 — Diagramme de séquence technique de l'authentification

Il est important de noter que `register()` fonctionne quasiment de la même manière, la différence étant que la requête sera à `POST /users` et non `POST /tokens`.

Gestion des erreurs d'authentification

Les erreurs d'authentification sont gérées par des classes qui sont retournées par les fonctions d'authentification. J'ai décidé de baser mon système de gestion d'erreurs sur le type de l'objet plutôt qu'avec un statut quelque part dans l'objet, car le système de typage de Dart rend ce procédé beaucoup plus propre.

Nous avons d'un côté la classe `AuthenticationSuccess`, qui représente une authentification avec les bons identifiants ou une création de compte qui a fonctionnée. De l'autre, nous avons `AuthenticationError`, une classe abstraite dont plusieurs autres classes héritent. Elle hérite de la classe `Error` par défaut de Dart, et nous permet d'avoir une interface commune et standarde à toutes ces erreurs. Si une de ces erreurs est retournée, le client pourra donc gérer le message à afficher au client en vérifiant le type de l'objet retourné. On peut apercevoir ce procédé dans l'implémentation de `login_screen.dart` et `register_screen.dart`.

Authentification des requêtes

Une fois la connexion / la création de compte effectuée, les autres requêtes qui seront effectuées vers le serveur pourront désormais être authentifiées avec le jeton d'accès et de rafraîchissement stockés dans le stockage sécurisé. Ils sont récupérés à l'aide de *getters* dans le `AuthProvider`, qui rendent le procédé beaucoup plus propre car le *getter* `accessToken` rafraîchit automatiquement le jeton d'accès si il venait à expirer, en décodant la base64 et en vérifiant la date d'expiration avec la date de l'appareil.

ROUTAGE

Le routage des pages est effectuée en utilisant `go_router`, qui permet de rediriger l'utilisateur•trice sur la page d'accueil de son compte dès qu'il•elle s'authentifie, ainsi que de s'occuper d'envoyer l'utilisateur sur les différentes pages de l'application comme la page d'une conversation, l'écran de recherche etc. `AuthProvider` étant ce qu'on appelle un *listenable* en Dart (un élément depuis lequel un autre peut réagir), à chaque fois que `notifyListeners()` est appelé (dans le cas de `AuthProvider`, après une connexion / création de compte qui a fonctionné), la logique définie dans `_router.refresh`, présente dans l'entrypoint de mon application, `main.dart` est lancée. Voici à quoi elle ressemble :

```
redirect: (context, state) async {
      // Determine if the user is in the onboarding flow (trying to login or register)
      bool onboarding = state.matchedLocation == '/login' ||
          state.matchedLocation == '/register' ||
          state.matchedLocation == '/landing';
      if (!authProvider.isLoggedIn) {
        return onboarding ? null : '/landing';
      }

      return onboarding ? '/' : null;
    },
    refreshListenable: authProvider
```

Figure 21 — Logique de redirection du routeur de Missive

Si la connexion échoue, une erreur est affichée à l'utilisateur (grâce au type de retour de `login()` et `register()`), et il lui est demandé de réessayer.

Conversations

La gestion des conversations est effectuée grâce à `ChatProvider`. Ce dernier contient des méthodes qui permettent de se connecter au serveur WebSocket, de réagir aux nouveaux messages en temps réel ainsi que de se charger du stockage des messages après leur chiffrement/déchiffrement.

STOCKAGE DES MESSAGES EN LOCAL

Une partie importante du fonctionnement de Missive est la manière dont les messages sont stockés en local. Pour se faire, une bibliothèque nommée Realm, publiée par MongoDB, est utilisée. Il s'agit d'une base de données locale, en NoSQL, qui permet d'effectuer des requêtes de manière efficace et supporte également le chiffrement de la base de données en natif.

Différentes solutions ont été considérées pour le stockage des messages. Tout d'abord, SQLite a été le premier choix, mais le chiffrement ne fonctionnait malheureusement pas. Ensuite, des essais ont été effectués avec Hive, une autre solution de stockage persistent en local qui supporte des requêtes, mais le souci était que l'intégralité des messages devaient être chargés si l'on souhaitait stocker quelque chose (également aucun support de pagination). Vu les soucis de performance, le choix de Realm a été fait, qui supporte le chiffrement de manière native, ainsi que la capacité de requêtes et l'efficacité d'une base de données comme SQLite.

La base de données Realm est chiffrée avec une clé générée à la création du compte, et stockée dans le stockage sécurisé du téléphone. Elle peut être accédée seulement quand l'utilisateur est connecté, et est seulement déchiffrée au besoin.

Voici un modèle de cette dernière :

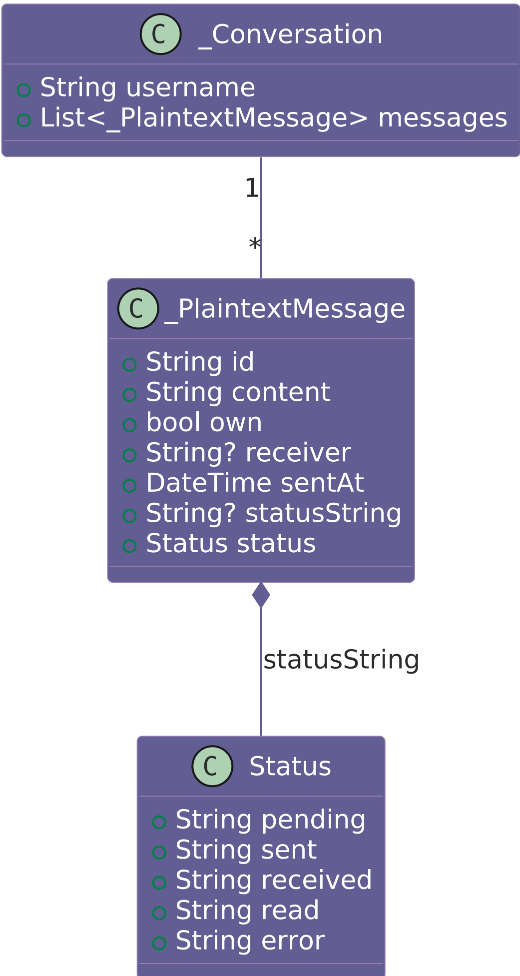


Figure 22 — Diagramme de classes de la base de données Realm des conversations

Les conversations sont des objets Flutter, séparées par le nom du destinataire, qui sert de clé.

AFFICHAGE DES CONVERSATIONS

Une fois la base de données locale mise en route, il est possible de voir ses conversations sur la page d'accueil de Missive. Elles sont récupérées ainsi que triées en fonction de la conversation avec le message le plus récent, et un petit badge avec le nombre de messages non lus est ensuite affiché afin de pouvoir correctement visualiser ses derniers messages. Cette logique visualisation est gérée par le widget `conversations_screen.dart`, qui sert également de point d'entrée de l'application après la connexion / la création de compte. Ce dernier initialise tout d'abord l'application, avec la logique suivante :

```
late AuthProvider _userProvider;
late SignalProvider _signalProvider;
late SecureStorageIdentityKeyStore identityKeyStore;
late ChatProvider _chatProvider;
late Future _initialization;
final Logger _logger = Logger('ConversationsScreen')

void initState() {
    super.initState();
    _userProvider = Provider.of<AuthProvider>(context, listen: false);
    _signalProvider = Provider.of<SignalProvider>(context, listen: false);
    _chatProvider = Provider.of<ChatProvider>(context, listen: false);
    _fToast.init(context);
    _initialization = initialize();
  }
```

```
/// Initializes stores, providers, and fetches pending data. It also installs the app (generates all required keys and upload them) in case it's the first time the user opens the appa.
Future<void> initialize() async {
  await _initializeSignalAsNeeded();
  await _chatProvider.setupUserRealm();
  try {
    _chatProvider.fetchPendingMessages();
    _chatProvider.fetchMessageStatuses();
  } catch (e) {
    _logger.log(Level.WARNING,
        'Error fetching pending messages: $e (error of type ${e.runtimeType})');
  }
  if (!mounted) return;
  await _chatProvider.connect();
  final info = await PackageInfo.fromPlatform();
  setState(() {
    _version = info.version;
  });
}
```

Figure 23 — Logique d'initialisation de Missive

Comme on peut le voir plus haut, le fait d'avoir séparé la logique le plus possible du client rend le code côté client beaucoup plus lisible. On peut également noter quelque chose qui pourrait paraître assez particulier à première vu : la logique de `initState()` est partiellement dans une fonction `initialize()`. C'est une partie assez importante de cet `initState()` ; en effet, certaines de ces fonctions étant asynchrones, nous ne pouvons pas utiliser l'application avant que ces fonctions soient complétées, car elles permettent d'initialiser correctement les Provider ainsi que les différentes connexions.

Pour remédier à cela, nous définissons un Future, une fonction asynchrone, qui nous permettra de le réutiliser plus bas. Flutter implémente un widget, appelé FutureBuilder, qui nous permet d'envelopper l'application complètement et montrer un écran de chargement à l'utilisateur le temps que cette fonction n'a pas fini son initialisation. Voici à quoi ressemble le code de l'application :

```
Widget build(BuildContext context) {
  const String logoName = 'assets/missive_logo.svg';
  final logo = SvgPicture.asset(
    logoName,
    width: 200.0,
    height: 200.0,
    colorFilter: ColorFilter.mode(
        Theme.of(context).colorScheme.onPrimary, BlendMode.srcIn),
  );
  return FutureBuilder(
      future: _initialization,
      builder: (context, snapshot) {
        if (snapshot.connectionState == ConnectionState.done) {
          return _buildBody();
        }
        return Container(
          color: Theme.of(context).colorScheme.surface,
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              logo,
              CircularProgressIndicator(
                strokeWidth: 8,
                valueColor: AlwaysStoppedAnimation<Color>(
                    Theme.of(context).colorScheme.onPrimary),
              ),
            ],
          ),
        );
      });
}
```

Figure 24 — Implémentation du FutureBuilder

En Flutter, un FutureBuilder a besoin d'un Future, c'est donc pour ça qu'on est venu le stocker dans `initState()`. Ce dernier est une fonction qui ne se lancera qu'une seule fois par "cycle de vie" du widget, c'est à dire tant qu'il est à l'écran.

Une fois ce Future assigné, nous pouvons implémenter ce que nous souhaitons faire dans le callback `builder:`, qui nous permet de retourner le widget de notre choix par rapport à l'état de `snapshot`. Dans ce cas-là, l'intégralité de l'application est stockée dans `_buildBody`, qui sera retournée uniquement après la complétion du Future. Dans le cas échéant, un écran de chargement pleine page, apparent à un *splashscreen* sera montré à l'utilisateur afin de l'informer visuellement et l'empêcher d'interagir avec une application non pleinement initialisée.

DÉMARRAGE D'UNE CONVERSATION

Si l'on souhaite démarrer une conversation avec un utilisateur, il suffit de cliquer sur le bouton en haut à droite. Ce dernier ouvrira une page via le routeur, que l'on peut retrouver dans `user_search_screen.dart`. Il s'agit d'un écran de recherche, qui à chaque changement de la barre de recherche, effectue un appel à l'API afin de récupérer la liste des utilisateurs. Voici à quoi ressemble son implémentation, au niveau de la logique :

```
  List<String> _usernames = [];
  final TextEditingController _searchController = TextEditingController();
  final _debouncer = Debouncer(milliseconds: 500);

  @override
  void initState() {
    super.initState();
    _searchController.addListener(_onSearchChanged);
  }

  void _fetchUsers() async {
    if (_searchController.text.isEmpty) {
      return;
    }

    final response = await dio.get('/users',
        queryParameters: {'search': _searchController.text},
        options: Options(headers: {
          'Authorization':
              'Bearer ${await Provider.of<AuthProvider>(context, listen: false).accessToken}',
        }));

    setState(() {
      _usernames = List<String>.from(
          response.data['data']['users'].map((user) => user['name']).toList());
    });
  }

  void _onSearchChanged() {
    if (_searchController.text.isEmpty) {
      setState(() {
        _usernames = [];
      });
      return;
    }

    _debouncer.run(
      () {
        _fetchUsers();
      },
    );
  }
```

Figure 25 — Implémentation de la logique de recherche des conversations

Comme on peut le constater, `AuthProvider` est utilisé via `Provider.of<Type>`. C'est une fonction assez pratique qui permet de récupérer la value d'un Provider dans le contexte de l'application (la fonction va remonter l'arbre des widgets et trouver le premier Provider qui correspond au type). Cela nous permet de récupérer le jeton d'accès, et d'accéder à l'API.

Une classe appelée `Debouncer` a également été implémentée : il s'agit d'une classe qui permet de ralentir la vitesse des requêtes, à un minimum de 500ms entre les différentes requêtes. Cela permet d'éviter de surcharger l'API inutilement, car la fonction `_fetchUsers` est appelée à chaque fois que la barre de recherche change. Voici son implémentation :

```
class Debouncer {
  final int milliseconds;
  VoidCallback? action;
  Timer? _timer;

  Debouncer({required this.milliseconds});

  void run(VoidCallback action) {
    if (_timer != null) {
      _timer!.cancel();
    }
    _timer = Timer(Duration(milliseconds: milliseconds), action);
  }
}
```

Figure 26 — Implémentation de la classe Debouncer

Une fois les utilisateurs récupérés, ils sont affichés au moyen d'une `ListView` sur la page, grâce à la variable `_usernames`.

```
Expanded(
            child: ListView.builder(
              itemCount: _usernames.length,
              itemBuilder: (context, index) {
                return ListTile(
                  title: Text(_usernames[index]),
                  onTap: () async {
                    final accessToken =
                        await Provider.of<AuthProvider>(context, listen: false)
                            .accessToken;
                    if (!context.mounted) return;
                    await Provider.of<SignalProvider>(context, listen: false)
                        .buildSession(
                            name: _usernames[index], accessToken: accessToken!);

                    if (!context.mounted) return;
                    await context
                        .push('/conversations/${_usernames[index]}')
                        .then(
                          (value) => context.pop('/userSearch'),
                        );
                  },
                );
              },
            ),
          ),
```

Figure 27 — Affichage des utilisateurs trouvés

Comme on peut le voir au dessus, si l'on veut commencer une conversation avec quelqu'un, les opérations suivantes vont se produire :

- Une session est construite avec `SignalProvider.buildSession`, en lui passant le nom d'utilisateur ainsi que le jeton d'accès afin de pouvoir récupérer le bundle de pré-clés
- Une nouvelle fenêtre de conversations est ouverte, avec l'addresse de routage interne `/conversations/{username}`
- La fenêtre de recherche est fermée

Cela nous permet de nous assurer qu'une session de chiffrement est présente au moment ou l'utilisateur ouvre la conversation pour la première fois.

ENVOI D'UN MESSAGE

Une fois la conversation accédée, l'utilisateur va se retrouver sur une page de conversation, implémentée sous `conversation_screen.dart`.

En accédant à cet écran, la première opération réalisée est la vérification que la conversation existe bel et bien dans la base de données locale Realm, avec la fonction `ensureConversationExists`. Si elle n'existe pas, elle sera créé afin de pouvoir s'assurer que le stockage pourra s'effectuer correctement.

Après l'initialisation de cette dernière, on peut écrire un message dans la boite de texte qui se situe en bas, et l'envoyer. Une fois le bouton pressé, la fonction `handleMessageSent` est appelée. Cette dernière va aller elle-même appeler la méthode `sendMessage` du `ChatProvider`. Voici une explication de son fonctionnement :

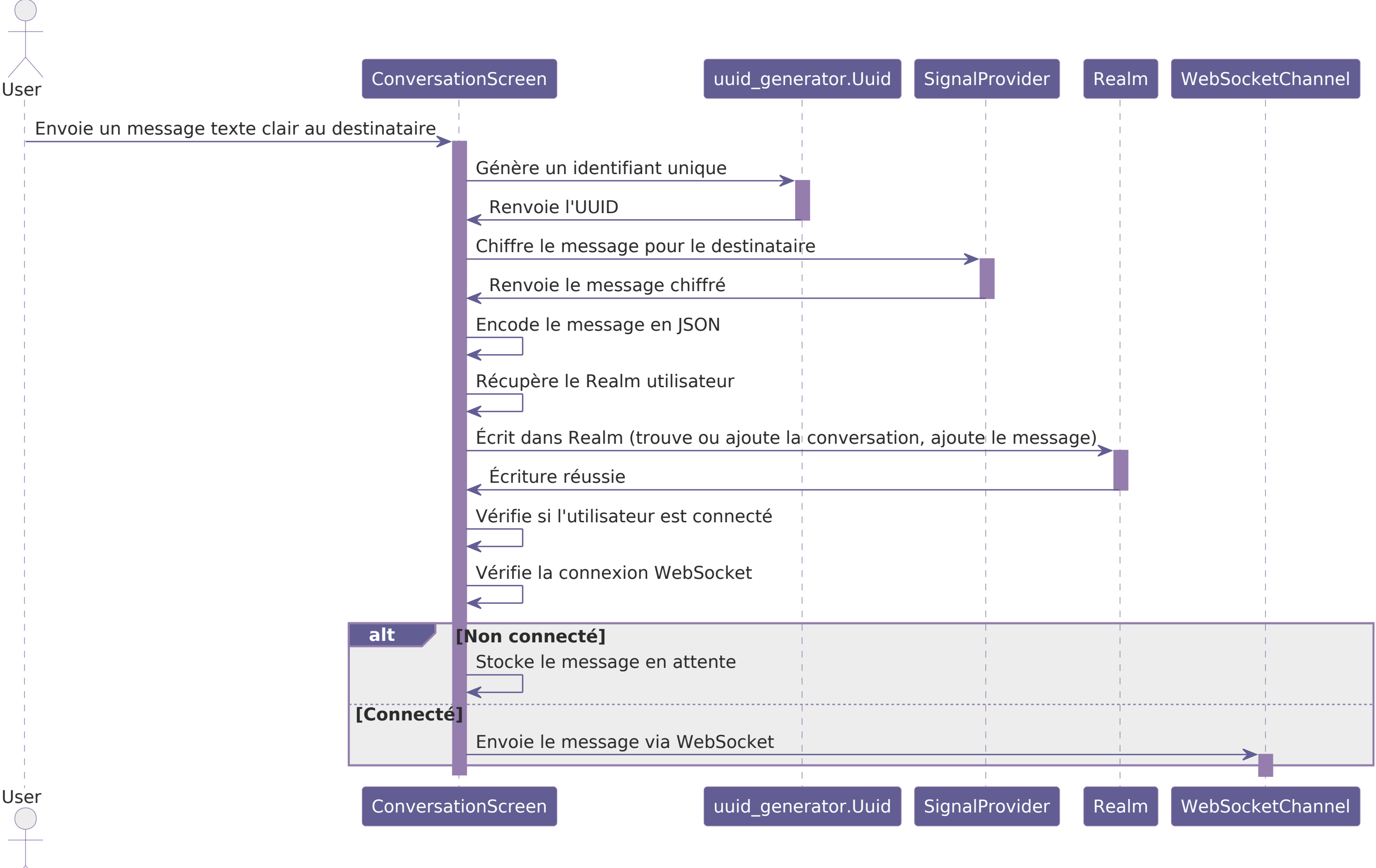


Figure 28 — Diagramme de séquence de la méthode sendMessage

On peut voir deux comportements possibles :

- L'envoyeur est connecté : le message est directement envoyé au WebSocket
- L'envoyeur est hors-ligne : le message est stocké dans une base de données Realm et sera envoyé à la reconnexion

Une deuxième base de données Realm a donc dû être mise en place, afin de stocker ces messages en attente. Cela nous assure qu'aucun message ne sera donc perdu, du moment que le serveur fonctionne correctement. Voici un schéma de cette base de données :

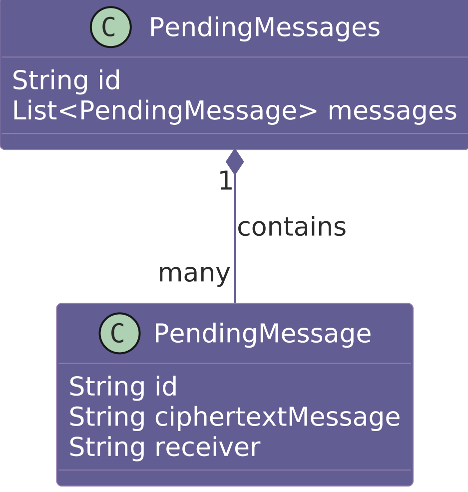


Figure 29 — Diagramme de classe de la base de données Realm PendingMessages

RÉCEPTION

Quand un message est reçu, la méthode `handleMessage` est appelée, qui se charge de déchiffrer le message et de le stocker dans la base de données locale. Voici comment elle fonctionne :

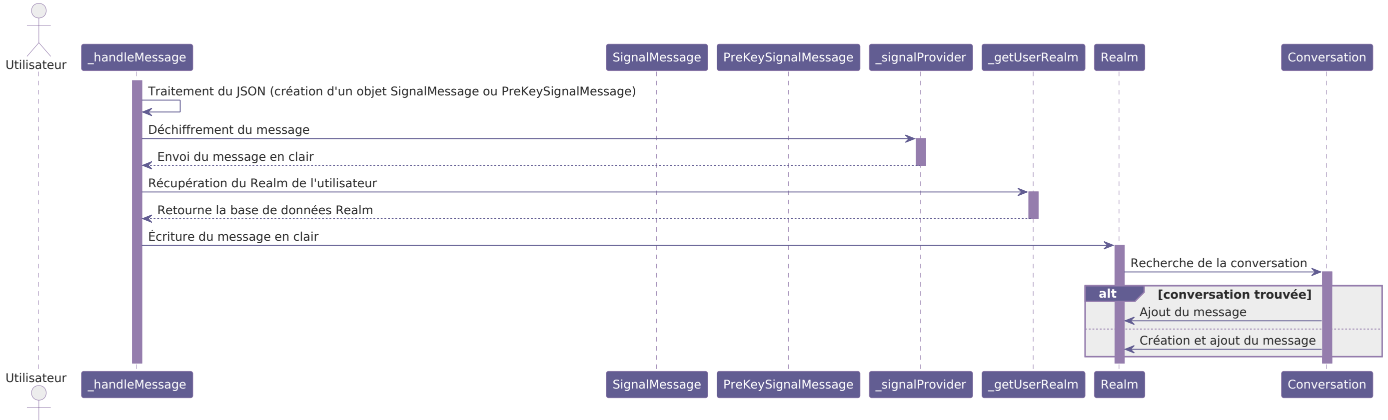


Figure 30 — Diagramme de séquence de la méthode pour gérer les messages reçus

COMMUNICATION EN TEMPS RÉEL

La communication en temps réel est gérée par le provider `ChatProvider`. Ce dernier permet de gérer la connexion au serveur de WebSocket, ainsi que l'envoi et la réception des messages. Il permet également de gérer les différentes erreurs qui peuvent survenir lors de la communication. Il dépend de `SignalProvider` pour chiffrer et déchiffrer les messages, ainsi que de `AuthProvider` pour récupérer les jetons d'accès, comme par exemple afin d'établir la connexion au serveur WebSocket, ou pour récupérer les messages en attente.

Il implémente une méthode `connect()` qui permet d'effectuer les opérations suivantes :

- Connecte l'application au serveur WebSocket de Missive
- Envoie les messages en attente au WebSocket si il y en a
- Gère la reconnexion si jamais la connexion est perdue
- Gère la réception des messages (leur déchiffrement ainsi que leur stockage dans la base de données locale)

GESTION DES MISES À JOUR DE STATUTS

Missive implémente également une mise à jour des statuts de lecture, sous forme de petites coches, inspiré par des applications comme WhatsApp et Telegram. Voici tout d'abord à quoi correspondent les différentes légendes, que vous pouvez retrouver sous les messages que vous avez envoyé :

- ✓ : message envoyé
- ✓✓ : message reçu
- ✓✓ : message lu

Ces statuts sont mis à jour également grâce au `ChatProvider`. Une dépendance Flutter est tout d'abord utilisé au niveau de la conversation,

2.2.2 Serveur

Le serveur de Missive est composé de deux parties distinctes : l'API, et le serveur WebSocket. Ces deux parties permettent de gérer l'authentification, l'autorisation, le stockage des messages, et la communication en temps réel entre les utilisateur·rice·s. Ils sont réalisés à l'aide du framework Fastify, qui est un framework back-end rapide, et qui permet de gérer les différentes routes de

manière efficace. Il permet également de gérer les différentes parties de l'application de manière découplée, ce qui est crucial dans le développement de cette application.

Concepts Fastify

Avant de commencer à détailler le fonctionnement du serveur, il est important de comprendre les concepts de base de Fastify, qui est le framework utilisé pour l'API et le serveur de WebSocket, car la nomenclature sera utilisée dans la suite de la documentation.

HOOKS

Les hooks Fastify sont des fonctions qui sont exécutées avant ou après une route. Ils permettent de gérer des opérations comme l'authentification, la validation des données, la vérification des permissions, etc. Ils sont très utiles pour gérer les différentes parties de l'application, et permettent de découpler les différentes parties de l'application. Ce sont l'équivalent des middlewares dans d'autres frameworks.

API

L'API est une API REST en TypeScript, qui permet de gérer l'authentification, le stockage des différentes clés publiques, ainsi que les utilisateurs. Le framework utilisé pour cette dernière sera Fastify, qui est un framework back-end rapide, qui permet de gérer les routes de manière efficace. Beaucoup de fonctionnalités sont déjà implémentées, comme la gestion des erreurs, la gestion des paramètres, etc. Cela me permettra de me concentrer au maximum sur le développement de l'application et de ses fonctionnalités.

La base de données est gérée par Prisma, qui est un ORM (Object-Relational Mapper) permettant de gérer les différentes tables de manière efficace, et de gérer les relations entre les différentes tables. Il permet également de gérer les migrations de manière efficace, et de gérer les différentes versions de la base de données (ce qui est crucial durant le processus de développement).

AUTHENTIFICATION

Le système d'authentification de l'API repose sur un système de jetons JWT. Ces derniers permettent une gestion de l'authentification complètement découplée du serveur (ce qui est important pour une API REST, qui doit rester *stateless*).

Après la connexion à l'application (route `POST /users`), l'utilisateur·rice reçoit :

- Jeton d'accès : permet de s'authentifier sur les différentes routes de l'API. Il est valide pendant 15 minutes, et contient les différentes permissions de l'utilisateur·rice. Il est envoyé dans le corps de la réponse, et doit être stocké localement de manière sécurisée.
- Jeton de rafraîchissement : permet de rafraîchir le jeton d'accès. Il est valide pendant 30 jours, et permet de générer un nouveau jeton d'accès sans avoir à se reconnecter. Il permet également de révoquer la connexion en cas de perte ou de vol du compte. Il est envoyé dans un cookie sécurisé HTTPOnly, et doit être stocké localement de manière sécurisée.

