Rapport du projet – Plot Those Lines! (Crypto Edition)

1. Introduction

1.1 Objectifs pédagogiques

Ce projet est conçu pour me permettre de mettre en pratique les compétences acquises dans les modules de programmation.

J'ai appliqué plusieurs notions clés du cours :

- la programmation orientée objet avec C# ;
- l'utilisation de LINQ comme approche moderne de traitement des données ;
- la gestion d'une API externe et la désérialisation JSON ;
- la mise en œuvre de tests unitaires afin d'assurer la qualité du code ;
- la création d'une interface utilisateur graphique (WinForms) ;
- la gestion de projet avec GitHub (projets, commits, versioning) ;
- la production d'une documentation professionnelle (Rapport, JDT, README).

L'objectif pédagogique est donc double : d'un côté, apprendre à développer une application concrète avec un client, et de l'autre, acquérir des bonnes pratiques professionnelles en termes de planification, organisation et communication technique.

Enfin, ce projet me permet de développer des compétences transversales comme l'autonomie dans la résolution de problèmes, la structuration de code maintenable et la documentation claire des processus de développement. J'ai également appris à planifier des tests basés sur des User Stories et à corriger des erreurs détectées lors des tests unitaires, ce qui renforce mon approche méthodique et rigoureuse.

1.2 Objectifs produit

Le produit attendu est une application Windows Forms qui permet de visualiser des séries temporelles sous forme de graphiques.

- L'utilisateur pourra importer et afficher les données issues d'une API externe (CoinGecko).
- L'application doit être capable d'afficher plusieurs séries sur un même graphique (par ex. Bitcoin + Ethereum).
- L'utilisateur doit pouvoir choisir la période d'analyse (7, 30, 90 jours).
- Des fonctions supplémentaires permettront d'analyser les données de manière flexible (zoom, légendes, choix de couleurs).

Le produit n'est pas seulement une démonstration technique : il doit être utilisable et compréhensible par un utilisateur non technique qui souhaite visualiser facilement l'évolution des cryptomonnaies.

2. Domaine et sources de données

Le projet s'inscrit dans le secteur des cryptomonnaies, un domaine en évolution rapide et riche en données quantitatives.

J'ai choisi le domaine des cryptomonnaies pour plusieurs raisons :

- C'est un secteur en forte croissance et très médiatisé.
- Les données financières et temporelles y sont particulièrement adaptées à la représentation graphique.
- Il existe plusieurs sources de données fiables et gratuites.

2.1 Sources de données

- CoinGecko API (https://www.coingecko.com/en/api)
  CoinGecko est une plateforme reconnue dans le domaine des cryptos. Son API gratuite fournit des données historiques et en temps réel au format JSON. C'est cette source que j'utiliserai pour alimenter mon application.

- Investing.com – Live Charts (https://www.investing.com/charts/live-charts)
  En analysant cette plateforme, j'ai pu m'inspirer des fonctionnalités de visualisation qui pourraient être intéressantes à intégrer (choix d'un actif, intervalle de temps, graphiques multi-séries).

