|  |
| --- |
| Tomé Rúben  23/05/2025 |

|  |
| --- |
| 3IND-3TPMa |
| CryptoTrak Rapport projet TPI |
| Application de suivi et gestion d’investissements en cryptomonnaies avec sécurité avancée |



Table des matières

[Introduction 3](#_Toc198538141)

[Matériels et technologies 4](#_Toc198538142)

[Matériel 4](#_Toc198538143)

[Technologies 4](#_Toc198538144)

[Conception 5](#_Toc198538145)

[Planification 5](#_Toc198538146)

[Journal de travail 5](#_Toc198538147)

[Architecture du site 6](#_Toc198538148)

[Identification des processus métier 8](#_Toc198538149)

[Documentation des processus métier 10](#_Toc198538150)

[Base de données 11](#_Toc198538151)

[MCD 11](#_Toc198538152)

[MLD 12](#_Toc198538153)

[Réalisation 13](#_Toc198538154)

[Arborescence du projet 13](#_Toc198538155)

[Serveur (Backend) 14](#_Toc198538156)

[API REST 15](#_Toc198538157)

[Sécurité 16](#_Toc198538158)

[Gestion de l’authentification (classique) 16](#_Toc198538159)

[Gestion de l’authentification (2FA, WebAuthn) 19](#_Toc198538160)

[Sécurisation de l’API REST 20](#_Toc198538161)

[Protection contre les attaques 21](#_Toc198538162)

[Gestion des erreurs 21](#_Toc198538163)

[Implémentation API CoinGecko 22](#_Toc198538164)

[Frontend 23](#_Toc198538165)

[Protocole de test 23](#_Toc198538166)

[Problèmes rencontrés et solutions 23](#_Toc198538167)

[Améliorations et perspectives 23](#_Toc198538168)

[Conclusion 23](#_Toc198538169)

[Bibliographie 23](#_Toc198538170)

[Annexes 24](#_Toc198538171)

[Documentation technique 24](#_Toc198538172)

[Guide d’utilisation 25](#_Toc198538173)

# Introduction

Ce projet s'inscrit dans le cadre de mon TPI lors duquel il a fallu réaliser un projet informatique sous 110 périodes. Ce projet a été encadré par M. Vadi, qui a été consulté à plusieurs reprises lorsque cela s’est avéré nécessaire afin d’obtenir diverses orientations. Le but du TPI est de réaliser de manière autonome un projet pratique encadré, en appliquant ses compétences techniques tout en documentant rigoureusement son processus et ses résultats afin de démontrer sa maîtrise professionnelle.

Mon projet consistait en la création d’une application web de suivi et gestion d’investissements en cryptomonnaies avec sécurité avancé. Il convient de préciser que ce projet a initialement été amorcé dans le cadre du Pré-TPI, où une première version de l’application avait été développée. Cette version proposait des fonctionnalités de base telles que l’affichage des cours des cryptomonnaies sous forme de graphiques interactifs, ainsi qu’un outil de conversion de devises. Par la suite, le projet m’a été proposé comme sujet de TPI, avec un cahier des charges plus complet. Plusieurs objectifs ont été fixés pour la réalisation de ce projet. L’application devait intégrer un ensemble d’objectifs qui figuraient dans le cahier des charges. Parmi les fonctionnalités attendues figuraient notamment la création de comptes utilisateurs, la mise en place d’une authentification renforcée via WebAuthn, la possibilité d’effectuer des dépôts et retraits d’argent fictif, ainsi que l’achat et le suivi de cryptomonnaies. L’application devait également offrir une interface de visualisation des investissements à l’aide de graphiques, permettre l’ajout de cryptomonnaies en favoris, proposer une gestion des paramètres du compte utilisateur, et inclure une option d’exportation des graphiques au format PNG. Pour finir, des exigences spécifiques portaient sur la sécurité de l’application, la qualité du code (par l'écriture de tests unitaires), l'intégration d'une documentation technique complète et la réalisation de tests fonctionnels pour valider le bon fonctionnement des différentes composantes.

Ce rapport retrace l’ensemble des étapes de réalisation du projet, de l’analyse du cahier des charges et de la documentation de ce projet.

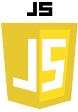
# Matériels et technologies

## Matériel

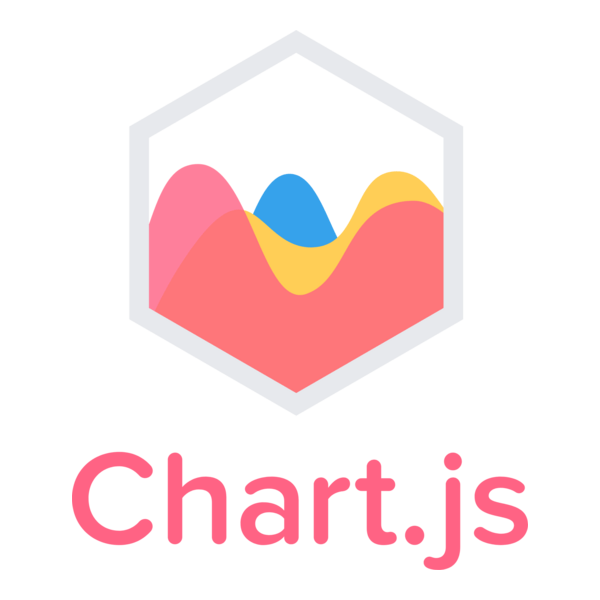
* Ordinateur sous Windows 10 (Clavier, souris, écran, etc…)

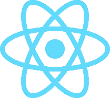
## Technologies

**Environnement de développement :**

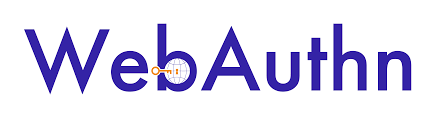
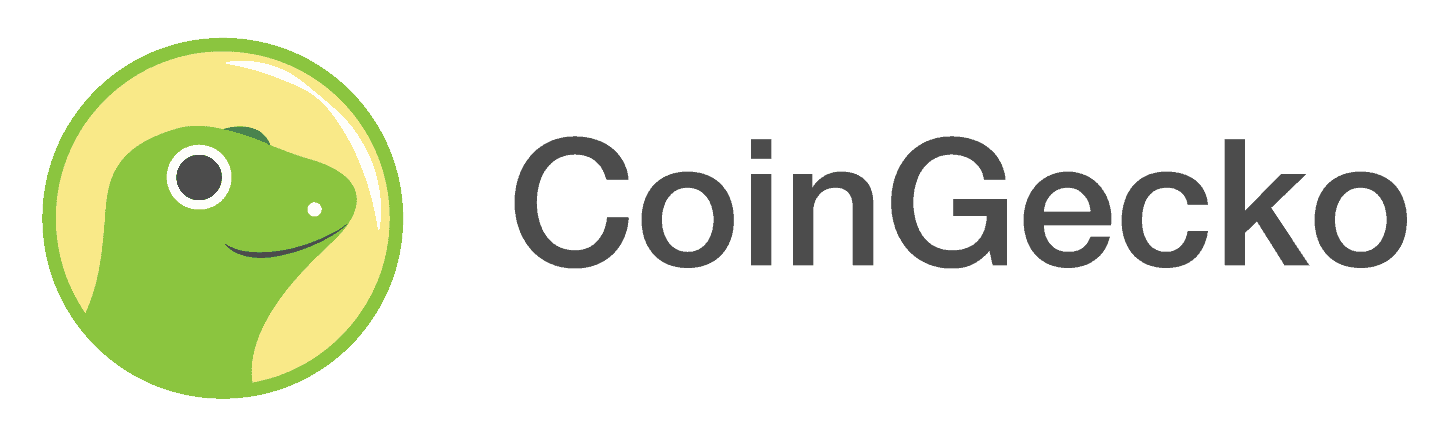


* Langage : JavaScript, HTML, CSS
* IDE : Microsoft Visual Studio Code

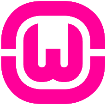
**Front-end :**

* React.js
* Chart.js (Graphique)

**Back-end :**

* Node.js
* Express.js (server)
* WebAuthn
* API CoinGecko

**Base de données :**

* MySQL 8.3.0
* Wampserver
* Looping (Conception)

**Documentation :**

* JSDoc (Documentation technique)
* Word (Rapport)

**Gestion de projet :**

* Excel
* GitHub

# Conception

## Planification

Ci-dessus, une capture d'écran présente l'ensemble des tâches de mon planning. J’ai organisé les tâches en commençant par les tâches de conception de l’application (planification, base de données…), ensuite toute la partie développement et implémentation des fonctionnalités décrites dans le cahier des charges, et pour finir tout ce qui concerne les tests et la documentation. Même si le rapport et les tests unitaires, je les ai rédigés tout au long du projet. En raison de la taille limitée d'affichage, il n'est pas possible de visualiser l'intégralité du diagramme de Gantt dans ce rapport. Pour consulter le planning complet et détaillé, veuillez-vous référer au fichier Excel joint.