Le contenu d'un jeton d'accès est le suivant :

```json
{
  "scope": [
    "profile:read",
    "profile:write",
    "keys:read",
    "messages:read"
  ],
  "iat": 1712660971,
  "sub": "aed060ac-75d5-44df-a82e-760b3d1d6636",
  "exp": 1712661871
}
```

Figure 31 — Contenu d'un jeton d'accès

- `scope` : les différentes permissions de l'utilisateur·rice
- `iat` : l'heure à laquelle le jeton a été émis (issued at)
- `sub` : l'identifiant unique de l'utilisateur·rice
- `exp` : l'heure à laquelle le jeton expire

Le jeton de rafraîchissement contient uniquement l'identifiant unique de l'utilisateur, ainsi que sa date d'expiration.

Ces derniers étant signés cryptographiquement via une clé privée, il est impossible de les modifier sans la clé privée correspondante. Cela permet de garantir l'intégrité des jetons, et de garantir que l'utilisateur·rice est bien celui·celle qu'il·elle prétend être. Cela permet également de ne pas avoir à gérer des sessions côté serveur, et laisse au client la responsabilité de gérer son propre état.

Diagramme de séquence du processus d'authentification

Figure 32 — Diagramme de séquence du processus d'authentification

Le processus d'authentification est géré par le hook `authenticationHook`.

AUTORISATION

Le processus d'autorisation de Missive repose sur un système de permissions. Chaque utilisateur·rice possède un ensemble de permissions, qui lui permettent d'accéder à certaines routes de l'API. Ces permissions sont stockées dans le jeton d'accès, et sont vérifiées à chaque requête. Si l'utilisateur·rice n'a pas les permissions nécessaires pour accéder à une route, il·elle recevra une erreur 403 Forbidden.

Voici le champ en question, qui a déjà été mentionné dans la partie Authentification :

```json
{
    "scope": [
        "profile:read",
        "profile:write",
        "keys:read",
        "messages:read"
    ]
}
```

Figure 33 — Scope d'un jeton d'accès

Le processus d'autorisation est géré par le hook `authorizationHook`, qui prend également en paramètre les permissions nécessaires pour accéder à une route.

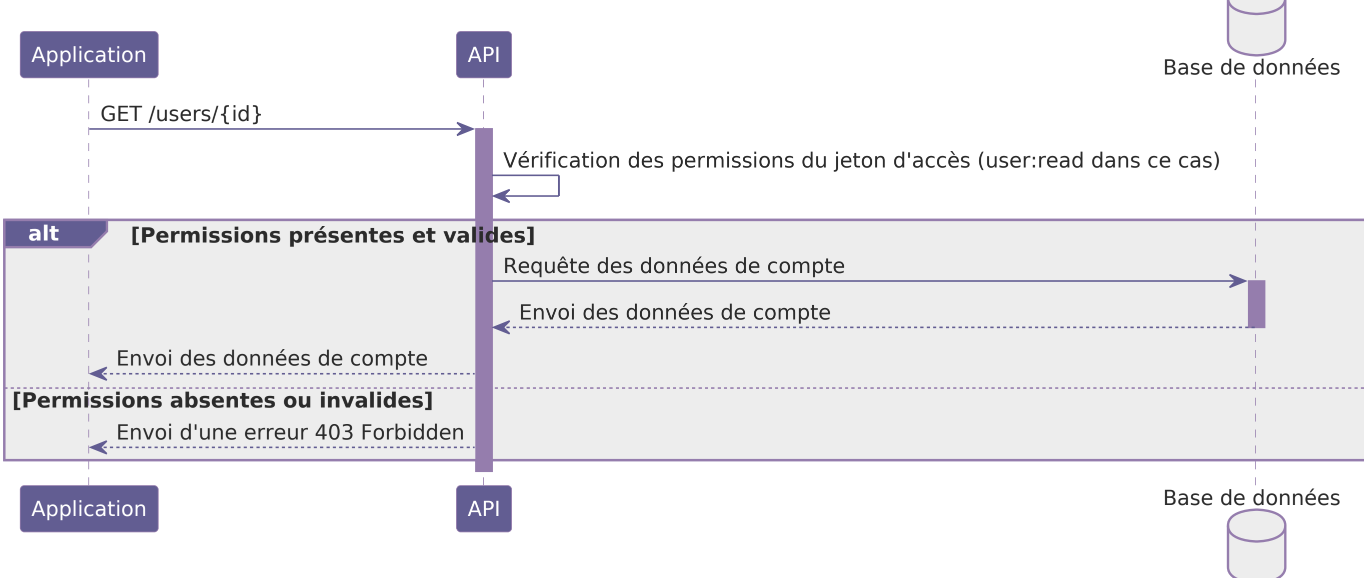


Figure 34 — Diagramme de séquence du processus d'autorisation

Serveur WebSocket

Le serveur WebSocket est un serveur en TypeScript, qui permet de gérer la communication en temps réel entre les utilisateur·rice·s. Il permet de gérer l'envoi des messages, ainsi que leur stockage si nécessaire. La bibliothèque utilisée pour ce dernier est un plugin Fastify, @fastify/websocket, qui encapsule le protocole WebSocket et permet d'utiliser les fonctionnalités de Fastify.

Ce dernier utilise le même système d'authentification que l'API (jeton d'accès) à la création de la connexion. Il peut également accéder à la base de données, ce qui lui permet de stocker les messages temporairement dans le cas où l'utilisateur·rice n'est pas connecté·e.

Il

permet d'envoyer des messages à un utilisateur•trice en utilisant son identifiant, ainsi que d'en recevoir et de gérer les statuts de lecture et de réception.

AUTHENTIFICATION / AUTORISATION

Comme expliqué auparavant, le serveur de Websocket emploie le même procédé de vérification d'identité et de permissions que l'API. Les jetons étant signés avec une clé privée propre au serveur, la vérification de l'identité utilise la même clé publique que l'API. Ce dernier a également accès à la même base de données, car l'instance de Prisma est partagée entre les deux parties du serveur.

CRÉATION DU WEBSOCKET

Le WebSocket est créé en effectuant une requête HTTP classique à la route `/`, qui est ensuite transformée en WebSocket. Cette route est protégée par le hook `authenticationHook`, qui vérifie l'authentification de l'utilisateur·rice. Si l'utilisateur·rice est authentifié·e, le serveur accepte la connexion, et permet à l'utilisateur·rice de communiquer en temps réel avec les autres utilisateur·rice·s.

ENVOI ET RÉCEPTION DE MESSAGES

Une fois la connexion établie, l'utilisateur·rice peut envoyer des messages à un autre utilisateur·rice en utilisant son identifiant unique. Le serveur vérifie que l'utilisateur·rice est bien connecté·e, et que l'utilisateur·rice destinataire est également connecté·e. Le corps du message qui doit être envoyé ressemble à ceci :

```
{
    "receiverId": "5443f5ef-9b6d-438b-9c0f-e00298725d13",
    "content": "This is an encrypted test message"
}
```

Figure 35 — Structure d'un message

Des informations sont ensuite ajoutées au message :

- L'identifiant de l'expéditeur
- L'heure d'envoi

Si l'utilisateur est connecté, le message est envoyé directement à l'utilisateur·rice destinataire, sans passer par la base de données. Un message de confirmation est ensuite envoyé à l'expéditeur pour lui indiquer que le message a bien été remis.

Si ce n'est pas le cas, le message est stocké dans la base de données, et sera envoyé à l'utilisateur·rice dès qu'il·elle se connectera. Après le stockage, un message de confirmation est ensuite envoyé à l'expéditeur pour lui indiquer que le message a bien été envoyé. La sécurité des données des utilisateur•trice•s est garantie par le chiffrement de bout en bout, ainsi que WSS, grâce à TLS, qui permet d'avoir une protection au niveau du contenu même du message et du transport. Les messages sont également supprimés de la base de données une fois qu'ils ont été demandés par le destinataire.

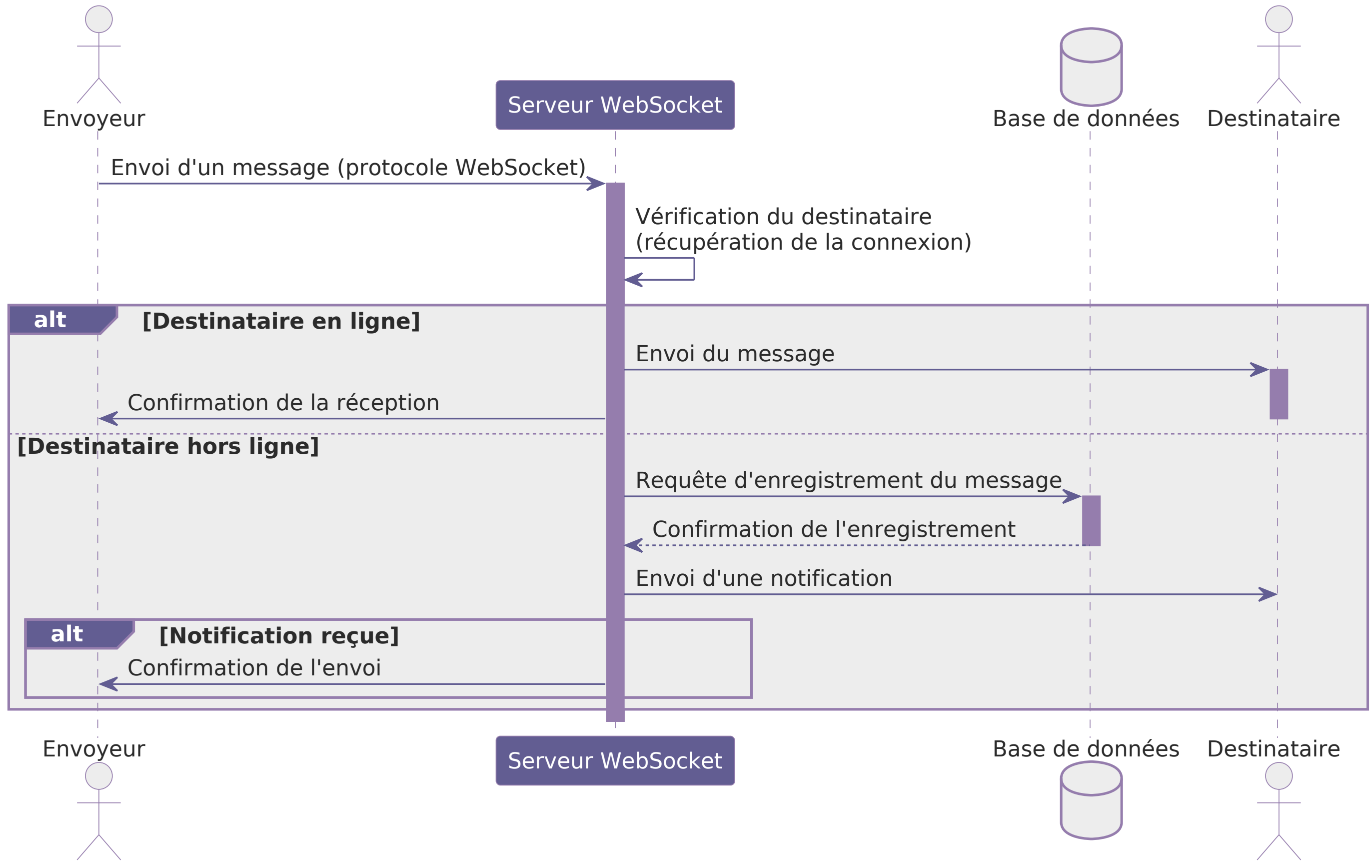


Figure 36 — Diagramme du processus d'envoi d'un message

GESTION DES STATUTS DE MESSAGES

Au niveau du serveur, le WebSocket permet également de publier des mises à jour de statuts de messages, qui sont à ce format-ci :

```json
{
    "messageId": "5443f5ef-9b6d-438b-9c0f-e00298725d13",
    "state": "received",
    "sender": "alice"
}
```

Figure 37 — Structure d'un message de statut

Ces dernières permettent aux clients de mettre à jour le statut de leur message. Il y a trois états possibles :

- `sent` : le message a été envoyé
- `received` : le message a été reçu
- `read` : le message a été lu

Ils peuvent être soit envoyés directement depuis le serveur, dans le cas d'un envoi de message (afin d'avertir l'envoyeur que son message a bien été reçu), soit envoyés par le client, dans le cas de la lecture d'un message. Ces derniers sont soit :

- Si l'utilisateur•trice est en ligne, envoyés directement au client
- Si l'utilisateur•trice est hors-ligne, stockés en base de données, et récupérés dès que l'utilisateur•trice se connecte

C'est un processus qui est similaire à celui de l'envoi de messages, mais qui utilise une table complètement différente, à savoir `MessageStatus`, afin d'éviter de garder les messages en base de données. Cela permet d'augmenter encore plus la sécurité des données des utilisateur•trice•s.

2.2.3 Base de données

La base de données est une base de données PostgreSQL, qui permet de stocker les utilisateur•trice•s, les messages non envoyés et les clés publiques. PostgreSQL a été retenu pour sa robustesse, sa fiabilité, et sa capacité à gérer de gros volumes de données. Il permet également de gérer les transactions, les clés étrangères, et les index de manière efficace. Un diagramme de la base de données est disponible ci-dessous :

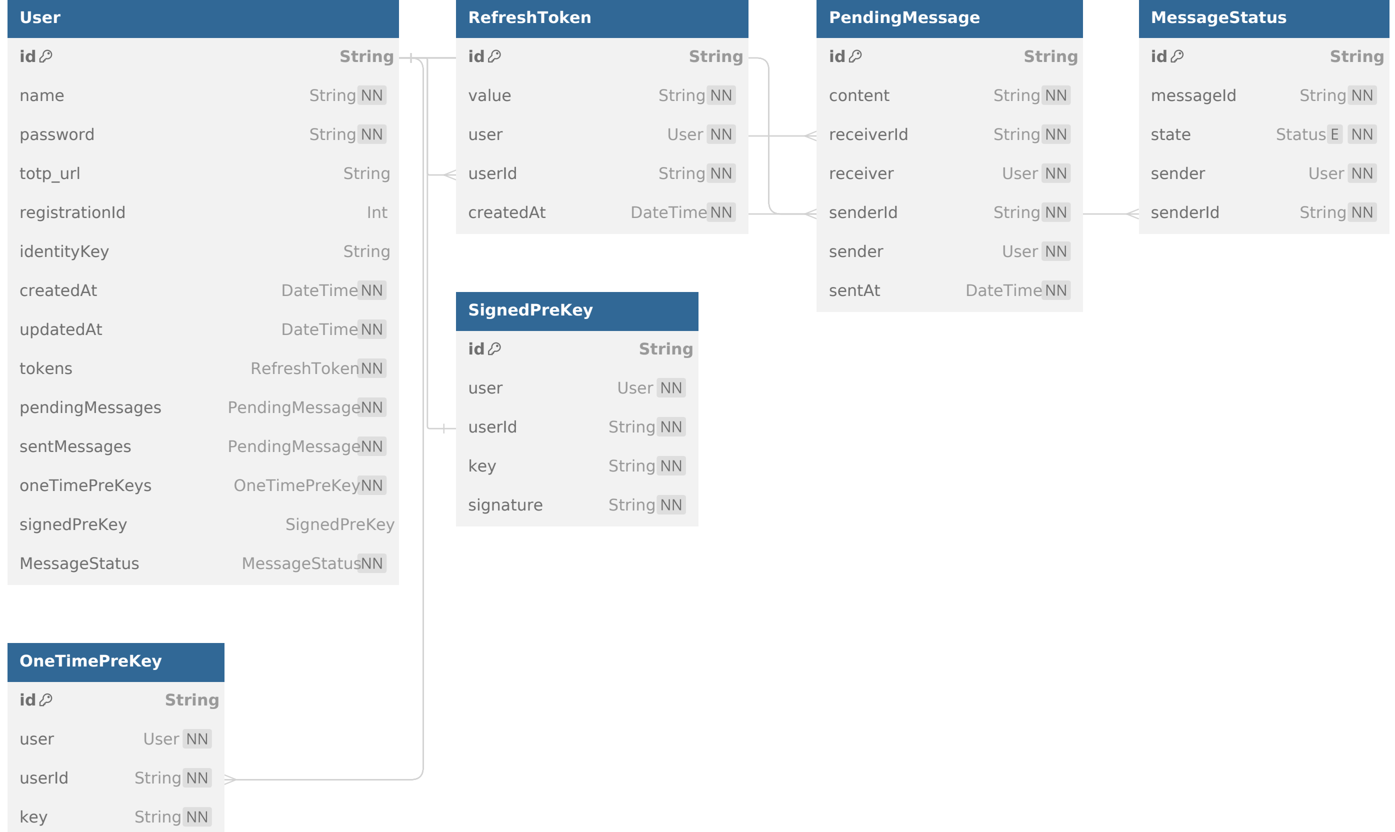


Figure 38 — Schéma de la base de données

La base de données est gérée par Prisma, qui est un ORM (Object-Relational Mapper) permettant de gérer les différentes tables de manière efficace, et de gérer les relations entre les différentes tables, en représentant les tables sous forme d'un schéma générique, qui peut être ensuite converti en un grand nombre de types de base de données.

Le choix d'une base de données SQL a été fait car il n'y avait aucune nécessité pour un modèle NoSQL, les données et les relations étant clairement définies et structurées.

Explication du schéma

USER

La table `User` permet de stocker les différents utilisateur•trice•s de l'application. Elle contient les informations suivantes :

Colonne	Description
`id`	l'identifiant unique de l'utilisateur•trice (UUID généré automatiquement, permet d'éviter les collisions et d'itérer sur les utilisateurs)
`name`	le nom d'utilisateur•trice (unique, sert également à identifier l'utilisateur•trice dans les différents cas où c'est la seule information que l'on a)
`password`	le mot de passe de l'utilisateur•trice (hashé, pour des raisons de sécurité, en utilisant Argon2)
`totp_url`	l'URL du TOTP (Time-based One-Time Password) de l'utilisateur•trice, qui permet de générer un code à usage unique pour la double authentification
`registrationId`	l'identifiant de l'enregistrement de l'utilisateur•trice. Fait partie du protocole Signal
`identityKey`	la clé d'identité publique de l'utilisateur•trice. Fait partie du protocole Signal
`createdAt`	la date de création de l'utilisateur•trice
`updatedAt`	la date de dernière mise à jour de l'utilisateur•trice
`tokens`	les différents jetons de rafraîchissement de l'utilisateur•trice (permet de révoquer les connexions en cas de perte ou de vol du compte)
`pendingMessages`	les messages en attente de l'utilisateur•trice (stockés en base de données, et récupérés dès que l'utilisateur•trice se connecte)
`sentMessages`	les messages envoyés par l'utilisateur•trice
`oneTimePreKeys`	les clés pré-clés à usage unique de l'utilisateur•trice. Fait partie du protocole Signal. Utilisées afin d'établir une session de chiffrement initiale avec un•e autre utilisateur•trice
`signedPreKey`	la clé pré-clé signée de l'utilisateur•trice. Fait partie du protocole Signal
`MessageStatus`	les différents statuts des messages de l'utilisateur•trice (envoyé, reçu, lu)

REFRESHTOKEN

La table `RefreshToken` permet de stocker les différents jetons de rafraîchissement des utilisateur•trice•s. Elle contient les informations suivantes :

Colonne	Description
`id`	l'identifiant unique du jeton de rafraîchissement (UUID généré automatiquement)
`userId`	l'identifiant de l'utilisateur•trice auquel appartient le jeton de rafraîchissement
`value`	la valeur du jeton de rafraîchissement (unique, permet de révoquer les connexions en cas de perte ou de vol du compte)

Cette table contient le minimum d'informations possibles, afin d'éviter de stocker une session côté serveur, et de garantir que l'application reste *stateless* le plus possible. Cela allège le travail du serveur et laisse la responsabilité au client de gérer son propre état, ce qui dans le cas de Missive, est un choix pertinent pour garantir la sécurité des données des utilisateur•trice•s.

SIGNEDPREKEY

La table `SignedPreKey` permet de stocker les différentes clés pré-clés signées des utilisateur•trice•s. Il y en a une seule par utilisateur•trice. Elle contient les informations suivantes :

Colonne	Description
`id`	l'identifiant unique de la clé pré-clé signée (UUID généré automatiquement)
`userId`	l'identifiant de l'utilisateur•trice auquel appartient la clé pré-clé signée
`keyId`	l'identifiant unique de la clé pré-clé signée (généré automatiquement, fait partie du protocole Signal)
`publicKey`	la clé publique de la clé pré-clé signée (permet d'établir une session de chiffrement initiale avec un•e autre utilisateur•trice)
`signature`	la signature de la clé pré-clé signée (permet de vérifier l'authenticité de la clé pré-clé signée, et d'éviter les attaques de type MITM)
`createdAt`	la date de création de la clé pré-clé signée

ONETIMEPREKEY

La table `OneTimePreKey` permet de stocker les différentes clés pré-clés à usage unique des utilisateur•trice•s. Elle contient les informations suivantes :

Colonne	Description
`id`	l'identifiant unique de la clé pré-clé à usage unique (UUID généré automatiquement)
`userId`	l'identifiant de l'utilisateur•trice auquel appartient la clé pré-clé à usage unique
`keyId`	l'identifiant unique de la clé pré-clé à usage unique (généré automatiquement, fait partie du protocole Signal)
`publicKey`	la clé publique de la clé pré-clé à usage unique (permet d'établir une session de chiffrement initiale avec un•e autre utilisateur•trice)
`createdAt`	la date de création de la clé pré-clé à usage unique

PENDINGMESSAGE

La table `PendingMessage` permet de stocker les différents messages en attente des utilisateur•trice•s. Elle contient les informations suivantes :

Colonne	Description
`id`	l'identifiant unique du message en attente (UUID généré automatiquement)
`userId`	ldentifiant de l'utilisateur•trice auquel appartient le message en attente
`senderId`	l'identifiant de l'utilisateur•trice qui a envoyé le message en attente
`content`	le contenu du message en attente (chiffré, pour des raisons de sécurité)
`createdAt`	la date de création du message en attente

MESSAGESTATUS

La table `MessageStatus` permet de stocker les différents statuts des messages. Elle contient les informations suivantes :

Colonne	Description
`id`	l'identifiant unique du statut de message (UUID généré automatiquement)
`userId`	l'identifiant de l'utilisateur•trice auquel appartient le statut de message
`messageId`	l'identifiant du message auquel appartient le statut de message
`state`	l'état du message (envoyé, reçu, lu)
`createdAt`	la date de création du statut de message
`updatedAt`	la date de dernière mise à jour du statut de message

2.3 Intégration continue

Missive possède une pipeline d'intégration continue, gérée par un fichier `.gitlab-ci.yml`. Ce dernier est configuré pour être activé à chaque push la branche `main`, et se charge d'effectuer les opérations suivantes :

1. Tests unitaires du serveur
2. Tests unitaires du client
3. Génération de la documentation d'API du client
4. Génération de la documentation mkdocs
5. Déploiement sur Gitlab Pages

Les tests unitaires du serveur et du client se déclenchent seulement si il y a eu des changements dans leurs dossiers respectifs, afin d'éviter d'effectuer des pipelines redondantes. Il n'a pas été possible de faire la même chose pour la documentation du client et du serveur, car ces derniers dépendent l'un de l'autre (la documentation du client se situe dans un sous-dossier du Gitlab Pages du serveur, afin de pouvoir être accessible facilement via le même domaine).

2.4 Déploiement

Le déploiement du serveur de Missive s'effectue à l'aide de Docker, qui a été retenu pour sa facilité à maintenir et pour sa capacité à recréer des environnements reproducibles. Un `Dockerfile` est présent dans le répertoire `server`, qui permet de générer une image du serveur.

Une fois cette image générée, un `docker-compose.yml` est présent à la racine du projet. Ce dernier comporte deux variantes :

- Développement (`docker-compose.dev.yml`) : permet de lancer une instance en développement, avec *hot reload*
- Production (`docker-compose.prod.yml`) : permet de lancer une instance en production, avec HTTPS automatique

La raison pour laquelle ces deux dernières ont été séparées en plusieurs fichiers vient du fait des légères différences entre les environnements. Avec ce système, la version en développement peut bénéficier du *hot reload*, comme mentionné auparavant, ce qui améliore grandement l'expérience développeur. De plus, le déploiement en production nécéssitant HTTPS, qui a besoin d'un nom de domaine afin de générer les certificats, ce n'est donc pas possible de le faire si le serveur tourne sur localhost.

Ce Compose permet également aussi de lancer une base de données postgres, qui est migrée automatiquement grâce au script `migrate-and-run.sh` dans le dossier serveur. Ce dernier lance les bonnes migrations par rapport à l'environnement dans lequel il se trouve (migrations de développement en développement, qui active également les seeders présents, et migrations de production en production, qui permettent de s'assurer que les données ne seront pas supprimées si nous venions à mettre à jour le modèle). Les différentes variables d'environnements nécéssaires sont également définies, si besoin est.

Finalement, il lance un serveur Caddy, qui permet de gérer le reverse proxy vers les différentes applications, afin de pouvoir avoir le serveur et un client pgAdmin sur la même URL sans avoir besoin de changer le numéro de port.

Une des autres raisons pour laquelle les deux versions sont séparées est le fait que Docker Swarm est utilisé pour le déploiement de Missive. Ce dernier est un outil de Docker, qui permet de rajouter des fonctionnalitées extrêmement pratiques pour le développement, comme la gestion des secrets de manière sécurisée (clés privées...), des configurations, et encore de pouvoir effectuer un *load balancing* automatiquement en connectant des nœuds ensemble.

2.4.1 Jelastic Cloud

Le choix de l'hébergeur pour Missive a été Jelastic Cloud de chez Infomaniak. Ce dernier a été choisi car les données de Missive doivent se trouver en Suisse, afin de respecter la confidentialité des utilisateurs. De plus, ayant déjà une expérience antérieure avec les produits Infomaniak, le choix me semblait tout à fait naturel.

Jelastic Cloud est une *PaaS* (Platform as a Service), qui permet d'héberger facilement des applications. Un Docker Swarm a été créé chez eux, ainsi qu'une interface Portainer afin de pouvoir permettre une gestion du déploiement simple. Portainer possède également une fonctionnalité très appréciable, qui permet de donner le lien vers un dépôt Git contenant un `docker-compose.yml`, afin de pouvoir gérer automatiquement les différentes versions de l'application.

2.5 Justifications techniques et réflexions

2.5.1 Choix de Flutter

Flutter a été retenu pour plusieurs raisons :

1. **Rapidité de développement** : Flutter permet de développer des applications de manière très rapide, grâce à son hot reload, qui permet de voir les changements en temps réel. Cela permet de gagner énormément de temps lors du développement.
2. **Expérience développeur** : Flutter possède une excellente documentation, ainsi qu'une communauté très active. Cela permet de trouver des solutions aux problèmes rencontrés très rapidement. J'ai pu d'ailleurs discuter avec des membres de la communauté, via leur serveur Discord, pour résoudre des problèmes rencontrés lors du développement.
3. **Facilité de déploiement** : Flutter permet de déployer des applications sur plusieurs plateformes (iOS, Android, Web, Desktop) avec une seule *codebase*. Cela permet de gagner énormément de temps, et de garantir une expérience utilisateur homogène sur toutes les plateformes.
4. **Gestion des mises à jour** : Flutter permet de gérer les mises à jour de manière très efficace, grâce à son système de packages. Cela permet de mettre à jour l'application de manière très simple, et de garantir une expérience utilisateur optimale.
5. **Capacité à gérer les différentes plateformes** : Flutter permet de gérer les différentes plateformes de manière très simple, grâce à son système de widgets. Cela permet de garantir une expérience utilisateur optimale sur toutes les plateformes.

