Nexium

Rapport de projet

Jean HERAIL
jean.herail@epita.fr

Milo DELBOS
milo.delbos@epita.fr

Antonin BESSIÈRES
antonin.bessieres@epita.fr

William VALENDUC
william.valenduc@epita.fr

Sommaire

1 Introduction

Dans le cadre du projet de quatrième semestre à l'EPITA, notre groupe a pris la décision de créer un outil que nous voulions pertinent, utile et ayant un sens pour les gens. C'est de cette conception particulière que naquit l'idée de **Nexium**.

Il s'agit d'un protocole transactionnel pair-à-pair, basé sur la cryptographie, et destiné aux membres d'EPITA (étudiants et personnels éducatifs). Basé sur la technologie de la blockchain[1], et construit selon le *Nakamoto Consensus*[2] établi avec le Bitcoin en 2008, Nexium s'assimile donc à une réelle crypto-monnaie décentralisée, *trustless*[3], indépendante et résiliente.

Le produit final de Nexium se décline sous deux formes binaires : un **client**, et un **serveur**. Nous expliquerons en détail dans ce rapport le rôle et l'agencement de ces deux pôles fondamentaux.

Conformément aux contraintes techniques requises par le sujet, tout le projet est développé en **Rust**. Ce langage de programmation, reconnu pour ses performances et sa *memory-safety*, est particulièrement propice et adapté dans le cadre d'un projet impliquant des notions de sécurité et de cryptographie comme Nexium. La popularité de Rust rend tout aussi appréciable la découverte de son fonctionnement : la richesse de sa documentation et l'engagement de sa communauté permettent de résoudre aisément d'éventuels problèmes que l'on rencontre en développant.

En outre, Nexium est un projet aussi ambitieux que captivant : cryptographie, architecture et communication réseau, développement web, et travail de groupe sont autant de compétences que chacun d'entre nous a développées dans la réalisation de cet objectif commun.

Nous abordons dans ce rapport l'entièreté des réalisations que nous avons implémentées tout au long du projet. Les notions techniques les plus importantes du projet seront explorées, et nous détaillerons également la situation globale du groupe tout au long du projet, son organisation et son ressenti final.

[1] La première et la plus capitalisée des cryptomonnaies, créée en 2009.

[2] Le consensus de Nakamoto fait référence à la manière spécifique dont le réseau décentralisé de Bitcoin parvient à un accord sur l'état de sa blockchain.

[3] Le terme "trustless" ("sans confiance" en français) désigne un système ou des transactions où les parties peuvent interagir et échanger de la valeur sans avoir besoin de faire confiance à un tiers. Dans le contexte des cryptomonnaies, la trustless est souvent réalisée grâce à des protocoles cryptographiques basés sur la notion de "preuve de travail".

2 Présentation des membres

2.1 Jean HERAIL

Mon rôle pour ce début de projet est de développer, en tandem avec **William**, l'architecture de la **blockchain** et du **réseau**. Nous écrivons aussi, au fur et à mesure de notre avancement, la **documentation théorique** de Nexium au format **Markdown**, intégrée dans le **site web officiel** de Nexium. Cela implique de comprendre les mécanismes de la **blockchain**, de la **cryptographie**, et de la **communication réseau**.

J'assume également le rôle de **chef de projet**, en charge de l'**organisation du groupe** et de la **répartition des tâches**. Je m'occupe également de l'écriture en LaTeX des différents documents rendus.

Étant moi-même passionné de **cryptographie**, c'est tout naturellement que j'ai initialement suggéré à mon groupe l'idée d'une **cryptomonnaie**. Accompagnée par un **synopsis** détaillant les grandes lignes théoriques du projet, l'idée a plu au groupe et a évolué au fil de plusieurs **réunions** et **débats**.

Je suis également très intéressé par le langage **Rust**, que j'ai découvert par curiosité en amont de la **SPÉ**, et que j'ai eu la chance de découvrir plus en profondeur lors des **cours du quatrième semestre**. Les **TPs hebdomadaires** me permettent régulièrement de me familiariser avec les concepts de Rust, et d'acquérir des **automatismes** dans l'écriture de code.

Ce projet m'enthousiasme donc tant dans sa **dimension technique** que dans sa **dimension humaine**. Je suis ravi de pouvoir découvrir et travailler sur la **blockchain** dans un cadre aussi concret. Je dois aussi reconnaître que c'est une **chance** pour moi d'avoir pu être dans un groupe dont les membres sont aussi **motivés**, et avec qui je m'entendais déjà très bien avant le début du projet. Cette bonne entente me semble favoriser une **communication fluide** et **efficace**.

2.2 Milo DELBOS

Je suis **Milo Delbos**, étudiant à **EPITA** et anciennement bachelier spécialité **mathématiques et physique**. Mon attrait pour les matières scientifiques est ce qui m'a amené à prendre ce chemin d'études et qui m'a mené jusqu'ici. Je me débrouille en **mathématiques** ainsi qu'en **théorie sur les raisonnements algorithmiques**.

Dans ce projet, le développement de fonctions **SHA-256** et **RSA** dans lesquelles j'ai participé m'ont permis d'agrandir mes compétences **algorithmiques** et en terme de **programmation**. Il me semblait également difficile de m'imaginer un jour toucher à de l'**HTML** ou du **CSS**, mais pour faire

l'application client, nous en avons eu besoin et cela s'est révélé être un meilleur moment que ce à quoi je m'attendais.

Il est aussi évident que mon rôle était d'apporter un avis objectif et utile aux travaux de mes camarades s'ils en demandaient afin d'améliorer le rendu global du projet. **Nexium** représentait une pente à gravir, c'était le premier gros projet de ce type dans lequel je participe. Ainsi, il m'a permis de développer des **compétences utiles** et meilleures dans tous les domaines touchés (mathématiques, algorithmique, design, esprit créatif, travail de groupe et compétences sociales).

Nous avons tous choisi ce projet car c'est une idée **innovante** et très intéressante d'utiliser une **blockchain** et une **cryptomonnaie** dans un cadre privé qui puisse répondre à beaucoup d'attentes et de requêtes et qui pourrait même être utile à l'école.

Non seulement il y a mon attrait pour ce domaine de la **cryptomonnaie**, mais aussi toutes les possibilités derrière qui ne dépendent que de notre **imagination**, ainsi ce projet nous permettra de transmettre ce que l'on désire tout en améliorant nos compétences.

2.3 Antonin BESSIÈRES

Je m'appelle **Antonin Bessières**, étudiant en deuxième année de prépa à l'**EPITA**, avec un **bachelor spécialité Maths et NSI**.

J'ai commencé à m'intéresser à l'**informatique** lors de mon **stage de troisième** que j'ai eu la chance de faire chez **Thales** à Paris. Cet intérêt m'a ensuite guidé dans mes choix d'orientation.

Dans le cadre du projet, avec **Milo**, nous nous sommes occupés en particulier de l'aspect **mathématique**, en conséquence le **chiffrement/déchiffrement** et tous les **calculs nécessaires** à l'avancement du projet. J'ai aussi reçu la tâche de m'occuper du **site web**.

Nexium est un projet que nous avons développé sur le langage **Rust** que j'ai pu découvrir tout au long du projet, un langage que j'ai particulièrement apprécié. Pour la réalisation du site web, j'ai pu découvrir **Astro** pour sa construction ainsi que **Markdown** pour le contenu. Ces derniers sont très bien réalisés et très intéressants à utiliser.

Ce projet m'a permis aussi d'aborder le **chiffrement** et tout un aspect **serveur** dont je n'avais que vaguement entendu parler auparavant.

En résumé, ce projet a été un moyen de développer quelque chose de **passionnant** et m'a permis une découverte de nombreuses **connaissances** ainsi

qu'un développement dans l'apprentissage du langage **Rust** et dans la **création/développement de site web**.

2.4 William VALENDUC

Après **3 ans de licence informatique** à la faculté des sciences et ingénierie de l'université **Paul Sabatier**, j'ai rejoint **l'EPITA** en septembre dernier.

De nature très **curieuse**, j'ai à cœur de comprendre le fonctionnement de technologies dans de multiples domaines.

Réalisation de la **blockchain**, de l'**architecture et communication réseau**, ainsi que du **serveur web** et la **documentation du projet**.

Ne connaissant pas grand-chose du monde des **crypto-monnaies** avant le début de **Nexium**, c'est très enrichissant de découvrir cet univers. Possédant un peu d'expérience dans certains domaines que nous avons à utiliser, c'est un plaisir pour moi de pouvoir **partager** mais également **agrandir mes compétences**.

3 Pourquoi une cryptomonnaie ?

Bien que l'étendue des notions techniques que recouvre **Nexium** soit vaste, nous pensons qu'il est important, avant de se lancer dans les détails techniques, de préciser le sens qu'a l'existence des **cryptomonnaies** de manière générale.

Les **cryptomonnaies**, particulièrement et en premier lieu le **Bitcoin**, sont apparues comme une réponse aux limites et faiblesses du système économique traditionnel. Une devise devrait avant tout être un **vecteur d'échange**, une unité de mesure permettant de faciliter les transactions et de stocker la valeur. Or la **monnaie moderne** repose sur la **confiance**.

Cette **confiance** en les **institutions financières** (telles que les **banques centrales**) qui émettent et régulent la masse monétaire, fait de ce système un système intrinsèquement faillible. Il repose sur des entités **centralisées**, susceptibles de manipulation, de politique monétaire inflationniste ou de **crises économiques imprévues**. Ces entités privées ou étatiques peuvent, indépendamment de la volonté du peuple, imposer des **restrictions sur l'accès aux finances** et rendre la **distribution financière déséquilibrée**.

Dans ce contexte apparu le **Bitcoin**. Porté par le mouvement **cypherpunk**[4] qui cherchait à préserver **vie privée**, **liberté d'expression**, et **garantie**

[4]Le mouvement cypherpunk dans les années 90 prônait la préservation de la vie privée, la liberté d'expression, et la sécurité des données dans un monde numérique.

de sécurité des données dans un monde qui devenait numérique et dont on ne mesurait pas toujours l'ampleur des enjeux sociétaux à l'époque, le **Bitcoin** a proposé une solution radicale : une **monnaie absolument décentralisée**, **open-source**, sans **intermédiaire de confiance**. Basée sur la technologie de **blockchain**, elle permet de **valider**, **enregistrer** et **émettre des transactions** de manière transparente, immuable et sécurisée, sans faire appel à quelconque **autorité centrale**. Ainsi, le **Bitcoin** introduit une forme de **trustless**, où la confiance n'est plus placée en un gouvernement ou une institution, mais dans le **code informatique** qui régit le système : une concrétisation de l'adage "*code is law*".