## Journal de travail

## Utilisateur - Icônes social gratuitesBase de données - Icônes la technologie gratuitesApi - Icônes ordinateur gratuitesArchitecture du site

Base de données

Interface utilisateur

React.js, Chart.js

Node.js, Express.js, WebAuthn

MySQL

Serveur

CoinGecko

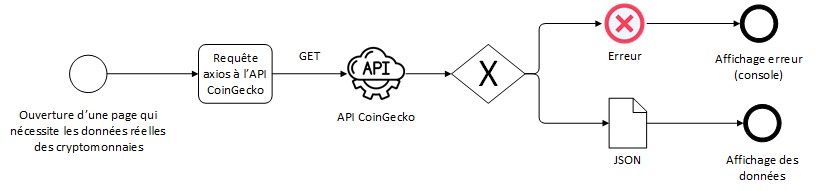
Le schéma ci-dessus illustre l’architecture globale de l’application. Il met en évidence les principales entités impliquées ainsi que leurs interactions.

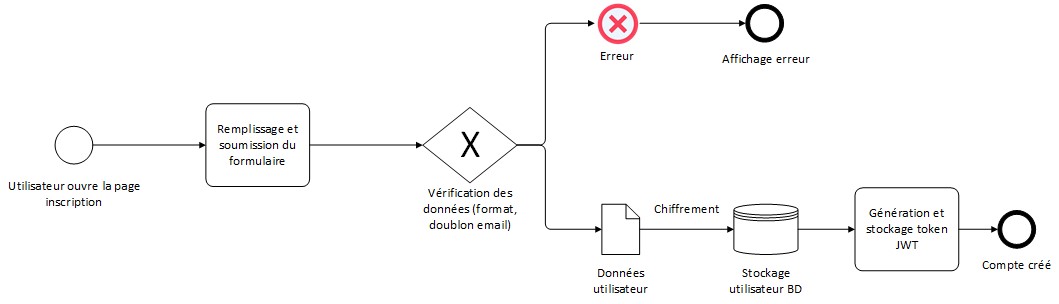
* **Interface utilisateur (front-end)** : L’utilisateur interagit avec l’application web via une interface créé à l’aide de React.js ainsi que Chart.js. Les requêtes de l’utilisateur sont transmises au serveur principal.
* **Serveur (back-end)** : Il constitue le cœur de l’application, il est développé avec Express.js. Le serveur gère l’authentification des utilisateurs (par mot de passe et passkey WebAuthn), l’interaction avec la base de données MySQL, l’exposition d’une API REST sécurisé qui permet de faire des requêtes à la base de données. La gestion des sessions y est aussi gérée en appliquant des mesures de sécurité via JWT, bcrypt et CORS.
* **Base de données**: Le serveur interagit avec la base de données pour lire et écrire les informations nécessaires (stockage utilisateur, transactions etc…)
* **API**: Le serveur communique également avec l’API CoinGecko qui permet de récupérer les données des cryptomonnaies.

## Identification des processus métier

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N° | Nom du processus | Événement déclencheur | Résultat attendu | Déclencheur du processus | Destinataire du résultat |
| 1 | Affichage des données réelles des cryptomonnaies | Ouverture d’une page nécessitant l’affichage des prix et données. | Données des cryptomonnaies récupérées et affichées en temps réel. | Ouverture de la page affichage données | Utilisateur |
| 2 | Création de compte utilisateur | Utilisateur remplit et soumet le formulaire d’inscription | Compte crée avec données chiffrées en base de données, token JWT stocké dans le localStorage ou message d’erreur si échec. | Soumission formulaire inscription (clic « Register ») | Utilisateur et système |
| 3 | Authentification (email/mot de passe) | Tentative de connexion avec identifiants (clic « Login ») | Utilisateur connecté, token JWT stocké ou message d’erreur | Soumission formulaire de connexion | Utilisateur et système |
| 4 | Activation 2FA (Yubikey, WebAuthn) | Demande d’activation 2FA depuis paramètres (clic « Register passkey ») | Clé 2FA enregistrée en base + message de confirmation | Clic sur « Register passkey » + validation physique | Utilisateur et système |
| 5 | Authentification (Yubikey, WebAuthn) | Connexion avec identifiants + 2FA activé | Utilisateur connecté, clé vérifiée, token JWT stocké ou message d’erreur | Soumission formulaire + validation 2FA | Utilisateur et système |
| 6 | Dépôt / retrait de fonds fictifs | Ajout / retrait de fonds au portefeuille en cliquant sur « Deposit », « Withdrawal ». | Montant ajouté ou retiré en base, solde portefeuille mis à jour | Utilisateur rempli le montant, la devise et le choix de déposer / retirer. | Utilisateur et système |
| 7 | Achat de cryptomonnaies | Utilisateur soumet un ordre d’achat | Ordre validé, portefeuille mis à jour, confirmation affichée | Soumission formulaire ordre (clic « Buy ») | Utilisateur et système |
| 8 | Échange (swap) de cryptomonnaies en monnaies fiat. | Utilisateur soumet un ordre d’échange | Échange validé, portefeuille mis à jour, confirmation affiché | Utilisateur soumet le montant de la cryptomonnaies et la devise d’échange | Utilisateur et système |
| 9 | Suivis des investissements | Ouverture de la page « inve |  |  |  |

## Documentation des processus métier

Processus N°1 :

Processus N°2 :

## Base de données

### C:\Users\TomeR1\AppData\Local\Packages\Microsoft.Windows.Photos_8wekyb3d8bbwe\TempState\ShareServiceTempFolder\MCD.jpegMCD

Voici le MCD (Modèle Conceptuelle de Données) de ma base de données. J’ai 7 entités principales, elles sont les suivantes :

|  |  |  |
| --- | --- | --- |
| Nom | Explication | Attributs |
| Watchlist | Représente une liste de cryptomonnaies ajoutées aux favoris par l’utilisateur pour un suivi rapide. | watchlist\_id, name, symbol, user\_id |
| User | Représente un utilisateur de l’application, qui possède un compte et peut interagir avec le portefeuille, les transactions et les favoris. | user\_id, username, email, password, created\_at |
| Passkey | Contient les données d’authentification WebAuthn (Yubikey) associées à un utilisateur pour sécuriser l’accès au compte. | passkey\_id, credential\_id, public\_key, counter, device\_type, backed\_up, transports, user\_id, created\_at, last\_used\_at |
| Wallet | Représente un portefeuille appartenant à un utilisateur, permettant de stocker différents soldes d’actifs. | wallet\_id, name, created\_at, user\_id |
| Currency | Contient la liste des cryptomonnaies ou monnaies disponibles dans l’application avec leurs informations de base. | currency\_id, currency\_name, currency\_code, currency\_flag\_image\_url, currency\_type |
| Balance | Représente un solde d’actif spécifique (crypto ou fiat) détenu dans un portefeuille donné. | balance\_id, asset\_type, name, symbol, amount, wallet\_id |
| Transaction | Représente une opération effectuée par l’utilisateur (achat, vente, dépôt ou échange d’actifs). | transaction\_id, type\_transaction, asset\_input, amount\_input, asset\_output, amount\_output, price\_at\_transaction, price\_at\_transaction\_currency, date\_at\_transaction, status, balance\_id |