2.5.2 Choix du protocole Signal

Le protocole Signal a été retenu pour plusieurs raisons :

1. **Fiabilité** : Le protocole Signal est utilisé par de nombreuses applications de messagerie sécurisée, comme WhatsApp, Signal, et Facebook Messenger. Cela permet de garantir une fiabilité optimale, et de s'assurer que les messages sont chiffrés de manière sécurisée.
2. **Documentation** : Le protocole Signal possède une excellente documentation, qui permet de comprendre son fonctionnement très rapidement. Cela permet de gagner énormément de temps lors du développement.
3. **Implémentations disponibles** : Le protocole Signal possède de nombreuses implémentations disponibles dans différents langages de programmation, ce qui permet de l'utiliser dans de nombreux contextes. Cela permet de garantir une flexibilité optimale, et de s'adapter à de nombreux environnements de développement.
4. **Sécurité** : Le protocole Signal permet de garantir une sécurité optimale, grâce à son chiffrement de bout en bout, le fait que chaque message est chiffré avec une clé différente, et à son système de signatures. Cela permet de garantir que les messages sont chiffrés de manière sécurisée, et que seul•e•s les utilisateur•rice•s peuvent les lire.

2.5.3 Choix de Fastify

Fastify a été retenu pour plusieurs raisons :

1. **Rapidité** : Fastify est un framework back-end rapide, qui permet de gérer les routes de manière très efficace. Cela permet de garantir une performance optimale, et de s'assurer que les requêtes sont traitées de manière rapide.
2. **Gestion des erreurs** : Fastify permet de gérer les erreurs de manière très simple et unifiée grâce à son système de hooks. Cela permet de s'assurer que les erreurs sont gérées de manière efficace.
3. **Simplicité** : Fastify est un framework très simple à utiliser, qui permet de se concentrer au maximum sur le développement des fonctionnalités. Cela permet de gagner énormément de temps lors du développement, et de garantir une expérience développeur optimale.
4. **Extensions disponibles** : Fastify possède de nombreuses extensions disponibles, qui permettent de gérer des fonctionnalités avancées de manière très simple. Cela permet de garantir une flexibilité optimale, et de s'adapter à de nombreux environnements de développement.

2.5.4 Choix de PostgreSQL

PostgreSQL a été retenu pour plusieurs raisons :

1. **Robustesse** : PostgreSQL est une base de données très robuste, qui permet de gérer de gros volumes de données de manière très efficace. Cela permet de garantir une performance optimale, et de s'assurer que les données sont stockées de manière sécurisée.
2. **Fiabilité** : PostgreSQL est une base de données très fiable, utilisée par de nombreuses entreprises pour stocker leurs données. Cela permet de garantir une fiabilité optimale, et de s'assurer que les données sont accessibles à tout moment.
3. **UUID en clé primaire** : PostgreSQL permet d'utiliser des UUID comme clés primaires, ce qui permet d'éviter les collisions, et de garantir que les identifiants sont uniques. Cela permet de garantir l'intégrité des données, et de s'assurer que les données sont stockées de manière cohérente. Cela permet également d'éviter que l'on puisse itérer les données, les identifiants étant simplement incrémentaux.

2.5.5 Sécurité des données

La sécurité des données des utilisateur•rice•s est une priorité absolue pour Missive. Plusieurs mesures ont été mises en place pour garantir la sécurité des données :

1. **Chiffrement de bout en bout** : Les messages sont chiffrés de bout en bout, ce qui permet de garantir que seul•e•s les utilisateur•rice•s peuvent les lire. Cela permet de garantir la confidentialité des messages, et de s'assurer que les données ne sont pas accessibles à des tiers.
2. **Stockage sécurisé des clés** : Les clés sont stockées de manière sécurisée dans le stockage sécurisé du système d'exploitation, ce qui permet de garantir que les clés ne sont pas accessibles à des tiers. Cela permet de garantir la sécurité des clés, et de s'assurer que les données sont chiffrées de manière sécurisée.
3. **Utilisation de jetons JWT** : Les jetons JWT permettent de gérer l'authentification de manière sécurisée, en garantissant que les jetons ne peuvent pas être modifiés sans la clé privée correspondante. Cela permet de garantir l'intégrité des jetons, et de s'assurer que les utilisateur•rice•s sont bien ceux•celles qu'ils•elles prétendent être. Ces derniers étant également *stateless*, ils laissent au client la responsabilité de gérer son propre état, ce qui, dans le cas de Missive, est un choix pertinent pour garantir la sécurité des données des utilisateur•rice•s.

En conclusion, la sécurité des données des utilisateur•rice•s est une priorité absolue pour Missive, et plusieurs mesures ont été mises en place pour garantir la sécurité des données. Ces mesures permettent de garantir que les données sont stockées de manière sécurisée, et qu'elles sont accessibles à tout moment. Elles permettent également de garantir que les données ne sont pas accessibles à des tiers, et que les utilisateur•rice•s sont bien ceux•celles qu'ils•elles prétendent être.

2.5.6 Notes supplémentaires et réflexions

2.5.7 Problèmes rencontrés

Voici quelques problèmes rencontrés durant le développement de Missive, ainsi que les solutions apportées :

1. **Problème de stockage des classes** : Le stockage sécurisé ne permet que le stockage de chaînes de caractères, ce qui pose problème pour le stockage des classes Signal, qui sont des objets complexes. La solution apportée a été de sérialiser les différentes classes en utilisant la fonction prodiguée par l'implémentation, `serialize`, et les encoder en base64 avant de les stocker, et de les désérialiser puis les décoder à la récupération.
2. **Problème de gestion des erreurs** : La gestion des erreurs est cruciale pour garantir la sécurité des données des utilisateur•rice•s. La solution apportée a été d'utiliser les hooks de Fastify pour gérer les erreurs de manière efficace, et de s'assurer que les erreurs sont gérées de manière cohérente. Missive utilisant Docker pour le déploiement, les erreurs peuvent être récupérées de manière standarde via les logs de Docker.
3. **Problème de reconnexion** : La reconnexion au serveur WebSocket est cruciale pour garantir la continuité de la communication en temps réel. La solution apportée a été d'implémenter un système de reconnexion automatique, qui permet de se reconnecter automatiquement au serveur en cas de perte de connexion, ainsi que d'envoyer les messages en attente dès que la connexion est rétablie.
4. **Problème de stockage des messages** : Le stockage des messages en attente est crucial pour garantir que les utilisateur•rice•s ne ratent aucun message. La solution apportée a été d'implémenter une table `PendingMessage` dans la base de données, qui permet de stocker les messages en attente, et de les récupérer dès que l'utilisateur•rice se connecte.

2.5.8 Améliorations futures

Voici quelques améliorations futures qui pourraient être apportées à Missive :

1. **Gestion des contacts** : La gestion des contacts permettrait de simplifier la recherche d'utilisateur•rice•s, et de faciliter l'envoi de messages. Cela permettrait également de garantir une meilleure expérience utilisateur.
2. **Support multi-appareils** : Le support multi-appareils permettrait de garantir que les utilisateur•rice•s peuvent utiliser l'application sur plusieurs appareils en même temps, et de synchroniser les messages entre les différents appareils. Cela permettrait de garantir une meilleure expérience utilisateur. Le protocole Signal étant par défaut un protocole multi-appareils, il serait intéressant de l'implémenter dans Missive.
3. **Amélioration de l'interface utilisateur** : L'amélioration de l'interface utilisateur permettrait de garantir une meilleure expérience utilisateur, en rendant l'application plus agréable à utiliser. Cela permettrait également de simplifier l'utilisation de l'application, et de garantir une meilleure compréhension de son fonctionnement.

2.6 Références

- Protocole Signal: https://signal.org/docs/
- Fastify: https://www.fastify.io/
- Prisma: https://www.prisma.io/docs/
- Flutter: https://flutter.dev/
- Docker: https://www.docker.com/
- Jelastic Cloud: https://www.infomaniak.com/en/hosting/dedicated-and-cloud-servers/jelastic-cloud
- Realm: https://docs.mongodb.com/realm/
- PostgreSQL: https://www.postgresql.org/
- GitLab CI/CD: https://docs.gitlab.com/ee/ci/
- Flutter Secure Storage: https://pub.dev/packages/flutter_secure_storage

3. Plan de tests

3.1 API

3.1.1 Introduction

Ce plan détaille les tests fonctionnels prévus pour valider les fonctionnalités de l'API Missive, qui gère les tokens d'accès et de rafraîchissement ainsi que les informations utilisateur.

3.1.2 Lien de la production

L'API est hébergée sous ce lien : `https://missive.nezia.dev/api/v1`

3.1.3 Authentification

Toutes les requêtes nécessitent une authentification par token Bearer, sauf la route `POST /tokens` et `PUT /tokens` qui est utilisée pour l'authentification.

3.1.4 Scénarios de Test

Endpoint	Méthode	Description	Objectif	Code Attendu	Statut	Impact
POST / tokens	POST	Authentifier l'utilisateur et fournir tokens	Vérifier l'authentification et la remise de tokens	200 OK	OK	Élevé
PUT / tokens	PUT	Rafraîchir le token d'accès	Vérifier la mise à jour du token d'accès	200 OK	OK	Élevé
DELETE / tokens	DELETE	Révoquer un refresh token	Vérifier la révocation du refresh token	204 No Content	X	Élevé
POST / users	POST	Créer un compte utilisateur	Vérifier la création de compte	204 No Content	OK	Élevé
GET /users/{id}	GET	Obtenir les données d'un utilisateur	Vérifier la récupération des données utilisateur	200 OK	OK	Élevé
DELETE / users/{id}	DELETE	Supprimer le profil d'un utilisateur	Vérifier la suppression du profil utilisateur	204 No Content	OK	Élevé
POST / users/{id}/ keys	POST	Stocker un lot de pré-clés	Vérifier le stockage des pré-clés	204 No Content	OK	Moyen
GET /users/{id}/keys	GET	Récupérer les pré-clés d'un utilisateur	Vérifier la récupération des pré-clés	200 OK	OK	Moyen
PATCH / users/{id}/ keys	PATCH	Mettre à jour la pré-clé signée d'un utilisateur	Vérifier la mise à jour de la pré-clé signée	204 No Content	OK	Moyen
GET /users/{id}/ messages	GET	Récupérer les messages en attente d'un utilisateur	Vérifier la récupération des messages en attente	200 OK	OK	Moyen
GET /users/{id}/ messages	GET	Récupérer les messages en attente d'un utilisateur	Vérifier la suppression après récupération	200 OK	OK	Moyen

Notes supplémentaires

- Le statut "X" indique que le cas de test doit encore être exécuté ou que le résultat est en attente de vérification.
- Ces tests doivent être effectués dans un environnement de test avant le déploiement en production.

3.1.5 Tests de Sécurité et de Performance

- Des tests de charge et de sécurité devraient également être effectués en utilisant des outils comme OWASP ZAP pour identifier toute vulnérabilité potentielle.

3.1.6 Automatisation des tests

- Envisager l'automatisation de ces tests à l'aide de Postman ou d'autres frameworks d'automatisation pour intégrer dans un pipeline CI/CD.

3.1.7 Rapport de tests

POST /tokens

L'UTILISATEUR PEUT S'AUTHENTIFIER ET REÇOIT DES TOKENS VALIDES

Une des grandes difficultés avec ce schéma d'authentification a été principalement de comprendre comment fonctionnaient les JWT. Je suis parti sur un schéma de token avec un `scope`, qui permet d'attribuer des permissions à l'utilisateur. J'ai utilisé la bibliothèque `jose` afin de générer ces différents jetons, ainsi que Prisma pour récupérer les informations de l'utilisateur et comparer le mot de passe haché avec `argon2`.

J'ai du étendre une des classes de `jose`, car j'avais besoin de rajouter un `scope` sur mes jetons, et vu que j'utilise Typescript, la gestion des types a été assez complexe. Après avoir approfondi, j'ai réussi à créer ma propre classe dérivée, et tout fonctionne correctement.

Ensuite, pour implémenter une des fonctionnalités les plus essentielles du JWT, la signature, j'ai tout d'abord opté pour une simple chaîne de caractères générée aléatoirement. Voulant encore plus renforcer ma sécurité, j'ai au final opté pour une paire de clés RSA, qui me permettront d'augmenter encore l'entropie et la complexité de la signature, rendant le risque de trouver cette fameuse clé privée encore plus minime, et également de standardiser encore plus la sécurité, une clé asymmétrique étant de manière générale plus standarde qu'une chaîne de caractères dans la cryptographie moderne.

Une fois cette signature implémentée, elle peut être vérifée au niveau du serveur, et une exception sera générée si il s'avère que le jeton a été modifié. Je pourrais donc refuser l'accès aux routes protégées en conséquence.

PUT /tokens

MISE À JOUR DU TOKEN D'ACCÈS

En ce qui concerne la mise à jour du token d'accès, j'ai dû créer une table dans ma base de données, avec Prisma et rajouter une migration afin de pouvoir avoir une table qui stocke un token de rafraîchissement avec un ID utilisateur. De ce fait, je garde mon modèle d'API au maximum *stateless*, ce qui est un des points principaux d'une API REST. Ensuite, avec le jeton de rafraîchissement, je viens vérifier sa validité ainsi que son appartenance, et je génère un nouveau token pour l'utilisateur. Toute cette logique fonctionne parfaitement.

DELETE /tokens

RÉVOCATION DU REFRESH TOKEN

Je n'ai pas eu le temps d'implémenter la révocation du refresh token. Il est par contre correctement supprimé du côté du client, ce qui ne poserait pas de problèmes, à moins que la base de données devienne vraiment grande.

POST /users

CRÉATION DU COMPTE UTILISATEUR

Pour la création du compte utilisateur, j'ai récupéré les informations de l'utilisateur via le corps de la requête, et utilisé Prisma afin de créer un compte. J'ai également haché le mot de passe en utilisant argon2, afin de ne pas le stocker en clair.

GET /users/{id}

OBTENIR LES DONNÉES D'UN UTILISATEUR

En ce qui concerne la récupération des données, je voulais faire en sorte que cette route retourne le minimum d'informations possibles : elle est utilisée afin d'effectuer une recherche des utilisateurs par tout le monde, il était donc important que le moins d'informations soient disponibles afin de respecter la confidentialité des utilisateurs.

J'ai donc dû développer une fonction qui me permet d'exclure des clés d'un type d'objet. Ensuite, après avoir récupéré les données avec Prisma, j'utilise cette fonction `exclude` afin d'envoyer le strict nécéssair au fonctionnement de l'application.

POST /users/{id}/keys

STOCKER UN BUNDLE DE PRÉ-CLÉS

Cette partie m'a posé un sacré défi : en effet, ayant réalisé mon serveur avant de faire l'application, je n'avais aucune idée d'à quoi allait ressembler un bundle de pré-clés de manière exacte. Ayant lu le *whitepaper* du protocole, j'ai essayé au mieux de créer une table qui me semblait correcte. Cependant, l'implémentation en Dart utilisait une structure assez différente, en rajoutant des champs auxquels je n'avais pas du tout pensé auparavant, comme le `registrationId`, ou même la gestion des différents périphériques.

J'ai donc dû passer beaucoup de temps sur la modélisation de mes données, et y revenir un bon nombre de fois. Je pense avoir fini sur une itération qui est complètement fonctionnelle. J'ai également fait attention à stocker uniquement la partie publique de ces clés, en les sérialisant en base64 dans la base de données afin de simplifier au maximum le stockage.

GET /users/{id}/keys

RÉCUPÉRER UN BUNDLE DE PRÉ-CLÉS

La récupération des pré-clés avait une petite subtilité : il faut supprimer les *one time pre keys* une fois utilisées. J'ai donc implémenté une logique similaire à cette pour récupérer un utilisateur, sauf que cette fois, je n'ai pas exclu de champ, et j'ai simplement supprimé la pré-clé correspondante après l'avoir récupérée, en me servant de son ID.

PATCH /users/{id}/keys

MISE À JOUR DE LA PRÉ-CLÉ SIGNÉE

Idéalement, une application implémentant le protocole Signal devrait se charger de pouvoir effectuer la rotation de la pré-clé signée, afin d'accroître encore plus la confidentialité. Missive implémente cette dernière, en utilisant encore une fois Prisma afin de pouvoir mettre à jour le bon champ. La route est fonctionnelle, malheureusement, il n'y a pas eu le temps de l'implémenter dans le client.

GET /users/{id}/messages

RÉCUPÉRATION DES MESSAGES EN ATTENTE

Cette fonction est une partie extrêmement importante de l'application, car elle permet à chaque lancement de l'application de récupérer les messages envoyés quand le destinataire était hors-ligne (application fermée ou en arrière-plan). J'ai donc passé un bon moment dessus, afin de m'assurer qu'elle fonctionne, et surtout qu'elle supprime bien les messages après avoir accédé à la route.

DELETE /users/{id}

SUPPRESSION DU PROFIL UTILISATEUR

Le profil utilisateur peut être supprimé de manière sécurisée.

3.2 Serveur WebSocket

3.2.1 Introduction

Ce plan détaille les tests fonctionnels prévus pour valider les fonctionnalités du serveur WebSocket de Missive, qui gère l'envoi et la réception des messages en temps réel ainsi que la gestion des statuts des messages.

Les tests unitaires du serveur de Missive sont réalisés avec Vitest, un framework de test moderne et rapide, simple à utiliser et extrêmement bien documenté et maintenu. Il est développé par Evan You, le créateur de Vue.JS. Il permet également de réaliser des mocks de manière simple, ce qui a été crucial étant donné que le serveur de Missive dépend d'un bon nombre de composants externes, comme Prisma pour la base de données.

Ils sont organisés en ajoutant .test.ts après le nom du fichier, ce qui permet d'avoir les tests directement à côté du fichier concerné dans l'arborescence.

Ils également mis en place de manière à générer un rapport JUnit, ce qui permet d'avoir une vue dans la pipeline Gitlab des différents tests.

3.2.2 Lien de la production

Le serveur WebSocket est hébergé sous ce lien : `wss://missive.nezia.dev`

3.2.3 Scénarios de Test

Description	**Objectif**	**Statut**	**Impact**
Établir une connexion WebSocket	Vérifier l'établissement de la connexion	OK	Élevé
Envoyer un message à un utilisateur spécifique	Vérifier l'envoi du message	OK	Élevé
Recevoir un message de l'utilisateur	Vérifier la réception du message	OK	Élevé
Stocker le message si le récepteur est hors ligne	Vérifier le stockage temporaire des messages	OK	Élevé
Envoyer le message directement si le récepteur est en ligne	Vérifier l'envoi en temps réel	OK	Élevé
Mettre à jour le statut du message à 'envoyé'	Vérifier la mise à jour du statut après l'envoi	OK	Moyen
Mettre à jour le statut du message à 'reçu'	Vérifier la mise à jour du statut après la réception	OK	Moyen
Mettre à jour le statut du message à 'lu'	Vérifier la mise à jour du statut après lecture	OK	Moyen

Historique

- 2024-03-29 : Les routes `POST /tokens`, `PUT /tokens` sont opérationnelles et validées.
- 2024-05-02 : Validation des routes `POST /users/{id}/keys`, `GET /users/{id}/keys`.
- 2024-05-16 : Amélioration de la gestion des WebSocket et validation des tests d'envoi et de réception de messages.
- 2024-06-03 : Refactoring et validation des tests de mise à jour des statuts des messages.
- 2024-06-05 : Documentation du code avec docstrings.
- 2024-06-08 : Implémentation de la gestion des WebSocket
- 2024-06-13 : Implémentation des tests de mise à jour des statuts des messages à "lu".
- 2024-06-15 : Validation de la reconnexion automatique et stockage temporaire des messages.

3.2.4 Rapport de tests

ÉTABLISSEMENT D'UNE CONNEXION WEBSOCKET

L'établissement d'une connexion WebSocket depuis le serveur étant essentielle pour Missive, j'ai décidé de me concentrer sur cet élément dès les premiers stades de mon projet. J'ai pu trouver une bibliothèque, `@fastify/websocket`, qui m'a permis d'intégrer des routes WebSocket à mon serveur Fastify existant. Cela a été primordial pour plusieurs raisons :

- Avoir tout le code au même endroit m'a grandement simplifié la tâche
- Avoir le WebSocket au même endroit que mon API m'a permis de réutiliser la même authentification (le hook `authenticationHook`), ce qui est très pratique car il fallait pouvoir réutiliser la même clé publique afin de vérifier la signature du JWT

Je me suis donc attaqué à la création d'une route WebSocket à la racine de mon projet. La documentation de `@fastify/websocket` étant très bonne, j'ai pu réussir à l'implémenter très rapidement, et même pu réutiliser mon authentification avec la même syntaxe que pour mon API.

ENVOI D'UN MESSAGE À UN UTILISATEUR / RÉCEPTION

Pour envoyer un message à un utilisateur, j'ai tout d'abord commencé par réfléchir à comment j'allais faire pour garder en mémoire les connexions WebSocket : pour envoyer un message à un utilisateur spécifique, il faut utiliser la méthode `send()` de sa connexion WebSocket. J'ai décidé de partir sur une liste de connexions stockée au dessus de mes routes, dans la mémoire du serveur, car ce ne sont pas des données persistantes, et ces connexions se déconnectant / reconnectant constamment, le choix de tout stocker dans une variable était je pense judicieux dans le cas de Missive.

J'ai ensuite réfléchi à une structure en JSON afin de pouvoir envoyer ce message : j'ai tout d'abord essayé d'envoyer le message via l'ID de l'utilisateur, ce qui fonctionnait au départ, mais malheureusement, je me suis rendu compte en réalisant le client que ce n'était pas une information à laquelle on pouvait avoir accès, la seule information connue par l'utilisateur étant le nom d'utilisateur. J'ai donc décidé d'utiliser ces derniers, qui sont uniques, et je stocke toutes les connexions WebSocket dans ma liste de connexions sous une clé, la clé étant leur nom.

A chaque nouveau message, je regarde à quel destinataire il appartient, je retrouve la connexion WebSocket correspondante, et j'envoie le message. Ça fonctionne !

STOCKAGE DU MESSAGE SI L'UTILISATEUR EST HORS-LIGNE

Un des premiers défis que j'ai rencontré en programmant mon serveur WebSocket était le souci d'un des deux utilisateurs étant hors-ligne : si le destinataire est hors-ligne, comment j'allais pouvoir gérer tout ça ? C'était essentiel dans le cas de Missive, car l'utilisation d'une application de messagerie instantanée sur téléphone se fait généralement en grande majorité en dehors de l'application, l'application étant seulement ouverte pour répondre à un message ou à l'envoyer (elle est très souvent fermée).

Ayant implémenté une route pour récupérer les messages temporaires au niveau de mon API, je me suis dit que j'allais stocker le message dans la base de données, en utilisant le client Prisma défini dans l'API (encore un autre avantage du partage de code). Je suis passé par de multiples itérations de ma logique, afin de garantir une expérience responsive pour l'envoyeur et le destinataire. La logique repose principalement sur la vérification de l'existence d'une connexion WebSocket dans la liste des connexions. Si elle n'existe pas, le message sera stocké temporairement en base de données (ce qui n'est pas un problème de sécurité dans le cadre de Missive, les messages étant chiffrés de bout en bout, et supprimés à la récupération.

J'envoie un message de statut au destinataire après le stockage de son message, pour lui confirmer qu'il a bien été reçu (ce message de statut sera ensuite traité dans le client).

MISE À JOUR DES STATUTS DE MESSAGES

J'aimerais revenir plus en détails sur la mise à jour des statuts de messages, qui est une partie importante d'une application de messagerie. Cette dernière fonctionne exactement comme l'envoi des messages, avec la différence que la structure d'un message de statut est différente de celle d'un message classique (elle ne possède pas de contenu, avec une propriété `state` à la place).

3.3 Client

3.3.1 Introduction

Ce plan détaille les tests fonctionnels prévus pour valider les fonctionnalités du client Missive, développé en Flutter, qui gère l'authentification des utilisateurs, la communication sécurisée via le protocole Signal, et l'envoi/réception de messages en temps réel.

Les tests unitaires du client de Missive sont réalisés avec flutter_test, une librairie native à Flutter. Ils utilisent aussi mockito afin de pouvoir créer les différents mocks, qui dans le cas du client est crucial (le client dépend d'un bon nombre d'éléments externes, l'exemple le plus évident étant l'API).

Ils sont organisés dans le répertoire test, ce qui est la convention pour les tests en Flutter. Un rapport JUnit est également généré afin d'avoir une visualisation claire directement dans la pipeline Gitlab.

3.3.2 Scénarios de Test

Fonctionnalité	Description	Objectif	Statut	Impact
Connexion utilisateur	Authentifier un utilisateur avec ses identifiants	Vérifier l'authentification et la remise de tokens	OK	Élevé
Création de compte utilisateur	Créer un nouveau compte utilisateur	Vérifier la création de compte et l'initialisation des clés	OK	Élevé
Récupération des données utilisateur	Obtenir les informations de l'utilisateur connecté	Vérifier la récupération des données utilisateur	OK	Élevé
Déconnexion utilisateur	Déconnecter l'utilisateur et révoquer les tokens	Vérifier la déconnexion et la révocation des tokens	OK	Élevé
Stockage et récupération des messages en local	Stocker et récupérer les messages en local	Vérifier la persistance des messages en local	OK	Élevé
Envoi de message	Envoyer un message chiffré à un autre utilisateur	Vérifier l'envoi et le chiffrement des messages	OK	Élevé
Implémentation du protocole Signal	Implémenter le protocole Signal en Dart, avec libsignal_protocol_dart	Vérifier le bon fonctionnement du protocole de chiffrement	OK	Élevé
Réception de message	Recevoir un message chiffré d'un autre utilisateur	Vérifier la réception et le déchiffrement des messages	OK	Élevé
Reconnexion automatique	Reconnecter automatiquement en cas de perte de connexion	Vérifier la reconnexion automatique et la reprise des messages	OK	Moyen
Gestion des notifications	Recevoir des notifications push en arrière-plan	Vérifier la réception des notifications	OK	Moyen
Statut des messages	Mettre à jour le statut des messages (envoyé, reçu, lu)	Vérifier la mise à jour des statuts des messages	OK	Moyen

Historique

- 2024-03-27 : Mise en place de l'environnement de développement et configuration initiale.
- 2024-05-17 : Validation de l'interface de connexion et des conversations.
- 2024-05-18 : Implémentation du stockage sécurisé des clés publiques.