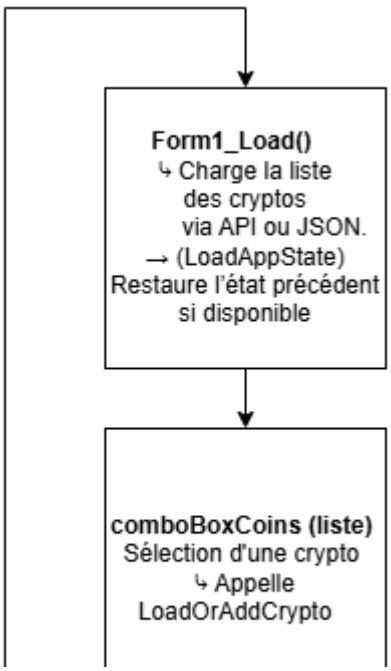
2.2 Justification du choix

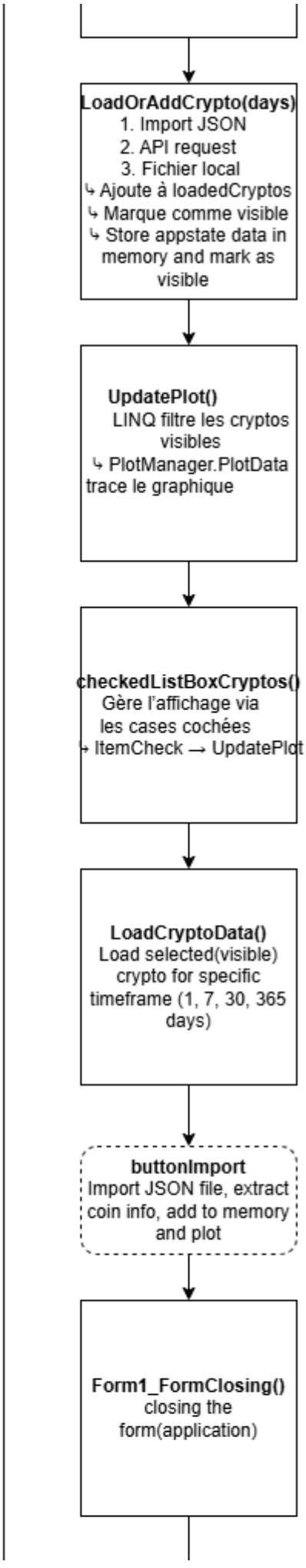
J'ai préféré l'utilisation d'une API JSON plutôt que de fichiers CSV pour plusieurs raisons :

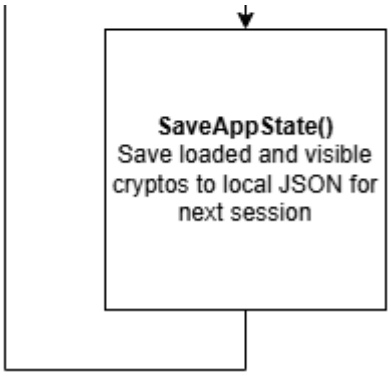
1. Les données sont toujours à jour.
2. L'API permet une intégration dynamique sans téléchargement manuel.
3. Le format JSON est mieux adapté à la sérialisation en objets C#.

2.4. Schéma de fonctionnement de l'application

Le schéma ci-dessous présente le fonctionnement général de l'application, incluant la récupération des données depuis l'API CoinGecko, leur traitement en C# et l'affichage dynamique via ScottPlot.




LoadOrAddCrypto(days)
1. Import JSON
2. API request
3. Fichier local
↳ Ajoute à loadedCryptos
↳ Marque comme visible
↳ Store appstate data in memory and mark as visible
UpdatePlot()
LINQ filtre les cryptos visibles
↳ PlotManager.PlotData trace le graphique
checkedListBoxCryptos()
Gère l'affichage via les cases cochées
↳ ItemCheck → UpdatePlot
LoadCryptoData()
Load selected(visible) crypto for specific timeframe (1, 7, 30, 365 days)
buttonImport
Import JSON file, extract coin info, add to memory and plot
Form1_FormClosing()
closing the form(application)



Le fonctionnement général de l'application PTL_Crypto repose sur un cycle complet de chargement, affichage et sauvegarde des données.
Lors du lancement, la méthode Form1_Load() tente de charger la liste des cryptomonnaies depuis l'API. En cas d'échec (limite d'accès ou absence de réseau), les données locales JSON sont utilisées.
L'utilisateur sélectionne ensuite une crypto dans la comboBoxCoins, ce qui appelle LoadOrAddCrypto().
Cette méthode essaie de récupérer les données dans l'ordre suivant :
1 fichier importé, 2 API, 3 fichier local.

Les cryptos chargées sont stockées en mémoire (loadedCryptos), marquées comme visibles, et le graphique est actualisé par UpdatePlot(), qui trace les données via PlotManager.PlotData.
L'affichage peut être ajusté grâce au checkedListBoxCryptos, où chaque case permet d'afficher ou de masquer une crypto.
Les boutons de période (1, 7, 30, 365 jours) appellent LoadCryptoData() pour recharger la crypto visible selon la période choisie.
Le bouton d'import JSON est optionnel et permet d'ajouter de nouvelles cryptos à partir d'un fichier local.

Enfin, lors de la fermeture (Form1_FormClosing()), l'état actuel (cryptos chargées et visibles) est sauvegardé dans un fichier state.json par SaveAppState().
Au prochain démarrage, LoadAppState() restaure automatiquement cet état, formant ainsi un cycle complet et autonome entre le chargement, l'affichage et la persistance des données.

3. Analyse et planification

3.1 User Stories

Voir README.md
(section User Stories) et le GitHub Project associé.