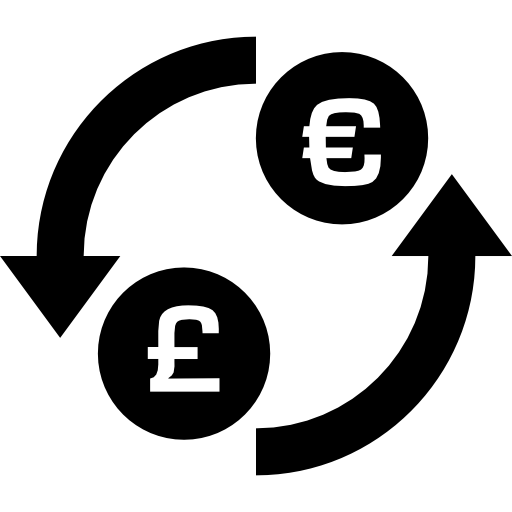
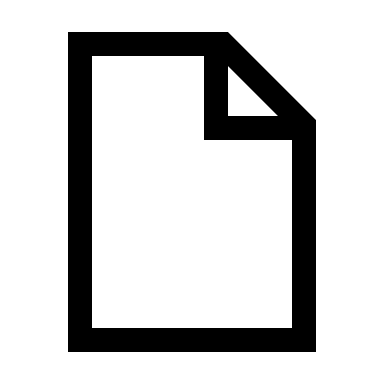
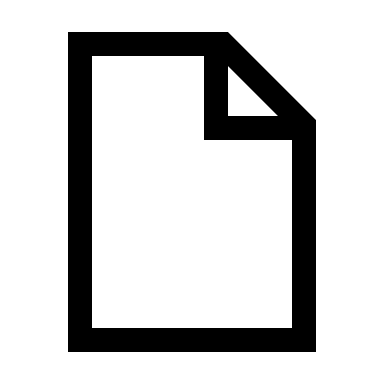
Chaque utilisateur peut avoir 0 ou 1 liste de favoris, mais une liste de favoris ne peut appartenir qu’à un seul utilisateur. Un utilisateur peut aussi avoir 0 ou plusieurs portefeuilles, mais dans mon application il en a un créé d’office et il ne peut pas en créer. Et un portefeuille lui peut contenir plusieurs soldes (balances), chacun correspondant à un actif (crypto ou fiat). Un solde appartient à un seul portefeuille. Chaque solde peut être associé à plusieurs transactions (par exemple : achats, ventes, dépôts). Une transaction est toujours liée à un seul solde. Un utilisateur peut avoir plusieurs clés d’authentification (passkeys), par exemple plusieurs Yubikey ou dispositifs WebAuthn. Cependant dans mon application l’utilisateur ne peut qu’enregistrer une seul passkey. Chaque passkey appartient à un seul utilisateur. La table des devises (Currency) est indépendante et contient uniquement les devises compatibles avec l’API CoinGecko. Les données sur les cryptomonnaies récupérées via l’API ne peuvent être obtenues que dans certaines devises spécifiques. Ces devises autorisées sont donc stockées dans la base de données pour garantir que seules les requêtes valides puissent être effectuées.

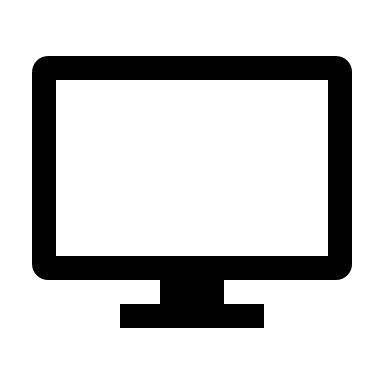
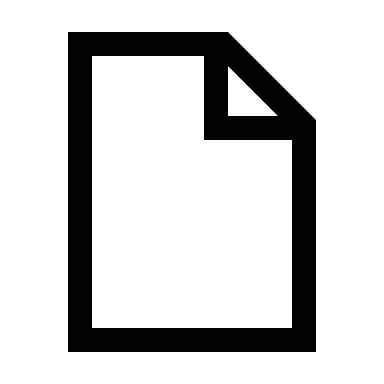
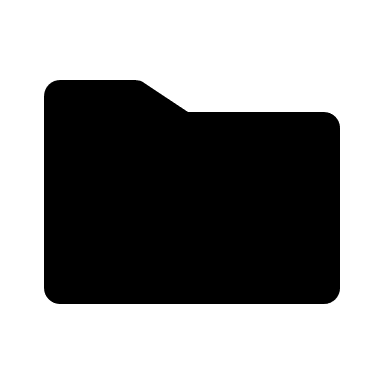
### C:\Users\TomeR1\AppData\Local\Packages\Microsoft.Windows.Photos_8wekyb3d8bbwe\TempState\ShareServiceTempFolder\MLD.jpegMLD

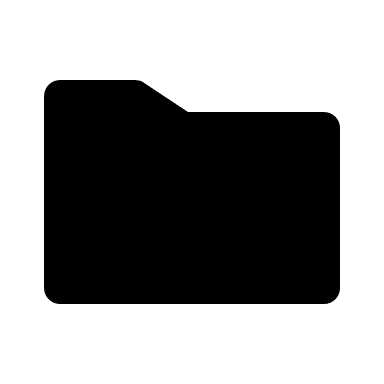
Voici mon MLD (Modèle Logique de Données) de ma base de données qui est différent du MCD et qui permet de visualiser les clés étrangères à avoir.

# Réalisation

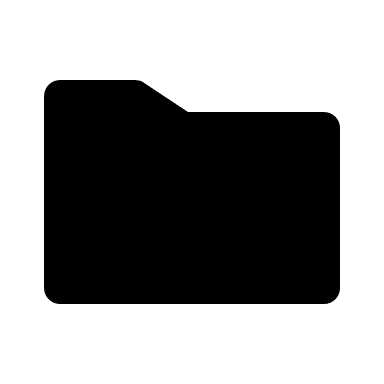
## Dossier avec un remplissage uniArborescence du projet



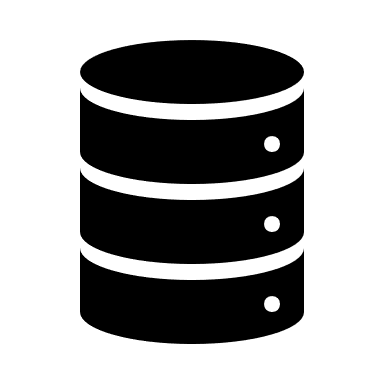
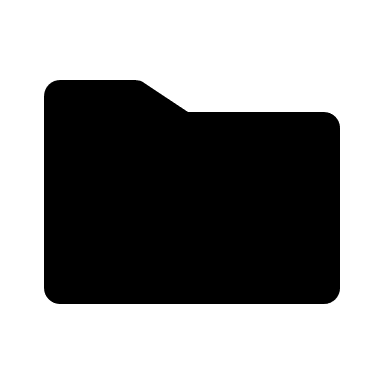


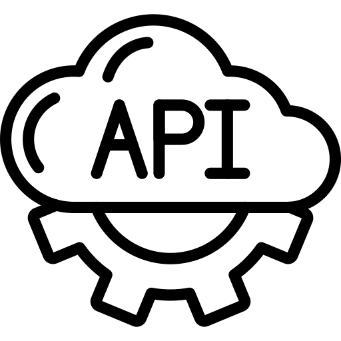


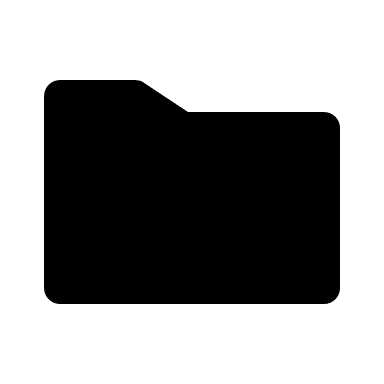
CryptoTrak



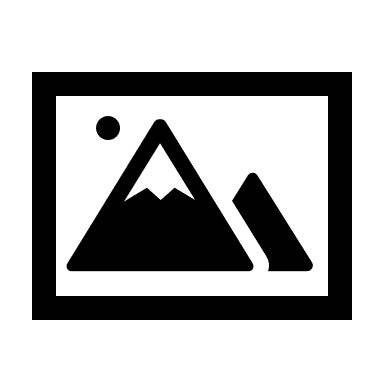
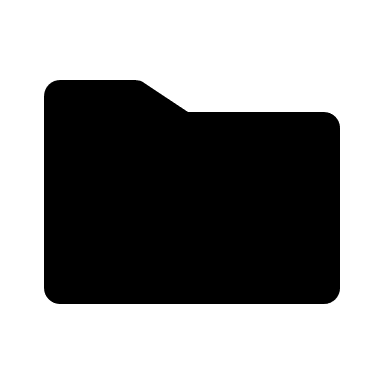
/src