- 2024-05-21 : Révision du stockage des clés publiques et validation de la création de sessions chiffrées.
- 2024-05-22 : Validation de la connexion WebSocket et de l'envoi/réception de messages chiffrés.
- 2024-05-23 : Gestion de la persistance des messages locaux avec Hive.
- 2024-05-25 : Validation de l'envoi et de la réception de messages avec mise à jour des statuts.
- 2024-05-26 : Amélioration de l'interface utilisateur.
- 2024-06-01 : Validation de l'écran de conversations.
- 2024-06-02 : Implémentation de la recherche d'utilisateurs.
- 2024-06-03 : Validation de la récupération et du stockage des messages temporaires.
- 2024-06-07 : Validation des notifications push avec Firebase.
- 2024-06-13 : Validation des tests de mise à jour des statuts des messages à "lu".
- 2024-06-15 : Validation de la reconnexion automatique et du stockage temporaire des messages.
- 2024-06-17 : Validation des notifications sur iOS et Android.
- 2024-06-28 : Déploiement sur Jelastic Cloud.

3.3.3 Rapport de tests

Création de compte

J'ai tout d'abord commencé par implémenter un système d'authentification avec le client. Ayant un système d'API fonctionnelle, j'ai tout d'abord commencé par créer un écran de démarrage, qui permet d'accéder à l'écran de création de compte, et à l'écran de connexion. Ensuite, j'ai implémenté mon `AuthProvider`, qui gère l'intégralité du statut de création de compte de l'utilisateur, en utilisant l'API, et gère également les jetons, en les stockant dans le stockage sécurisé du téléphone. Le `AuthProvider` gère également l'upload des clés publiques sur le serveur, afin que les autres utilisateurs puissent les trouver.

J'ai ensuite implémenté un routeur, avec `go_router`, qui me permet d'avoir des routes nommées, et d'également de permettre une redirection automatique avec la propriété `refreshListenable`, qui écoute les changements sur mon `AuthProvider`, et dès qu'il y en a, la fonction `redirect` est appelée, qui contient la logique de redirection par rapport au statut de connexion de l'utilisateur.

Authentifier un utilisateur

J'ai rajouté une fonction de connexion dans mon `AuthProvider`, qui va venir faire un appel à l'API, et vérifier les informations de l'utilisateur. Quand la connexion réussit, les jetons sont stockés comme pour la création de compte, et l'utilisateur est renvoyé sur la page d'accueil (la page des conversations).

Récupération des données utilisateur

J'ai implémenté un getter dans mon `AuthProvider`, qui permet de récupérer les informations utilisateur si elles ne sont pas disponibles. Cela permet notamment d'afficher le nom de l'utilisateur dans la barre à gauche.

Déconnexion utilisateur

La déconnexion de l'utilisateur a été implémentée en ajoutant une fonction `logout` dans mon `AuthProvider`, qui se charge de supprimer tous les jetons, remettre le statut `isLoggedIn` à `false`, et de supprimer toutes les données de connexion. La redirection au *landing screen* est ensuite effectuée automatiquement, grâce au routeur, qui écoute les changements sur `AuthProvider`.

Envoi des messages

Une fois tout ça implémenté, car j'avais besoin des jetons pour tester la connexion en temps réel, j'ai commencé à réfléchir à comment j'allais gérer les messages en temps réel. C'est une étape qui m'a pris du temps, car je n'avais jamais travaillé avec des WebSocket auparavant, je n'étais donc pas sûr de comment le faire. Je n'avais en plus jamais fait de Flutter de toute ma vie, donc j'ai dû me pencher sur la manière dont on stocke l'état global de l'application en Flutter de manière générale. L'architecture de l'application est d'ailleurs une des choses qui m'a pris le plus de temps, car les frameworks à composants non dogmatiques ont tendance à devenir extrêmement désorganisés si on ne réfléchit pas à son architecture.

J'ai donc décidé de programmer toute la logique dans un Provider, `ChatProvider`, afin de pouvoir séparer la logique le plus possible de l'interface. Le système de widgets de Flutter pouvant devenir assez complexe à suivre, en raison de l'imbrication des différentes parties, j'ai donc implémenté pour commencer une fonction `connect()`, qui permet de connecter l'application au serveur WebSocket. Une fois que tout ça a été réalisé, j'ai implémenté une fonction `sendMessage`, qui permet d'envoyer un message en clair pour commencer à tester le WebSocket. Une fois que la bonne structure JSON a été implémentée, tout a fonctionné !

Réception des messages

Une fois les messages envoyés, il était temps de programmer la manière dont j'allais les afficher. J'ai décidé, de manière temporaire, de les afficher de manière désorganisée sur la page d'accueil afin de juste pouvoir tester si j'arrivais à avoir une application réactive. Vu que ma connexion WebSocket a été enveloppée dans un Provider, ces derniers donnent accès à une fonction, `notifyListeners`, qui permet de notifier l'application et les parties de l'interface qui ont besoin d'être rechargées. Cela veut dire qu'à chaque message, on peut utiliser `notifyListeners` afin de notifier l'interface que les messages ont besoin d'être mis à jour. Il m'a ensuite fallu utiliser un widget `ListView`, qui prend une liste quelconque, en l'occurence les messages, et permet d'afficher des widgets en itérant sur la liste automatiquement. Cela permet également d'ajouter des séparateurs, et de contrôler le comportement de visualisation, ce qui sera très utile par la suite.

Implémentation du protocole Signal

Maintenant que la connexion au WebSocket fonctionne, il est temps de s'attaquer au gros morceau du projet, à savoir implémenter le protocole Signal. Une des raisons principales pour lesquelles j'ai choisi Flutter pour mon projet, est l'existence d'une implémentation de ce protocole en Dart. Après avoir lu comment fonctionnait le protocole Signal, avec le *whitepaper* que la fondation a publiée sur son site, j'ai commencé à me pencher sur comment utiliser l'implémentation en Dart.

Pour faire simple, la librairie implémente toutes les fonctions mathématiques et la logique dont une application Signal a besoin. La manière dont la librairie fonctionne est très intéressante, car elle nous laisse libre arbitre sur la manière dont on souhaite implémenter le stockage des différentes clés privées et publiques. L'idée est que il est nécéssaire d'implémenter des classes, appelées des stores, qui vont gérer les interactions avec la manière de stocker de votre choix. Vous trouverez ci-dessous un exemple d'une de ces interfaces :

```
abstract mixin class PreKeyStore {
  Future<PreKeyRecord> loadPreKey(
      int preKeyId); //  throws InvalidKeyIdException;

  Future<void> storePreKey(int preKeyId, PreKeyRecord record);

  Future<bool> containsPreKey(int preKeyId);

  Future<void> removePreKey(int preKeyId);
}
```

Figure 39 — Classe abstraite de PreKeyStore

Ces différentes méthodes doivent être implémentées de la manière dont l'on souhaite. Le seul souci de ce procédé était malheureusement le fait que la librairie était assez mal documentée, j'ai donc dû passer beaucoup de temps à déchiffrer les implémentations et à essayer de comprendre la logique derrière.

Une fois tout ça terminé, il suffit simplement d'utiliser les classes qu'ils ont implémenté, comme `SessionCipher`, qui prend toutes nos classes et utilise leurs fonctions internes, et le chiffrement fonctionne ! J'ai fait en sorte de tester ces classes unitairement de manière assez extensive, afin de m'assurer du bon fonctionnement de ces dernières. J'ai également à terme créé un provider, `SignalProvider`, qui me permet d'envelopper la logique un peu plus bas niveau de ces derniers dans des fonctions simples, comme `encrypt`, `buildSession` et `decrypt`. Cela permet de rendre la logique beaucoup plus lisible à terme, et beaucoup plus facilement maintenable.

Reconnexion automatique

La reconnexion automatique est extrêmement importante pour Missive, car étant une application téléphone, la qualité de la connexion de l'appareil peut changer constamment car l'utilisateur est en mouvement.

J'ai donc rajouté de la logique dans ma fonction `connect()`, qui me permet de gérer tout ça. Voici à quoi la logique ressemble :

```
    Future<void> connect() async {
    ...
    // If reconnection attempts have been made, fetch pending messages as there could have been messages sent while disconnected, and reset the attempts
    try {
    if (_reconnectionAttempts > 0) {
      fetchPendingMessages();
      fetchMessageStatuses();
      _reconnectionAttempts = 0;
      notifyListeners();
    }

    _channel!.stream.listen(
      (message) async {
        await _handleMessage(message);
      },
      onDone: _handleConnectionClosed,
      onError: (error) =>
          _logger.log(Level.SEVERE, 'WebSocket Error: $error'),
    );
  } catch (e) {
    _logger.log(Level.SEVERE, 'Failed to connect: $e');
    _connected = false;
    _scheduleReconnection();
  } finally {
    _isConnecting = false;
  }
  ...
}

void _handleConnectionClosed() {
  _logger.log(Level.WARNING, 'WebSocket connection closed.');
  _channel = null;
  _scheduleReconnection();
}

void _scheduleReconnection() {
  if (_disposed) return;
  if (_reconnectionTimer != null && _reconnectionTimer!.isActive) return;

  _reconnectionAttempts++;
  final delay = min(_reconnectionAttempts * 2,
      30); // take more and more time to reconnect after each failed attempt (max 30s)
  _reconnectionTimer = Timer(Duration(seconds: delay), () {
    _logger.log(Level.INFO,
        'Attempting to reconnect... (Attempt $_reconnectionAttempts)');
    connect();
  });
}
```

Figure 40 — Logique de reconnexion automatique

Une fois cette partie implémentée, il a fallu trouver un moyen de gérer le cas où un message est envoyée si la connection se ferme, afin d'éviter de perdre un message.

Il a fallu créer une base de données locale Realm, `PendingMessages`, qui fonctionne de la même manière que le stockage des conversations. J'ai rajouté un bout de code dans `sendMessage` qui vérifie si la connexion WebSocket est établie, et si elle ne l'est pas, le message sera stocké avec `_storePendingMessage`.

Ensuite, une fois la reconnexion effectuée au WebSocket, une fonction est appelée, `_sendPendingMessages`, qui s'occupe d'envoyer l'intégralité des messages en attente. L'interface est également modifiée, au niveau de `conversation_screen.dart`, pour faire en sorte d'afficher un *spinner* de chargement si le message est en attente.

Gestion des notifications

Pour la gestion des notifications, comme dit précédemment, Firebase Cloud Messaging a été utilisé. Comme le serveur s'occupe d'envoyer les notifications, il a fallu également installer l'impléemntation de Firebase Cloud Messaging sur le client.

J'ai simplement suivi le tutoriel sur le site de Firebase, qui explique en détails comment relier son application à son projet Firebase.

Un des soucis principaux que j'ai rencontré était avec la mise en place des notifications iOS, qui a été plus complexe que la gestion sur Android. Il a fallu ajouter un bon nombre de profils et de clés dans ma console développeur Apple, ainsi que de les relier via XCode.

Une fois toutes ces opérations effectuées, il a simplement fallu instancier Firebase Cloud Messaging, ainsi que de générer un jeton qui permet de relier le périphérique de l'utilisateur à la base de données des connexions. Voici à quoi ressemble le code :

```
    if (Platform.isAndroid || Platform.isIOS) {
      FirebaseMessaging messaging = FirebaseMessaging.instance;
      notificationId = await messaging.getToken();
```

```
    } else {
      notificationId = null;
    }
```

Figure 41 — Génération de l'ID de notification

Ce code est utilisé dans la fonction `login` et `register`, car il est nécéssaire d'éviter d'envoyer des notifications si l'utilisateur est déconnecté, ou de recevoir des notifications qui ne concernent pas la connexion actuelle.

Statut des messages

La gestion des statuts de messages étant prise en compte au niveau du serveur, il a fallu également l'implémenter côté client. Cette gestion est prise en charge dans ma fonction `_handleMessage` : si une propriété `state` est présente (mise à jour de statut), et le message stocké en local sera modifié.

Ensuite, une logique a été implémentée au niveau de `conversation_state.dart`, qui permet d'afficher les petites coches, ou le spinner si le message est en attente, afin d'informer l'utilisateur du statut de lecture / envoi de leur message.

4. Bilan personnel

4.1 Bilan personnel

4.1.1 Introduction

Mon travail de diplôme sur la création de Missive a été une expérience de développement intense et enrichissante. J'ai abordé des aspects variés du développement logiciel, depuis la conception initiale de l'architecture de l'API et de la base de données jusqu'au déploiement final sur le cloud Infomaniak. Voici une rétrospective détaillée de ce que j'ai réalisé, des défis rencontrés et des apprentissages tirés de ce projet.

4.1.2 Réalisations clés

1. Conception et architecture

- Conception et mise en place d'une API robuste en utilisant la spécification OpenAPI 3.0.
- Conception d'une base de données relationnelle performante, modélisée avec Prisma.

2. Implémentation technique

- Développement et test des routes critiques pour la gestion des utilisateurs, des clés publiques, et des messages.
- Implémentation du protocole Signal pour le chiffrement de bout-en-bout des messages, avec gestion du stockage sécurisé.
- Intégration et gestion des communications en temps réel via WebSocket.

3. Déploiement et DevOps

- Création d'environnements de développement et de production robustes en utilisant Docker et Docker Swarm.
- Migration et déploiement de l'application sur Jelastic Cloud, garantissant la disponibilité et la résilience.

4. Interface utilisateur

- Développement de maquettes intuitives et de l'interface utilisateur en utilisant Figma et Flutter, avec une attention particulière à l'expérience utilisateur.
- Implémentation des notifications push pour Android et iOS via Firebase Cloud Messaging.

5. Tests et documentation

- Création de tests unitaires et d'intégration pour assurer la fiabilité du code.
- Maintien d'une documentation détaillée et accessible pour tous les aspects du projet.

4.1.3 Défis rencontrés

1. Protocole Signal

- La complexité de l'implémentation du protocole Signal et la gestion des clés publiques ont nécessité une recherche approfondie et une rigueur dans la mise en œuvre.
- Le manque de documentation sur l'implémentation en Dart a rendu la compréhension de la bibliothèque plus complexe, ce qui a nécéssité plus de temps passé dessus. L'avantage principal étant que cela m'a permis de comprendre beaucoup plus en profondeurs comment le tout fonctionnait ensemble.

2. Gestion des connexions

- La gestion des connexions WebSocket a prouvé être un défi intéressant, notamment au niveau de la gestion de son cycle de vie au niveau du client. Ce problème a été réglé rapidement, en me penchant d'avantage sur le cycle de vie des providers et des composants avec Flutter.

3. Gestion des notifications

- La gestion des notifications push sur différentes plateformes a été particulièrement délicate, notamment en raison des restrictions spécifiques à iOS. Ce défi a été surmonté en prenant le temps de bien créer les bons certificats et clés, ainsi qu'en passant du temps sur XCode afin de comprendre les différentes parties.

4. Déploiement en production

- L'intégration et le déploiement de l'application sur l'environnement Infomaniak avec Docker Swarm a posé des défis en termes de configuration et de gestion des secrets. Cela m'a poussé à réadapter mes fichiers Docker, ainsi que de passer sur une gestion des secrets différentes.

5. Tests unitaires

- La création de tests unitaires et la gestion des mocks pour Prisma et les WebSockets ont demandé une compréhension approfondie des outils de test. J'ai rencontré pas mal de difficultés avec les mocks, en particulier avec Typescript qui m'a poussé à changer plusieurs fois de librairies de test, mais j'ai finalement fini par trouvé une solution solide et maintenable.

4.1.4 Ce que le projet m'a appris

1. Compétences Techniques

- J'ai amélioré mes compétences en conception et en architecture d'API et de bases de données.
- Une compréhension beaucoup plus en profondeur des protocoles de chiffrement, en particulier le protocole Signal, et comment les implémenter dans une application.
- Des compétences renforcées en déploiement et gestion d'environnements de développement et de production avec Docker et Docker Swarm.
- Apprentissage du Dart avec Flutter, que je voulais découvrir et approfondir depuis un bon moment.

2. Gestion de projet

- Utilisation efficace d'outils comme Trello pour la gestion des tâches et la planification du projet.
- Importance d'une documentation claire et complète pour la pérennité et la maintenabilité du projet, pour moi-même ainsi que pour les autres.

3. Résolution de Problèmes

- Capacité à s'adapter et à trouver des solutions alternatives face aux défis techniques.
- Apprentissage de la persévérance et de la patience dans la résolution de problèmes complexes.

4.1.5 Conclusion

Ce projet a été une expérience inestimable qui m'a permis de développer une gamme complète de compétences en développement logiciel, de la conception à la mise en production. Les défis rencontrés ont renforcé ma capacité à résoudre des problèmes complexes et à m'adapter avec des contraintes temporelles. Je suis fier de la robustesse et de la sécurité de l'application Missive, ainsi que de la qualité de la documentation et des tests. Je suis convaincu que les compétences acquises lors de ce projet seront extrêmement bénéfiques pour mes futurs projets professionnels.

Au-delà des compétences techniques, ce projet m'a également permis de développer une certaine éthique et méthodologie de travail. J'ai appris à gérer mon temps de manière efficace, à prioriser les tâches les plus critiques, et à maintenir un niveau de qualité élevé tout au long du développement. Cette expérience m'a aussi montré l'importance de la collaboration et de la communication, des qualités essentielles pour réussir dans n'importe quel projet technologique.

5. Glossaire

Termes	Description
Authentification 2FA	Implémentation de l'authentification à deux facteurs pour renforcer la sécurité de l'application.
Bun	Runtime utilisé pour le backend en TypeScript, offrant de meilleures performances que Node.js.
Caddy	Serveur web utilisé pour gérer les certificats SSL et le reverse proxy.
ChatProvider	Composant Flutter gérant la communication en temps réel via WebSocket.
Chiffrement de bout en bout	Technique de sécurité assurant que seuls les interlocuteurs peuvent lire les messages échangés.
Docker	Technologie utilisée pour le déploiement des services backend.
Fastify	Framework utilisé pour l'API et le serveur WebSocket.
Firebase Cloud Messaging	Service de notifications push utilisé pour envoyer des notifications sur iOS et Android.
Flutter	Framework utilisé pour développer le client de l'application.
FlutterSecureStorage	Librairie Flutter utilisée pour le stockage sécurisé des clés et autres données sensibles sur les appareils des utilisateurs.
Gitlab CI/CD	Outil utilisé pour l'intégration continue et le déploiement.
Gitlab Runner	Utilisé pour automatiser la construction de l'application iOS.
Healthcheck	Vérifications de l'état du service pour s'assurer que les dépendances, comme la base de données, sont disponibles avant de démarrer l'application.
Hive	Base de données locale NoSQL utilisée pour stocker les messages de manière performante.
Jelastic Cloud	Plateforme de cloud computing utilisée pour l'hébergement de l'application.
JUnit	Format de rapport utilisé pour les résultats de tests dans la pipeline Gitlab.
Lazy loading	Technique utilisée par Realm pour améliorer les performances en chargeant les objets uniquement lorsque nécessaire.
Load balancer	Fonctionnalité intégrée à Docker Swarm utilisée pour répartir la charge entre plusieurs répliques du service.
Node.JS	Environnement d'exécution initialement considéré pour le backend avant de passer à Bun.
Notifications Push	Fonctionnalité permettant d'envoyer des notifications en arrière-plan aux utilisateurs.
OpenAPI 3.0	Spécification utilisée pour décrire l'API de l'application.
OWASP ZAP	Outil utilisé pour identifier les vulnérabilités potentielles de sécurité.
PlantUML	Outil utilisé pour créer le diagramme de la base de données.
PreKeysStore	Composant de stockage des clés publiques utilisé pour gérer les clés de chiffrement dans l'application.
Prisma	ORM utilisé pour interagir avec la base de données et générer les migrations.
Protocole Signal	Protocole de chiffrement sécurisé de bout en bout, utilisé pour sécuriser les communications.
Realm	Base de données locale NoSQL utilisée pour stocker les messages de manière performante.
SecureStorage	Utilisé pour stocker des données de manière sécurisée sur les appareils des utilisateurs.
SecureStorageManager	Gestionnaire de stockage sécurisé implémenté pour gérer les clés et les données sensibles dans Flutter.
SessionBuilder	Composant de la librairie Signal utilisé pour construire des sessions chiffrées entre utilisateurs.
Typescript	Langage utilisé pour le développement du backend.

Termes	Description
UUIDv4	Type d'identifiant utilisé pour les utilisateurs, permettant d'empêcher les attaques par devinette des identifiants.
Vitest	Framework utilisé pour les tests unitaires du serveur Missive.
WebCrypto	API de chiffrement native à Node.JS, non disponible sur React Native, influençant le choix de Flutter pour le développement multi-plateformes.

6. Table des figures

Figure	Type	Description	Section
Figure 1	**Image**	Logo de l'application	`Documentation Missive > Missive`
Figure 2	**Image**	Schéma de l'architecture de l'application	`Documentation Missive > Fonctionnement`
Figure 3	**Image**	Écran de création du compte	`Documentation Missive > Fonctionnement > Analyse fonctionnelle > Création du compte`
Figure 4	**Image**	Diagramme de séquence de la création d'un compte	`Documentation Missive > Fonctionnement > Analyse fonctionnelle > Création du compte`
Figure 5	**Image**	Diagramme de séquence de la génération des clés	`Documentation Missive > Fonctionnement > Analyse fonctionnelle > Génération des clés`
Figure 6	**Image**	Écran des conversations	`Documentation Missive > Fonctionnement > Analyse fonctionnelle > Génération des clés`
Figure 7	**Image**	Écran de recherche d'un utilisateur	`Documentation Missive > Fonctionnement > Analyse fonctionnelle > Début de la conversation`
Figure 8	**Image**	Écran de conversation	`Documentation Missive > Fonctionnement > Analyse fonctionnelle > Envoi du message`
Figure 9	**Image**	Écrans de conversation - messages lus	`Documentation Missive > Fonctionnement > Analyse fonctionnelle > Réception du message`
Figure 10	**Image**	Diagramme de composants de l'application	`Documentation Missive > Fonctionnement > Analyse organique`
Figure 11	**Image**	Mindmap du fonctionnement des Provider	`Documentation Missive > Fonctionnement > Analyse organique > Client > Concepts Flutter > Provider`
Figure 12	**Codeblock**	Arborescence de l'application	`Documentation Missive > Fonctionnement > Analyse organique > Client > Arborescence`
Figure 13	**Codeblock**	Exemple d'implémentation du store PreKeyStore	`Documentation Missive > Fonctionnement > Analyse organique > Client > Protocole Signal > Implémentation des stores`
Figure 14	**Codeblock**	Méthode encrypt de SignalProvider	`Documentation Missive > Fonctionnement > Analyse organique > Client > Protocole Signal > Utilisation des stores`
Figure 15	**Image**	Diagramme de classe de SignalProvider	`Documentation Missive > Fonctionnement > Analyse organique > Client > Protocole Signal > SignalProvider`
Figure 16	**Image**	Diagramme de séquence de SignalProvider.initialize()	`Documentation Missive > Fonctionnement > Analyse organique > Client > Protocole Signal > Initialisation du protocole`
Figure 17	**Image**	Diagramme de séquence de l'établissement d'une session	`Documentation Missive > Fonctionnement > Analyse organique > Client > Protocole Signal > Établissement d'une session`
Figure 18	**Codeblock**	Implémentation de la méthode encrypt	`Documentation Missive > Fonctionnement > Analyse organique > Client > Protocole Signal > Chiffrement d'un message`
Figure 19	**Codeblock**	Implémentation de la méthode decrypt	`Documentation Missive > Fonctionnement > Analyse organique > Client > Protocole Signal > Déchiffrement d'un message`

Figure	Type	Description	Section
Figure 20	**Image**	Diagramme de séquence technique de l'authentification	`Documentation Missive > Fonctionnement > Analyse organique > Client > Authentification > AuthProvider`
Figure 21	**Codeblock**	Logique de redirection du routeur de Missive	`Documentation Missive > Fonctionnement > Analyse organique > Client > Authentification > Routage`
Figure 22	**Image**	Diagramme de classes de la base de données Realm des conversations	`Documentation Missive > Fonctionnement > Analyse organique > Client > Conversations > Stockage des messages en local`
Figure 23	**Codeblock**	Logique d'initialisation de Missive	`Documentation Missive > Fonctionnement > Analyse organique > Client > Conversations > Affichage des conversations`
Figure 24	**Codeblock**	Implémentation du FutureBuilder	`Documentation Missive > Fonctionnement > Analyse organique > Client > Conversations > Affichage des conversations`
Figure 25	**Codeblock**	Implémentation de la logique de recherche des conversations	`Documentation Missive > Fonctionnement > Analyse organique > Client > Conversations > Démarrage d'une conversation`
Figure 26	**Codeblock**	Implémentation de la classe Debouncer	`Documentation Missive > Fonctionnement > Analyse organique > Client > Conversations > Démarrage d'une conversation`
Figure 27	**Codeblock**	Affichage des utilisateurs trouvés	`Documentation Missive > Fonctionnement > Analyse organique > Client > Conversations > Démarrage d'une conversation`
Figure 28	**Image**	Diagramme de séquence de la méthode sendMessage	`Documentation Missive > Fonctionnement > Analyse organique > Client > Conversations > Envoi d'un message`
Figure 29	**Image**	Diagramme de classe de la base de données Realm PendingMessages	`Documentation Missive > Fonctionnement > Analyse organique > Client > Conversations > Envoi d'un message`
Figure 30	**Image**	Diagramme de séquence de la méthode pour gérer les messages reçus	`Documentation Missive > Fonctionnement > Analyse organique > Client > Conversations > Réception`
Figure 31	**Codeblock**	Contenu d'un jeton d'accès	`Documentation Missive > Fonctionnement > Analyse organique > Serveur > API > Authentification`
Figure 32	**Image**	Diagramme de séquence du processus d'authentification	`Documentation Missive > Fonctionnement > Analyse organique > Serveur > API > Authentification`
Figure 33	**Codeblock**	Scope d'un jeton d'accès	`Documentation Missive > Fonctionnement > Analyse organique > Serveur > API > Autorisation`
Figure 34	**Image**	Diagramme de séquence du processus d'autorisation	`Documentation Missive > Fonctionnement > Analyse organique > Serveur > API > Autorisation`
Figure 35	**Codeblock**	Structure d'un message	`Documentation Missive > Fonctionnement > Analyse organique > Serveur > Serveur WebSocket > Envoi et réception de messages`

Figure	Type	Description	Section
Figure 36	**Image**	Diagramme du processus d'envoi d'un message	`Documentation Missive > Fonctionnement > Analyse organique > Serveur > Serveur WebSocket > Envoi et réception de messages`
Figure 37	**Codeblock**	Structure d'un message de statut	`Documentation Missive > Fonctionnement > Analyse organique > Serveur > Serveur WebSocket > Gestion des statuts de messages`
Figure 38	**Image**	Schéma de la base de données	`Documentation Missive > Fonctionnement > Analyse organique > Base de données`
Figure 39	**Codeblock**	Classe abstraite de PreKeyStore	`Documentation Missive > Plan de tests > Client > Rapport de tests > Implémentation du protocole Signal`
Figure 40	**Codeblock**	Logique de reconnexion automatique	`Documentation Missive > Plan de tests > Client > Rapport de tests > Reconnexion automatique`
Figure 41	**Codeblock**	Génération de l'ID de notification	`Documentation Missive > Plan de tests > Client > Rapport de tests > Gestion des notifications`
Figure 42	**Image**	Schéma initial de la base de données	`Documentation Missive > Journal de bord > 2024-03-28`
Figure 43	**Image**	Capture d'écran de l'interface	`Documentation Missive > Journal de bord > 2024-05-03`
Figure 44 & 45	**Images**	← Envoi d'un message à un utilisateur connecté via Postman → Envoi d'un message à un utilisateur hors-ligne via Postman	`Documentation Missive > Journal de bord > 2024-05-12`
Figure 46	**Image**	Maquette de l'authentification	`Documentation Missive > Journal de bord > 2024-05-17`
Figure 47	**Image**	Maquette des écrans de conversation	`Documentation Missive > Journal de bord > 2024-05-17`
Figure 48	**Codeblock**	Exemple de sérialisation des sessions	`Documentation Missive > Journal de bord > 2024-05-21`
Figure 49	**Image**	Chiffrement de bout-en-bout	`Documentation Missive > Journal de bord > 2024-05-22`
Figure 50 & 51	**Images**	← Écran d'accueil → Écran de connexion - informations vides	`Documentation Missive > Journal de bord > 2024-05-26`
Figure 52 & 53	**Images**	← Écran de connexion - informations remplies → Écran de chargement	`Documentation Missive > Journal de bord > 2024-05-26`
Figure 54	**Image**	Écran de recherche	`Documentation Missive > Journal de bord > 2024-06-02`
Figure 55	**Image**	Poster de Missive	`Documentation Missive > Journal de bord > 2024-06-03`
Figure 56	**Codeblock**	Exemple de règle pour tester les changements sur le submodule	`Documentation Missive > Journal de bord > 2024-06-06`
Figure 57	**Image**	Missive sur Jelastic Cloud	`Documentation Missive > Journal de bord > 2024-06-28`

Figure	Type	Description	Section
Figure 58	**Codeblock**	Arborescence de l'application (au 11 février 2024)	`Documentation Missive > Proof of concept > Journal de bord > 11 février 2024`
Figure 59	**Image**	Diagramme de fonctionnement de BLoC	`Documentation Missive > Proof of concept > Journal de bord > 11 février 2024`

7. Journal de bord

7.1 2024-03-27

Aujourd'hui, je commence mon travail de diplôme. Après un briefing avec M. Garcia¸ j'ai décidé de commencer à mettre en place l'environnement de développement de mon projet : j'aimerais m'assurer que je peux travailler sur mon projet depuis n'importe quel ordinateur, ainsi que de ne pas avoir de soucis au niveau des outils que j'utilise.