Cette **décentralisation**, loin d'être un simple détail technique, pose des questions profondes sur les notions de **contrôle**, d'**autonomie** et de **souveraineté individuelle**. Dans un monde où les **gouvernements** peuvent influencer la **monnaie nationale** par des **politiques économiques** et où les **institutions bancaires** détiennent le pouvoir de décision sur les **flux monétaires**, le **Bitcoin** a offert un espace de **liberté**, où le peuple peut être souverain de sa propre monnaie, sans intervention étatique. Ce modèle a non seulement des implications **économiques**, mais aussi **philosophiques**. Il remet en question les fondements mêmes de l'organisation de notre société, et offre une **alternative radicale** à la centralisation du pouvoir économique.

C'est dans cet esprit de **décentralisation**, de **souveraineté individuelle** et de **liberté** que **Nexium** trouve sa place. Nous voulons de **Nexium** qu'il incarne l'idée d'un futur où la **finance** et la **monnaie** ne seront pas seulement contrôlées par des entités **centralisées**, mais où chaque citoyen, chaque individu, peut participer, contribuer et décider des règles du monde dans lequel il vit.

Cette contextualisation nous paraissait naturelle à la lumière des cours d'**Éthique** que nous avons pu suivre lors du **troisième semestre** : le développement d'une **technologie** doit, tant que possible, s'accompagner d'une **réflexion sur ses enjeux sociétaux et humains**. Ainsi, **Nexium** s'inscrit dans une réflexion plus large sur l'évolution des sociétés, et sur l'émergence de nouveaux systèmes de **valeurs partagées**.

4 Architecture de la blockchain

Comme brièvement mentionné dans la présentation du projet, la **cryptomonnaie Nexium** (dont le symbole de devise est le **NXM**) est basée sur la **blockchain**. Définissons donc les grandes lignes de cette technologie, et comment elle est implémentée dans **Nexium**.

Chaque **transaction** effectuée par un utilisateur est **signée**, puis ajoutée

à un **bloc** contenant un certain nombre de transactions. Ce bloc est ensuite ajouté à la **blockchain**, une chaîne de blocs contenant l'**historique** de toutes les transactions effectuées sur le réseau. La blockchain est un **registre public**, **partagé** et **immuable**, qui permet de garantir la **transparence** et la **sécurité** des transactions.

Chaque bloc est lié au bloc précédent par un **hash cryptographique** à l'image d'une liste chainée, ce qui assure l'**intégrité de la chaîne**. En effet, si un bloc est modifié, son **hash** est changé et le hash du bloc suivant ne correspond plus. Cela permet de détecter d'éventuelles **tentatives de modification** de la blockchain, et de les rejeter.

La méthode de **hashage** utilisée est celle du **SHA-256**, que nous détaillerons plus loin. Cette **fonction de hashage** est utilisée pour **signer les transactions**, **générer les blocs**, et **lier les blocs entre eux**.

4.1 Stockage de la blockchain

La **blockchain** est stockée dans un **fichier binaire** `blockchain.dat` qui contient les **données codées des blocs**, du **genesis block** (bloc initial) jusqu'au **dernier bloc**.

Nous pouvons rechercher n'importe quel **bloc** de la blockchain en **parcourant successivement les blocs suivants**, en partant du **bloc initial**.

Pour vérifier l'**intégrité** ou la **validité** d'une **transaction**, nous pouvons vérifier son **existence dans un bloc donné** à l'aide de la **racine Merkle**, avec une complexité de **O(log n)**.

4.2 Mémoire tampon (mempool)

La **mémoire tampon** est un **stockage temporaire** pour toutes les **transactions** qui ne sont pas encore incluses dans un **bloc**.

Cette mémoire tampon est continuellement **synchronisée** entre tous les **nœuds du réseau Nexium**.

Lorsque la mémoire tampon est pleine, une **vérification des transactions** a lieu et tous les nœuds commencent à **miner** en gardant uniquement les **transactions valides** pour créer le **nouveau bloc**. Le premier nœud qui trouve le **Nonce** qui satisfait la **cible de difficulté** diffuse le bloc sur le réseau.

C'est ainsi que la **preuve de travail** est réalisée : le **proof of work** consiste à fournir la **preuve cryptographique de minage**, c'est-à-dire le

Nonce[5] qui satisfait la **cible de difficulté**[6] actuelle.

4.3 Structure d'un bloc

Un **bloc** dans la blockchain de **Nexium** est structuré de la manière suivante :

En-tête du bloc (82 octets) :

- **version** (2 octets) : Version de la structure du bloc.
- **previous_block_hash** (32 octets) : Hachage du bloc précédent.
- **merkle_root** (32 octets) : Racine Merkle des transactions du bloc.
- **timestamp** (4 octets) : Horodatage du bloc.
- **difficulty_target** (4 octets) : Cible de difficulté du bloc.
- **nonce** (4 octets) : Nonce du bloc.
- **transaction_size** (4 octets) : Taille en octets des transactions du bloc.

Transactions (variable) : Liste des transactions.

- **transaction_header** (73 octets) :
  - **transaction_size** (4 octets) : Taille de la transaction.
  - **timestamp** (4 octets) : Horodatage de la transaction.
  - **fees** (2 octets) : Frais de la transaction.
  - **emitter** (64 octets) : Login de l'émetteur de la transaction (prenom.nom).
  - **data_type** (1 octet) : Type de la transaction.
- **data** (1 à 1719 octets) : Données de la transaction.
- **signature** (256 octets) : Signature de la transaction.

[5] Le Nonce est une valeur que les mineurs doivent ajuster de telle sorte que le hash du bloc concerné remplisse certaines conditions (dans le cas du Bitcoin, il doit commencer par un certain nombre de 0.)

[6] La cible de difficulté est un paramètre évoluant au cours du temps selon le consensus du réseau de la blockchain, et qui établit quelles contraintes doivent être remplies pour que le Nonce fournisse un hash valide.

Cette conception, pour laquelle nous nous sommes directement inspirés du **Bitcoin**, permet de stocker les **informations essentielles** de chaque bloc.

Étant donné que les données sont directement écrites en **binaire** sans label ni séparation, il est important de respecter scrupuleusement l'**ordre de lecture/écriture** de chaque champ pour éviter les problèmes de **données corrompues**. Si on prend en exemple le cas de l'en tête des blocs, sur le buffer de 82 octets, on retrouvera sur l'intervalle 0-2 la version du bloc, puis sur 2-34 le hash du bloc précédent, puis sur 34-66 le merkle root, 66-70 le timestamp, et ainsi de suite.

Cette structure permet également un **parcours fluide** de la blockchain depuis le **genesis block** : en sautant à chaque fois de n octets (avec n la taille du bloc), on peut accéder efficacement à n'importe quel bloc.

L'**identifiant unique** d'un bloc correspond à son **double hash SHA-256**, c'est-à-dire le **hash du hash du bloc**. Cela garantit l'**unicité** de chaque bloc et facilite la **recherche** ainsi que la **vérification de l'intégrité** de la blockchain.

$$\boxed{\text{id(bloc)} = \text{SHA256(SHA256(bloc))}}$$

Un **arbre de Merkle (merkle root)** est une **structure de données arborescente** qui permet de **résumer efficacement un grand nombre de transactions**, assurant ainsi l'**intégrité** et une **vérification rapide** des données à travers des hachages successifs.

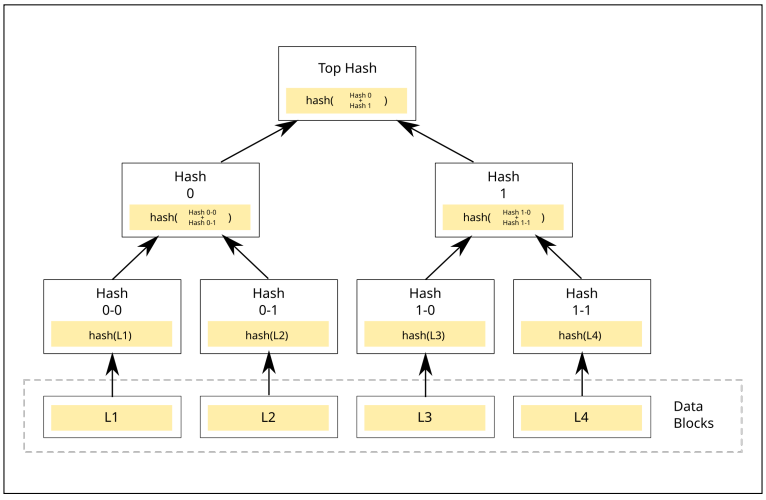

Figure 1: Arbre de Merkle.

Ainsi, à partir de la **racine Merkle**, on peut vérifier avec une **complexité logarithmique** si une **transaction** est incluse dans un bloc donné.

4.4 Fonctionnement

Tout commence avec la création d'une **transaction**. Il en existe plusieurs types mais ont toutes un **fonctionnement identique**. Prenons en exemple une transaction dite classique (`ClassicTransaction`), contenant plusieurs informations.

Toutes les transactions ont en commun une **en-tête** que l'on appelle **header** contenant des informations sur l'**émetteur**, la **date**, les **frais**, ainsi que la **taille** et le **type** de son contenu, que l'on appelle **data**. C'est cette partie **data** qui diffère selon les différents types de transactions, et si l'on prend l'exemple pris précédemment, les **transactions classiques**, qui servent tout simplement à transmettre des **Nexiums**, contiennent des **informations complémentaires** pour mener à bien la transaction. On y retrouve évidemment le **montant** et la **personne réceptrice**, mais également une **description optionnelle** et un **indicateur** révélant si oui ou non il y a une description.

Une fois toutes ces informations regroupées, nous pouvons alors faire une **signature** à l'aide d'une **clé RSA** et ces données, **signature** que nous ajoutons à la fin de la transaction. Cette **signature** a pour but de confirmer l'**identité de l'émetteur** de la transaction.

Une fois complète, cette transaction est alors **envoyée à un serveur** qui, après une simple **vérification de la signature**, l'ajoutera à la **mempool**. Elle sera alors **placée en attente** parmi d'autres. Une fois remplie, la mempool sera **vidée** et toutes les **transactions** y figurant seront **traitées** avant d'entamer la **création du bloc**.

La **vérification** pour les **transactions classiques** consiste à assurer que les **montants ne provoquent pas de solde négatif** et que les **destinataires existent bel et bien**. Les **transactions invalides** sont alors **rejetées** et ne seront pas prises en compte dans la suite.

Lorsque la vérification est terminée, l'étape du **"Proof of Work"**, plus communément appelé **"minage" du bloc** commence. En effet, chaque **nœud** va **hasher le bloc** de manière successive, et plus précisément **hasher deux fois successivement** pour faire un **double hachage**, en faisant varier le **nonce** afin que le **hash** corresponde au **critère de difficulté**, c'est-à-dire que le hash débute par un **nombre défini de 0**. Cette étape **intensive pour les machines** a pour but de **dissuader les tiers malintentionnés**.