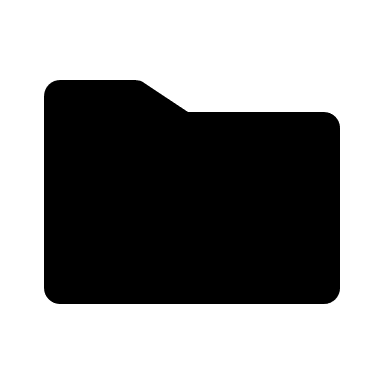


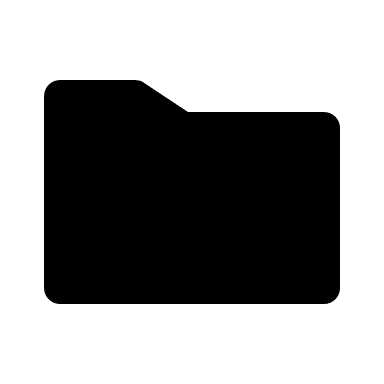




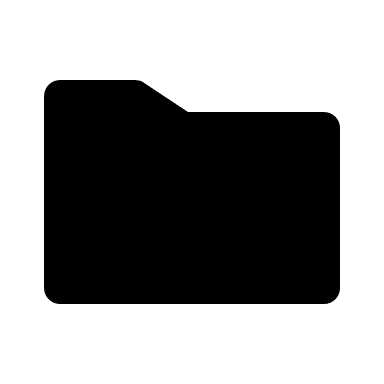
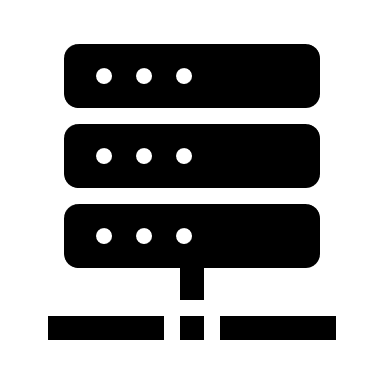
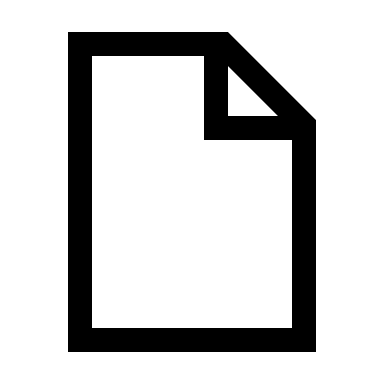
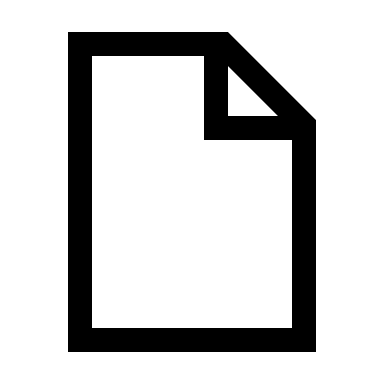
/public



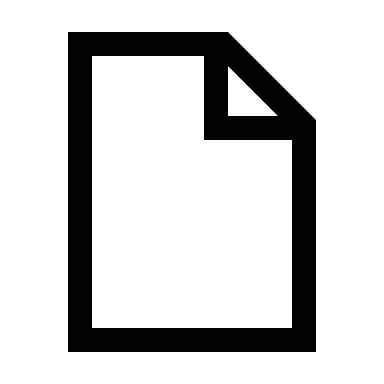




/server



/tests



Pour assurer une meilleure lisibilité et maintenabilité. J’ai organisé l’ensembles des fichiers et dossiers du projet selon une structure simple en 3 grandes parties principales. Cette organisation m’a extrêmement facilité le développement puisque chaque petite tâche ou fonction du processus est divisé.

1. **/src**

Ce dossier contient l’ensemble du code qui s’exécute côté client, c’est-à-dire l’interface visible par l’utilisateur. Ce dossier est lui-même séparé en 3 parties :

* Components : Il regroupe tous les composants React. Un composant est une partie fonctionnelle bien précise d’une page (par exemple, le graphique d’investissement ou le convertisseur de devises). Chaque composant est composé de ses propres fichiers React et CSS associés pour le style. Ces composants sont conçus pour être réutilisables dans différentes pages si nécessaire.
* Pages : Ce dossier contient les différentes pages de l’application, c’est-à-dire les vues principales que l’utilisateur voit à l’écran (comme la page d’accueil, la page portefeuille, etc.). Ces fichiers importent les composants du dossier components pour construire l’interface complète de chaque page.
* Redux : Ce dossier contient les fichiers JavaScript nécessaires à la gestion de l’état global de l’application. On y trouve notamment les fichiers qui permettent d’effectuer les requêtes au serveur (backend) ainsi qu’à l’API externe CoinGecko, pour récupérer les prix des cryptomonnaies et gérer les actions côté client.

1. **/public**

Le dossier « public » contient toutes les ressources statiques de l’application, principalement les images utilisées sur le site (par exemple : logos, icônes, images des drapeaux de devises). Ces fichiers sont directement accessibles par le navigateur.

1. **/server**

Ce dossier regroupe tout ce qui concerne le backend de l’application, c’est-à-dire la partie qui gère la base de données et la logique serveur, Il comprend :

* server.js : Fichier principal de l’API REST. Il permet de gérer l’ensemble des interactions avec la base de données (par exemple : récupérer les données utilisateur, gérer l’authentification, exécuter les transactions, etc.).
* migrations : Ce sous-dossier contient un script automatisé qui permet, lorsqu’il est lancé, de remplir la table currency avec toutes les données nécessaires sur les cryptomonnaies (par exemple : noms, codes, symboles). Cela permet d’initialiser rapidement la base de données.

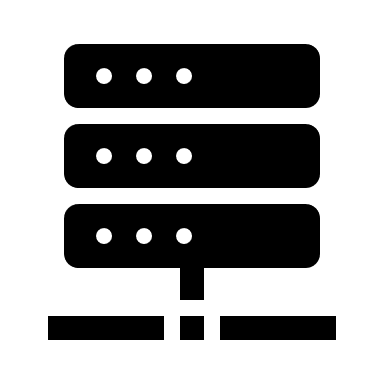
## Serveur (Backend)

La partie serveur de l’application a été développé en utilisant le framework Express.js pour Node.js. J’ai choisi ce framework pour sa simplicité, sa flexibilité et surtout pour une création d’API REST simple d’utilisation. Il facilite la gestion des routes et de l’intégration de middlewares ainsi que sa comptabilité avec Node.js.

Le rôle principal du serveur est de gérer l’ensemble des échanges entre l’interface utilisateur (frontend), la base de données MySQL. Consultez le schéma à la section [Architecture du site](#_Architecture_du_site).

Le serveur gère :

* L’authentification et la connexion



http://localhost:8080

* L’authentification à deux facteurs avec WebAuthn
* Gestion des utilisateurs
* Gestion des portefeuilles
* Gestion des devises
* Gestion des soldes
* Gestion des transactions
* Gestion des watchlist
* Gestion des sessions
* Sécurité

### API REST

L’API REST que j’ai développée sur le serveur agit comme un pont sécurisé entre le backend (où se trouve la logique métier) et la base de données, mais aussi entre le frontend (React) et le backend. Pour rendre l’API accessible, il suffit de démarrer le serveur avec la commande node server.js.

Par exemple, si je souhaite récupérer les utilisateurs de la table user, je crée une route /user sur le serveur, qui fait le lien entre le backend et la base de données. Ensuite, côté frontend, j’utilise la bibliothèque Redux pour envoyer une requête vers cette route /user.