Je suis également en train de créer les différentes tâches, sur mon Trello, synchronisé aux *issues* Gitlab, qui permettront de suivre l'avancement de mon projet. J'essaie au maximum de découper mon projet en tâches logiques, ainsi de les ordonner de telle manière à ce que je puisse travailler de manière efficace. Je m'occupe également de faire un diagramme de Gantt grâce à un *power up* (extension) Trello.

Je vais commencer par architecturer mon API ainsi que ma base de données : j'utiliserais la spécification OpenAPI 3.0 pour décrire mon API, ce qui me fera gagner du temps car le format est bien documenté, et je pourrais générer de la documentation à partir de cette spécification grâce à Postman.

Pour la base de données, je vais tout d'abord créer un diagramme grâce à PlantUML, qui me permettra de visualiser les différentes tables et relations entre celles-ci. Ensuite, je vais implémenter tout ça grâce à un schéma Prisma, qui me permettra de générer du code TypeScript pour interagir avec ma base de données au niveau de l'API ainsi que de créer toutes les migrations.

En fin d'après-midi, j'ai commencé à travailler sur la spécification API, vu qu'il me restait un peu de temps. J'ai terminé de définir mes routes `/tokens`, et commence à travailler sur `/users`.

7.2 2024-03-28

Aujourd'hui, je continue de travailler sur le design de mon API. Je suis en train de terminer de définir les routes `/users`, et je vais ensuite m'attaquer à la route `/users/{id}/keys`, qui permettra la gestion des clés publiques au niveau du serveur.

Les routes `/users/{id}` étant terminées, je me renseigne un peu plus sur le protocole Signal. Je vais devoir implémenter un mécanisme qui permettra d'ajouter et retirer des clés publiques à un utilisateur de manière périodique (les pre keys sont à usage unique, et doivent être régénérées régulièrement quand le serveur n'en possède plus beaucoup). Il faut également idéalement que la clé publique signée change, de manière moins fréquente. Il faudra creuser ces détails pour voir comment je vais implémenter tout ça. Il sera important de stocker un timestamp de création pour toutes les clés, ainsi que de mise à jour pour la clé signée, pour pouvoir les régénérer au bon moment.

Je viens de réfléchir au mécanisme de mise à jour de mes clés : est-ce qu'il ne serait pas plus pratique de mettre à jour la clé publique non signée automatiquement après qu'elle ait été lue pour la première fois ? Cela permettrait de ne pas avoir à gérer de mécanisme de mise à jour côté client. Le faire du côté de l'API me semble plus simple. Il faudrait réfléchir à ça.

Au niveau des clés primaires de mes tables, je pense que des UUIDv4 seraient idéal pour les utilisateurs. Cela permettrait de pouvoir empêcher un potentiel attaquant de deviner les identifiants des ressources pouvant être accédées par identifiant, comme les utilisateurs.

Je viens de terminer le design de mon API. Je suis plutôt satisfait du résultat, et je pense que le modèle que j'ai décidé d'utiliser est plutôt robuste. Bien évidemment, des révisions seront possibles si jamais je me rends compte que quelque chose ne va pas. Les routes sont pour l'instant disponibles sur ce lien, mais à terme, elles seront disponibles directement sur une documentation Postman. Je m'attaque maintenant au design de la base de données, que je réalise avec PlantUML.

Pour la base de données, j'ai décidé de partir sur un schéma qui sépare les pre-keys signées des non signées : le nombre de clés signées étant bien inférieures au nombre de clés non signées (ces dernières étant générées en masse), cela permettra de rendre l'une des deux requêtes plus rapides.

J'ai mis à jour le schéma de base de données afin d'inclure ces nouvelles tables et relations.

Voici le résultat :

Schéma initial de la base de données

Figure 42 — Schéma initial de la base de données

Je me suis aussi occupé de rajouter des exemples de tokens dans la spécification OpenAPI, afin de simplifier la compréhension globale du projet. Journée finalement très productive, je suis content de mon avancement et me sens prêt à attaquer la suite, qui sera l'implémentation de l'API. Cependant, avant de m'y mettre, je vais encore peaufiner la documentation, et m'assurer que tout soit bien clair et défini.

7.3 2024-03-29

Aujourd'hui, ayant terminé le design de mon API et de ma base de données, je m'occupe de copier ce que j'avais pour le POC, qui va me servir de base, et de l'adapter un peu pour coller à ce que j'ai défini dans mes spécifications. Je me charge maintenant de modifier la base de données afin de coller à ce que j'ai défini dans mon diagramme.

7.4 2024-05-02

Aujourd'hui, je m'attaque à la programmation des routes. Ayant déjà une route fonctionnelle pour les tokens, je vais juste m'assurer que tout soit bien en ordre, et je vais ensuite m'attaquer aux routes `/users`.

Les routes `/tokens` sont maintenant en ordre. J'ai dû changer quelques détails, par rapport à la structure qui a été définie dans ma spécification. Je m'attaque maintenant aux routes `/users`, plus précisément à `/users/{id}/keys`. J'aimerais pouvoir mettre en place la gestion de clés publiques pour les utilisateurs, et je vais commencer par implémenter la route `GET /users/{id}/keys`, qui permettra de récupérer les clés publiques d'un utilisateur.

J'ai réussi à implémenter la route `GET /users/{id}/keys`, qui récupère la première clé publique non signée et la supprime après l'avoir récupérée, et la clé signée d'un utilisateur. J'ai rajouté un sous plugin Fastify dans mon plugin utilisateurs, qui me permet de séparer tout ça proprement. J'ai dû également rajouter une permission KEYS:READ, qui donne le droit à un utilisateur de lire les clés publiques d'un autre utilisateur.

Il va me falloir mettre à jour ma spécification, car je me suis rendu compte que la structure n'était pas tout à fait juste par rapport à mon implémentation. Je pense surtout à ma route `GET /users/{id}/keys`, qui ne renvoie pas un tableau de clés, mais bien un objet contenant une clé signée et une clé non signée. Je vais également mettre à jour les exemples pour cette route.

J'ai malheureusement rencontré des problèmes lors de l'utilisation de ma pipeline Gitlab: en effet, vu que j'utilise un submodule, la pipeline n'est pas lancée automatiquement quand des changements de documentation sont effectués. J'ai dû donc lancer la pipeline manuellement, ce qui n'est pas idéal. Je vais devoir trouver une solution pour automatiser tout ça.

J'ai réussi à faire fonctionner la pipeline correctement ! Il suffisait juste d'utiliser le nom du submodule directement dans changes, au lieu du dossier (Gitlab a l'air de vérifier le pointeur vers le commit, ce qui simplifie grandement la tâche). J'ai également mis à jour la spécification API afin d'inclure le bon corps de réponse pour `GET /users/{id}/keys` (à savoir preKey et signedPreKey).

7.5 2024-05-03

Aujourd'hui, je décide de m'intéresser de plus près à l'implémentation en elle-même du protocole Signal. Je n'étais pas sûr de comment nommer certaines choses, je suis donc allé voir un exemple que les développeurs de la librairie que je vais utiliser pour Flutter ont réalisé en Typescript, qui est disponible ici. Leur interface est extrêmement intuitive, et explique bien comment fonctionne le protocole, ainsi de comment l'implémenter au niveau du client. Voici ci-dessous une capture d'écran de leur interface :

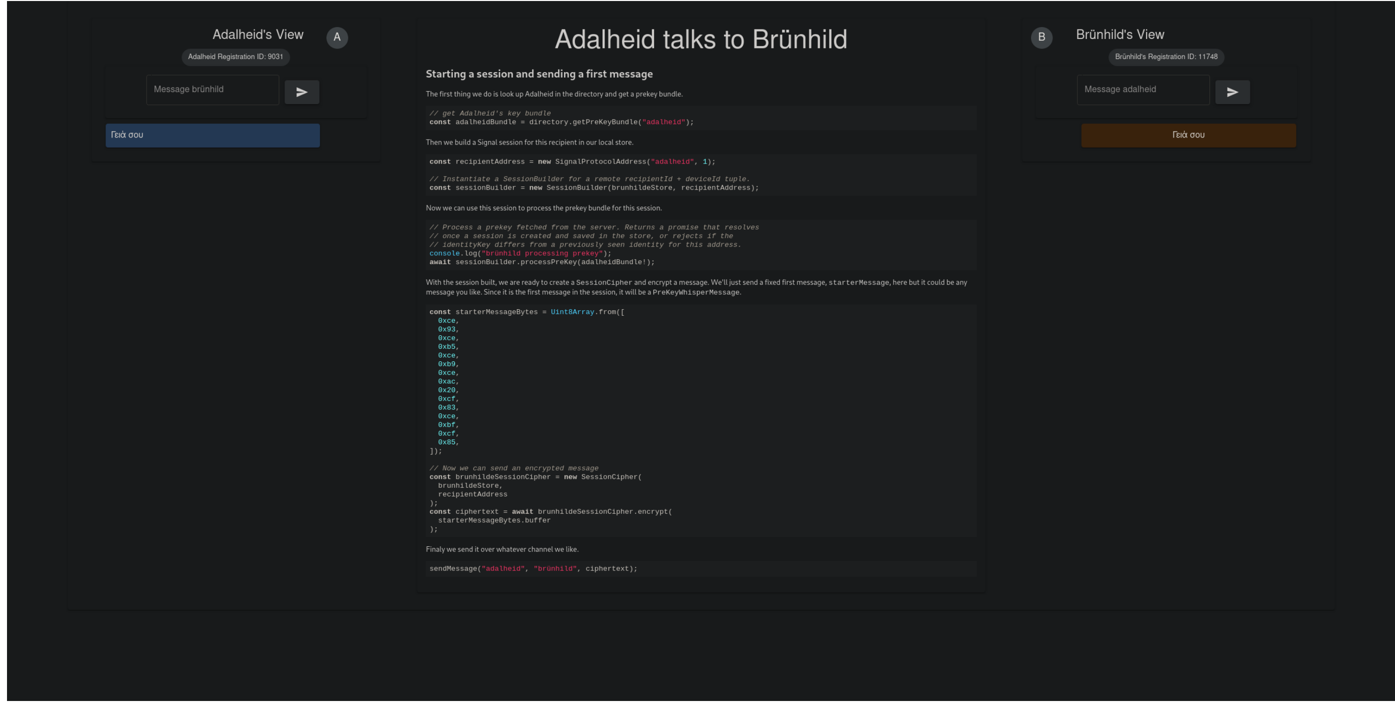


Figure 43 — Capture d'écran de l'interface

Je met donc à jour ma spécification API, afin de coller à ce que j'ai vu dans l'exemple. La première chose que je vais faire est de renommer mon champ preKey en oneTimePreKey afin d'éviter les ambiguïtés.

7.6 2024-05-04

Aujourd'hui, je me suis occupé d'écrire la spécification pour la route `POST /users/{id}/messages`, qui permettra au serveur WebSocket de stocker les messages chiffrés pour un utilisateur, si son périphérique est déconnecté.

Je viens de me rendre compte qu'il sera nécéssaire de partager une clé entre l'API et le serveur WebSocket, afin de pouvoir vérifier les jetons JWT. Je vais donc passer sur un système de signature asymétrique, qui me permettra de signer les jetons JWT côté API avec une clé privée, et de les vérifier côté serveur WebSocket avec la clé publique. Je vais donc générer une paire de clés, et stocker la clé publique côté serveur WebSocket, et la clé privée côté API. Je dois également modifier le code afin de prendre en compte ce changement.

J'ai réussi à effectuer le changement. J'ai généré une paire de clés RSA256 avec OpenSSL, et j'ai stocké la clé privée et publique à la racine de `api`, bien évidemment ignorés avec .gitignore. J'ai rajouté la procédure pour générer ces dernières dans mon README. Maintenant, les JWT sont signés côté API grâce à la clé privée, ce qui améliore encore la sécurité de mon application, et il suffira juste d'utiliser la clé publique côté serveur WebSocket pour vérifier les jetons, ce qui évite de devoir partager une clé secrète entre les deux services. Je pense que ce changement est une bonne idée, car il permet de ne pas avoir deux potentiels points d'attaque, et de simplifier la gestion des clés.

Je devrais également regarder si il ne serait pas plus judicieux de faire la même chose pour le secret des cookies.

J'ai également créé une fonction dans le fichier utils afin de charger les clés publiques et privées, pour éviter de le faire sur chaque fichier. Je suis vraiment satisfait de ma progression aujourd'hui, et je m'attaquerais au serveur WebSocket demain.

Finalement, j'ai réglé un souci avec la version de Node, qui ne supportait pas le *globbing* afin de trouver les tests unitaires. J'ai donc dû mettre à jour la version de Node, et tout fonctionne correctement maintenant. J'ai également réglé quelques soucis dans les tests car je préfixe maintenant les routes avec `/v1` (afin de garder une version stable), ce qui n'était pas le cas avant.

7.7 2024-05-05

Aujourd'hui, je vais préparer le projet afin de pouvoir implémenter les WebSocket. Je me suis rendu compte que Fastify avait des fonctionnalités qui permettent d'avoir un serveur WebSocket sur la même application. Je vais donc restructurer le projet, afin de pouvoir bien séparer les différentes parties. Il serait je pense plus pratique d'avoir un dossier `api` et un dossier `websocket`, qui contiendront respectivement l'API et le serveur WebSocket.

Je vais maintenant implémenter les routes pour les messages, qui permettront de stocker les messages chiffrés pour un utilisateur, si son périphérique est déconnecté. Il faudra également implémenter une stratégie d'authentification différente, car les messages seront envoyés par le serveur WebSocket, et non par l'utilisateur. Comme défini dans ma spécification, une simple clé d'API au format UUID sera utilisée. Il me faut donc étendre mon plugin d'authentification pour supporter cette stratégie.

J'ai étendu mon plugin d'authentification afin de supporter différentes stratégies d'authentification. Je suis encore en train de réfléchir à comment stocker la clé d'API, que je pense simplement stocker dans une variable d'environnement (elle a seulement besoin d'être vérifiée).

Mes routes `GET users/{id}/messages` et `POST users/{id}/messages` fonctionnent désormais. J'ai dû effectuer des petits changements dans la base de données, notamment pour rajouter le contenu du message ainsi que changer le nom des champs afin qu'ils soient moins ambigüs. J'ai aussi reflété ces changements de base de données dans ma spécification OpenAPI. Il faudrait également donner un moyen au serveur de rajouter l'identifiant de l'envoyeur du message, ce qui n'est pas encore fait.

7.8 2024-05-08

Aujourd'hui, je m'attaque à la partie WebSocket. Fastify ayant son propre plugin WebSocket, j'ai décidé de l'utiliser pour plusieurs raisons pratiques:

- Il est intégré directement dans Fastify, ce qui me permet de ne pas avoir à gérer un autre serveur
- Il me permet de réutiliser ma stratégie d'authentification, ce qui me facilite énormément la tâche
- Il est bien documenté, ce qui me permet de ne pas perdre de temps à chercher comment l'utiliser

Je viens également de me rendre compte que si j'utilise le plugin WebSocket de Fastify, je n'aurais pas besoin d'avoir une route pour créer les messages temporaires: je pourrais simplement utiliser Prisma, mon ORM, afin de stocker les messages directement dans la base de données. Cela me permet également d'éviter d'utiliser une clé d'API, qui n'est plus nécéssaire. Le code devra donc être adapté afin de retirer cette dernière. J'aurais beaucoup aimé avoir trouvé ce plugin plus tôt, car il m'aurait évité de perdre du temps à implémenter des routes que je ne vais pas utiliser.

J'ai réussi à implémenter tout ça. J'ai été agréablement surpris de la facilité d'implémentation, ayant fait tout le code pour gérer les autorisations. C'est un plaisir de pouvoir tout réutiliser !

Pour l'instant, j'ai seulement implémenté une route exemple. Je m'occuperais de commencer à implémenter la logique de traitement des messages demain.

Il faudra également implémenter une validation au niveau du corps de la requête, pour s'assurer que les différentes requêtes envoyées par l'utilisateur•trice sont bien formées. Fastify intègre ça de manière assez élégante et native en Typescript, en utilisant des schémas JSON qui peuvent aussi servir de types. Voici le lien vers la documentation .

Apparemment, il est possible de générer un schéma JSON directement depuis Prisma, qui prend le schéma Prisma et créé un schéma JSON qui peut être utilisé pour valider les requêtes. Cela me semble être une bonne idée, car cela me permettra de ne pas avoir à réécrire les schémas de validation, et de m'assurer que les données envoyées par l'utilisateur•trice sont bien formées. Le souci étant que les requêtes, typiquement celle de connexion, n'est pas la même que le schéma de la base de données (il suffit de donner un nom et un mot de passe). Il faudra donc que je réfléchisse à comment je vais gérer ça. Voici le lien vers la documentation.

J'ai également refactor un peu mon application, et transformé mon client Prisma en plugin Fastify. Cela me permet de gérer l'instance de Prisma de manière beaucoup plus propre, ainsi que d'alléger les routes (mon client est maintenant importé une seule fois, et peut être utilisé partout dans l'application).

Finalement, j'ai aussi créé un Dockerfile de base, qui me permet de servir mon application facilement. Il y a encore quelques points de friction, comme un message `FATAL: database "prisma" does not exist". Il faudra que je m'en occupe plus tard. Également éviter de stocker le mot de passe de la base de données en clair dans le Dockerfile.

7.9 2024-05-09

Aujourd'hui, je me suis occupé de nettoyer un peu le code de certaines fonctions que je n'utilisais pas, et ait supprimé des importations inutiles. J'ai également supprimé la route `POST /users/{id}/messages`, qui n'est pas nécéssaire sachant que le serveur WebSocket peut interagir directement avec la base de données via Prisma. J'ai également mis à jour la spécification OpenAPI pour refléter ce changement, et je m'attaque maintenant à documenter tout ça. Ayant intégré un bon nombre de fonctionnalités, je vais m'occuper de documenter tout ça dans la page Fonctionnement.

7.10 2024-05-11

Aujourd'hui, journée documentation. Je m'occupe de documenter le travail qui a été fait durant ces derniers jours, afin de s'assurer que tout soit bien clair. Je vais également commencer à réfléchir à un plan de tests.

J'ai également réussi à faire fonctionner un déploiement sur un VPS Amazon Lightsail à l'aide du `docker-compose.yml`. J'ai eu un peu de peine, car j'avais eu des soucis de versions, notamment liés au générateur automatique de fichier dbdiagram que j'avais mis en place, mais après être repassé sur le générateur Prisma par défaut, tout fonctionne. J'ai donc rajouté un reverse proxy sur le `docker-compose.yml` pour rediriger les requêtes vers le bon service.

7.11 2024-05-12

Aujourd'hui, j'ai travaillé sur l'envoi des messages entre différents utilisateurs. J'ai commencé par créer la connexion WebSocket en la stockant dans une Map qui contient en clé, l'id de l'utilisateur à qui elle appartient, et en valeur, la connexion WebSocket. Cela me permettra de directement envoyer des messages à un utilisateur spécifique si il est connecté.

Ensuite, j'ai implémenté la logique derrière l'envoi. Elle fonctionne comme ceci:

1. L'envoyeur envoie un message à un destinataire (après avoir vérifié son identité)
2. Le serveur vérifie si l'envoyeur est connecté
3. Si l'utilisateur est connecté, le serveur envoie le message directement au destinataire et informe l'envoyeur que le message a bien été reçu
4. Si l'utilisateur n'est pas connecté, le serveur stocke le message dans la base de données, et informe l'envoyeur du statut du message dès qu'il est stocké
5. Après stockage du message, le serveur WebSocket envoie une notification push au destinataire, qui lui permettra de récupérer le message. Le statut du message change également en "remis", et l'envoyeur est informé. (cette étape n'est pas encore implémentée)

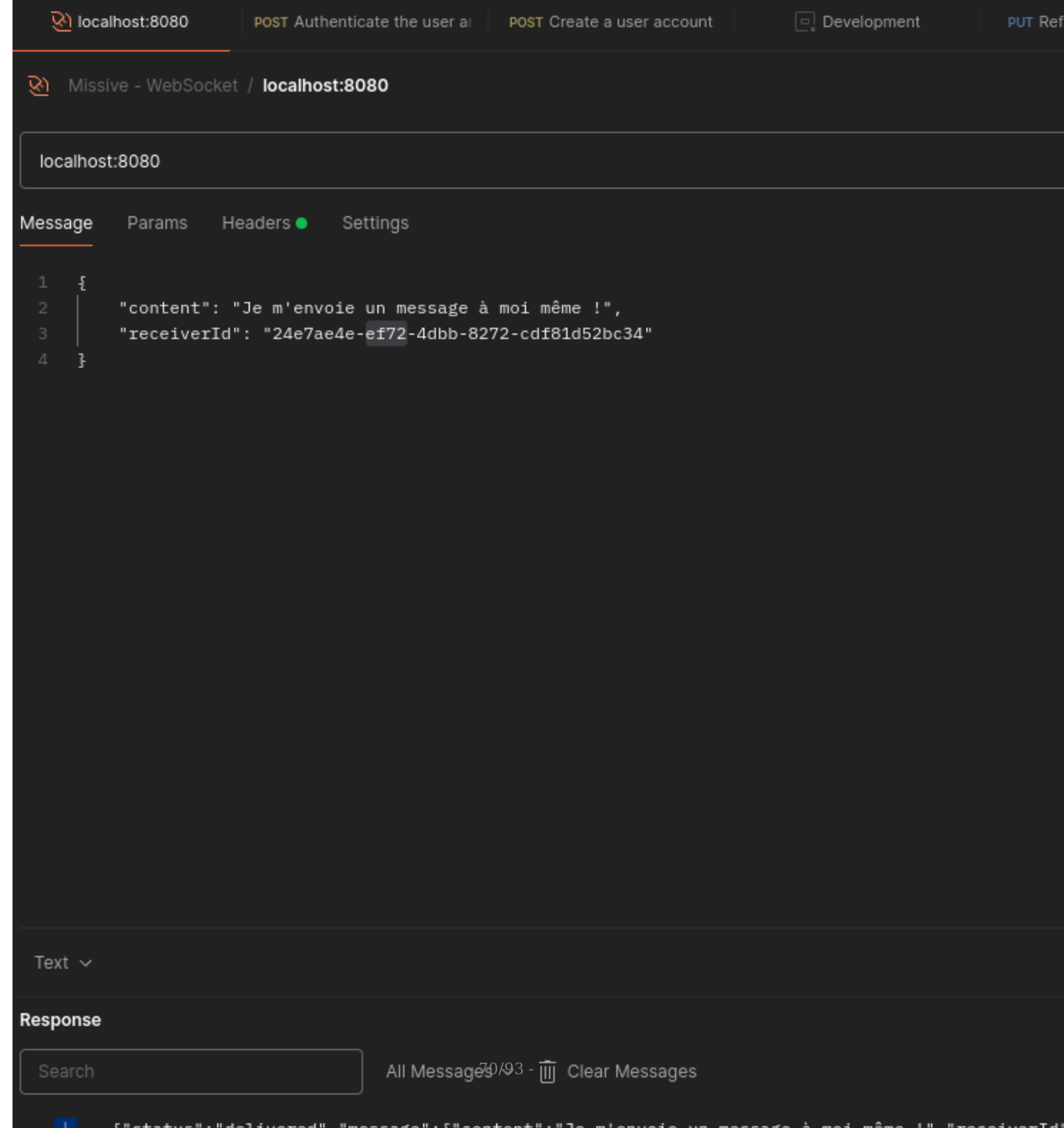

localhost:8080
POST Authenticate the user ar
POST Create a user account
Development
PUT Refr
Missive - WebSocket / localhost:8080
localhost:8080
Message
Params
Headers
Settings
{
"content": "Je m'envoie un message à moi même !",
"receiverId": "24e7ae4e-ef72-4dbb-8272-cdf81d52bc34"
}
Text
Response
Search
All Messages
Clear Messages

Figure 44 & 45 — Envoi d'un message à un utilisateur connecté via Postman | Envoi d'un message à un utilisateur hors-ligne via Postman

J'ai rencontré quelques soucis avec cette approche, et quelques points qui ne sont pour l'instant pas encore clairs:

- Si le destinataire récupère le message depuis la base de données, comment informer l'envoyeur que le message a bien été reçu ? En sachant que l'idée de base était de récupérer avec une route dédiée dans l'API, et non directement dans le serveur WebSocket.
- Si l'envoyeur envoie un message à un utilisateur et ferme ensuite son application, il y a une forte possibilité pour que le statut du message ne soit jamais mis à jour, car il compte sur la connexion WebSocket pour être informé du statut du message.