Exemples:

- Importer des données JSON depuis l'API CoinGecko.
- Afficher un graphique temporel avec ScottPlot.
- Comparer plusieurs cryptomonnaies.
- Documentation (Rapport, JDT, README, Planification).

3.2 Planification

J'ai planifié le projet sur **8 semaines** pour un total de 24 à 32 périodes, avec une marge de sécurité.

Consulter le fichier séparé Planification.md

3.3 Expression des besoins (User Stories)

Cette section présente les principales **User Stories** du projet **Plot Those Lines (Crypto Edition)**.
Elles ont été rédigées selon la méthode « *En tant que… je veux… afin de…* », accompagnées de leurs **critères d'acceptation [TA]**.

---

1. Prototype API

En tant que développeur, je veux que l'application récupère des données JSON depuis **CoinGecko** pour une cryptomonnaie donnée, afin de vérifier la connexion et les données reçues.

[TA] Critères d'acceptation :

- Quand je lance la récupération, un appel HTTP est effectué via `HttpClient`.
- Si l'appel réussit, le JSON brut s'affiche dans une `TextBox` ou la console.
- Si l'appel échoue (erreur réseau ou crypto inconnue), un `MessageBox` d'erreur s'affiche et je reste sur la même interface.

En tant qu'utilisateur, je veux voir les données de prix d'une cryptomonnaie affichées dans l'application, afin de vérifier qu'elles proviennent bien de la source officielle.

[TA] Critères d'acceptation :

- Quand je choisis une crypto (ex. Bitcoin) et que je lance la recherche, les données apparaissent dans l'application.
- Si la crypto existe, je vois les valeurs correctes.
- Si la crypto est inconnue ou la connexion échoue, un message clair m'informe du problème.

---

2. Parsing JSON + LINQ

En tant que développeur, je veux parser le JSON et le transformer en objets `CryptoPrice` à l'aide de **LINQ**, afin de préparer les données à l'affichage graphique.

[TA] Critères d'acceptation :

- Le timestamp du JSON est correctement converti en `DateTime`.
- LINQ est utilisé pour transformer et filtrer les données.
- Des tests unitaires vérifient la cohérence des objets `CryptoPrice`.

En tant qu'utilisateur, je veux que les données du graphique soient cohérentes dans le temps.

[TA] Critères d'acceptation :

- Les dates sont correctes sur l'axe temporel.
- Les valeurs affichées correspondent à celles de l'API.
- Si les données sont invalides, un message clair apparaît.

---

3. Premier graphique

En tant qu'utilisateur, je veux voir un graphique **ScottPlot** des prix d'une cryptomonnaie sur une période donnée, afin de visualiser son évolution.

[TA] Critères d'acceptation :

- Au démarrage, un graphique est visible (Bitcoin sur 7 jours par défaut).
- Les axes X et Y sont correctement configurés.
- Une légende indique la cryptomonnaie correspondante.

---

4. Flexibilité utilisateur

En tant qu'utilisateur, je veux choisir la cryptomonnaie et la période d'affichage (7, 30, 90 jours), afin d'adapter l'analyse à mes besoins.

[TA] Critères d'acceptation :

- Une `ComboBox` me permet de sélectionner la cryptomonnaie.
- Une autre `ComboBox` me permet de choisir la période.
- Le graphique se met à jour automatiquement.
- Si la crypto est inconnue, un `MessageBox` d'erreur s'affiche.

---

5. Afficher un graphique temporel avec ScottPlot

En tant qu'utilisateur, je veux visualiser les données sous forme de graphique temporel interactif, afin de comprendre l'évolution du prix.

[TA] Critères d'acceptation :

- L'axe X correspond au temps, Y au prix.
- Je peux zoomer et déplacer le graphique.
- La légende indique quelle courbe correspond à quelle crypto.

---

6. Comparer plusieurs cryptomonnaies

En tant qu'utilisateur, je veux comparer plusieurs cryptomonnaies sur un même graphique, afin d'analyser leurs performances relatives.

[TA] Critères d'acceptation :

- Je peux sélectionner plusieurs cryptos.
- Chaque crypto a une couleur distincte.
- Une légende précise à quelle crypto correspond chaque courbe.

---

7. Maquette Figma

En tant qu'utilisateur, je veux une maquette interactive dans **Figma** de l'application, afin de visualiser et tester toutes les fonctionnalités avant le développement.