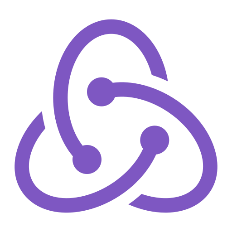
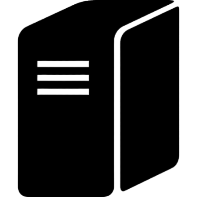
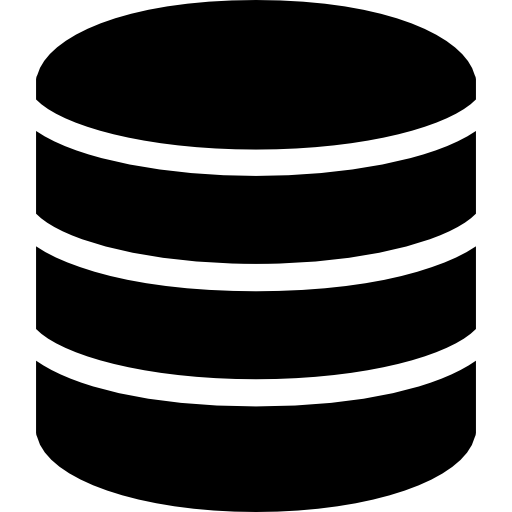


Table user

Route /user

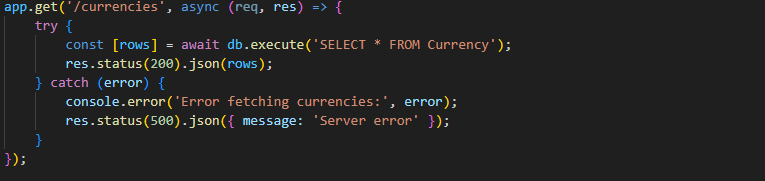
Requête /user

Requête GET / POST

Response.data

Requête SQL

Réponse de la BD

  
Pour créer une route, il suffit d’indiquer le type de requête, comme GET (pour lire) ou POST (pour écrire), puis de définir le endpoint, c’est-à-dire le chemin d’accès à une ressource — ici, « /currencies ». Ensuite, dans cette route, on effectue simplement une requête SQL pour récupérer les données, puis on les renvoie en réponse. Cela signifie que lorsqu’on démarre le serveur, en accédant à l’adresse <http://localhost:8080/currencies> , les devises sont affichées à cet emplacement.

Pour chaque interaction avec la base de données une route dédié est créée comme pour la connexion, la création de compte etc… Pour le format des données, j’ai choisi d’utiliser le JSON, car c’est plus simple et largement utilisé pour les échanges entre le client et le serveur. Comme les requêtes envoyées depuis le front sont au format JSON, Express propose un convertisseur automatique qui permet de parser le corps des requêtes. Grâce à cela, je peux facilement manipuler les données en JSON côté serveur. À noter que les données renvoyées par le serveur sont également au format JSON (comme le montre l'encadré rouge sur l’image ci-dessus). Le choix d’utiliser le format JSON permet d’unifier les échanges de données, puisqu’en récupérant les informations depuis l’API CoinGecko, celles-ci sont aussi reçues en JSON.

## Sécurité

### Gestion de l’authentification (classique)

L’authentification classique repose sur la vérification des informations fournies par l’utilisateur (Email, mot de passe). L’utilisateur rentre ses identifiants, le serveur vérifie ces informations dans la base de données. Si elles sont correctes, l’accès est accordé à l’utilisateur.

**Schéma Register :**

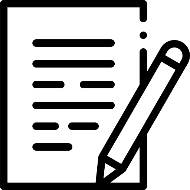
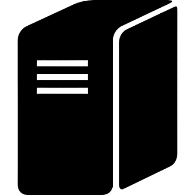
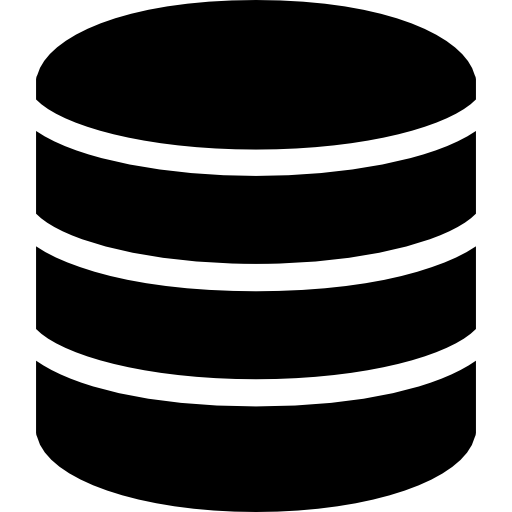
JWT token

**4**

Cadenas - Icônes sécurité gratuites

Nom d’utilisateur  
Email  
Mot de passe

Ruben  
[Ruben.Tome@rpn.ch](mailto:Ruben.Tome@rpn.ch)  
$2b$10$zJ0rCf.KPb



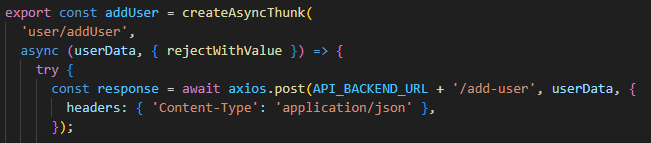
**3**

**2**

**1**

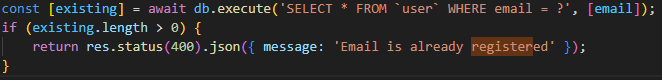
  
Lorsque l’utilisateur souhaite se créer un compte, il doit se rendre sur la page « /regsiter ». Le formulaire lui demande de rentrer un nom d’utilisateur, email et mot de passe. Lorsqu’il clique sur le bouton « Register », il appelle la méthode adduser et qu’il lui passe en paramètre les données de l’utilisateur.

**1**

Cette fonction envoie une requête au serveur **API\_BACKEND\_URL/add-user**. Il spécifie la route « add-user » qui a été au préalablement créer dans le serveur.

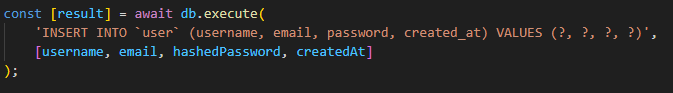
**2**

Le serveur va ensuite récupérer les informations de l’utilisateur envoyé lors de la requête :

  
Il va également vérifier qu’aucun autre utilisateur à la même adresse mail. Nous pouvons nous créer un seul compte avec un email :

Avant de stocker dans la base de données, il faut hashé et salé le mot de passe afin de ne pas afficher le mot de passe en dur dans la base de données. Le hachage transforme le mot de passe en une chaîne de caractères fixe, qui est difficile à inverser, tandis que le salage ajoute   
  
  
une valeur aléatoire unique à chaque mot de passe. Pour cela j’ai utilisé la librairie bcrypt qui nous permet grâce à la méthode hash(), de hashé la chaîne de charactère.   
Cette méthode prend en paramètre la chaine de caractère à hashé ainsi que le nombre de « salt rounds » qui permet de déterminé la complexité du hachage.



  
Après avoir hashé le mot de passe nous pouvons stocker l’utilisateur dans la base de données avec une requête SQL simple :

**3**

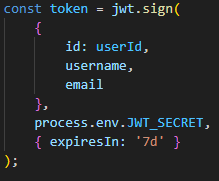
**3**

L’utilisateur est maintenant enregistré dans la base de données en plus de ça nous allons aussi lui créer un wallet par défaut nommé « My wallet ».



**4**

Et finalement un token JWT est nécessaire. Un token JWT (JSON Web Token) est un moyen sécurisé de transmettre des informations entre un client et un serveur. Lorsqu’un utilisateur se connecte ou crée un compte, le serveur génère un JWT qui contient des informations sur l’utilisateur (userId, username, email) et le renvoie au client.

J’ai utilisé la librairie JWT sur node pour réaliser la création du token. La méthode sign() nous permet de créer un nouveau token. Le token contient donc les informations de l’utilisateur (userId, username, email). Le « process.env.JWT\_SECRET » est une variable dans mon fichier «.env » qui permet de signé notre token. J’ai utilisé [JWTSecret](https://jwtsecret.com/generate) pour générer mon JWT\_SECRET.

Cette signature est essentielle, elle permet de garantir que le contenu du token n’as pas été altéré après sa création et que ce token provient bien du serveur qui l’a émis. Et enfin la durée de vie du jeton, moi j’ai mis 7 jours après 7 jour le token n’est plus valide et il faut se reconnecter.