Je pense que ces deux problèmes pourront être réglés en ajoutant le statut du message dans la base de données. Cependant, il faudrait quand-même tester, car il est possible que cela ne soit pas suffisant.

7.12 2024-05-15

Aujourd'hui, je vais passer du temps sur la documentation. J'ai beaucoup avancé et après une discussion avec M. Garcia, il m'a fortement conseillé de commencer à la polir. Je vais donc m'occuper de tout ça. Vu qu'il était actuellement avec un élève, j'ai décidé de séparer mes déploiements Docker : Caddy ayant besoin d'un nom d'hôte différent et d'une configuration différente en développement (car il ne peut pas générer de certificat HTTPS depuis localhost), il m'a fallu séparer tout ça et créer un Caddyfile spécifique pour le développement. Tout a l'air de fonctionner maintenant, mais petit souci : quand je lance le docker compose pour la première fois, les migrations ne fonctionnent pas car il me semble que la base de données se lance toujours après le serveur, malgré les efforts que j'ai mis en place pour régler ce problème. Il faudra se pencher un peu plus sur ça.

Il faudrait :

- Mettre en place une architecture de documentation qui permette d'être imprimable facilement (tester avec mkdocs-with-pdf)
- S'assurer que l'application soit compatible avec les normes de protections de données en Suisse
- Faire un plan de tests solide

J'ai ajouté des paramètres VSCode afin de pouvoir garantir une impression standard du code, en enlevant tout ce qui n'est pas nécéssaire et non écrit par moi-même (node_modules etc.).

Je m'occupe maintenant de réaliser un plan de tests, que je vais effectuer tout d'abord sur Postman, et que j'inclurais ensuite dans ma documentation. Il faudra potentiellement des tests unitaires au niveau du code, mais il faut encore que je réfléchisse à ça.

J'ai rajouté un plan de tests dans ma documentation. J'ai également programmé quelques tests d'API dans Postman, mais il faudrait continuer demain afin d'avoir un *coverage* plus complet.

7.13 2024-05-16

Aujourd'hui, j'ai encore travaillé sur la documentation. J'ai rajouté des explications ainsi que des diagrammes sur le fonctionnement de l'API et du serveur WebSocket.

Maintenant, je suis en train de m'attaquer aux maquettes de l'application mobile, qui me permettront de mieux visualiser comment l'application va fonctionner. J'ai décidé de partir sur un design simple, qui reprend les éléments de base de ce qu'on attend d'une application de messagerie. J'ai également décidé de partir sur un design Material, qui est bien documenté et qui me permettra de gagner du temps. Il est aussi intégré directement dans Flutter.

7.14 2024-05-17

Aujourd'hui, j'ai continué à travailler sur les maquettes. Je commence à avoir quelque chose qui ressemble à ce que je veux, en ce qui concerne l'authentification et les conversations. Voici des captures d'écran des maquettes:

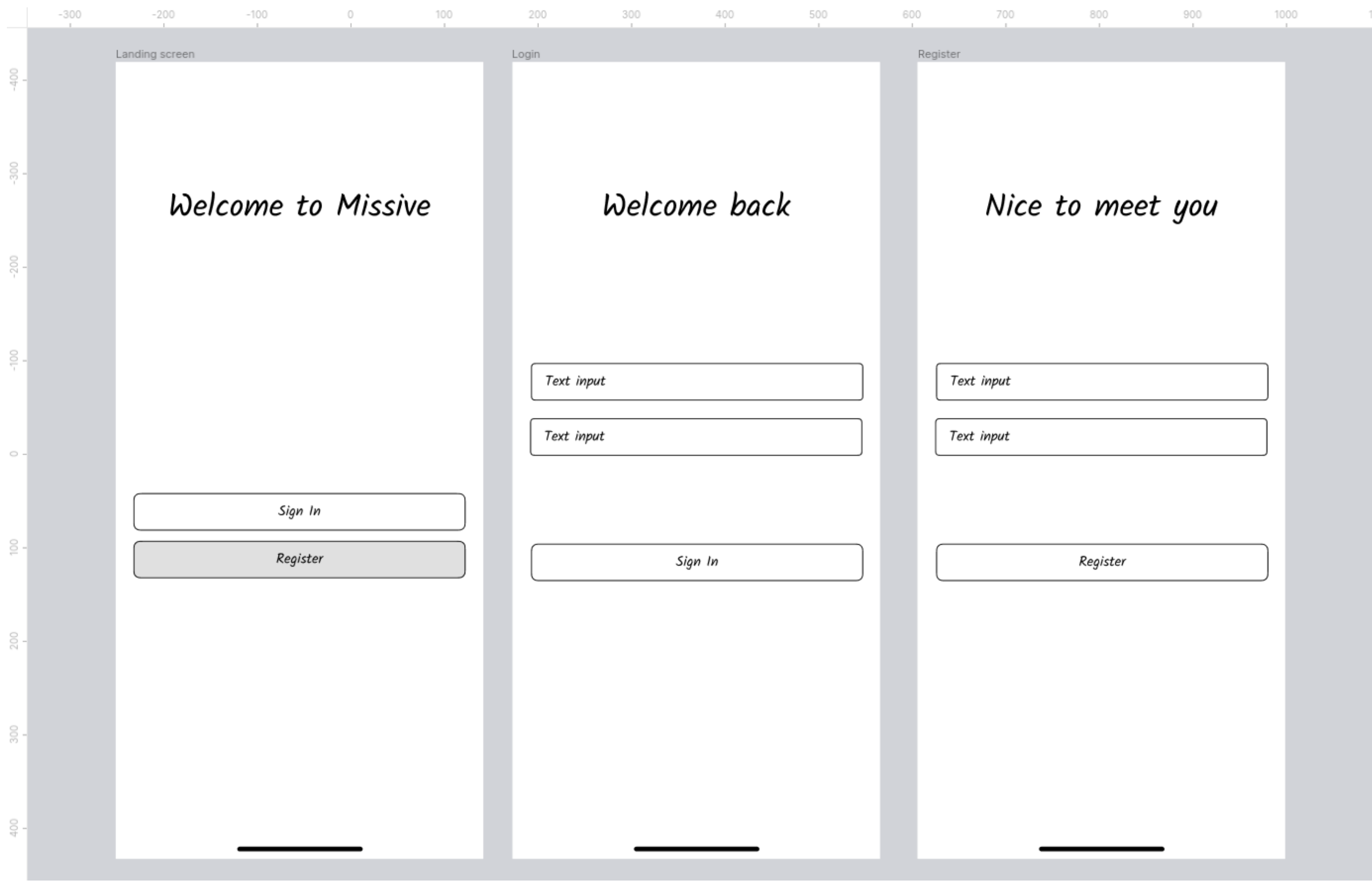


Figure 46 — Maquette de l'authentification

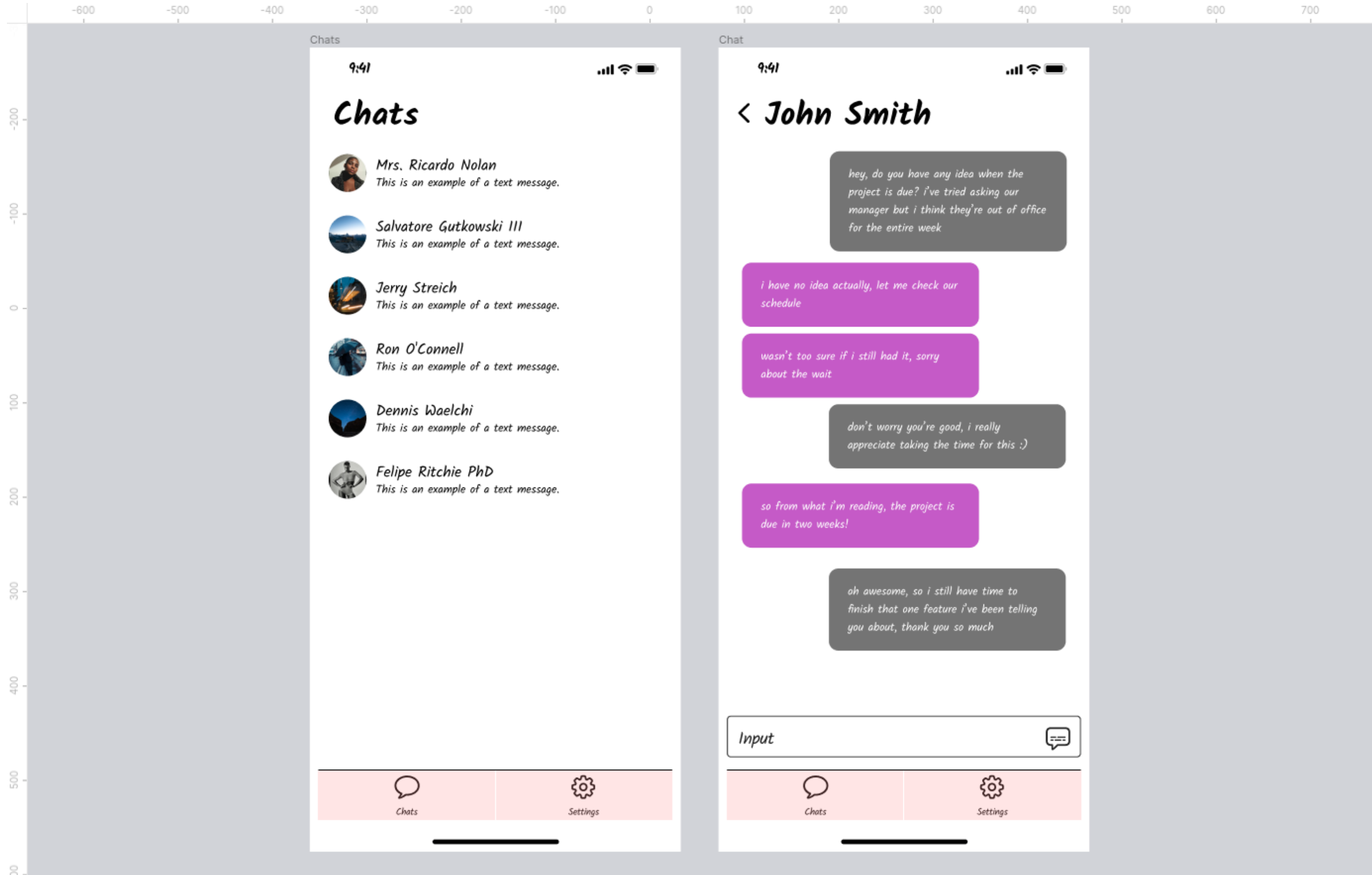


Figure 47 — Maquette des écrans de conversation

Il me reste encore à m'occuper des maquettes de l'écran de paramètres. Accessoirement, je vais commencer à m'occuper de copier ce que j'avais sur le POC dans mon dossier `client`. Il est une bonne base sur laquelle j'avais passé beaucoup de temps, et j'aimerais bien pouvoir réutiliser ce que j'avais déjà fait.

Finalement, vers la fin de journée, j'ai adapté le client afin de fonctionner avec les nouvelles routes, et la nouvelle addresse, qui sont légèrement différentes. J'ai aussi simplifié le processus d'authentification et le stockage des jetons (j'ai finalement tout stocké dans le SecureStorage), et j'ai aussi changé isLoggedIn, qui vérifie juste si le refreshToken est défini dans le SecureStorage. Après de multiples tests, l'application fonctionne exactement comme mon POC, ce qui est très rassurant pour la suite.

Je pensais peut-être rééssayer Dio comme client HTTP afin de simplifier la gestion des requêtes, mais il faudrait que je regarde si j'aurais les mêmes erreurs que pendant le POC.

J'ai aussi pu commencer à implémenter mon client Signal: j'ai commencé par réfléchir à comment j'allais stocker tout ça, et Flutter a une excellente librairie, flutter_secure_storage, qui se connecte directement au stockage sécurisé de l'appareil.

J'ai donc décidé que j'allais stocker toutes ces données sensibles dans ce stockage sécurisé, sous forme de JSON. Si il y a un object à stocker, il sera sérialisé auparavant en base64.

Aujourd'hui j'ai pu implémenter une partie du PreKeysStore, qui va s'occuper de la gestion des clés publiques. La méthode storePreKey et loadPreKey ont l'air de bien fonctionner, après m'être pris la tête avec la sérialisation.

7.15 2024-05-18

Aujourd'hui, j'ai continué sur ma lancée avec l'implémentation du protocole Signal. Pour l'instant, j'ai rajouté un store complètement fonctionnel pour les clés d'identité, qui permet de stocker:

- Notre paire de clés d'identité
- Les clés publiques d'identité des autres utilisateurs

Il permet aussi de le stocker et de les retrouver de manière sécurisée, en utilisant le SecureStorage de Flutter. Il sérialise les objets en base64 (et ensuite en JSON si nécéssaire, dans le cas d'une liste ou d'un tableau de Maps), et les désérialise de la même manière. J'ai eu quelques soucis avec le flux de données, car l'implémentation des classes en Signal propose uniquement une sérialisation en UInt8List (liste d'entiers signés sur un octet), et non en String. J'ai donc dû rajouter pas mal de logique pour gérer tout ça.

Maintenant, après avoir continué à coder, je me suis rendu compte que je n'avais pas implémenté la fonctionnalité pour rafrâichir le jeton d'accès : j'ai donc décidé d'utiliser Dio à la place, qui est un autre client HTTP(S) en Dart. Il permet d'avoir une configuration globale, et de plus facilement gérer mes requêtes.

J'ai finalement réussi à implémenter le rafraîchissement automatique du jeton, en mettant la logique dans le getter accessToken. Cela me permet d'abstraire complètement le processus.

Je suis en train de m'occuper du stockage des pré clés, je viens de me rendre compte de quelque chose : il faudra également synchroniser les clés publiques avec le serveur. Il sera important de s'assurer que l'état entre le client et le serveur soit le même, car les clés publiques sont utilisées par un utilisateur qui souhaiterait nous envoyer un message, afin de réaliser son Diffie-Hellman. Il faudra donc implémenter une méthode qui permet de synchroniser les clés publiques avec le serveur, qui devra être appelée à chaque modification des clés publiques. Il faudra peut-être que ça fonctionne dans le cas où on perd la connexion.

Je viens de finir d'implémenter le store des pré clés signées. Il faudra maintenant tester un peu tout ça, voir si il est possible de chiffrer un message.

Je viens également de me rendre compte de quelque chose : au niveau du serveur, ma manière de stocker les clés publiques n'est pas bonne. En effet, pour une application Signal, il faut stocker ce qu'on appelle un "pre key bundle", qui contient :

- La clé publique d'identité
- La pré-clé publique signée
- La signature de la pré clé publique signée
- (optionnel) une pré clé à usage unique

Malheureusement, avec l'API que j'ai actuellement, il n'y a pas moyen de récupérer tout ça simplement. Il faudra donc que je réfléchisse à comment je vais gérer ça, potentiellement réorganiser certaines de mes routes. Je verrais tout ça demain.

Il faudra également aussi stocker le registrationID de l'utilisateur ainsi que son deviceID. Il est souvent recommandé de faire une table à part pour les appareils, mais je pense que pour l'instant, je vais sûrement le mettre dans la table des utilisateurs. Si on voudrait améliorer le système dans le futur et le rendre multi-appareils, il faudra réfléchir à le séparer.

7.16 2024-05-19

Aujourd'hui, je vais commencer à revoir un peu la manière dont fonctionne mon API. Il faut que je rajoute différents champs à la création de l'utilisateur :

- registrationID
- identityKey

Ces champs peuvent être stockés dans la table des utilisateurs. Je vais donc mettre à jour ma spécification OpenAPI, et mon schéma Prisma afin de refléter ces changements. J'aimerais implémenter :

- Une route pour créer un bundle de clés
- Une route pour récupérer un bundle de clés d'un utilisateur
- Une route pour effectuer la rotation des clés uniques

Je me suis également rendu compte qu'un setup avec plusieurs appareils rajouterait beaucoup trop de complexité. Nous allons donc assumer un ID de périphérique de 1 pour tous les utilisateurs, qui va simplifier énormément la tâche (ID utilisé dans la documentation du protocole Signal en Dart).

J'ai énormément travaillé ce matin. J'ai pu tout d'abord implémenter la page de création de compte, ce qui m'a obligé à faire une landing page pour l'application afin de pouvoir soit se créer un compte, soit se connecter.

Ensuite j'ai mis à jour ma spécification OpenAPI, mon schéma Prisma, ainsi que mon API pour prendre en compte les nouveaux champs. J'ai modifié la route pour créer le bundle de clés afin d'inclure tous les champs nécéssaires, j'ai modifié la route pour récupérer un bundle de la même manière. J'ai aussi réglé quelques problèmes de permissions qui étaient mal configurées (les utilisateurs n'avaient pas la permission `keys:write` par défaut).

J'ai aussi changé la manière dont je stockais les clés publiques dans le client Signal, afin de pouvoir stocker un bundle de clés, qui contient la clé publique d'identité, la clé publique signée, la signature de la clé publique signée, et une clé publique à usage unique. J'ai également rajouté un champ pour stocker le registrationID, qui est également nécéssaire pour notre implémentation.

Ensuite, j'ai fini par implémenter mon flow de création de compte au niveau du client. Il fonctionne de cette manière :

1. L'utilisateur rentre son nom d'utilisateur et son mot de passe
2. L'application génère une paire de clés d'identité, un registrationID, ainsi qu'une clé publique signée
3. L'application stocke les clés dans le SecureStorage
4. L'application envoie ces clés au serveur, qui les stocke (les clés publiques sont sérialisées, puis encodées en base64)

J'ai du créer un nouveau Provider, qui me permet de gérer ce processus de manière globale à l'application. Je pourrais également m'en servir quand j'aurais besoin de réactivité (quand on créé une session, par exemple).

7.17 2024-05-21

Aujourd'hui, j'ai pu me rendre compte de quelques soucis dans mon implémentation actuelle:

- Les différentes clés et données étaient et sont toujours mal stockées (elles ne sont pas sérialisées dans le SecureStorage, ce qui les rend impossible à récupérer)
- Mon API ne permettait pas de récupérer un bundle avec le nom d'utilisateur (ce qui est la seule donnée que l'on a à ce moment-là)

J'ai donc changé ma spécification OpenAPI afin de prendre en compte ces changements, et j'ai modifié mon API pour qu'elle prenne un nom d'utilisateur pour les clés. Au niveau du client, j'ai rajouté une méthode pour récupérer un bundle de clés. J'ai aussi du modifier mon implémentation de SignedPreKeys. Il faudra modifier les autres fichiers afin de tout bien stocker en base64 car sérialiser ne suffit pas, vu que SecureStorage ne prend que des chaînes de caractères. Les stocker juste en Uint8List ne permettrait donc pas de le déserialiser, car la représentation en chaîne de caractères est stockée et non les données en elles-mêmes.

J'ai quand même décidé de continuer à travailler sur la création de sesssions, avec une méthode de SignalProvider, buildSession. Cette dernière prend un nom d'utilisateur, un jeton d'accès et permet de créer une session chiffrée avec un autre utilisateur. Tout avait l'air de fonctionner, donc je suis passé sur la fonction d'envoi de message. C'est là que j'ai rencontré un problème : je n'arrivais pas à chiffrer le message, car j'avais énormément d'erreurs de type, et certaines valeurs manquantes.

Je me suis donc penché sur le problème, et j'ai réalisé que les sessions étaient mal sérialisées :

```
@override
  Future<void> storeSession(
      SignalProtocolAddress address, SessionRecord record) async {
    final sessions = await _getSessions();

    if (sessions == null) return;

    sessions[address.toString()] = base64Encode(record.serialize());

    await _secureStorage.write(key: 'sessions', value: jsonEncode(sessions));
  }
```

Figure 48 — Exemple de sérialisation des sessions

Je ne m'étais pas rendu compte de mon erreur, mais le `if (sessions == null) return;` empêchait de stocker les sessions si elles n'étaient pas déjà stockées. J'ai donc enlevé cette ligne, et initialisé une Map vide si les sessions n'étaient pas déjà stockées. J'ai également rajouté une méthode pour récupérer les sessions, qui permet de les désérialiser correctement.

Vu que la librairie est vraiment bien faite, après avoir implémenté toutes ces interfaces, il suffit juste de les passer dans le SessionBuilder, puis dans le SessionCipher pour chiffrer un message. La librairie s'occupe toute seule d'appeler les différentes méthodes afin de gérer l'état de la session.

J'ai testé la création de session avec l'utilisateur actuellement connecté, car je n'ai pas encore pu tester avec un autre utilisateur. Il me semble que tout fonctionne, je n'ai pas eu d'exceptions après avoir réglé le souci de sessions ! Il faudrait tester avec un autre utilisateur demain, afin de s'assurer que les clés soient les bonnes valeurs, car si ne serait-ce que l'une des clés est mauvaise, le déchiffrement ne pourra pas fonctionner.

7.18 2024-05-22

Aujourd'hui, je me suis attaqué à la connexion WebSocket quand l'utilisateur se connecte à l'application. J'ai pour cela commencé à créer un Provider qui me permet de gérer la communication en temps réel, et je me suis rendu compte que mon serveur WebSocket prenait malheureusement un ID au lieu d'un nom d'utilisateur. Il va donc falloir modifier ça, car il est seulement possible d'envoyer un message avec des données qu'on connaît déjà (l'ID est généré par le serveur, et n'est pas connu de l'utilisateur).

En tout cas, le lancement de l'application sur l'ordinateur de l'école marche correctement ! J'ai eu quelques soucis liés au cache de l'application qui était encore présent de l'ancienne version, mais après avoir tout nettoyé, tout fonctionne correctement. Il faudra rajouter de la gestion d'erreurs, car pour l'instant, l'application crash si on lui fournit un nom d'utilisateur qui n'existe pas à la récupération du bundle de pré clés.

J'ai réussi à faire fonctionner le chiffrement de bout-en-bout !! Voici une capture d'écran du processus en action:

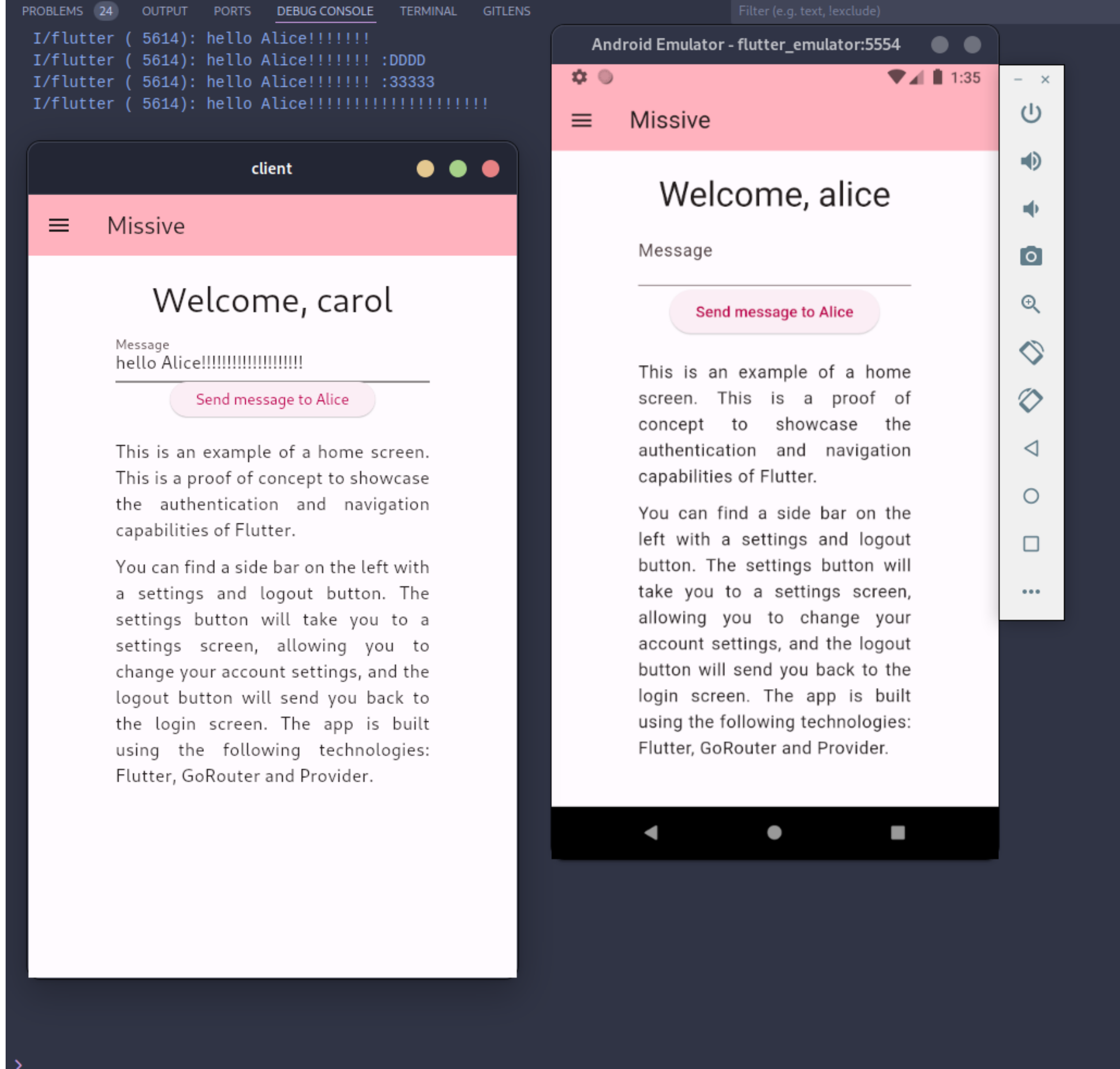


Figure 49 — Chiffrement de bout-en-bout

Malheureusement, en essayant d'ajouter un StreamBuilder afin de visualiser le flux des messages en temps réel, je me suis heurté à un problème : ayant du marquer les variables en "late", ce qui signifie qu'elles seront initialisées plus tard dans le code, mon interface n'attend pas la connexion au WebSocket et crashe complètement. Il sera possible de régler ça avec un FutureBuilder, un widget qui permet d'attendre la fin d'un Future avant de construire le widget. Cela nous permettra aussi d'avoir un écran de chargement pendant le processus de génération des clés, ce qui sera beaucoup plus agréable et propre pour l'expérience utilisateur.

7.19 2024-05-23

Aujourd'hui, je me suis rendu compte de quelques soucis sur l'application :

- Quand on se déconnecte, puis se reconnecte à l'application, les clés peuvent changer dans le cas où l'utilisateur a créé un nouveau compte entre temps. Il faudrait potentiellement stocker toutes les clés des utilisateurs qui se sont déjà connectés, et utiliser les bonnes clés en fonction de l'utilisateur. Pour l'instant, ce n'est pas un problème car on teste sans jamais se déconnecter, mais cela serait une amélioration future cruciale. Il suffirait simplement d'implémenter la bonne logique dans les stores Signal.