Fonctionnalités représentées dans la maquette :

- Graphiques multi-séries temporelles.
- Zoom, déplacement et filtrage interactifs.
- Importation de données depuis JSON, CSV ou API.
- Affichage de plusieurs intervalles pour une même donnée.
- Menu clair, tooltips et design harmonieux.
- Tous les composants nécessaires au développement inclus.
- (Optionnel) Un onglet pour afficher des fonctions mathématiques (`x²`, `sin(x)`, etc.) avec saisie libre.

[TA] Critères d'acceptation :

- Navigation fluide entre toutes les fonctionnalités.
- Chaque fonctionnalité est visuellement annotée.
- Le design respecte la charte graphique.

---

8. Importer des données JSON depuis CoinGecko

En tant que développeur, je veux importer les données de prix depuis l'API **CoinGecko**, afin d'avoir des informations toujours à jour.

[TA] Critères d'acceptation :

- Les données sont récupérées automatiquement au lancement.
- Le JSON est parsé et affiché dans l'interface.
- En cas d'erreur réseau, un message d'erreur s'affiche.

En tant qu'utilisateur, je veux voir des données toujours actualisées pour pouvoir suivre l'évolution des cryptomonnaies.

[TA] Critères d'acceptation :

- Au lancement, je vois les prix actuels.
- Quand je change de crypto, le graphique se met à jour.
- En cas d'échec, un message clair s'affiche.

---

9. Bouton d'importation de fichier JSON pour les prix de crypto-monnaies

En tant qu'utilisateur de l'application PTL_Crypto,
je souhaite pouvoir sélectionner un fichier d'une extension correspondante aux données de cryptomonnaies sur mon ordinateur et **importer les données de prix**,
afin de visualiser et analyser les prix historiques sans passer par l'API.

[TA] Critères d'acceptation :

- L'utilisateur clique sur le bouton **"Importer JSON"**.
- Une fenêtre de sélection de fichier (`OpenFileDialog`) s'ouvre et permet de choisir un fichier `.json`.
- Le fichier JSON est lu et les données de prix sont converties en objets `CryptoPrice`.
- Les données sont affichées dans la `ComboBox` prévue.
- Les données importées peuvent ensuite être utilisées pour tracer un graphique via `formsPlot1`.

Tâches techniques :

- Ajouter un bouton `button1` avec le texte **"Importer JSON"**.
- Implémenter un `OpenFileDialog` pour la sélection du fichier.
- Utiliser la méthode `FileClient.LoadPricesFromFile` pour lire et convertir le JSON.
- Afficher les données dans `textBoxRawData`.
- Ajouter une gestion des erreurs si le fichier est invalide ou vide.

---

10. Ajouter la persistance de l'état de l'application (local JSON)

En tant qu'utilisateur de l'application PTL_Crypto,
je veux que mon état (cryptos sélectionnées et visibles sur le graphique) soit sauvegardé,
afin de retrouver le même affichage au redémarrage de l'application.

[TA] Critères d'acceptation :

- Lors de la fermeture de l'application, l'état courant est enregistré dans un fichier state.json dans le dossier local_data.
- Le fichier contient :
    - les symboles des cryptomonnaies chargées (LoadedCryptos),
    - les symboles des cryptomonnaies visibles (VisibleCryptos).
- Au démarrage, si le fichier state.json existe, l'application :
    - recharge les données locales correspondantes,
    - restaure les éléments affichés dans checkedListBoxCryptos1,
    - affiche automatiquement le graphique précédent.
- Si aucune donnée n'est trouvée, l'application démarre avec les paramètres par défaut (une crypto sur 7 jours).

Objectif technique :
Implémenter un système de persistance d'état à l'aide d'un fichier JSON local et le restaurer automatiquement à chaque ouverture.

4. Réalisation

4.1 Choix techniques

- **Framework** : C# Windows Forms (facile à mettre en œuvre et compatible avec ScottPlot).
- **API externe** : CoinGecko (format JSON).

- **Parsing JSON** : System.Text.Json.
- **Manipulation de données** : LINQ (remplace les boucles for).
- **Graphiques** : ScottPlot (librairie simple et efficace).