Une fois ce **hash** trouvé, le **bloc nouvellement créé** est ajouté à la suite de la **blockchain**. Le **nœud** ayant trouvé le bloc peut alors le **partager aux autres** qui pourront très facilement **vérifier s'il est bon** en le **hashant également**.

4.5 Implémentation et parcours

Afin d'avoir une **implémentation facile à utiliser**, nous avons fait le choix d'implémenter une structure `Blockchain` contenant **trois attributs** nous permettant un **parcours bi-directionnel** des blocs.

Le premier étant `size`, soit la **taille en octets de la blockchain**, nous permettant un **parcours en ordre chronologique** en partant de 0, en ajoutant la taille du bloc lu, puis en s'arrêtant lorsque cette taille est atteinte. Elle est **calculée à l'initialisation** de la structure lors du **contrôle de l'intégrité** de la blockchain (vérification de la **liaison des hash**), puis **incrémentée** à chaque ajout de bloc. Bien que non indispensable, cet attribut permet de **réduire les erreurs** lors de la lecture ou même de les **détecter** en cas de valeurs incohérentes.

Les deux autres attributs sont bien plus intéressants puisqu'ils nous permettent un **parcours inverse**. Nous avons donc `last_hash`, qui comme son nom l'indique est le **hash du dernier bloc** de la chaîne, associé à un **cache** sous forme de `HashMap` contenant les **hash en clé** et un **offset en valeur**.

En passant `last_hash` dans le cache, on obtient l'**offset dans le fichier** du dernier bloc, en lisant ce bloc nous obtenons le **hash précédent**, permettant d'obtenir l'offset du bloc précédent, et ainsi de suite jusqu'à tomber sur le **premier hash ne contenant que des zéros**.

5 Architecture du réseau

5.1 Décentralisation

Nous avons choisi une **approche décentralisée** afin de garantir une **disponibilité élevée** et ainsi assurer un **réseau fiable** face aux pannes. Chaque **nœud** est donc **indépendant** des autres, pouvant ainsi continuer à fonctionner normalement même en cas de nœud hors ligne ou défaillant.

Afin de garantir la **communication** entre tous les nœuds et d'**optimiser** les échanges au sein du réseau, nous nous sommes appuyés sur la **théorie des graphes**, notamment les notions de **connexité**, **bi-connexité** et d'autres propriétés pertinentes.

Voici une illustration du concept de **forte connexité**, que nous avons utilisé pour la distribution du réseau :

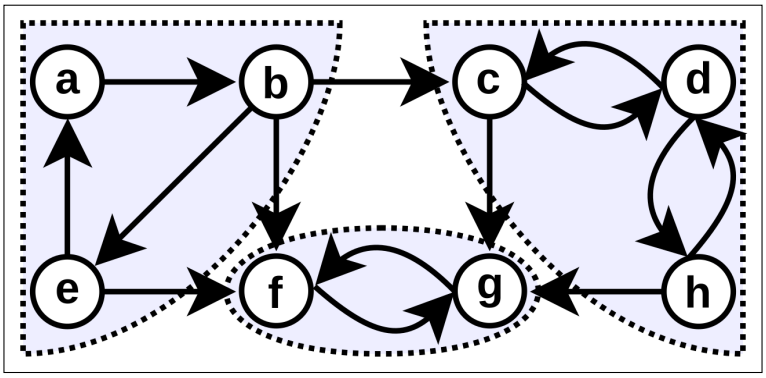

Figure 2: Exemple de réseau fortement connexe.

La **sécuarisation d'un réseau décentralisé** repose principalement sur la maintenance par un **grand nombre de participants** dans une communauté.

Il nous a donc semblé **plus fiable de déployer Nexium de manière centralisée** dans un premier temps, afin de **limiter les attaques** et les **problèmes potentiels** liés à une adoption initiale encore trop faible.

5.2 Sécurité

Nous avons songé au **problème** qui se poserait dans l'éventualité où une personne tenterait de posséder une part au moins égale à **51% des nœuds du réseau**. Le choix a été fait, pour cette raison, de **restreindre les nœuds** à un par membre d'**Épita**, en se basant sur le **login unique** de chaque élève et professeur, que l'on retrouve notamment dans son adresse mail `@epita.fr` et qui se compose par `<prénom>.<nom>`.

Nous pouvons ainsi, par un système que nous avons mis en place, accéder à la **clé publique** de n'importe quel utilisateur via l'**API de Gitlab**. La requête suivante permet de récupérer la **clé publique** d'un login donné :

```
GET /login/gpg_keys/key_id
```

6 Hashage SHA-256

Pour notre projet, nous avons fait le choix d'avoir recours à une **méthode de hachage** pour garantir la **sécurité** et l'**intégrité** des informations/données lors de leur **transmission**, mais aussi comme **preuve de travail** (**Proof of Work**, **PoW**) présente dans la **blockchain**.

6.1 Définition d'une méthode de hachage

Définissons tout d'abord ce qu'est une **méthode de hachage**. C'est une **fonction mathématique** dont le but est de transformer une **chaîne de caractères** de **taille quelconque** en une nouvelle chaîne de **longueur fixe et unique**.

Avoir une fonction mathématique, c'est bien, mais alors pourquoi s'en servir ? La particularité d'une **fonction de hachage** est que la **transformation est irréversible** due aux nombreux **décalages de bits**, **XOR** et autres **opérations** appliquées sur la chaîne de caractères initiale.

Mais, en plus d'être **irréversible**, le **moindre changement** dans l'entrée initiale **change complètement** la valeur de sortie.

6.2 Utilités des fonctions de hachage

Maintenant que nous savons ce qu'est une **méthode de hachage**, intéressons-nous à toutes ses **utilités**, mais surtout à celles qui nous concernent.

Passons d'abord aux **utilités possibles** qui ne sont pas en relation directe avec notre projet. Parmi celles-ci, on retrouve :

- le **stockage et l'authentification de mots de passe**,
- la **création de signatures numériques**,
- la **détection de logiciels malveillants**.

À présent, regardons les **deux points qui nous concernent** dans l'utilisation du **hachage** dans la **blockchain**.

6.2.1 Intégrité des données

Un premier point déjà évoqué est l'**intégrité des données**, permettant de **repérer les données possiblement corrompues ou falsifiées** grâce à une simple **comparaison** entre le **résultat attendu** et le **résultat calculé**.

6.2.2 Preuve de travail

Le second point est la **preuve de travail** (**Proof of Work**, **PoW**) au sein de la **blockchain**. Ce **mécanisme de consensus** permet d'obtenir un **accord sur le réseau de blockchains**, pour **confirmer les transactions** et **produire de nouveaux blocs** sur la chaîne.

6.3 Utilisation du SHA256 dans le Bitcoin

Avant de parler de l'**implémentation du SHA-256**, voyons son **utilisation dans le Bitcoin** :

- **Vérification de l'intégrité des transactions** : une transaction ne doit pas pouvoir être modifiée. Grâce au **hachage**, la modification de n'importe quel caractère entraîne une **différence de résultat** et donc la **détection immédiate** de la moindre anomalie.
- **Lien entre les blocs** dans un **Arbre de Merkle**, ce qui permet de **condenser** et **sécuriser la mémoire**.
- **Utilisation dans le processus de preuve de travail**, déjà évoqué.

6.4 Implémentation du SHA256

Tous les **points** et les **définitions** ayant été abordés, regardons de quelle manière nous avons **implémenté le SHA-256**. Pour ce faire, nous avons fait le choix de **séparer en deux** la **transformation de la chaîne de caractères** :

6.4.1 Prétraitement des données

La fonction **preprocessing** prend une **chaîne de caractères** en entrée et la convertit en un **vecteur d'octets**. Elle ajoute ensuite un **bit de fin** (`0x80`) et remplit le vecteur avec des **zéros** jusqu'à ce que sa **longueur** soit un **multiple de 512 bits moins 64 bits**.

Enfin, la **longueur initiale** de la chaîne, exprimée en **bits**, est ajoutée à la fin du vecteur. Cette étape prépare les données pour le traitement en assurant qu'elles respectent le **format requis par l'algorithme de hachage SHA-256**.

6.4.2 Traitement des données

La fonction **processing** commence par initialiser des **valeurs de hachage** et des **constantes de rondes** spécifiques à l'algorithme **SHA-256**. Puis, elle :

- **Découpe** les données en **blocs de 64 octets** et prépare un **tableau de 64 entiers de 32 bits** pour chaque bloc.
- Effectue une série d'**opérations bit à bit** et **arithmétiques** pour **mettre à jour les valeurs de hachage**, en utilisant des **variables temporaires** calculées à partir des données et des constantes de rondes.

- Après avoir **traité tous les blocs**, la fonction **combine les valeurs de hachage finales** pour produire une **chaîne hexadécimale** représentant le **hachage final**.

Cet algorithme renvoie une **sortie fixe de 256 bits**.

7 Chiffrement RSA

Notre **réseau** utilise le **chiffrement RSA**. Il s'agit d'un concept de **cryptographie asymétrique** composé d'une **clé privée** et d'une **clé publique** par utilisateur, qui leur permettront de s'échanger des **messages** tout en restant **secrets** et **sécurisés**.

7.1 Fonctionnement du RSA

Ainsi, un utilisateur **A** pourra envoyer un message en le **cryptant** avec sa **clé publique** à un utilisateur **B** qui, pour le lire, devra **décrypter** ce message avec sa **clé privée** et vice-versa.

Cela rend **impossible**, sans connaître la **clé privée du receveur**, d'intercepter un message qui ne vous est pas destiné.

Notre travail est donc de **créer/générer ces clés** et de faire en sorte que chaque personne utilisant **Nexium** puisse **crypter** et **décrypter des messages** grâce à leur propre **clé privée** et **publique**.

7.2 Génération de clés

Tout d'abord, la **génération de clé** se passe sous la forme d'un **algorithme mathématique** qui forme des clés en utilisant cinq nombres, que nous appelons **p**, **q**, **n**, **e** et **d**, où :

- **p** et **q** sont deux **nombres premiers distincts**,
- **n** est leur **produit**, $n = p \times q$,
- **e** est un **nombre premier** avec la valeur de l'indicatrice d'Euler en **n**,
- **d** est l'inverse **modulaire** de **e mod($\varphi(n)$)** où $\varphi(n) = (p-1)(q-1)$.

La **clé publique** sera donc le couple (n, e) tandis que la **clé privée** est le nombre **d**.