J'ai réussi à faire fonctionner tout ça : j'ai simplement rajouté une classe, SecureStorageManager, qui fonctionne comme un FlutterSecureStorage mais qui rajoute un préfixe à chaque clé (le nom de l'utilisateur à qui les données appartiennent). Cela permet d'avoir plusieurs comptes connectés par périphériques, et à terme, de pouvoir gérer ces clés (si on veut les renouveller, les supprimer...). Cela a réglé un grand nombre de problèmes que j'avais, notamment les soucis à la déconnexion et la perte de l'état global. Je suis donc très content car tout fonctionne comme prévu ! J'ai aussi réglé la logique de déconnexion et de connexion, qui remet l'état de l'application à zéro si jamais le compte qui vient de se connecter est nouveau sur l'appareil.

Maintenant que tout fonctionne, il va falloir s'occuper du stockage des messages. En effet, le protocole Signal s'occupant uniquement du chiffrement, il va falloir trouver une solution pour persister les messages déchiffrés sur le périphérique.

Après avoir cherché un peu et étudié les différentes solutions, je pense partir sur Hive, qui est une implémentation d'une base de données NoSQL en Dart. Elle est rapide, simple d'utilisation, et permet également de créer des bases de données chiffrées. Je pense que stocker les messages dans le secure storage serait une assez mauvaise idée, car les requêtes pourraient rapidement devenir lentes et peu performantes. Il faudra donc que je réfléchisse à comment je vais gérer tout ça.

7.20 2024-05-24

Aujourdhui, j'ai commencé à m'occuper du stockage persistant des messages, qui pour l'instant sont seulement stockés dans la mémoire vive de l'application : l'idée est de tout stocker sur Hive à la réception d'un message, ou d'une mise à jour de statut, et de les chiffrer grâce au chiffrement qui est prodigué par Hive.

J'ai eu beaucoup de mal à comprendre comment le tout fonctionnait, et j'ai également pu régler quelques bugs, mettre un peu à jour l'apparence de l'application. Il faudra continuer à travailler sur ce stockage demain. J'ai eu des soucis de type, qui ont été quasiment réglés par l'utilisation de ce que Hive appelle un adapteur, qui permet de typer les données stockées dans la base de données. Malheureusement, en essayant de stocker directement mon message, je me suis heurté à un souci de cast qui ne fonctionnait pas bien. Il faudra que je réfléchisse à tout ça demain. J'ai trouvé ce lien, qui explique comment stocker des objets non primitifs dans Hive, et je vais m'occuper d'implémenter tout ça demain.

7.21 2024-05-25

Aujourd'hui, ayant travaillé depuis la maison, j'ai passé la matinée à recréer mon environnement. C'est en faisant ça que je me suis rendu compte qu'il y avait quelques soucis avec ma configuration, notamment au niveau de la configuration du reverse proxy en développement : il écoutait les requêtes sur localhost, ce qui faisait qu'il était impossible d'y accéder depuis un autre appareil sur le réseau local. J'ai donc du modifier tout ça. Je me suis aussi heurté à quelques soucis au niveau de la configuration de mon environnement Flutter, notamment une configuration Gradle qui ne correspondait pas à la version de Java sur la machine. J'ai donc modifié tout ça.

J'ai aussi rajouté une configuration VSCode afin de pouvoir lancer les deux clients en même temps et en mode debug. J'ai aussi configuré le client afin d'utiliser des variables d'environnement, pour éviter de hard coder les valeurs de l'API.

On peut maintenant utiliser `--dart-define=API_URL=http://localhost:3000` pour changer l'URL de l'API, et `--dart-define=WEB_SOCKET_URL=ws://localhost:3000` pour changer l'URL du serveur WebSocket. Ces dernières sont déjà configurées dans le `launch.json`, afin de fonctionner avec un serveur qui est lancé en local.

J'ai réussi à régler les soucis de type avec Hive ! J'utilisais mal les génériques en Dart, car Hive ne supporte pas les génériques imbriquées pour le type de la Box. Le stockage fonctionne maintenant parfaitement !! Il me reste juste à l'implémenter pour l'envoi des messages, donc à la réception d'une mise à jour de statut. Il faudrait également récupérer les messages temporaires au démarrage de l'application, ainsi que de les stocker comme il faut. Il faudrait également résoudre le souci de la mise à jour des

statuts dans le cas où l'une des deux personnes n'est pas connectée au WebSocket. Je pense qu'on pourrait le faire dès qu'un utilisateur se reconnecte au WebSocket, et de le stocker en base de données en attendant que l'autre se reconnecte, mais pour l'instant, ce n'est pas la priorité.

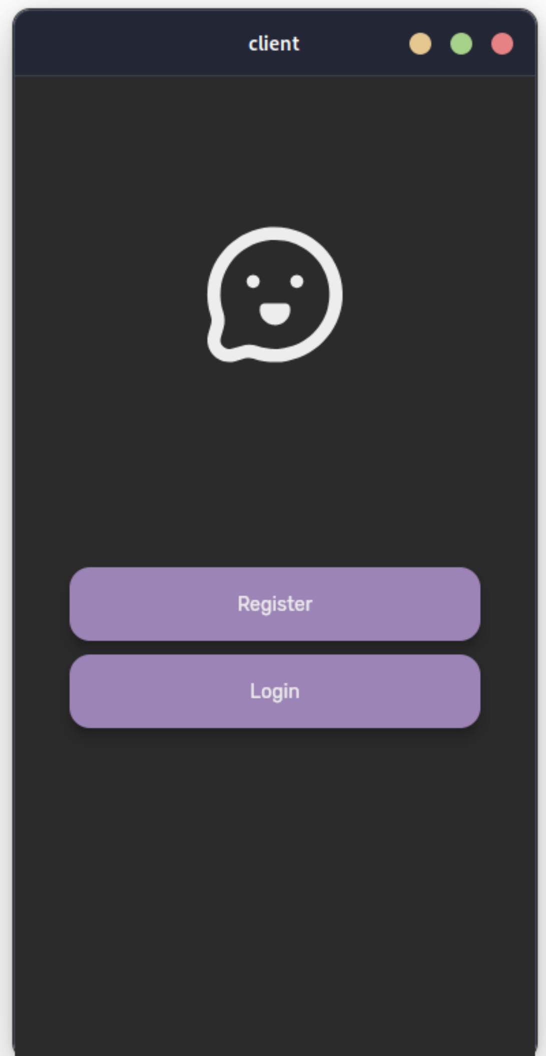
7.22 2024-05-26

Aujourd'hui, j'ai modernisé le look de mon application. J'ai essayé différentes palettes que j'ai généré grâce à ChatGPT, et j'ai finalement trouvé une palette qui me plaisait. J'ai donc changé les couleurs de l'application, et j'ai également mis à jour l'apparence de mes boutons, mes entrées texte afin d'avoir une apparence un peu plus agréable et fluide.

J'ai également implémenté le stockage des messages envoyés en local, dans la conversation du destinataire.

Il faudra également rajouter un ID au message, généré depuis le client, afin de pouvoir mettre à jour les statuts (envoyé, reçu, lu) de manière correcte. Il faudra également rajouter un champ pour stocker le statut du message, qui sera mis à jour en fonction des événements reçus du serveur WebSocket. Il faudra encore s'assurer que même si l'un des deux utilisateurs est déconnecté, le statut du message sera mis à jour au démarrage de l'application.

J'ai également rajouté un écran de chargement qui prend tout l'écran pendant le chargement des clés, afin de ne pas laisser l'utilisateur dans le flou, et de rendre le tout plus agréable à utiliser.



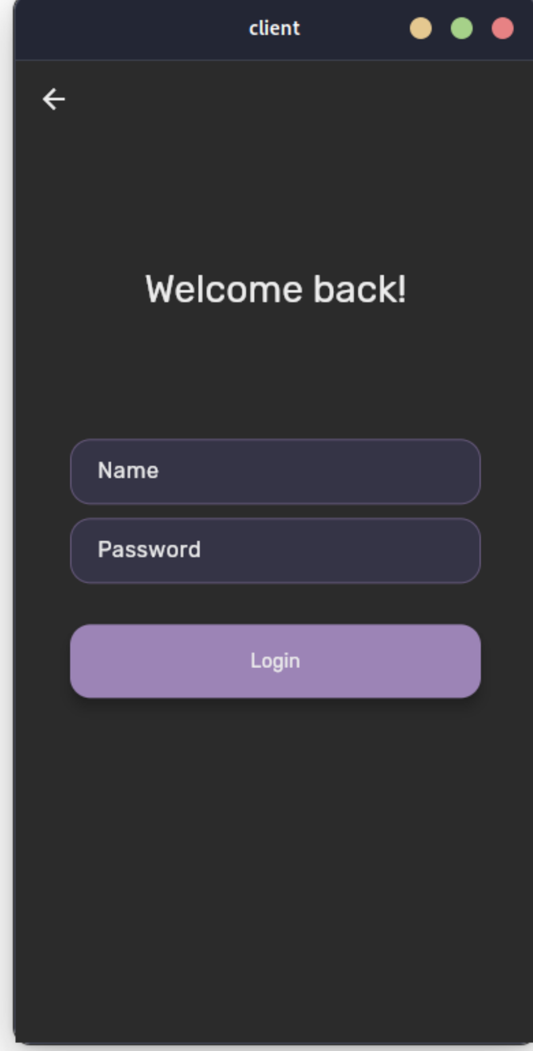
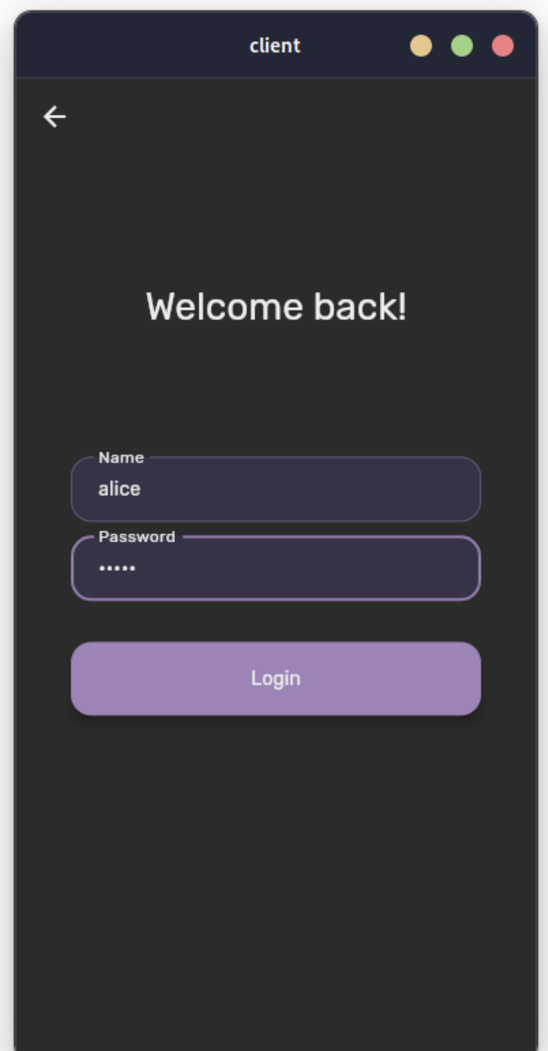


Figure 50 & 51 — Écran d'accueil | Écran de connexion - informations vides



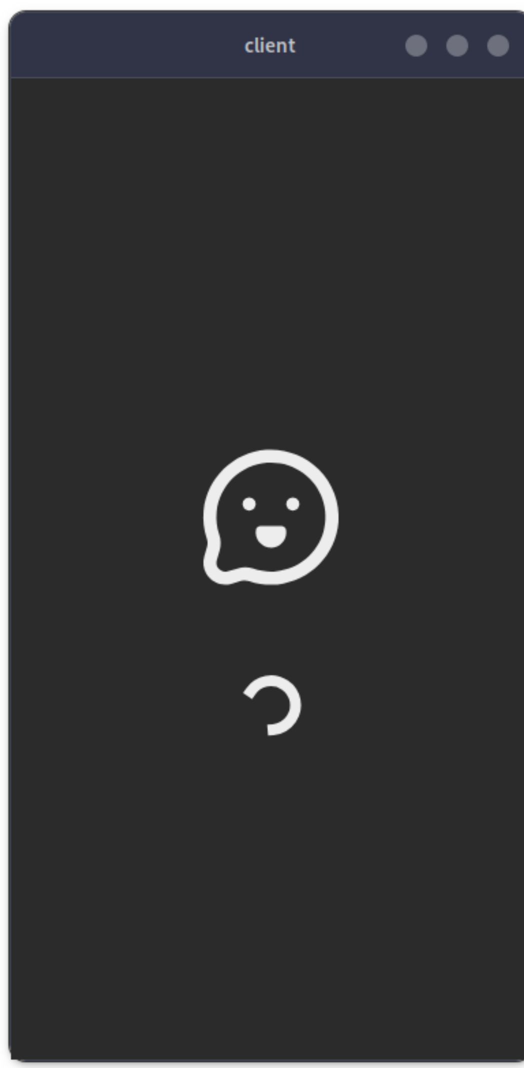


Figure 52 & 53 — Écran de connexion - informations remplies | Écran de chargement

7.23 2024-05-28

Aujourd'hui, j'ai procédé à un léger refactor de mes Provider : j'ai remarqué que l'interface était légèrement trop couplée avec la logique de mon application, et je n'aimais pas la manière dont on injectait AuthProvider dans ChatProvider. J'ai donc décidé de retirer les arguments qui dépendaient de AuthProvider, comme name, qui était le nom de l'utilisateur actuellement connecté, et j'ai décidé d'utiliser une fonctionnalité de Provider, ProxyProvider, qui permet d'utiliser d'autres providers dans un autre.

Cela me permet d'avoir une meilleure séparation des responsabilités, et de pouvoir directement dépendre de la valeur de AuthProvider sans avoir à la passer en argument, tout en conservant la réactivé de l'application. Cela retire encore un peu de complexité de l'interface, que j'aimerais garder la plus "pure" possible.

Finalement, j'ai pu régler un de mes derniers soucis : l'un de mes Provider utilisé dans ProxyProvider, ChatProvider, me donnait une erreur Flutter, car il était apparemment déjà "disposed", c'est-à-dire qu'il avait déjà été détruit. Je me suis donc rendu compte que je recrééais un Provider à chaque fois qu'il se mettait à jour, ce qui déclenchait cette erreur (on ne peut pas réutiliser un Provider déjà déconstruit en Flutter). J'ai donc créé des fonctions internes pour gérer les dépendances, et tout fonctionne comme avant maintenant !

7.24 2024-05-29

Aujourd'hui, j'ai réglé un souci que j'avais depuis l'implémentation du stockage des messages : quand on changeait de compte, on restait encore sur l'ancienne connexion WebSocket. C'était en fait dû à la manière dont j'initialisais Hive : je le faisais dans le initState de mon écran d'accueil, au lieu de le faire dans le main de l'application. J'ai donc déplacé tout ça, et tout fonctionne maintenant comme prévu.

J'ai aussi rajouté un ID au message généré depuis le client, afin d'avoir un moyen de mettre à jour les statuts des messages dans le futur.

J'ai également commencé à travailler sur l'interface des conversations. J'ai rajouté un Listenable dans mon ChatProvider, qui me permet de mettre à jour l'interface en temps réel quand on rajoute des messages dans la base de données locale.

Ensuite, je viens utiliser un widget Flutter, ValueListenableBuilder, qui permet de reconstruire le widget quand la valeur du Listenable change. Dans le cas de cette interface, le nom de l'utilisateur et son dernier message est récupéré. Cela me permet d'avoir une interface en temps réel, qui se met à jour dès qu'un message est reçu. Il faudra sûrement à terme modifier cette logique pour ne pas avoir à parcourir l'intégralité des messages à chaque fois, ce qui est extrêmement inefficace. Il faudrait idéalement trouver un moyen de stocker ces conversations (une liste de noms d'utilisateur ainsi que leur dernier message) quelque part dans la mémoire. Cela fonctionne pour l'instant, mais c'est une solution temporaire.

Il y a également un autre petit souci : quand on change de compte, l'historique complet des messages est perdu. Il faudra que je regarde pourquoi cela se produit-il, mais je suspecte une mauvaise gestion de la base de données (je ne préfixe pas la base de données avec le nom de l'utilisateur, ce qui écrase sûrement les messages de l'autre compte).

Le problème a été réglé ! Il a suffit de rajouter un champ qui contient le nom de l'utilisateur dans mon ChatProvider, un ValueListenable ne pouvant pas être asynchrone (il ne peut donc pas récupérer directement le nom depuis AuthProvider). Je le récupère ensuite depuis ma méthode connect(), qui en plus d'initialiser la connection WebSocket initialise le nom afin d'être utilisé par Hive. Mes messages sont donc stockées dans la base `$name_messages`, ce qui permet de ne pas écraser les messages de l'autre utilisateur, et d'avoir une clé de chiffrement différente pour chaque utilisateur, et permet d'encore plus isoler les messages des différents comptes connectés sur l'appareil.

7.25 2024-05-30

Aujourd'hui, j'ai décidé de m'attaquer à la création d'un écran afin de pouvoir consulter une conversation, et de pouvoir envoyer des messages. Malheureusement, je me suis heurté à un problème : je me suis rendu compte que Hive, la base de données que j'utilise, ne supporte pas la pagination, ce qui veut dire que l'intégralité des messages est chargée en mémoire à chaque fois. Cela peut poser problème si l'utilisateur a beaucoup de messages, et peut ralentir énormément l'application.

J'ai donc cherché une autre solution de base de données, et j'ai vu que sqflite avait un support pour la pagination, et le chiffrement avec sqflite_cipher. Malheureusement, ce dernier fonctionne seulement avec Android et iOS. Je ne peux donc pas l'utiliser car l'application doit fonctionner également sur Linux / Windows / macOS.

Après avoir effectué mes recherches, j'ai découvert Realm, une solution de base de données locale NoSQL publiée par MongoDB. Elle est rapide (les objets sont *lazy loaded*), simple d'utilisation, et supporte le chiffrement. Elle est également multi-plateforme, ce qui est un énorme avantage pour moi. J'ai donc décidé de l'implémenter dans mon application. Je vais effectuer des tests aujourd'hui pour voir si elle correspond à mes besoins.

J'ai finalement réussi à tout passer sur Realm ! J'ai même pu gagner en performance par rapport à Hive, grâce au *lazy loading*. Je me charge maintenant de créer l'écran de conversation.

J'ai créé un composant MessageBubble qui me permet d'afficher les messages de manière agréable. J'ai réussi à récupérer tous les messages comme il faut, et à les afficher dans l'ordre. Il faudra maintenant que je m'occupe d'ajouter une boite de texte pour envoyer des messages, ainsi que d'afficher l'heure de réception des messages.

7.26 2024-06-01

J'ai réussi à implémenter l'écran de conversations sans aucun problèmes aujourd'hui. Il m'a suffit de rajouter un TextField pour envoyer des messages, et de rajouter un bouton pour envoyer le message. La communication a fonctionné directement de manière fluide et bidirectionnelle, ce qui est très rassurant pour la suite.

Il faudrait maintenant réfléchir à un moyen de trouver comment envoyer le premier message, car il n'y a pas de liste de contacts pour l'instant. Je pensais rajouter un bouton en haut à droite de l'écran d'accueil afin de pouvoir taper un nom d'utilisateur et envoyer un message directement. Ce n'est pas la solution la plus pratique, il faudrait idéalement avoir une liste de contacts quelque part ainsi que la possibilité d'ajouter un utilisateur en tant que contact / ami.

7.27 2024-06-02

Aujourd'hui, je vais commencer à implémenter une recherche d'utilisateurs. J'aimerais avoir une barre de recherche, qui permet de rechercher un utilisateur par son nom. Tout d'abord, je vais implémenter une simple route à GET /users qui permet de récupérer tous les utilisateurs, puis je vais rajouter un paramètre de recherche qui permet de filtrer les utilisateurs par leur nom.

J'ai réussi à implémenter cette recherche. J'ai rajouté ma route `GET /users`, qui permet de récupérer le nom des utilisateurs que l'on recherche (sauf l'utilisateur connecté, pour éviter de pouvoir s'envoyer des messages à soi-même). J'ai également rajouté un paramètre `search` qui permet de filtrer les utilisateurs par leur nom.

Ensuite, j'ai créé une page qui permet de rechercher un utilisateur, et à chaque fois que l'on tape quelque chose dans la barre de recherche, une requête est envoyée au serveur pour récupérer les utilisateurs correspondants (j'ai mis un timeout de 500ms avec une classe Debouncer, qui prend un temps en millisecondes et qui empêche de pouvoir être lancée avant la fin de ce timer). Quand on clique sur l'utilisateur auquel on souhaite envoyer un message, une session est créée avec lui grâce au SignalProvider, et l'écran de conversation est affiché.

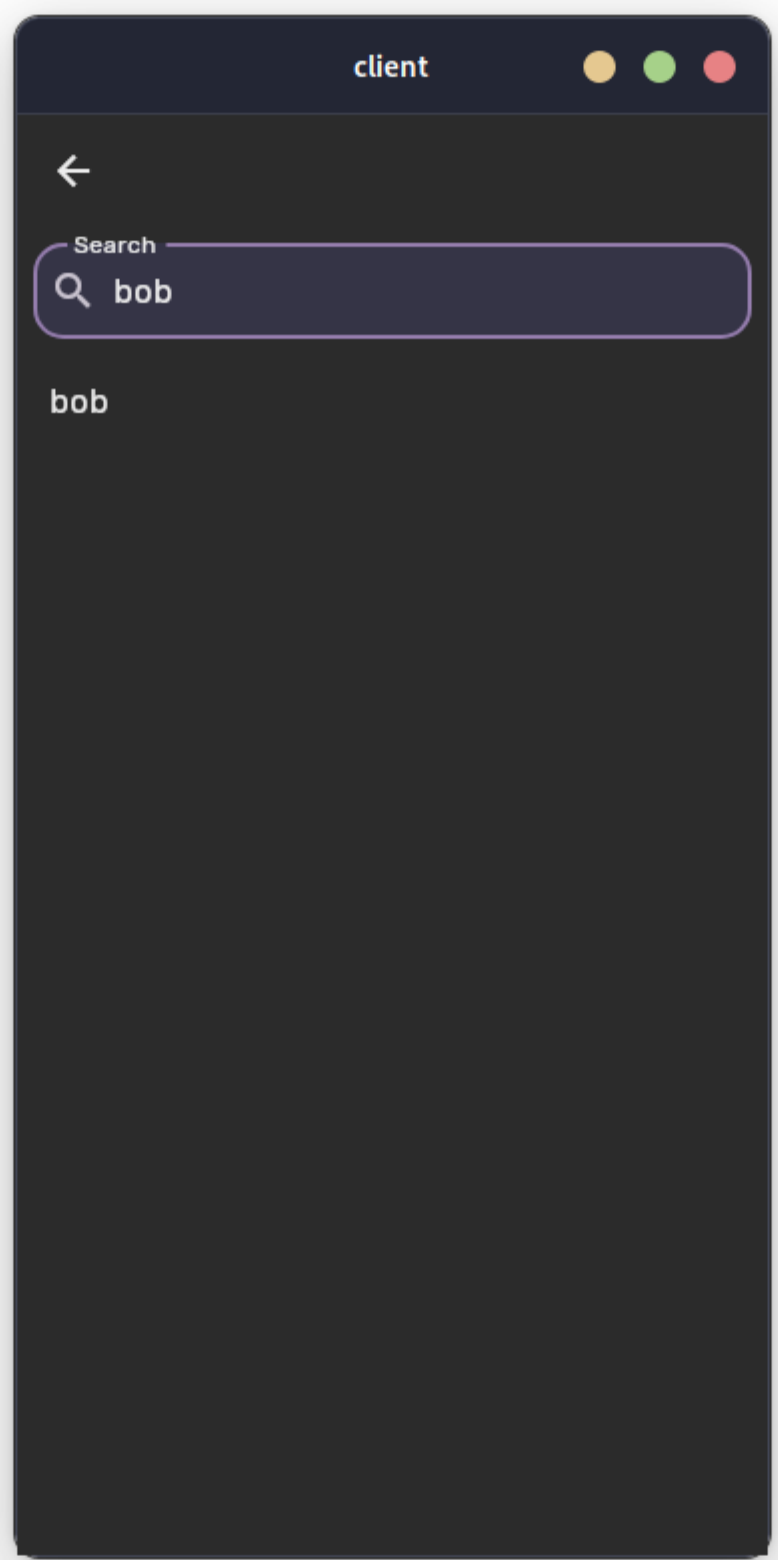

client
Search
bob
bob

Figure 54 — Écran de recherche

Maintenant que la communication initiale peut être établie au niveau de l'interface, j'aimerais m'occuper de la récupération des messages temporaires au démarrage de l'application. Je pense rajouter une méthode dans mon ChatProvider, qui permet de récupérer ces messages, créer une session au besoin, et les stocker dans la base de données locale.

J'ai aussi réglé un bug assez gênant qui faisait qu'un utilisateur s'envoyait un message à lui-même au premier message. Ce bug était causé par le fait que Flutter ne disposait pas du ChatProvider après le logout, car on ne le recréeait pas. J'y ai passé énormément de temps, ça m'a pris toute la journée, mais j'ai finalement réussi à régler le problème en retournant un nouveau ChatProvider quand le statut isLoggedIn devient false, dans mon update. Tout fonctionne maintenant !

7.28 2024-06-03

Aujourd'hui, j'ai principalement passé du temps sur quelques fioritures au niveau de la page de conversation. J'ai rajouté des timestamps pour chaque message, si ils sont séparés de plus de 5 minutes, ou la date complète si ils sont séparés de plus de 24 heures. Je me suis également occupé de créer un poster pour l'application, qui sera utilisé pour la présentation de mon projet. Voici à quoi il ressemble pour l'instant :

Figure 55 — Poster de Missive

7.29 2024-06-05

Aujourd'hui, je me suis concentré sur la documentation de mon code. Dart possède une fonctionnalité très intéressante, les docstrings, qui permettent de documenter le code de manière très simple et efficace. J'ai donc passé la journée à documenter tout mon code, afin de pouvoir le rendre plus lisible et compréhensible. Il est également possible de générer une documentation en ligne à partir de ces docstrings, ce qui permet de rendre le tout encore plus accessible, et de le publier sur un site internet. Je vais m'en servir pour générer la documentation API de mon client.

7.30 2024-06-06

Aujourd'hui, vu que le code est bien avancé, je suis en train de me concentrer sur la partie DevOps du projet. Je suis en train d'implémenter des tests unitaires pour le client, en priorisant pour l'instant le test des fonctionnalités de chiffrement et de déchiffrement.