Le token JWT est constitué de 3 parties :

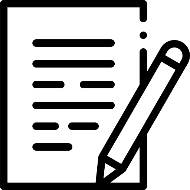
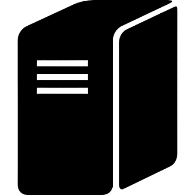
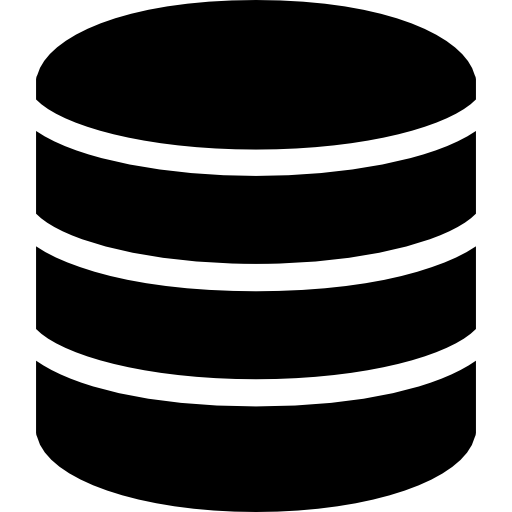
1. **Header (en-tête)** : type de token (JWT) et l’algorithme utilisé pour signer le token « HS256 » dans mon projet.
2. **Payload (corps)** : Contient les informations que l’on veut transmettre, comme les données de l’utilisateur « userId », « username » et « email » dans mon projet.
3. **Signature**: C’est ce qui permet de vérifier que le token n’as pas été altéré. La signature est générée en combinant le header, le payload et une clé secrète « JWT\_SECRET »

Le serveur renvoie donc les informations de connexion ainsi que le token en réponse. Ensuite, il faut stocker le token dans le localStorage afin que, pour chaque requête nécessitant un token valide, celui-ci soit disponible. La présence du token dans le localStorage permet également de savoir si l'utilisateur est connecté ou non.

Je récupère le token de la réponse envoyée par le serveur et je le stock dans le localStorage.

**Schéma login :**

JWT token

Cadenas - Icônes sécurité gratuites

[Ruben.Tome@rpn.ch](mailto:Ruben.Tome@rpn.ch)

Email

Mot de passe

$2b$10$zJ0rCf.KPb

Pour le login, le processus est très similaire. Lorsque l’utilisateur souhaite se connecter, il doit renseigner son email ainsi que son mot de passe. Grâce à une requête vers la base de données, on récupère les informations de l’utilisateur à partir de l’email qu’il a fourni. Ensuite, avec bcrypt.compare(), nous allons comparer le mot de passe saisi avec celui stocké en base de données. Bcrypt ne cherche pas à « déchiffrer » le mot de passe haché, car un hash n’est pas réversible. Il va plutôt hasher le mot de passe saisi par l’utilisateur en utilisant le même sel intégré dans le hash existant, puis comparer les deux résultats.



Le même processus est appliqué pour le login, si les deux mots de passe sont identiques alors on crée un token JWT et le stock localement chez l’utilisateur.

### Gestion de l’authentification (2FA, WebAuthn)

En plus de l’authentification classique (email, mot de passe), j’ai intégré une authentification à double facteur (2FA) basée sur WebAuthn. J’ai choisi d’ajouter cette couche de sécurité supplémentaire afin de renforcer la protection des comptes utilisateurs. Étant donné qu’il s’agit d’une plateforme liée aux cryptomonnaies, le risque de vol d’argent est élevé. L’utilisation d’une clé physique, comme une YubiKey, permet de garantir une authentification forte et de limiter les risques. Contrairement aux autres systèmes de 2FA, tels que la réception d’un code par SMS, où un pirate pourrait potentiellement intercepter le message pour se connecter, une clé physique comme la YubiKey nécessite que le pirate possède réellement la clé. Il doit donc l’obtenir en personne, rendant l'attaque beaucoup plus difficile et limitée aux situations où l'attaquant a un accès physique direct à l’utilisateur.

WebAuthn (Web Authentication) est une norme web développée par le W3C et la FIDO Alliance qui permet une authentification forte, directement dans le navigateur. Elle utilise des clés cryptographiques stockées de manière sécurisée sur un appareil dans note cas c’est une clé physique YubiKey. Des bibliothèques prêtes à l'emploi sont disponibles pour faciliter l'intégration de WebAuthn côté serveur et côté client, permettant de gérer facilement l’enregistrement, l’authentification. J’ai choisi d’utiliser la bibliothèque SimpleWebAuthn, qui propose une documentation claire et facile à comprendre.   
Elle est composée de deux modules distincts :

* @simplewebauthn/server : destiné à la partie backend.
* @simplewebauthn/browser : utilisé côté frontend.

Chaque module propose des méthodes prêtes à l’emploi, ce qui simplifie grandement l’intégration de WebAuthn dans une application web.

Clé Physique

Backend

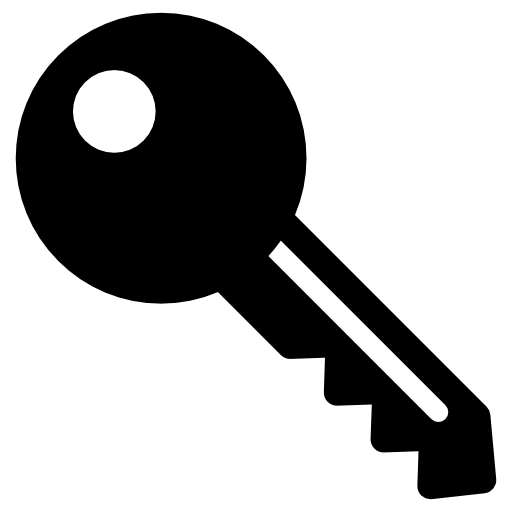
Frontend

### Base de données - Icônes la technologie gratuitesUtilisateur - Icônes social gratuites

Registration response

verifyRegistrationResponse()

**6**



**5**

Génère paire clés (privé/public)

startRegistration()

**4**

**8**

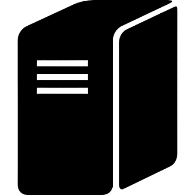
**7**

**3**

generateRegistrationOptions()

**2**

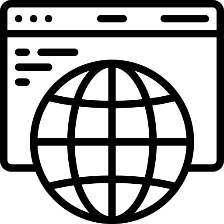
Registration options



Utilisateur

**1**

Navigateur



**- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -**

**- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -**

**1**

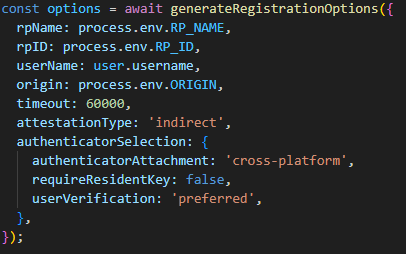
La première étape consiste pour l’utilisateur à initier l’enregistrement d’une nouvelle clé de sécurité, qui servira comme méthode d’authentification à deux facteurs (2FA).

**2**

Avant de pouvoir enregistrer une YubiKey (ou tout autre passkey), le navigateur a besoin d’un "plan d’action" : une sorte de contrat entre le site, l’utilisateur et l’appareil. C’est ce qu’on appelle les « registrations options ». Le frontend envoie donc une requête au backend vers la route « /webauthn/register », en envoyant un token d’authentification afin de demander les registrations options. Le token JWT est nécessaire puisque seulement les utilisateurs qui possèdent un compte peuvent activer la 2FA.