Nous avons rencontré des **problèmes** lors de nos premières générations de clés. Il s'agissait de **problèmes d'optimisation** liés à la **taille en bits**, en

rapport donc avec les **types natifs** de Rust qui sont limités en taille et qui deviennent donc insuffisants si nous avons besoin de manipuler des **nombres très grands**. Or, le standard **RSA** requiert généralement des **nombres de l'ordre de 1024 bits** pour être **sécurisé**.

Ces types de **taille fixe en mémoire** ont donc posé problème pour la génération. Pour résoudre ce problème, nous avons dû utiliser les **crates** `num-bigint` et `num-primes`.

Ces **crates** nous permettent de **générer de grands entiers** sans limite de taille. Par exemple, `num_bigint::BigUint` est une **structure dynamique** qui s'adapte à la taille du nombre, contrairement aux **types natifs de Rust** (grâce à l'utilisation de **vecteurs u32** ou **u64** pour limiter la taille en mémoire). Nous avons également utilisé `num_primes` pour **générer aléatoirement** et manipuler des **nombres premiers efficacement**.

7.3 Chiffrement et Signature

Grâce à une fonction de génération de clé **RSA**, nous avons pu implémenter l'ensemble des fonctions nécessaires pour réaliser tous les principes liés au **chiffrement RSA**, à savoir : la **signature** et la **vérification de signatures**, ainsi que le **chiffrement** et le **déchiffrement**. Ces fonctions reposent sur les fondements de **RSA** (nombres premiers, clés publique et privée, **n**, **d**, **e**) tout en intégrant un **hachage cryptographique** (**SHA-256**) afin de garantir l'**intégrité des messages**.

Signature (sign) La première fonction, `sign`, permet de générer une **signature numérique** pour un message donné. Elle repose sur le principe selon lequel seul le détenteur de la **clé privée** doit être capable de produire une signature valide. Voici le déroulé de l'opération :

- Le message est tout d'abord transformé en un **condensé** (hash) grâce à l'algorithme **SHA-256** implémenté plus tôt dans le projet. Ce **hachage** garantit que la signature porte uniquement sur le contenu du message, indépendamment de sa taille.
- Le condensé est ensuite converti en un **entier** (`BigUint`) pour être manipulé par les fonctions **RSA**.
- L'entier est signé en le mettant à la puissance **d** (exposant privé) modulo **n** :

$$S = H(m)^d \mod n$$

- La fonction renvoie une erreur si le message est vide ou si la taille du haché dépasse celle du module **n**.

- Cette **signature** peut ensuite être transmise avec le message pour vérification.

Vérification de signature (check_signature) La fonction `check_signature` permet de s'assurer que la signature reçue correspond bien au message. Elle utilise la **clé publique** du signataire, ce qui garantit que seule la personne possédant la **clé privée** a pu produire cette signature. Voici le déroulé de l'opération :

- Le message reçu est haché de la même manière que dans la fonction de signature.
- La signature est déchiffrée en la mettant à la puissance **e** (exposant public) modulo **n** :
$$m' = S^e \mod n$$
- Si le résultat du déchiffrement correspond au **hash recalculé**, alors la signature est considérée comme valide.
- Cela confirme que le message n'a pas été modifié ou falsifié, et qu'il provient bien de l'expéditeur déclaré. Ce mécanisme est au cœur de la **validation d'identité** dans les protocoles sécurisés.

Chiffrement (crypt) La fonction `crypt` permet de **chiffrer** un message de manière **confidentielle**. Ce **chiffrement** est réalisé à l'aide de la **clé publique** du destinataire. Voici le déroulé de l'opération :

- Le message est tout d'abord converti en un **entier**.
- Il est ensuite chiffré par l'opération RSA classique :
$$c = m^e \mod n$$
- Une vérification est effectuée pour s'assurer que la valeur à chiffrer est inférieure au module **n**, sinon une **erreur** est renvoyée.
- Ce **chiffrement** garantit que seul le détenteur de la **clé privée** pourra déchiffrer le message, assurant ainsi la **confidentialité**, même en cas d'interception du message.

Déchiffrement (decrypt) Le **déchiffrement** est l'opération inverse du **chiffrement**. Il s'effectue avec la **clé privée**. Voici le déroulé de l'opération :

- Le message chiffré (déjà sous forme d'entier) est déchiffré par l'opération :
$$m = c^d \mod n$$

- On obtient ainsi le message **original**, à condition que la **clé privée** utilisée corresponde à la **clé publique** ayant servi au chiffrement.
- Comme toujours, si le message est vide ou trop grand, une **erreur** est renvoyée.

7.4 Format PEM

Après avoir généré notre paire de clés **RSA** (publique et privée), nous avons souhaité les placer dans un **format standard**, **sécure**, et reconnu par les outils de chiffrement tels que **GnuPG**. Pour cela, nous les avons soumises à un processus de **sérialisation** vers le format **OpenPGP**, puis encapsulées dans un conteneur **ASCII Armor** au format **PEM**.

La transformation de la structure **RSA** en format **OpenPGP** suit plusieurs étapes, conformément à la **RFC 4880** (format des messages OpenPGP).

Paquet de clé publique En premier lieu, nous construisons un **paquet de clé publique** contenant :

- La **version** du format OpenPGP (4) ;
- Un **timestamp** correspondant à la date de création ;
- L'identifiant de l'**algorithme** (1 pour RSA) ;
- Les deux entiers principaux de la clé RSA : **n** et **e**.

Ces derniers sont encodés au format **MPI** (**Multi Precision Integer**) :

- Une **entête de deux octets** encode la longueur de l'entier en bits ;
- Une **série d'octets** représente la valeur de l'entier en **Big Endian**.

L'encodage d'un entier se fait donc en deux parties, permettant une **taille réduite** comparée à la valeur représentée. Par exemple, **e** (souvent égal à 65537) possède 17 bits significatifs et est représenté sur 5 octets (2 pour la taille, 3 pour la valeur). Le format MPI est utilisé dans tous les paquets contenant des entiers cryptographiques dans le protocole OpenPGP (paquets de clé publique, clé privée, signature, sous-clés, ou messages chiffrés).

Paquet de clé privée Le paquet de **clé privée** suit le même principe que celui de la clé publique, avec en plus les autres entiers RSA (**d**, **p**, **q**, et **u**). Ces valeurs sont assemblées, puis hachées avec **SHA-1** pour vérification. Le tout est ensuite chiffré avec **AES-128** en mode **CFB**, à l'aide d'un **mot de passe** fourni par l'utilisateur.

Ce chiffrement utilise un mécanisme appelé **S2K** (**String to Key**) qui dérive une clé AES à partir :

- du **mot de passe** ;
- d'un **salt de 8 octets** ;
- d'un **nombre d'itérations** ;
- d'un **vecteur d'initialisation aléatoire**.

L'ensemble du bloc chiffré est ensuite inclus dans le paquet avec les **métadonnées nécessaires**. Ce paquet possède un **tag** différent de celui de la clé publique.

Paquet d'identifiant utilisateur Un **paquet d'identifiant utilisateur** est également présent, sous forme de chaîne **UTF-8** au format :

```
prénom.nom <email@epita>
```

Paquet de signature Il existe aussi un **paquet de signatures** dont le rôle est de lier cryptographiquement la **clé publique** à un identifiant utilisateur. Il est composé :

- d'un **en-tête** ;
- de sous-paquets **hachés** et **non hachés** ;
- d'une **empreinte SHA-256** ;
- d'une structure **ASN.1** suivie d'une **signature RSA** faite avec la **clé privée**.

Comme pour les autres entiers cryptographiques, cette signature est encodée au format **MPI**.

Encapsulation PEM Une fois tous les paquets assemblés pour la **clé publique** (ou la **clé privée**), ils sont :

- **concaténés en binaire** ;
- **encodés en base64** ;
- **complétés par un CRC24** (contrôle d'intégrité sur 3 octets).

Enfin, le tout est entouré d'une **balise ASCII**, différente selon qu'il s'agisse d'une **clé publique** ou d'une **clé privée**.

Ce format est lisible par les outils **OpenPGP**, tout en étant transportable sous forme de texte. Le processus complet, conforme à la **RFC 4880**, assure aux clés une **compatibilité avec les outils** tels que **GPG**.

Vous trouverez ci-dessous un exemple de clé RSA transformée par ce processus de formatage PEM :

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

xsBNBGfyzYYBB/sGc+nlGYhmO6KVxZ8ii+ajK+HQ5jD0J9U+ZXzM3fO2FBkMFzoj
rZYvPtwV7UJ2QeF255SMCZJAYfPexdfznLn5MbbPgb7KxMdfQ1bnYFnjOTU3FzZT
7lWx1klP5er4Qht7OvuMs2KdfoJLBGm9AyCwh2setZ9aoKJq0zqit1Q4ofC+r6Xb
m3GmFy3+H6xOn5+591KGfnfxTHwCKEgbFnPKPHHb68eD9+z+eDzgHSo5KxINlLfH
shZFWVelP2x4nKi5awElO5ARH2TgZfOXLAaWfgyXg364fhZrjryovrOLOBN5r4IR
UQHZNAA1AFnghD5T4UK4fqQibfTiWNqryZqLABEBAAHNImplYW4uaGVyYWlsIDxq
ZWFuLmhlcmFpbEBlcGl0YS5mcj7CwFwEEwEIABAFAmfyzYYJEPzyLMmr8iq8AABI
ugf5AY+c/ECk/A2YonfSfFO7hTVP7Jbhv0FgwovIy5BwZrILBea4dQ4spq/ovd2
qItm1/tQdmj17ncoduKBQsLcnr+GDMA6GzgOi6wadGD2i3InfxTqRhftxj4d3auk
bcujcTzMVKC59x64PVRR2ed4pLeLaTu1gZJQLPdUWra1ZGPEKUini8bK6HVktr93
aRxx59giJHEh6QIdT6l1x9aS1HufEkjY9ZYylb42ERr0tORrvPqPqfNI8xsAUMak
OjdkKHI7RldUCPfwiLk9DDuQRz+yG7LrvI+eNzfe2x9cIEl2klcgBu1cBg8UCzyQ
GImS1ic0Je1gXGOnaPhZalb0kA==
=XWDw
-----END PGP PUBLIC KEY BLOCK-----
```

public_key.pem

8 Gitlab

Dans le cadre de notre projet, nous avons développé une **API GitLab** en **Rust**, afin d'interfacer facilement notre application avec l'instance GitLab d'**Épita** (`gitlab.cri.epita.fr`). Cette **API maison** encapsule plusieurs **fonctionnalités essentielles**, telles que :

- l'**authentification OAuth2** (via le flux d'autorisation **PKCE**),
- la **vérification de la validité** d'un **jeton d'accès**,
- la **récupération des identifiants utilisateurs** et de leurs **clés GPG**,
- l'**ajout d'une clé GPG** à un compte utilisateur.

Ce **module** est principalement utilisé pour **interagir avec les utilisateurs** via leurs **identifiants EPITA**, ce qui permet de garantir la **sécurité** et la **traçabilité** des actions au sein du **réseau Nexium**.

8.1 Authentification OAuth2

Pour les **clients**, l'**authentification** se fait via un **navigateur web** ouvert localement. Une fois l'utilisateur connecté, un **petit serveur HTTP local** récupère le **code d'autorisation** retourné par **GitLab**, puis échange ce code contre un **token d'accès API**.

Cette opération est encapsulée dans la méthode suivante :