En parallèle, j'ai commencé à regarder comment je pourrais implémenter les notifications Push sur l'application. J'ai trouvé un service, nommé OneSignal, qui permet de gérer les notifications Push sur Android et iOS de manière très simple. Il suffit de rajouter un SDK dans l'application, et de configurer le serveur pour envoyer des notifications. J'ai commencé à implémenter tout ça, et j'ai réussi à envoyer une notification depuis le serveur, qui est bien reçue par l'application. Il faudra maintenant que je rajoute un champ `oneSignalId` dans la base de données, qui permettra de stocker l'ID de l'utilisateur sur OneSignal, et de lui envoyer des notifications.

Il faudra également mettre à jour la base de données après la connexion / déconnexion, un ID OneSignal appartenant à un seul périphérique en même temps (sachant qu'on ne peut avoir qu'un compte connecté en même temps). Il faudra donc mettre à jour ma route `PUT /users/{id}` pour mettre à jour ce champ.

Pour l'instant, je me suis remis sur les tests unitaires car c'est la priorité pour l'instant. Je regarderais plus tard cette histoire de notifications, car il faudra également implémenter les reçus de messages.

J'ai un petit souci avec mon gitlab CI : la documentation ne se met pas à jour automatiquement. Il faudra que je regarde pourquoi, car c'est un peu embêtant pour le moment. Je pense que c'est lié au fait qu'elle soit un submodule, donc je ne peux pas vérifier le contenu de ce dernier comme je le ferais pour un dossier.

J'ai trouvé comment faire ! Il me suffisait simplement de tester les changements sur le nom du submodule, au lieu de son contenu. Voici un exemple fonctionnel :

```
rules:
  - changes:
    - documentation
```

Figure 56 — Exemple de règle pour tester les changements sur le submodule

7.31 2024-06-07

Aujourd'hui, j'ai encore avancé sur la documentation. J'ai mis à jour la page fonctionnement, et passé beaucoup de temps sur le poster afin de s'assurer qu'il serait prêt pour la présentation. J'ai également commencé à travailler sur la mise à jour des statuts de message, qui permettra de savoir si un message a été envoyé, reçu, ou lu.

J'ai rajouté un modèle MessageStatus dans mon modèle Prisma, lié au modèle PendingMessage. Il faudra maintenant implémenter au niveau des routes, et j'ai également rajouté une route `GET /messages/{id}/status` qui permet de récupérer le statut de tous les messages. Quand on récupère les messages au lancement de l'application, il faudra également rajouter la mise à jour des statuts de ces messages afin que l'envoyeur soit au courant de l'état de ses messages. Il faudra aussi s'occuper de le faire via le WebSocket, si l'autre utilisateur est connecté.

7.32 2024-06-08

Aujourd'hui, j'ai pu implémenter la mise à jour des statuts de messages. J'ai rajouté la route qui récupère les statuts des messages que l'utilisateur connecté a envoyé, et j'ai modifié ma route WebSocket afin de mettre à jour automatiquement les statuts à l'envoi (envoi direct si en ligne, sinon stockage temporaire).

Au niveau du client, j'ai rajouté un champ `status` dans mon modèle `PlaintextMessage`, qui me permet de stocker ce dernier et de le mettre à jour en temps réel dans l'interface. Vu que Realm rend ça très simple et réactif, il m'a juste suffit de rajouter les bonnes icônes dans mon widget `MessageBubble`. Il ne me restera plus qu'à implémenter le statut "lu", qui sera un peu plus complexe à implémenter, sachant que ma route WebSocket devra gérer en plus d'un envoi de message classique, l'envoi d'une mise à jour de statut.

Je suis également passé sur un framework de logging, qui me permet de gérer les logs de manière plus propre et plus efficace. J'ai choisi `logging`, qui est une librairie très simple d'utilisation, développée par Google même.

7.33 2024-06-09

J'ai commencé à implémenter le statut lu au niveau du client. J'utilise un package Flutter, `visibility_detector`, qui permet de détecter si un widget est visible à l'écran, et de mettre à jour le statut en conséquence.

Au niveau du serveur, je réutilise une logique similaire à celle des messages en attente : je vérifie tout d'abord si le message qui a été envoyé est une mise à jour de statut. Si c'est le cas et que le receveur est connecté, cela lui envoie la mise à jour de statut via le WebSocket. Sinon, on stocke en base de données. Cela fonctionne quasiment, mais au niveau du serveur, vu que l'ID du message n'est pas forcément lié à un PendingMessage, la logique crashe car il y a une contrainte de clé étrangère sur MessageStatus avec PendingMessage. Il faudra régler tout ça demain.

7.34 2024-06-13

J'ai enfin réussi à implémenter le statut lu ! J'ai réglé ce problème de clé étrangère, et je me suis aperçu que j'avais des problèmes au niveau du flux de mes données et de la logique de mon API, ce qui faisait que le mauvais ID était stocké dans la table MessageStatus (il fallait stocker l'ID de l'envoyeur du message, car c'est lui qui doit savoir si son message a été envoyé correctement). Tout a l'air de fonctionner pour l'instant. Il faudra maintenant rendre l'application un peu plus solide, notamment au niveau de la perte de connexion, car l'application crashe souvent si on perd la connexion au serveur WebSocket, ou les messages ne s'envoient jamais. Il faut implémenter :

1. Un système de reconnexion automatique
2. Un système de stockage temporaire des messages si l'utilisateur est déconnecté, et de les envoyer dès qu'il se reconnecte

Je vais chercher le dépôt des librairies Flutter, pour voir si il existe déjà une dépendance qui pourrait me permettre d'implémenter une vérification du statut de connexion en temps réel. Il me semble déjà avoir vu quelque chose de la sorte, donc je vais creuser un peu.

7.35 2024-06-14

Aujourd'hui, j'ai essayé d'ajouter des tests d'intégration. J'ai passé la matinée à essayer de le faire fonctionner avec mon application, mais je n'ai malheureusement pas réussi : j'ai des problèmes avec mon `MockAuthProvider`, qui refuse de mettre à jour le statut d'authentification de l'utilisateur ce qui rend la partie authentification impossible à tester pour l'instant.

J'ai ajouté une fonctionnalité qui permet de se reconnecter automatiquement au WebSocket quand on perd la connexion. Il faudra maintenant le mettre en pause si l'utilisateur perd la connexion à internet, et reprendre les tentatives quand une connexion est retrouvée. Il faudra également rajouter un système de stockage temporaire des messages, qui permet de les envoyer dès que l'utilisateur se reconnecte. Il est malheureusement impossible de réutiliser les mêmes messages que dans Realm, car ce sont les messages en clairs, et non les messages chiffrés. Il faudra donc rajouter une table dans la base de données qui stocke les messages chiffrés si besoin, et qui les envoie dès que l'utilisateur se reconnecte.

7.36 2024-06-15

J'ai réussi à tout régler ce matin ! La notification de lecture des messages fonctionne maintenant. J'ai du régler la logique au niveau du serveur et du client, en envoyant le nom du receveur de la notification sur le champ JSON `receiver`, au lieu d'avoir une logique complexe en utilisant le sender, car ce terme est relatif. Je suis parti du principe que la personne qui reçoit la notification de statut est le `receiver`, ce qui a amplement simplifié la logique.

En ce qui concerne la reconnexion automatique, j'ai réussi à le faire fonctionner : ma logique n'était pas complète, car je ne changeais pas la variable `_isConnecting` après chaque tentative de reconnexion. J'ai également mis à jour les niveaux de logs, afin de pouvoir mieux comprendre ce qui se passe en cas de problème, et que le tout soit encore plus cohérent.

Cette après-midi, j'ai mis en place un système au cas où l'on perde la connexion au serveur WebSocket : j'ai rajouté un champ `_connected` dans mon ChatProvider, qui permet de savoir si l'on est connecté ou non. Si l'on perd la connexion, on stocke les messages dans la base de données Realm de l'utilisateur (une base séparée, `PendingMessages`), et on les envoie dès que l'on se reconnecte. J'ai également rajouté un système de reconnexion automatique, qui permet de se reconnecter dès que l'on perd la connexion.

J'ai également mis en place une récupération des messages ainsi que des statuts de lectures si il y a eu une reconnexion, pour éviter de devoir recharger l'application dans ce cas là. Tout fonctionne comme prévu, et je suis très content du résultat !

La dernière chose qu'il faudrait mettre en place serait une persistence de la connexion à l'application, même hors-ligne : en effet, vu qu'un bon nombre d'opérations est possible à effectuer hors-ligne, comme consulter ses messages car ils sont stockés en local, ou envoyer un message qui sera stocké en local, il serait intéressant de pouvoir utiliser l'application même sans connexion à internet. Pour l'instant, l'application déconnecte l'utilisateur si il n'arrive pas à récupérer un jeton d'accès ou si l'une des requêtes d'authentification ne passe pas. Il faudrait donc rajouter un système permettant la reconnexion, un peu comme pour le `ChatProvider`. Il faudrait regarder tout ça demain.

7.37 2024-06-16

Aujourd'hui, j'ai pris la journée afin de me concentrer sur la documentation. Je voulais m'assurer que la partie Fonctionnement était bien expliquée, et j'ai également rajouté des réflexions qui étaient uniquement présentes dans mon journal de bord, afin d'avoir une documentation complète et précise du déroulé du projet. J'ai également rajouté des captures d'écran de l'application, afin de montrer le fonctionnement de l'application.

7.38 2024-06-17

Aujourd'hui, j'ai envie de faire fonctionner les notifications, car c'était une des parties essentielles de mon application. Je me suis rendu compte que les notifications ne fonctionnaient pas de la même manière sur téléphone et PC :

- iOS/Android : un système de notifications push est disponible, ce qui permet d'envoyer une notification depuis le serveur dès que le message est reçu
- PC : il n'y a pas de système de notifications push, il faudra donc utiliser un système de notifications locales. J'ai trouvé un package Flutter, `flutter_local_notifications`, qui permet de gérer les notifications locales sur toutes les plateformes. Il faudra donc l'implémenter pour les notifications sur PC, et garder le système de notifications push pour les téléphones.

La partie PC sera un peu plus complexe, car il faudra implémenter un système de *poll* des messages : je pense réutiliser le WebSocket afin d'envoyer une notification à l'utilisateur dès que le message est reçu. La partie notifications de statut étant déjà implémentée, ajouter les notifications PC sera sûrement beaucoup plus simple. Je vais commencer par la partie téléphone.

OneSignal sera utilisé afin de s'occuper des notifications téléphone : c'est un service qui propose un système de notifications unifié pour Android et iOS. Il a été retenu pour sa simplicité d'utilisation, dépendant d'un *Player ID*, qui est généré par le SDK OneSignal.

Après avoir essayé OneSignal, je me suis rendu compte que le système de notifications push n'était pas aussi simple que je le pensais : je n'ai pas réussi à envoyer de notifications depuis le serveur avec. J'ai donc décidé de passer sur Firebase Cloud Messaging, un service de Firebase, qui permet de faire la même chose de manière beaucoup plus simple. Après avoir implémenté, tout fonctionne !

Le seul souci qu'il me reste, et que je ne pense pas pouvoir résoudre durant le reste du travail de diplôme, car beaucoup trop de temps serait nécessaire, est le fait que iOS impose des restrictions sur les tâches de fond, et ne permettrait donc pas de déchiffrer les messages en arrière-plan. Il faudrait donc que l'utilisateur ouvre l'application pour pouvoir déchiffrer les messages, ce qui est un peu embêtant. Je suis pour l'instant obligé d'envoyer une notification générique, sans le contenu du message.

En cherchant un peu comment Signal fait pour déchiffrer ses messages en fond, je me suis aperçu qu'ils utilisent un procédé assez ingénieux, les notifications silencieuses d'iOS : cela permet d'envoyer une notification sans que l'utilisateur ne la voit, et de déchiffrer le message en arrière-plan, car cela réveille l'application pendant un bref moment. Cela me semble être une solution viable, mais je ne pense pas avoir le temps de l'implémenter et préfère me concentrer sur les fonctionnalités essentielles.

La dernière étape des notifications est de rajouter un système de notifications locales pour les PC. Comme dit auparavant, il va falloir que je recherche un peu plus, car le système est un peu différent : le système serait entièrement géré côté client, car les

notifications push n'existent pas sur ordinateur. Je pense utiliser le WebSocket lui-même afin de *poll* les nouveaux messages, et d'envoyer une notification si l'application n'est pas au premier plan.

7.39 2024-06-21

Aujourd'hui, commençant à arriver au bout du travail de diplôme, je me suis dit que j'allais commencer à préparer le déploiement. Sachant que je compte utiliser Docker pour le déploiement, j'aimerais prendre avantage de Docker Swarm, qui permet de gérer des clusters de conteneurs Docker. Cela permettrait au serveur d'avoir des fonctionnalités vraiment pratiques, comme un *load balancer* intégré, et une meilleure gestion des secrets avec `docker secret`.

J'ai donc commencé à regarder comment je pourrais mettre cela en place. Je me suis rendu compte que mes Dockerfile ainsi que `docker-compose.yml` actuels sont assez mal écrits, et non adaptés pour un déploiement en production. J'ai donc commencé à les réécrire, en me heurtant à beaucoup d'erreurs de variables d'environnement. J'aimerais le système suivant :

- Développement : `docker-compose up` qui lance le serveur Prisma, le serveur WebSocket, et le client Flutter à l'aide de variables d'environnement afin d'avoir un développement plus rapide et plus simple
- Production : `docker stack deploy` qui déploie le tout sur un cluster Docker Swarm, avec des secrets pour les clés privées/publiques

Afin de pouvoir utiliser Docker Swarm, il faut une image du Dockerfile (on ne peut pas la build directement sur le serveur). Je me suis donc rendu compte que mon image n'était pas vraiment indépendante, notamment qu'elle dépendait de la configuration de l'environnement. En plus, les healthcheck que j'utilisais qui me permettaient de ne pas lancer l'application et donc de ne pas lancer les migrations avant que la base de données soit opérationnelle sont apparemment *deprecated*. J'ai donc dû trouver une autre solution, et j'ai décidé d'utiliser `wait-for-it.sh`, un script qui permet d'attendre qu'un service soit disponible avant de lancer une commande. J'ai aussi pas mal *refactor* mon Dockerfile, et j'ai réussi à le rendre beaucoup plus propre et plus lisible. La partie dev fonctionne maintenant comme prévu, il faudra maintenant que je m'attaque à la partie production.

J'ai réussi à faire fonctionner tout ça ! J'ai réussi à déployer mon application sur un cluster Docker Swarm, sur mon VPS, avec des secrets pour les clés privées/publiques, et tout fonctionne comme prévu. J'ai également mis à jour la partie `docker-compose.dev.yml` afin d'avoir un *hot reload* qui fonctionne, ce qui est crucial en développement. Il aura fallu monter le répertoire du serveur dans le conteneur, afin que les fichiers qui soient réellement dans le conteneur soit les mêmes que ceux sur la machine hôte.

7.40 2024-06-22

Aujourd'hui, j'ai passé la plus grande partie de ma journée à installer une machine macOS afin de pouvoir build mon application dessus, et tester si elle fonctionne correctement. J'ai réussi à le faire fonctionner, après avoir tout configuré comme il faut au niveau macOS, mais j'ai eu quelques soucis avec les dépendances de Flutter : en effet, `flutter_secure_storage` se comporte différement sur iOS, et envoie une exception si la valeur n'est pas trouvée, car l'implémentation est différente. J'ai dû légèrement modifier le code pour que ça fonctionne correctement.

J'ai également eu des soucis avec le fait que les Mac soient des machines Arm64, avec les nouvelles puces M, et que les conteneurs Docker ne soient pas encore complètement supportés sur ces architectures (certaines dépendances doivent être compilées à la main). J'ai donc perdu du temps là dessus, il faudra repasser du temps dessus demain pour s'assurer que le déploiement iOS fonctionne comme le déploiement Android.

7.41 2024-06-23

Aujourd'hui, j'ai avancé sur la documentation. J'ai essayé de faire fonctionner une librairie qui me permettrait de générer une documentation OpenAPI à partir de mon code Fastify, mais je n'ai pas réussi à la faire fonctionner correctement. J'ai donc décidé de rester sur le fichier `openapi.yml` que j'ai écrit à la main, et de le mettre à jour manuellement.

7.42 2024-06-24

Aujourd'hui, je veux impérativement faire fonctionner les notifications sur iOS. Je me suis donc penché sur le sujet, et j'ai passé la matinée à configurer le service d'Apple. J'ai réussi à obtenir un certificat de développement, et à le configurer dans mon serveur de notifications Firebase. J'ai également réussi à envoyer une notification depuis le serveur, qui est bien reçue par l'application.

Il reste encore quelques soucis à régler, comme le fait que les notifications Push sont seulement envoyées quand l'application est fermée (ce qui est normal, mais il faudrait aussi que ça fonctionne quand l'application est en tâche de fond). J'essaie donc de mettre à jour mon serveur, afin d'envoyer la notification dans tous les cas, et de la gérer côté client si l'application est en tâche de fond.

J'ai réussi à régler le problème ! Il suffisait de les envoyer dans tous les cas. Le souci était que je n'envoyais pas de notification tant qu'il y avait une connection WebSocket, ce qui est le cas quand l'utilisateur a son application en tâche de fond.

J'ai par contre remarqué des soucis au niveau de la version Android, notamment certaines personnes qui ne peuvent pas envoyer de messages. Il faudrait que je m'y penche un peu plus lundi.

7.43 2024-06-27

Aujourd'hui, je m'occupe de peaufiner au maximum l'application afin qu'elle soit le plus agréable à utiliser. J'avais eu quelques retours sur certaines fonctionnalités et améliorations potentielles, que j'ai donc implémenté :

- J'ai trié les conversations pour que la conversation avec le message le plus récent soit en haut
- J'ai rajouté un timestamp sur l'écran des conversations afin de pouvoir s'y retrouver plus facilement
- Possibilité de copier le nom d'utilisateur en appuyant longtemps dessus
- Ajout d'une version de l'application en bas du menu Drawer
- Ajout d'un badge qui indique le nombre de messages non lus dans la liste des conversations, si il y en a

Le souci lié à la version Android est d'ailleurs lié à une migration du schéma Realm (ma base de données locale), qui devait être faite sur les anciennes versions de la base de données, car certaines valeurs sont maintenant requises, comme `sentAt`. Cela faisait crasher l'application si elle avait une ancienne version de la base de données. Ce ne sera pas un problème en production, et je ferais attention aux migrations locales si jamais je venais à changer le schéma.

7.44 2024-06-28

Aujourd'hui, je me suis décidé à déplacer le déploiement depuis DigitalOcean vers Jelastic Cloud, afin d'avoir les données stockées en Suisse. J'ai donc commencé à regarder comment je pourrais déployer mon application sur Jelastic, et j'ai commencé à le faire. J'ai réussi à déployer le serveur Prisma, le serveur WebSocket, et le client Flutter, et tout fonctionne comme prévu. J'ai eu beaucoup de soucis ce matin, notamment avec la gestion des secrets et des configurations avec Docker Swarm. J'ai décidé d'utiliser les configs de Docker Swarm, qui permettent de stocker des configurations dans le cluster, et de les utiliser dans les conteneurs. Cela m'a permis de stocker la configuration Caddy afin de la séparer complètement du code source de l'application.

Je vais également m'occuper de mettre en place une réplication, qui permettra d'effectuer un *load balancing*. J'aimerais aussi mettre en place une réplication des bases de données, car mon instance Jelastic Cloud possède deux machines dans la swarm.

La réplication a été mise en place au niveau de l'application. En revanche, c'est un peu plus complexe pour la base de données, et cela nécéssite des scripts de backup et de restauration, qui ne sont pas encore implémentés. Il faudra que je m'y penche un peu plus tard, si j'ai le temps. En tout cas, pour l'instant, l'application tourne sur trois répliques sur les serveurs d'Infomaniak !

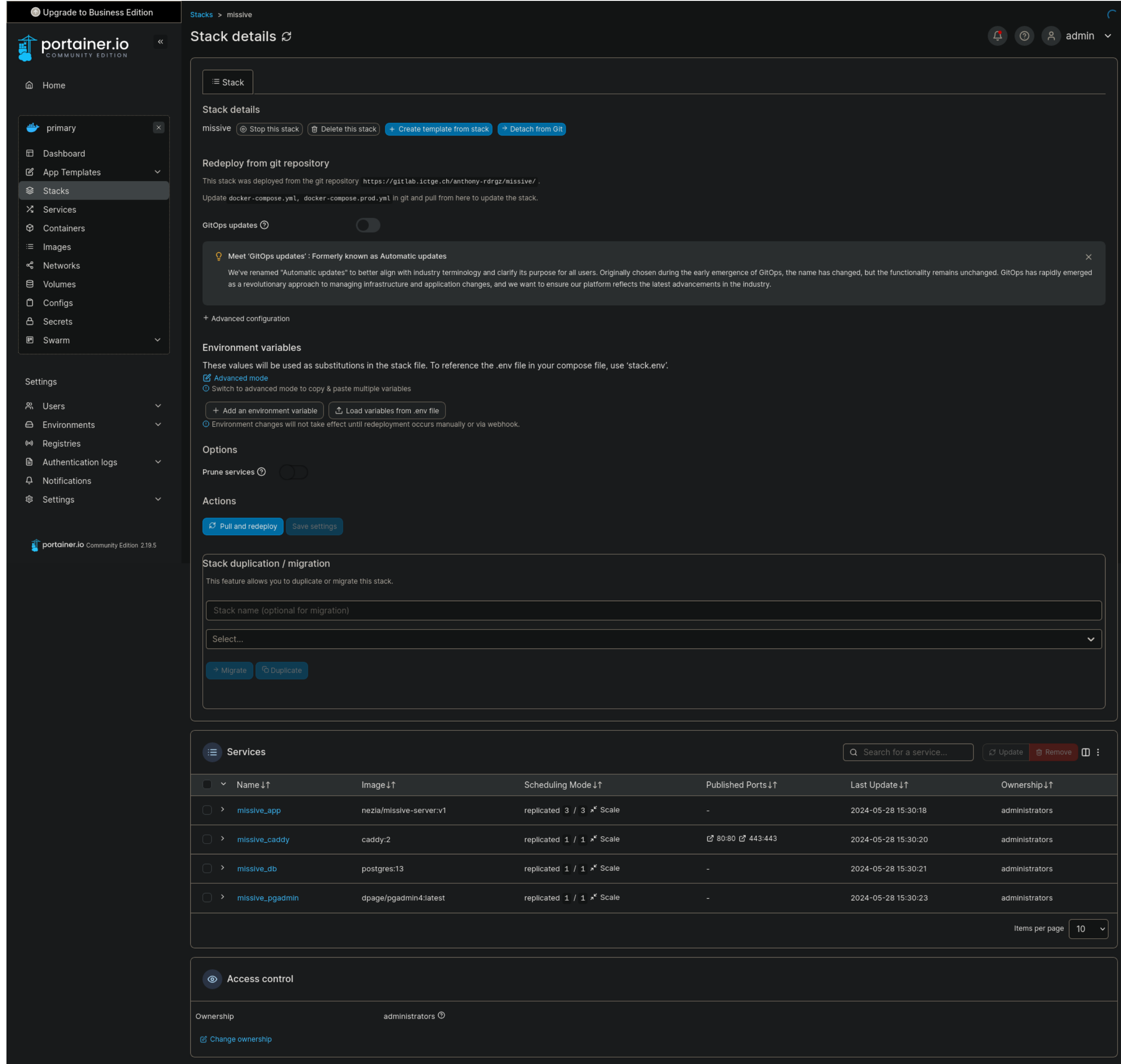


Figure 57 — Missive sur Jelastic Cloud

7.45 2024-06-30

Aujourd'hui, c'est une journée principalement documentation. Le projet étant quasiment terminé, il est temps de mettre à jour la documentation, et de la rendre la plus complète possible. J'ai commencé par travailler sur le plan de tests, en rajoutant les tests du client ainsi qu'un historique afin de pouvoir suivre la progression du projet de manière plus simple.

7.46 2024-06-31

Aujourd'hui, il est temps de s'attaquer aux tests unitaires. Ces derniers étant je trouve assez peu pertinents au niveau du serveur, j'ai décidé de les refaire en appliquant des principes solides de modularité et de testabilité.

Pour commencer, j'ai réécrit des parties du code afin de le rendre plus modulaire : certaines parties utilisaient des fonctions en interne, comme mon `authenticationHook`, par lequel j'ai décidé de commencer. J'ai fait en sorte qu'on puisse y injecter en tant que dépendance la fonction `verifyAndDecodeScopedJWT`, afin de pouvoir lui faire retourner une valeur attendue. J'ai aussi du modifier mon plugin Fastify qui me permettait d'avoir Prisma à `fastify.prisma`, afin qu'il puisse prendre une instance optionnelle personnalisée afin de pouvoir la mock complètement.

Malheureusement, je me suis heurté à de nombreux problèmes, qui sont principalement liés au mock : en effet, il s'est avéré que mocker mon client Prisma s'est avéré beaucoup plus complexe que prévu, et les différents essais entre différentes librairies de mock n'ont pas vraiment aidé. Je vais essayer de m'y pencher lundi, mais je pense que le fait que j'utilise Prisma comme plugin Fastify pose beaucoup de problèmes. Il faudrait que j'essaie d'utiliser une instance singleton afin de simplifier le processus.

Finalement, j'ai réussi ! Il me suffisait juste de passer sur une instance singleton de Prisma, et tous mes soucis ont été résolus. Il a fallu jsute apprendre à utiliser vitest, qui est ma bibliothèque de choix pour ce projet. Elle permet de développer des tests unitaires efficaces ainsi que de réaliser des mocks de manière efficace.

7.47 2024-07-03

Aujourd'hui, je me suis réintéressé aux tests unitaires. La fin du diplôme approchant, je me suis rendu compte que je n'avais pas de tests unitaires pour mes hooks d'authentification et d'autorisation, ainsi que pour mon serveur WebSocket.

La première partie était relativement simple, car faisant partie de l'API de Missive, et vu que j'avais déjà testé unitairement des routes, j'ai créé une route `GET /test-authentication` et `GET /test-authorization` depuis le test unitaire, qui me permet de tester ces deux parties séparément. Après avoir configuré les mocks comme il fallait, les tests sont maintenant opérationnels.

La partie WebSocket m'a malheureusement pris plus de temps : `@fastify/websocket` propose une solution pour injecter une requête sans avoir le serveur qui tourne (comme fastify.inject pour la partie API de Fastify), mais elle fonctionne bien différement, en raison du fait que ce soit un serveur WebSocket avec des évènements. J'ai réussi à faire fonctionner la plupart des tests unitaires pour ce dernier, en me servant de l'expérience que j'ai pu développer avec les mocks de l'API (Prisma et autres), mais j'ai un souci avec une connexion WebSocket qui refuse de fermer, et qui m'empêche de tester que la fonction `prisma.pendingMessage.create`, est correctement appelée. Il faudra que je me penche dessus un peu plus tard. Pour l'instant, je pense que je commence à être pas mal au niveau de ces tests, cependant, il faudra maintenant que je documente tout ça correctement.