**3**

L’étape suivante consiste à générer les options d’enregistrement côté serveur. Pour cela, on utilise la méthode « generateRegistrationOptions() » fournie par la librairie « @simplewebauthn/server », qui permet de définir toutes les informations requises pour l’enregistrement d’une passkey. Voici les informations nécessaires :

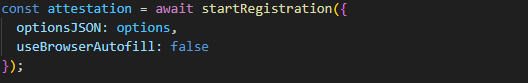
* **rpName** : le nom du site (CryptoTrak)
* **rpID** : son domaine (localhost)
* **userName** : nom d’utilisateur
* **origin** : son origine (<http://localhost:5173>)
* **timeout** : un délai
* **authenticatorAttachment** : définit le type d’authentificateur accepté (clé physique, biométrie, etc.). Comme on exige une clé physique, j’ai choisi "cross-platform", qui restreint l’usage aux dispositifs externes comme les YubiKeys.
* **requireResidentKey** : indique si la clé publique doit être conservée directement dans la clé physique. En la laissant sur false, on simplifie l’usage, car toutes les clés ne supportent pas ce mode.
* **userVerification** : demande une vérification de l’utilisateur (touché physique ou PIN)

 Un challenge est une chaîne aléatoire générée par le serveur pour sécuriser l’enregistrement. Il sert à vérifier que la réponse reçue vient bien du bon utilisateur, au bon moment.

Dans cette étape, un challenge est généré puis stocké dans la session. Il sera comparé plus tard avec celui contenu dans la réponse du navigateur, pour s’assurer que l’enregistrement est authentique et récent.

**4**

Le navigateur via la méthode « startRegistration() » mis à disposition par librairie « @simplewebauthn/browser » car c’est lui qui gère cette étape côté client. Le navigateur affiche une fenêtre système (popup native) demandant à l’utilisateur d’insérer sa clé physique (YubiKey) et d’interagir avec elle en la touchant ou en mettant son PIN.



**5**

Suite à cette interactive qui demande de rentrer la clé, le navigateur va transmettre les options à la clé physique, elle va ensuite :

* Générer une paire de clés (publique/privée)
* Signer le challenge généré par le serveur avec sa clé privée.

La clé privée restera secrète et stocké dans la clé physique (elle ne quitte jamais l’appareil). La clé publique elle pourra être envoyée au serveur.

**6**

La clé renvoie au navigateur ce qu’on appelle une attestation, qui contient :

* **Credential ID** : Un identifiant unique généré pour cette clé. Il permet au serveur de reconnaître cette clé plus tard lors des connexions.
* **Clé publique** : Elle sera stockée par le serveur pour vérifier les futures connexions.
* **Signature clé privée** : La clé signe le challenge fourni par le serveur, ce qui prouve qu’elle possède bien la clé privée correspondante à la clé publique envoyée.
* **Challenge (serveur)** : C’est une valeur aléatoire que le serveur avait générée et envoyée au navigateur. Il permet de s’assurer que cette attestation est bien créée à l’instant T et pour cet utilisateur, empêchant toute réutilisation.
* **Informations origine** : Le navigateur inclut aussi l’origin (localhost) pour prouver que l’attestation vient bien du bon site web.

Tout ça forme une preuve cryptographique qui prouve que la clé privée a bien signé le challenge, l’opération vient du bon site et qu’elle est bien liée à l’utilisateur.

**6**

Le navigateur va envoyer toutes les informations nécessaires, qui seront ensuite vérifiées à l’aide de la fonction « verifyRegistrationResponse ». Le serveur va récupérer le challenge précédemment envoyé et le comparer avec celui stocké en session, afin de s’assurer qu’il n’a pas été modifié entre-temps. Vérifier si la requête provient bien du bon site, en comparant l’origin reçu avec l’origin attendu. Et enfin analyser la signature fournie par la clé, en utilisant la clé publique générée pour s’assurer que le challenge a bien été signé avec la clé privée, qui ne quitte jamais l’appareil.

**8**

La dernière étape est d’enregistrer toutes les données requises pour que l’utilisateur puisse ensuite se connecter. Dans la base de données il est stocké le credentialID, la clé publique et d’autres informations qui ne sont pas nécessaires au login comme type d’appareil (device) ou encore le transport (USB, NFC etc…). Aussi à cette étape, si toutes les vérifications sont réussies, un token JWT est généré et envoyé à l'utilisateur afin de confirmer son authentification et lui permettre d'accéder aux ressources protégées de l'application.

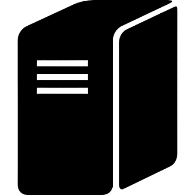
Clé Physique

Backend

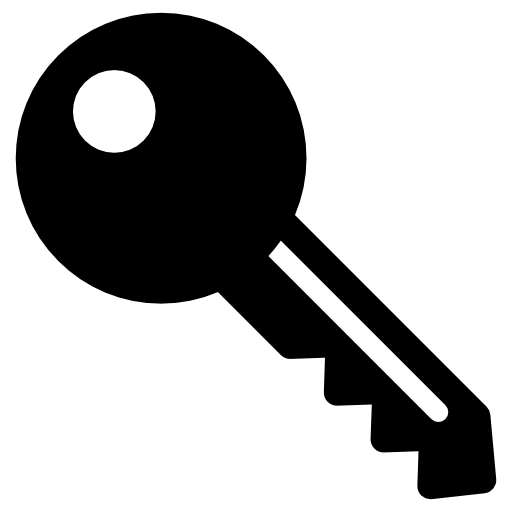
Frontend

### Cache - Icônes ordinateur gratuitesBase de données - Icônes la technologie gratuitesUtilisateur - Icônes social gratuites

**2**



**4**



**6**

Signature avec la clé privée du challenge

verifyAuthenticationResponse()

**4**

generateAuthenticationOption()s

**3**

Authentification options

Authentification response

**7**

startAuthentication()

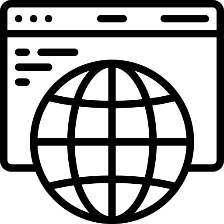
**5**

**8**

Utilisateur

**1**

Navigateur



**- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -**

**- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -**

**1**

Pour la connexion avec WebAuthn, c’est très similaire à l’enregistrement. L’utilisateur va bien sûr vouloir se connecter à son compte via le formulaire de connexion. Le serveur va d’abord vérifier l’email et le mot de passe, et si l’utilisateur a activé la 2FA avec une clé, le processus de login avec WebAuthn commence.

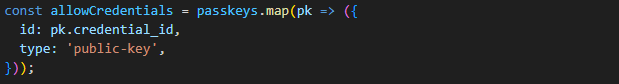
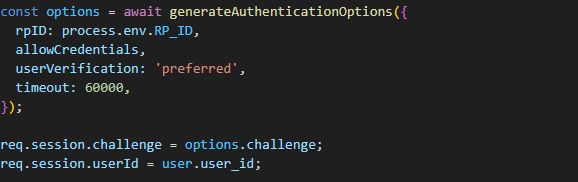
**2**

Le serveur crée d’abord une session sécurisée côté serveur, qui permettra de stocker le challenge en attendant la confirmation via la clé physique.

**3**

Comme pour l’enregistrement, le navigateur va demander au serveur de générer les options, mais cette fois pour l’authentification.

**4**

La fonction « generateAuthenticationOptions », va créer les options nécessaires pour lancer l’authentification WebAuthn. J’ai d’abord récupéré les identifiants des clés (credential IDs) associées à l’utilisateur depuis la base de données afin de créer une liste des clés autorisées pour l’authentification.   
La propriété « type: 'public-key' » dans l’objet allowCredentials indique le type d'identifiant d’authentification que le navigateur doit utiliser lors de la phase d’authentification WebAuthn. Elle précise que l’identifiant correspond à une clé publique, utilisée pour vérifier la signature générée par la clé privée stockée dans l’appareil de l’utilisateur.

Le serveur enregistre le challenge généré dans la session, puis renvoie au client les options d’authentification créées.

**5**

Le navigateur avec la méthode « startAuthentication() », Le navigateur affiche une fenêtre système (popup native) demandant à l’utilisateur d’insérer sa clé physique (YubiKey). L’utilisateur doit donc toucher la clé ou rentré le code PIN.

**6**

La clé physique signe le challenge envoyé grâce à sa clé privée qui est enregistré dans la clé.

Elle renvoie une **assertion** (preuve d’authentification) au navigateur, qui contient, le credentialId utilisé, la signature du challenge ainsi que d’autre informations. Ces données sont ensuite transmises au serveur pour vérification.