```
pub fn get_token() -> Result<String, GitlabError>
```

Listing 1: Obtention d'un token GitLab via OAuth2

8.2 Structure principale

La structure `GitlabClient` centralise l'état de la connexion :

```
pub struct GitlabClient {
    api_url: String,
    token: String,
}
```

Elle est instanciée à l'aide d'un jeton d'accès :

```
pub fn new(token: String) -> Self
```

8.3 Fonctions principales

Voici un aperçu des principales fonctionnalités de l'API :

- **Vérification de la validité d'un token** :

  ```
  pub fn check_token(&self) -> Result<bool,
      GitlabError>
  ```

  Cette fonction vérifie que le token fourni donne bien accès aux informations de l'utilisateur.

- **Récupération des clés GPG d'un utilisateur** :

  ```
  pub fn get_gpg_keys(&self, login: &str) -> Result
      <Vec<String>, GitlabError>
  ```

  Elle récupère les clés GPG associées au compte Gitlab d'un utilisateur donné.

- **Ajout d'une clé GPG** :

```
pub fn add_gpg_key(&self, gpg_key: &str) ->
    Result<(), GitlabError>
```

Cette fonction permet à l'utilisateur d'ajouter une nouvelle clé GPG à son compte.

8.4 Erreurs possibles

Nous avons défini une énumération `GitlabError` pour capturer proprement les erreurs liées à l'authentification ou à la communication avec l'API :

```
pub enum GitlabError {
    InvalidToken,
    NetworkError,
    UserNotFound,
    UnknownError,
    NoGPGKeys,
    BadGPGFormat,
    NoWebBrowser,
}
```

8.5 Utilisation dans le projet

L'**API** est utilisée pour **authentifier les utilisateurs** de **Nexium** avec leur **compte GitLab EPITA**, récupérer leur **clé GPG** et ainsi leur permettre de **signer les transactions ou blocs** avec une **preuve d'identité vérifiable**.

Cette intégration permet un **couplage fort** entre l'**identité réelle** d'un utilisateur et son **identité sur la blockchain Nexium**, renforçant la **sécurité** et la **confiance du système**.

9 Gestion de la configuration serveur

Le **serveur Nexium** nécessite plusieurs informations de **configuration** au démarrage, telles que les **chemins vers la base de données** ou la **clé privée**, les **informations GitLab de l'utilisateur**, ainsi que le **port d'écoute**. Ces informations sont regroupées dans une structure `Config`, définie dans le module `config.rs`.

Ce module permet de **générer un fichier de configuration**, de **le lire depuis un fichier JSON**, ou encore de **sauvegarder les paramètres courants**.

9.1 Structure de configuration

Les différents paramètres sont encapsulés dans la structure suivante :

```
pub struct Config {
    pub key_filepath: String,
    pub key_password: String,
    pub listen: String,
    pub port: u16,
    pub user_login: String,
    pub gitlab_token: String,
}
```

Listing 2: Structure de configuration

9.2 Génération interactive

La configuration peut être générée de manière interactive à l'aide de la méthode `generate`, qui demande à l'utilisateur les différentes informations via l'entrée standard. Des valeurs par défaut sont proposées, et certains champs (comme le login) sont validés :

```
pub fn generate(path: &Path) -> Config
```

Les valeurs par défaut sont les suivantes :

- Adresse d'écoute : `0.0.0.0`
- Port : `4242`
- Clé privée : `private-key.pem`
- Mot de passe de la clé : `""` (vide)
- Identifiant utilisateur : `prenom.nom` (doit être validé)
- Token GitLab : `""` (doit être validé)

Le `token Gitlab` peut également être lu depuis la variable d'environnement `GITLAB_TOKEN`, si elle est présente.

9.3 Lecture et sauvegarde

Une fois générée, la configuration peut être sauvegardée sous forme de fichier JSON lisible :

```
pub fn to_file(&self, path: &Path)
```

À l'inverse, un serveur peut être initialisé à partir d'un fichier JSON existant :

```
pub fn from_file(path: &Path) -> Config
```

Le fichier doit contenir les champs suivants :

```
{
    "key": "private-key.pem",
    "key_password": "",
    "listen": "0.0.0.0",
    "port": 4242,
    "user_id": "prenom.nom",
    "gitlab_token": "XYZ"
}
```

9.4 Validation et ergonomie

Pour améliorer l'**ergonomie**, la méthode `Config::get_user_input` permet de poser des questions à l'utilisateur avec une gestion simple des **retours clavier**. Un **validateur de login** est aussi inclus dans un module `Login` que nous avons créé pour vérifier ces structures :

```
pub enum LoginError {
  EmptyLogin,
  TooMuchPoints,
  NoPoint,
  InvalidCharacter,
  MissingField,
  InvalidFirstName,
  InvalidLastName,
  UnknownError,
}