**7**

**8**

Finalement, le serveur reçoit l’assertion envoyée par le navigateur et effectue plusieurs vérifications. Il récupère les passkeys associées à l’utilisateur et compare le credential ID contenu dans l’assertion avec ceux stockés en base de données pour identifier la bonne clé. Grâce à la fonction « verifyAuthenticationResponse », le serveur vérifie que le challenge n’a pas été modifié, en le comparant avec celui qui avait été stocké dans la session. Il s’assure également que la signature a bien été générée à l’aide de la clé privée correspondant à la clé publique enregistrée, et valide d'autres éléments comme le rpID et l’origin. Si toutes les vérifications sont réussies, un token JWT est généré et renvoyé au client pour authentifier l’utilisateur.

### Sécurisation de l’API REST

Grâce à mon système d’authentification basé sur les JSON Web Tokens (JWT), certaines routes de mon API REST sont protégées et accessibles uniquement aux utilisateurs authentifiés. Pour cela, j’ai mis en place un middleware de vérification de jeton (verifyToken) qui intercepte les requêtes vers ces routes.

Un middleware en Express est une fonction qui s'exécute entre la réception de la requête et l'envoi de la réponse. Il sert à traiter ou modifier la requête, vérifier des conditions (comme l’authentification), ou gérer des erreurs.

Le middleware que j’ai créé vérifie la présence et la validité du token JWT envoyé dans l’en-tête Authorization. Si le token est valide, l’utilisateur peut accéder à la ressource ; sinon, l’accès est refusé avec un message d’erreur.

Ce mécanisme permet de sécuriser efficacement les ressources sensibles de l’API en s’assurant que seules les personnes connectées y ont accès. C’est-à-dire que lorsque l’utilisateur veut récupérer les données d’une route sécurisée par le middleware, il doit envoyer son token dans l’en-tête de la requête pour prouver au serveur qu’il a bien les droits d’accéder à ces ressources.

Concrètement, si l’on veut par exemple récupérer les informations d’un utilisateur, il faut, lors de la définition de la route, préciser qu’il faut d’abord exécuter la fonction verifyToken, comme ceci :



La méthode verifyToken récupère l’en-tête de la requête pour vérifier la présence d’un jeton JWT. Elle extrait ce jeton depuis l’en-tête Authorization.



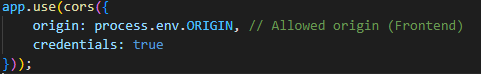
Il le décode et le valide à l’aide d’une clé secrète définie dans les variables d’environnement. Si le jeton est valide, l’utilisateur est authentifié et la requête peut continuer ; sinon, une réponse d’erreur est renvoyée.

Ce processus permet donc de sécuriser l’accès aux ressources de mon API en s’assurant que seules les requêtes provenant d’utilisateurs authentifiés, possédant un jeton valide, peuvent accéder aux routes protégées.

### Protection contre les attaques

Pour garantir la sécurité de mon application web, j’ai mis en place plusieurs mesures afin de se protéger contre les attaques courantes sur le web.

**CORS (Cross-Origin Ressource Sharing):**

Le CORS a été configuré côté serveur pour contrôler quelles origines sont autorisées à accéder aux ressources de l’API. Cela permet d’éviter les requêtes non autorisées provenant de domaines externes, ce qui renforce la protection contre certaines attaques cross-site.

Dans mon serveur, j’ai seulement autorisé les requêtes provenant de l’adresse de mon frontend (localhost:5173), qui est définie dans mon fichier .env. J’autorise également mon frontend à envoyer des cookies ainsi que des tokens JWT utilisés pour l’authentification.

**Injections SQL**

Pour éviter les attaques par injection SQL, toutes les requêtes vers la base de données utilisent des requêtes préparées (paramétrées).

Ici, le ? est automatiquement remplacé par la valeur userId de manière sécurisée, sans être interprétée comme du code SQL.

**XSS (Cross-Site Scripting)**

Côté frontend, l’application est développée avec React, qui intègre par défaut des protections contre les attaques XSS. React échappe automatiquement les données injectées malveillantes, ce qui empêche l’exécution de scripts malveillants transmis par l’utilisateur.

Et évidemment, l’utilisation des tokens JWT pour l’authentification, ainsi que les middlewares utilisés dans le serveur pour protéger les routes, garantissent également une sécurité renforcée.

### Gestion des erreurs

Pour ce qui est de ma gestion des erreurs dans mon serveur Express, j’ai mis en place dans chaque route des « try…catch ». Cela me permet de récupérer les erreurs qui pourraient survenir (variable vide, erreur SQL etc…) et éviter que le serveur ne plante.

A l’intérieur du « catch », j’utilise la méthode res.status() d’Express qui permet de renvoyer un code HTTP selon la situation :

* 200 : une réponse réussie.
* 400 : une erreur côté client (par exemple : données invalides, token incorrect).
* 404 : une ressource demandée non trouvé.
* 500 : une erreur interne du serveur.

Ce renvoie permet d’envoyer au frontend des messages clairs afin de facilité le debug.

## Implémentation API CoinGecko

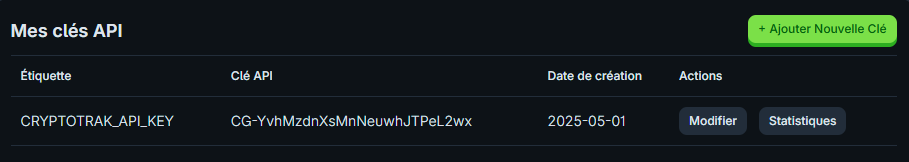
Pour mon projet, qui nécessite des données en temps réel des cryptomonnaies, j’ai donc choisi d’utiliser l’API de CoinGecko. Leur documentation est très claire, complète et facile à comprendre ce qui m’as facilité pour l’écriture des requêtes.

Cependant, durant la réalisation mon pré-TPI, je me suis rapidement confronté à une limitation du nombre de requête, l’API gratuite ne permet que d’envoyer 15 requêtes par minutes. Cette restriction risquait de poser problème à la fois pour la suite du développement de mon TPI ainsi que pour ma démonstration de mon projet devant les experts.

Pour résoudre ce problème pour le TPI, j’ai décidé de contacter directement l’équipe de CoinGecko par mail pour leur présenter ma situation, en expliquant que j’étais un étudiant suisse utilisant leur API dans leur cadre d’un projet de fin de diplôme.

La conversation a duré environs deux mois dans lesquelles, nous avons discuté pour accéder à un plan supérieur adapté à ce que j’ai besoin. Ils m’ont ensuite envoyé une candidature officielle à remplir. Après l’analyse de ma candidature, un membre de la direction de CoinGecko m’a envoyé un mail en m’informant que ma candidature avait été accepté. Ils m’ont donné accès à un plan qui permet d’envoyer 500 requêtes par minutes, ce qui m’as permis de résoudre ce problème de limitation.

Cette collaboration m’a grandement permis de facilité mon développement et d’assuré une démonstration possible sans interruption.

Suite à cela, j’ai pu créer ma propre clé API CoinGecko à partir du compte qu’ils m’ont fournis.

Cette clé me permet, à chaque fois que j’envoie une requête à l’API, de l’intégrer dans l’en-tête de la requête afin de prouver que j’ai le droit d’effectuer jusqu’à 500 requêtes par minute, comme ceci :   


## Frontend

# Protocole de test

# Problèmes rencontrés et solutions

# Améliorations et perspectives

# Conclusion

# Bibliographie

* [Flaticon](https://www.flaticon.com/packs/countrys-flags) (Icones des devises)
* [Dribbble](https://dribbble.com/shots/15090067-Cryptocurrency-Trading-Platform) (Idée d’interface)
* [SimpleWebAuthn](https://simplewebauthn.dev/docs) (Documentation / utilisation WebAuthn)
* [ChatGPT](https://chatgpt.com/) (Debug et corrections)

# Annexes

## Documentation technique

## Guide d’utilisation

Pour pouvoir utiliser mon application correctement vous devrez suivre ce guide d’utilisation. Je vais expliquer les différentes étapes pour lancer mon application web.

1. **Installer les dépendances**

* Installation de **NodeJS** [ici](https://nodejs.org/en/download). (Choisir l’installeur « Windows Installer (.msi) »)
* Installation de **Visual Studio Code** [ici](https://code.visualstudio.com/download)
* Une image contenant texte, Police, ligne, nombre

  Le contenu généré par l’IA peut être incorrect.Installation de WampServer [ici](https://wampserver.aviatechno.net/) (à noter que les ordinateurs de l’école en sont déjà équipés)