pub struct Login {
  pub login: String,
}

impl Login {
  pub fn new(login: String) -> Result<Self, LoginError>
    {}
}
```

Listing 3: /lib/src/login/mod.rs

Ce système permet de garantir que l'**identifiant utilisateur** respecte bien le format `prenom.nom` imposé par l'infrastructure **GitLab** de l'Épita.

9.5 Intégration dans le projet

Cette **gestion centralisée de la configuration** rend le **serveur Nexium** facile à **déployer**, **portable**, et **cohérent** avec le reste du projet. Elle permet notamment d'assurer que chaque instance est bien connectée à une **identité GitLab valide**, et qu'elle dispose de toutes les **ressources nécessaires** à son fonctionnement dès son démarrage.

10 Design Web

10.1 Tauri

Pour la réalisation de notre application pour les utilisateurs de Nexium, nous avons choisi d'utiliser **Tauri**, un framework qui permet de créer des applications de bureau multiplateformes en combinant du code front-end web (HTML, CSS, JavaScript ou des frameworks comme *Svelte*) avec un back-end écrit en **Rust**.

Tauri s'intègre parfaitement dans notre projet codé dans son entièreté avec Rust. Il permet aussi de générer des exécutables sécurisés et adaptés à une utilisation multiplateforme (Windows, Linux, macOS), ce qui facilite sa diffusion et son utilisation à grande échelle, notamment pour tous les étudiants d'**EPITA**.

Grâce à Tauri, nous avons pu développer une interface graphique fluide, correspondant à ce que nous imaginions, avec de bonnes performances. La communication entre le front-end et le back-end (Rust) se fait grâce à un système de **commandes sécurisées**, ce qui permet de déclencher des fonctions bas niveau (accès fichiers, cryptographie, communication réseau) depuis l'interface utilisateur, sans exposer de failles de sécurité.

10.2 Svelte et SvelteKit

Nous avons choisi d'utiliser **Svelte** et **SvelteKit** pour le développement de l'application web. Svelte est un framework JavaScript front-end permettant de créer des composants d'interface utilisateur interactifs en les connectant à des fonctions, notamment celles que nous avons réalisées en Rust via des appels. **SvelteKit**, quant à lui, intègre ces éléments dans une application web complète, avec une communication fluide avec le back-end, qui dans notre cas est développé en Rust.

Pour citer des exemples d'utilisation dans notre code, nous pouvons mentionner la création de modales lors de l'ouverture des paramètres, la création

d'une facture ou d'une transaction, ou encore pour consulter l'historique des transactions.

Ces modales sont chacune indépendantes et contiennent du code qui leur est propre. Prenons comme exemple la modale *paramètres* : elle comporte plusieurs champs que l'utilisateur doit remplir. Nous avons utilisé Svelte pour faciliter l'interaction avec l'utilisateur, notamment pour des fonctionnalités telles que la connexion à GitLab via le *token OAuth* — une fonction implémentée en Rust et appelée par le script JavaScript de la modale Svelte *paramètres*.

Nous avons également intégré des fonctions de sauvegarde et de chargement de fichiers au format `.json` (elles aussi réalisées en Rust), permettant de conserver ou de restaurer la configuration, évitant ainsi à l'utilisateur de devoir la saisir à chaque lancement de l'application.

10.3 Tailwind CSS

Pour le design et la mise en forme rapide de l'interface, nous avons utilisé **Tailwind CSS**, un framework utilitaire qui permet de styliser les composants directement via les classes HTML. Grâce à des classes comme `flex`, `gap`, `justify`, etc., nous avons pu construire une interface esthétique et cohérente, sans avoir à créer immédiatement un fichier CSS dédié pour chaque composant.

Tailwind nous a permis de gagner en productivité, de tester rapidement différents agencements, et de maintenir une base de code **claire**, **lisible** et **facilement modifiable**.

10.4 HTML / CSS

Pour l'interface graphique de **Nexium**, nous avons choisi d'utiliser **HTML** et **CSS**, deux langages répandus et standards dans le monde du développement web. Ce choix est également motivé par leur simplicité d'utilisation, leur caractère universel et leur compatibilité avec **Tauri**.

Le HTML nous permet de concevoir facilement l'interface souhaitée avec une grande liberté, tandis que le CSS nous offre une personnalisation fine des détails (titres, boutons, etc.). Grâce à cette combinaison, nous n'avons pas eu besoin de recourir à un framework supplémentaire, qui aurait pu nous imposer certaines restrictions.

Nous avons ainsi pu définir précisément la disposition des éléments, l'organisation visuelle des informations, ainsi que les interactions utilisateur, sans dépendre de composants ou de conventions externes. De plus, le contrôle total sur le

style nous a permis de garantir une **cohérence graphique** entre les différentes parties de l'application.

Enfin, l'utilisation de HTML et CSS facilite la maintenance du code, en le rendant lisible et modifiable rapidement par tous les membres du groupe, y compris ceux ayant peu d'expérience en développement web.

Voici à quoi ressemble l'interface finale de Nexium, une fois l'application configurée :

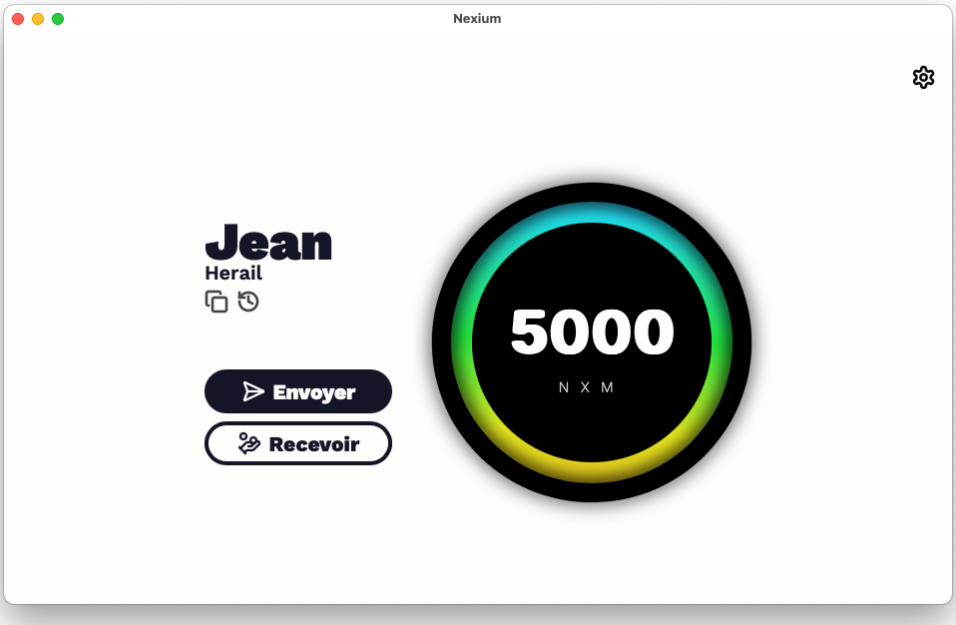

Figure 3: Application Nexium, menu principal

10.5 Composition de l'interface graphique

Pour la conception de l'**interface graphique** de notre application, nous avons utilisé le logiciel **Sketch**, une alternative à **Figma**, bien connue dans le domaine du **design numérique**. Sketch est un outil de **conception vectorielle** qui permet de créer des **interfaces utilisateur**, des **maquettes** et des **prototypes interactifs**. Il est particulièrement apprécié pour sa **simplicité d'utilisation**, ses **fonctionnalités de collaboration en temps réel** et sa capacité à produire des designs à la fois **modernes** et **fonctionnels**.

Le principe de Sketch repose sur une **interface de travail intuitive**, qui permet de concevoir des éléments graphiques à l'aide de **formes vectorielles**, de **calques** et de **styles réutilisables**. Ces éléments sont ensuite assemblés

pour constituer des **maquettes interactives** représentant les différentes vues de l'application. Le logiciel propose également une large gamme de **plugins**, ce qui permet d'étendre ses fonctionnalités et de faciliter certaines étapes du processus de conception, comme la gestion des **typographies**, des **couleurs** et des **composants réutilisables**.

Dans le cadre de notre projet, nous avons pu bénéficier d'une **licence gratuite de Sketch**, grâce à notre statut d'**étudiants**. Cette licence nous a permis de réaliser nos **prototypes** sans contrainte financière et de tirer pleinement parti des outils et des **fonctionnalités avancées** que le logiciel offre. Cela a été particulièrement utile pour la création de **maquettes interactives** et la gestion des éléments de design de manière **centralisée**.

Les **prototypages design** ont été réalisés avant même le début du développement du code. Cette approche nous a permis de définir une **ligne directrice claire** pour l'interface, assurant ainsi une **direction artistique stable** et cohérente tout au long du développement de l'application. En concevant d'abord les **maquettes**, nous avons pu anticiper les **besoins des utilisateurs**, tester différentes interfaces et ajuster les **choix graphiques** avant de nous lancer dans le codage. Ce processus a été crucial pour éviter des changements esthétiques ou fonctionnels trop tardifs, permettant une **intégration fluide** des éléments visuels dans l'application.

Ainsi, **Sketch** a joué un rôle essentiel dans le développement de l'**interface graphique** de notre projet, en nous offrant un outil puissant pour concevoir des **prototypes interactifs** et en nous permettant de maintenir une **cohérence visuelle** tout au long de l'élaboration de l'application.

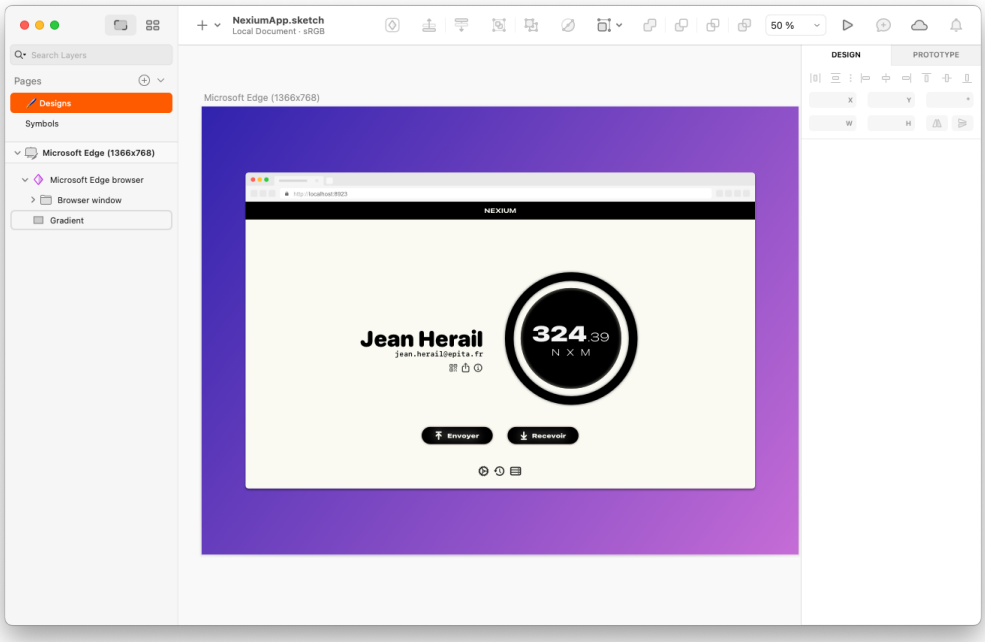

Figure 4: Premier prototype de l'application Nexium (mars 2025).

11 API Client Nexium

Située dans le fichier `nexium_api.rs` l'API Nexium est crucial pour permettre l'interaction entre l'application et le serveur Nexium, facilitant des opérations telles que la consultation de solde, l'envoi de transactions et la récupération de l'historique.

11.1 Structures de Données Clés

Plusieurs structures de données sont définies pour gérer les communications et les informations échangées avec le serveur Nexium :

- **`NexiumAPIError`** : Cette énumération robuste capture divers scénarios d'erreurs pouvant survenir lors des interactions avec l'API, allant des problèmes de réseau ou de parsing JSON à des erreurs métier spécifiques comme des fonds insuffisants ou des montants invalides. Sa gestion détaillée permet un traitement précis des échecs de requêtes.

```
pub enum NexiumAPIError {
    UnknownError,
    NoServerPublicKey,
    NoServerResponse,
    InvalidResponseFromServer,
```

```
    InvalidJsonResponse,
    NoServerLogin,
    NoServerSigSample,
    NoServerGpgKeys,
    InvalidSigSample,
    NoBalanceField,
    InvalidBalanceFormat,
    NegativeOrZeroAmount,
    InvalidTransactionAmount,
    InvalidFees,
    InsufficientFunds,
    InvalidAmount,
    BalanceFetchError,
    ReceiverNotFound,
    InvalidReceiver,
    SenderAndReceiverSame,
}
```

- **`TransactionResponse`** : Représente la structure attendue pour une réponse JSON du serveur lors de la récupération des transactions. Elle contient les informations brutes d'une transaction, telles que son en-tête (`transaction_header`) et les données associées (`data`).

```
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct TransactionResponse {
    pub transaction_header: String,
    pub data: String,
}
```

- **`ClassicTransactionReceived`** : Une structure plus conviviale pour afficher les transactions aux utilisateurs. Elle transforme les données brutes de `TransactionResponse` en un format lisible, incluant le récepteur, l'émetteur, une description, le montant, la date formatée et le type (IN/OUT).

```
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct ClassicTransactionReceived {
    pub receiver: String,
    pub emitter: String,
    pub description: String,
    pub amount: String,
    pub date: String,
    pub inorout: String,
}
```

- **`ServerKeys`** : Utilisée pour désérialiser la clé publique RSA du serveur,

essentielle pour la vérification des signatures.

```
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct ServerKeys {
    pub rsa_pub_key: String,
}
```

- **SignatureSampleResponse** : Contient l'exemple de signature de la clé GPG du serveur, utilisée pour valider l'authenticité des clés GPG récupérées.

```
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct SignatureSampleResponse {
    pub signature: String,
    pub message: String,
}
```

11.2 Fonctionnalités de l'API Client

Le module `nexium_api.rs` expose plusieurs fonctions asynchrones pour interagir avec le serveur Nexium :

- **`get_server_public_key(client: &Client, config: &Config) -> Result<String, NexiumAPIError>`** : Cette fonction est responsable de la récupération de la clé publique RSA du serveur Nexium. Cette clé est fondamentale pour des opérations de sécurité telles que la vérification des signatures des transactions ou l'établissement de communications sécurisées. Elle effectue une requête HTTP GET vers l'endpoint `/get_server_public_key` du serveur.

- **`get_balance(client: &Client, login: &String, config: &Config) -> Result<BigUint, NexiumAPIError>`** : Permet de consulter le solde (balance) d'un utilisateur donné sur la blockchain Nexium. Elle envoie une requête HTTP GET à l'endpoint `/get_balance/<login>` et parse la réponse JSON pour extraire le solde en tant que `BigUint`, gérant les erreurs de format ou de données manquantes.

- **`send_transaction(client: &Client, transaction: &Transaction, config: &Config) -> Result<(), NexiumAPIError>`** : Envoie une transaction signée au réseau Nexium. La transaction est sérialisée et envoyée via une requête HTTP POST à l'endpoint `/send_transaction`. Cette fonction est cruciale pour toutes les opérations impliquant un transfert de valeur ou de données sur la blockchain. Une vérification du statut de la réponse est effectuée pour confirmer le succès de l'envoi.

- **`get_transactions(client: &Client, login: &String, config: &Config) -> Result<Vec<ClassicTransactionReceived>, NexiumAPIError>`**

: Récupère l'historique des transactions pour un utilisateur spécifique. Elle interroge l'endpoint `/get_transactions/<login>` et analyse chaque transaction renvoyée. Pour chaque transaction, elle détermine s'il s'agit d'une transaction "IN" ou "OUT" et la convertit en une structure `ClassicTransactionReceived` pour un affichage simplifié. Seules les transactions de type `TransactionType::Classic` sont traitées.

- **`get_login_from_gpg_key(client: &Client, gpg_pub_key: &String, config: &Config) -> Result<String, NexiumAPIError>`** : Associe une clé publique GPG à un login utilisateur en interrogeant le serveur Nexium. Cette fonction est essentielle pour vérifier la correspondance entre une clé GPG et l'identité d'un utilisateur. Elle envoie la clé GPG à l'endpoint `/get_login_from_gpg_key` et attend en retour le login associé.

- **`get_signature_sample(client: &Client, config: &Config) -> Result<SignatureSampleResponse, NexiumAPIError>`** : Récupère un exemple de signature généré par la clé GPG du serveur. Cet exemple est utilisé pour valider l'authenticité de la clé GPG du serveur lors de l'initialisation ou d'autres vérifications de sécurité, garantissant que les communications proviennent bien du serveur légitime.

11.3 Gestion des Erreurs et Validation

Le module `nexium_api.rs` intègre une gestion d'erreurs approfondie via l'énumération `NexiumAPIError`. Chaque fonction de l'API renvoie un `Result` encapsulant soit la valeur attendue en cas de succès, soit une erreur `NexiumAPIError` en cas d'échec. Cela permet une identification précise de la cause de l'erreur (problèmes réseau, données JSON invalides, règles métier non respectées, etc.) et une meilleure robustesse de l'application. Des vérifications sont effectuées sur les montants des transactions (positifs, non nuls), les frais, et la disponibilité des fonds avant l'envoi des transactions.

11.4 Intégration et Dépendances

L'API client Nexium s'appuie sur plusieurs crates Rust pour son fonctionnement :

- **`reqwest::blocking::Client`** : Pour l'envoi de requêtes HTTP bloquantes au serveur Nexium.

- **`json`** : Pour le parsing et la manipulation des réponses JSON.

- **`nexium`** : La bibliothèque core de la blockchain Nexium, utilisée pour les structures de transactions (`Transaction`, `TransactionHeader`, `TransactionType`),

la cryptographie RSA, et les utilitaires liés à la blockchain.

- **`chrono::DateTime`** : Pour la gestion et le formatage des horodatages des transactions.
- **`num_bigint::BigUint`** : Pour la manipulation des clés RSA, des chiffrements et autres opérations cryptographiques.
- **`serde`** et **`serde_json`** : Pour la sérialisation et la désérialisation des structures de données en JSON.

L'API client Nexium fournit une interface complète et sécurisée pour interagir avec le serveur Nexium, encapsulant la logique de communication réseau et la gestion des données blockchain.

12 Serveur

12.1 Serveur HTTP

Le serveur se base sur la structure `TcpListener` de la librairie standard. Il constitue le **cœur** même du serveur, puisqu'il écoute les requêtes HTTP entrantes et ouvre les connexions sous forme de *stream*.

Afin de nous aider à interpréter et répondre aux requêtes, nous avons implémenté plusieurs structures dans le but de faciliter la suite du développement. La première que nous allons aborder est la structure `Request`, qui joue un rôle crucial dans la compréhension de la requête.

Une fois ouverte, le *stream* est envoyé à `Request` qui, dans un premier temps, va lire le contenu de la requête. Pour des raisons de sécurité, nous avons fixé la taille maximale des requêtes à **32 kilo-octets**. Au-delà de cette limite, la requête est directement rejetée et la connexion fermée.

Une fois le contenu récupéré, vient alors la phase de **déstructuration** (ou *parsing*). Le contenu est analysé pour extraire :

- la **méthode** (GET, POST, ...),
- le **chemin** ou `path` (`/nexium`, `/balance`),

ces deux éléments étant les plus importants puisqu'ils influencent directement le comportement du serveur et sa réponse.

Ensuite, sont extraits des paramètres optionnels :

- les **paramètres d'URL**, appelés `query` ou `search` (après le caractère `?`),
- les **headers**, sous forme de paires clé/valeur,
- et enfin le **contenu** (ou `body`).

Voici un exemple de requête reçue par le serveur :

```
GET / HTTP/1.1
Host: 127.0.0.1:8080
```

Une fois entièrement décortiquée, le serveur peut alors correctement prendre en charge la requête, en fonction de la **méthode** et du **chemin**.

Le traitement de la requête terminé, la dernière étape concerne la **réponse**. Pour cela, nous avons également créé une structure `Response`, dont le but est de construire la réponse HTTP de manière simple.

Un élément central de cette structure est le **statut**, représenté sous la forme d'une paire `code/texte`, indiquant principalement l'issue de la requête. Il existe une multitude de statuts, parmi lesquels :

- `200 OK`
- `400 Bad Request`
- `404 Not Found`

La réponse embarque également des éléments optionnels, comme les `headers` et le `body`, similaires à ceux déjà évoqués pour la requête.

Dans notre implémentation, le serveur renvoie :

- `404` si l'*endpoint* (que nous détaillerons par la suite) n'est pas reconnu,
- `400` si la requête ne contient pas les informations nécessaires ou des données valides,
- `500` (*Internal Server Error*) en cas d'erreur inattendue,
- et `200` si tout s'est bien passé.

Voici un exemple de réponse émise par le serveur :

```
HTTP/1.1 200 OK
Content-Type: text/plain

Hello world!
```

12.2 API Nexium

L'API Nexium est l'ensemble des **endpoints** que le serveur expose pour permettre aux clients de communiquer avec lui. Chaque endpoint correspond à une fonctionnalité spécifique, comme la récupération du solde d'un utilisateur, l'envoi d'une transaction, ou la vérification de l'identité du serveur.

12.2.1 `/nexium`

Permet de vérifier que le serveur concerné est bien un serveur Nexium, et de récupérer les informations de ce serveur.

- **Type** : `GET`
- **Endpoint** : `/nexium`

Réponse :

```
{
  "login": <login>,
  "sigSample": <sigsample>
}
```

- `login` : Identifiant de l'utilisateur à qui appartient le serveur.
- `sigSample` : Permet au client d'identifier quelle clé publique est utilisée par le serveur.

12.2.2 `/balance`

Renvoie le solde actuel sur un utilisateur.

- **Type** : `GET`
- **Endpoint** : `/balance/<login>`

Réponse :

```
{
  "balance": <balance>,
  "noise": <noise>
}
```

- `noise` : Chaîne de caractères de longueur 8, générée aléatoirement. Sert à empêcher les attaques par brute-force.

12.2.3 `/transactions`

Récupère les `n` dernières transactions d'un utilisateur.

- **Type** : `GET`
- **Endpoint** : `/transactions/<login>?n=<n>`
- **Paramètre** : `n` (optionnel, par défaut 3) : Nombre de transactions à récupérer.

Réponse :

```
{
  "transactions": [
    <transaction_1>,
    <...>,
    <transaction_n>
  ]
}
```

12.2.4 `/new_transaction`

Publie une nouvelle transaction pour l'ajout à la blockchain.

- **Type** : `POST`
- **Endpoint** : `/new_transaction`

Body :

```
{
  <transaction>
}
```

12.3 Header

Structure des headers de requêtes Nexium.

Toutes les réponses sont renvoyées sous forme de **JSON chiffré**, selon la logique suivante :

```
réponse = chiffrement(json(données))
```

- Le JSON est chiffré par le serveur avec la **clé publique du client**, obtenue via le login dans le header.
- Le contenu des requêtes `POST` est chiffré par le client avec la **clé publique du serveur**.

Header standard :

```
Login: <login>
Sig-Sample: <sigsample>
```

- `Sig-Sample` est la signature de la chaîne `"NEXIUMREQ"` faite avec la clé privée du client.
- Elle permet au serveur de retrouver la bonne clé publique, si plusieurs sont présentes sur le compte GitLab du login.

- Elle évite les requêtes non authentifiées ou de type *spam*.

12.4 Structure d'une transaction

Une transaction est envoyée au format **JSON** :

```
{
    "transaction_header": {
        "transaction_size": <size>,
        "timestamp": <timestamp>,
        "fees": <fees>,
        "emitter": <login>,
        "data_type": <data_type>
    },
    "data": <data>,
    "signature": <signature>
}
```

Champs :

- `transaction_size` : Taille du champ `data` (en octets)
- `timestamp` : Horodatage UNIX de la transaction
- `fees` : Frais de transaction (en µNEX/octet)
- `emitter` : Login de l'émetteur (au format `prenom.nom`)
- `data_type` : Type de données contenues dans `data`
- `signature` : Signature du couple `transaction_header + data`

13 Site Web

Astro permet une approche simple et rapide pour la création d'une **documentation web**. Utilisant **Vite**, **Astro** offre une **rapidité de développement** accrue avec le **SSR** (Server Side Rendering), permettant une **actualisation dynamique des pages** à chaque modification.

Prenant en charge **Markdown**, il nous est possible d'écrire la documentation en respectant la syntaxe **Markdown**, puis de laisser **Astro** faire le **build**, générant les fichiers **html**, **css** et **js** nécessaires pour les **navigateurs web**. L'avantage de cette approche est que toute l'équipe peut contribuer à la documentation sans avoir besoin de connaissances approfondies en développement web, en se concentrant sur le contenu plutôt que sur la structure technique.

Nous avons choisi d'héberger le site sur **Gitlab**, ce qui permet une **inté-**

gration continue simple et **gratuite**. Le site est accessible publiquement à l'adresse suivante :

Cette séparation avec le dépôt principal de **Nexium** nous permet :

- De maintenir une **organisation claire** entre le **code de Nexium** et la **documentation**.
- De gérer indépendamment les **mises à jour documentaires** sans impacter le développement du **logiciel**.

Un pipeline **GitLab** est configuré pour **déployer automatiquement le site** à chaque mise à jour de la **branche principale** (`main`). Cela permet des mises à jour rapides et efficaces de la documentation, sans nécessiter d'intervention manuelle pour le déploiement.

```
---
title: Qu'est-ce que Nexium ?
description: Presentation du projet Nexium
order: 1
---

Nexium est une crypto-monnaie decentralisee, ...
```

web/contents/blockchain.md

14 Organisation du projet

L'idée originelle de **Nexium** a suscité chez chacun des membres de notre groupe un **intérêt** et **enthousiasme** laissant présager un bon processus de développement sur le projet. Afin de pouvoir concrétiser le plus solidement et durablement possible le projet, nous avons accordé une place importante à l'**organisation du travail de groupe**.

14.1 Outils utilisés et mis en place

Comme mentionné précédemment, une **organisation claire** et **structurée** a permis une fluidification de la **communication**. Ainsi, nous avons mis en place plusieurs solutions pour tenir un **suivi régulier** de l'avancement du projet.

14.1.1 Discord

Nous avons utilisé un serveur **Discord** pour **communiquer**, relever des **problèmes**, planifier des **réunions** et partager des **ressources**. Plusieurs **salons textuels** ont été dédiés aux différentes parties du développement, ainsi

qu'à la centralisation des différents **liens**. Ce serveur **Discord** a été, en outre, la **pierre angulaire** de notre **communication**, la centralisation de nos **échanges** et **ressources**.

14.1.2 Gitlab

Gitlab est une solution d'hébergement de **repositories Git**. Nous avons choisi d'utiliser l'instance **Gitlab** hébergée par **Épita** pour **Nexium**. Cela nous permet une meilleure gestion des **collaborateurs** (avec les **logins Épita**), et une congruence avec le projet **Nexium**, puisque c'est de cette instance **Gitlab** que nous avons tiré l'idée de récupération des **clés publiques GPG**.

De plus, nous avons choisi d'utiliser **Gitlab**, qui propose un outil d'hébergement de **pages web** (**GitLab Pages**), que nous utilisons pour automatiquement déployer notre **site web**, sans avoir à gérer de **serveur web** ou de dépendre d'un prestataire externe.

14.2 Répartition des tâches

Au-delà des **réunions hebdomadaires** planifiées lors de tous les **week-ends**, notre équipe a pris la décision, pour la première phase de développement de **Nexium**, de se séparer en **deux groupes distincts**, chacun ayant une **mission bien précise**.

- **Architecture** : Ce groupe était composé de **Jean** et **William**. Ils étaient responsables de l'**architecture de la blockchain** et du **réseau**, ainsi que du **développement du serveur web**. Ils étaient également responsables de la **rédaction de la documentation théorique de Nexium**.

- **Cryptographie** : Ce groupe est composé de **Milo** et **Antonin**. Ils étaient en charge de la partie **mathématique** du projet, c'est-à-dire le **chiffrement/déchiffrement** et les **calculs nécessaires** pour les **changements de type**. Ils ont aussi la charge du **SHA-256**.

Pour la seconde phase du projet, la répartition a évolué selon les compétences de chacun :

- **Client** : Cette tâche a été menée par **Jean** et **Milo**, à l'aide du framework **Tauri** et des technologies web (**HTML/CSS**, **Svelte**, **Tailwind**). Ils ont conçu l'**interface utilisateur**, intégré les **fonctionnalités côté client**, développé l'**API de Gitlab**, et implémenté une **API Client** pour **Nexium**.

- **Blockchain** : **William**, avec l'aide de **Jean**, a établi les **structures fondamentales de la blockchain** : **transactions**, **blocs**, **headers**,

autant de **données** qu'il est crucial de pouvoir stocker et convertir dans différents formats (en **binaire** pour le stockage sur serveur, et en **JSON** avec **sérialisation/désérialisation** pour les échanges réseau).

- **Serveur** : **William** a pris en charge la construction du **serveur HTTP**, la gestion des **endpoints de l'API**, l'implémentation de la **blockchain** et l'**architecture globale du backend**.

- **Site Web** : **Antonin** a été responsable de la **conception** et du **développement du site web documentaire** de Nexium. **Astro** et **Markdown** ont été utilisés pour structurer et générer un **site statique** hébergé sur **GitHub Pages**, permettant de documenter et valoriser le projet.

Voici la répartition des tâches finales pour Nexium :

	Jean	***William***	***Milo***	***Antonin***
Serveur		X		
Blockchain	X	X		
API	X	X	X	
Application	X		X	
Chiffrement	X		X	
Site Internet	X	X	X	X

15 Conclusion

Ce projet nous a permis d'**approfondir nos connaissances** tant **théoriques** que **pratiques** dans un **cadre concret et collaboratif**. En confrontant nos idées, en surmontant les **imprévus techniques** et en affinant progressivement nos **méthodes de travail**, nous avons acquis une **meilleure compréhension** du domaine traité ainsi que des **compétences transversales** précieuses telles que la **gestion de projet**, la **communication en équipe** et la **rigueur dans le développement**.

Les **objectifs fixés** au départ ont été, dans l'ensemble, **atteints**. Les **résultats obtenus** témoignent de notre **investissement** et de notre **capacité à mener à bien un projet de bout en bout**, depuis la phase d'**analyse** jusqu'à la **mise en œuvre** et la **validation finale**.

Ce travail ouvre également la voie à de nombreuses **perspectives d'amélioration et de prolongement**. Il pourrait notamment être enrichi par l'**intégration de fonctionnalités supplémentaires**, une **optimisation des performances**, ou encore une **évaluation plus poussée** dans un **contexte réel**.

Nous tirons de cette expérience un **bilan très positif**, à la fois sur le plan **technique** et **humain**.

Bibliographie

[1] AdaCore Blog. ***Memory Safety in Rust***. Jan. 2024.

[2] Anthropic. ***GitHub Actions***.

[3] Atlassian. ***What is Continuous Integration***.

[4] Auth0. ***Authorization Code Flow with Proof Key for Code Exchange (PKCE)***.

[5] Matt Blaze. "**Protocol Failure in the Escrowed Encryption Standard**". In: *Proceedings of the 2nd ACM Conference on Computer and Communications Security*. ACM, Aug. 1994, pp. 59–67.

[6] Blockchain Council. ***What are Peer-to-Peer Transactions?*** May 2024.

[7] T. Bray. ***The JavaScript Object Notation (JSON) Data Interchange Format***. Dec. 2017.

[8] J. Callas et al. ***OpenPGP Message Format***. Nov. 2007.

[9] David Chaum. "**Security without Identification: Transaction Systems to Make Big Brother Obsolete**". In: *Communications of the ACM* 28.10 (1985), pp. 1030–1044.

[10] Cloudflare. ***Astro • Cloudflare Pages docs***. May 2025.

[11] Contensis. ***What is server-side rendering (SSR)***. Dec. 2023.

[12] Douglas Crockford. ***JSON***.

[13] Joan Daemen and Vincent Rijmen. ***AES Proposal: Rijndael (Version 2)***. Originally submitted to NIST as part of the AES competition. Sept. 1999.

[14] Ecma International. ***ECMAScript® 2024 Language Specification***. Ecma International. June 2024.

[15] ETSI. ***Abstract Syntax Notation One (ASN.1)***.

[16] R. Fielding et al. ***Hypertext Transfer Protocol – HTTP/1.1***. 1999.

[17] GitHub. ***documentation • GitHub Topics***.

[18] GitLab. ***Documentation | The GitLab Handbook***.

[19] Google. ***Markdown style guide***.

[20] D. Hardt. ***The OAuth 2.0 Authorization Framework: Bearer Token Usage***. Oct. 2012.

[21] IBM. ***What Is an API (Application Programming Interface)?***

[22] Internet Assigned Numbers Authority (IANA). ***Hypertext Transfer Protocol (HTTP) Status Code Registry***. Nov. 2024.

[23] ITU-T. ***Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation***. Feb. 2021.

[24] Khan Academy. ***Binary data (video) | Bits and bytes***.

[25] Ralph C. Merkle. "**A Digital Signature Based on a Conventional Encryption Function**". In: *Advances in Cryptology – CRYPTO '87*. Springer, Berlin, Heidelberg, 1988, pp. 369–378.

[26] Mozilla Contributors. ***An overview of HTTP***. Mar. 2025.

[27] Satoshi Nakamoto. ***Bitcoin: A Peer-to-Peer Electronic Cash System***. https://bitcoin.org/bitcoin.pdf. 2008.

[28] National Institute of Standards and Technology. ***Advanced Encryption Standard (AES)***. Tech. rep. FIPS 197. Updated May 9, 2023. National Institute of Standards and Technology, Nov. 2001.

[29] National Institute of Standards and Technology. ***Recommendation for Block Cipher Modes of Operation Methods and Techniques***. Tech. rep. NIST SP 800-38A. National Institute of Standards and Technology, Dec. 2001.
[30] National Institute of Standards and Technology. ***Secure Hash Standard (SHS)***. Tech. rep. FIPS 180-4. National Institute of Standards and Technology, Aug. 2015.
[31] Scott Nevil. ***What Is Proof of Work (PoW) in Blockchain?*** May 2024.
[32] OSL. ***Bitcoin Mempool: What Happens to Unconfirmed Transactions?*** Mar. 2025.
[33] Ray's Notebook. ***GnuPG S2K***.
[34] Ling Ren. "**Analysis of Nakamoto Consensus**". In: *Cryptology ePrint Archive* (2019).
[35] R.L. Rivest, A. Shamir, and L. Adleman. "**A Method for Obtaining Digital Signatures and Public-Key Cryptosystems**". In: *Communication ACM* 21.2 (1978), pp. 120–126.
[36] N. Sakimura et al. ***Proof Key for Code Exchange by OAuth Public Clients***. Sept. 2015.
[37] Sketch. ***Sketch • Design, prototype, collaborate and handoff***.
[38] Stephanie Susnjara and Ian Smalley. ***What Is Blockchain?*** Apr. 2025.
[39] Tailwind Labs. ***UI Blocks Documentation - Tailwind Plus***.
[40] Carla Tardi. ***Genesis Block: Bitcoin Definition, Mysteries, and Secret Message***. May 2024.
[41] tauri-apps. ***tauri-docs: The source for all Tauri project documentation.***
[42] Teleport. ***UNIX Timestamp to Date Converter | Instant UNIX to Date Conversion***.
[43] The "Notes on OpenPGP" project. ***5. Managing private key material in OpenPGP — OpenPGP for application developers***. 2024.
[44] The Investopedia Team. ***Nonce: What It Means and How It's Used in Blockchain***. May 2024.
[45] The Investopedia Team. ***What Is Cryptocurrency Difficulty? Definition and Bitcoin Example***. July 2024.
[46] The Rust Community. ***chrono - Rust - Docs.rs***.
[47] The Rust Community. ***num_bigint - Rust - Docs.rs***.
[48] The Rust Community. ***num_prime - Rust - Docs.rs***.
[49] The Rust Community. ***reqwest - Rust - Docs.rs***.
[50] The Rust Community. ***serde - Docs.rs***.
[51] The Rust Community. ***serde_json - crates.io: Rust Package Registry***.
[52] The Rust Project. ***Documentation - The Rust Programming Language***. 2018.
[53] The Rust Project. ***TcpListener in std::net - Rust Documentation***.
[54] The Svelte Team. ***@sveltejs/kit • Docs • Svelte***.
[55] The Svelte Team. ***Getting started • Docs - Svelte***.
[56] Paul Tierno. ***Introduction to Cryptocurrency***. Apr. 2025.
[57] Vercel. ***Vite on Vercel***.
[58] W3C. ***Understanding the CSS Specifications***. Feb. 2025.
[59] Wikipedia contributors. ***Cypherpunk***. May 2025.

[60] Wikipedia contributors. ***ECMAScript***. May 2025.
[61] Wikipedia contributors. ***Endianness***. May 2025.
[62] Wikipedia contributors. ***Merkle tree***. May 2025.
[63] World Wide Web Consortium (W3C). ***Web Standards***.