4.2 Organisation du code

- Namespace principal : PTL_Crypto.
- Classes :
    - Form1 (point d'entrée de l'application et interface principale, Gère les événements utilisateur, appelle les autres classes).
    - AppState (stockant l'état de l'application entre les sessions).
    - FileClient (lis fichier JSON).
    - CoinInfo ("Coin" information et Override lui à ToString).
    - CryptoPrice (modèle de données).
    - ApiClient (récupération JSON).
    - PlotManager (gestion graphique).
- Méthodes d'extension :
    - Conversion timestamp → DateTime.
    - Conversion liste JSON → séries exploitables par ScottPlot.

5. Tests

Tests unitaires

Les tests unitaires ont été conçus pour valider la fiabilité et la cohérence des différentes étapes de traitement des données dans l'application.
Chaque test correspond à une partie spécifique du flux logique — de la récupération des données jusqu'à leur affichage sur le graphique.

- a. **Vérification du parsing du JSON depuis l'API CoinGecko**
  Objectif : s'assurer que les données téléchargées (format JSON) sont correctement désérialisées en objets `CryptoPrice`.
  Méthode : test de désérialisation simulée à partir d'un échantillon JSON.
  Résultat attendu : chaque objet contient un symbole valide, un prix positif et une date correcte.
  Lien avec l'application : cette étape correspond au chargement initial des données depuis l'API dans la logique principale.

- b. **Vérification de la conversion timestamp → DateTime**
  Objectif : garantir que les timestamps Unix reçus de l'API sont correctement convertis en format `DateTime` lisible par C#.
  Méthode : utilisation d'une méthode d'extension personnalisée (`FromUnixTimestamp`).
  Résultat attendu : les dates sont en ordre chronologique croissant et au format UTC.

Lien avec l'application : cette conversion est essentielle pour l'affichage temporel correct des prix sur le graphique.

- c. **Vérification du filtrage des périodes avec LINQ**
  Objectif : s'assurer que le filtrage des périodes (7, 30, 90 jours) renvoie les bons intervalles de données.
  Méthode : application de requêtes LINQ sur les listes d'objets `CryptoPrice` avec filtrage par `DateTime`.
  Résultat attendu : seules les entrées correspondant à la période sélectionnée sont conservées.
  Lien avec l'application : cette étape correspond au choix de la période d'affichage du graphique par l'utilisateur.

- d. **Vérification de la transformation des données en séries ScottPlot**
  Objectif : contrôler la correspondance entre les données brutes (JSON) et les séries prêtes à être tracées sur le graphique.
  Méthode : simulation d'un graphique à partir de listes de valeurs pour les axes X (temps) et Y (prix).
  Résultat attendu : les points affichés correspondent aux prix réels, dans le bon ordre temporel.
  Lien avec l'application : cette étape valide la préparation des données avant leur affichage visuel via ScottPlot.

- e. **Test de validité et d'ordre temporel des prix**

  Objectif : vérifier que chaque objet `CryptoPrice` contient un symbole valide, un prix positif et que les dates sont dans le bon ordre chronologique.

  Méthode : création d'une liste de données simulées (`BTC` à différents instants) et utilisation des assertions XUnit pour contrôler :

  - que les symboles ne sont pas vides,
  - que les prix sont positifs,
  - que les timestamps ne sont pas dans le futur,
  - que les dates sont bien ordonnées.

  Résultat attendu : toutes les vérifications passent sans erreur, confirmant la cohérence et la validité des données.

  Lien avec l'application : ce test garantit que les données manipulées et affichées dans les graphiques sont logiquement correctes, évitant les anomalies comme des prix négatifs ou des dates inversées.

Tests d'acceptation

Les tests d'acceptation permettent de vérifier que le comportement global de l'application correspond aux attentes de l'utilisateur.

Cas de test	Description	Résultat attendu	Statut
1	Lancer l'application et rechercher "bitcoin"	Le graphique Bitcoin s'affiche (30 jours par défaut)	✅ Réussi
2	Ajouter "Ethereum" et cocher les deux cryptos	Les deux courbes apparaissent distinctement sur le graphique	✅ Réussi
3	Décocher "Ethereum" dans la liste	La courbe Ethereum disparaît, seule Bitcoin reste visible	✅ Réussi
4	Changer la période d'analyse (7 → 30 → 90 jours)	Le graphique se met à jour avec la nouvelle période	✅ Réussi

6. Journal de travail (JDT)

Le Journal de travail est disponible dans JDT.xlsx

Il contient les tâches réalisées à chaque séance, la durée et un commentaire (progrès, difficultés, corrections).

7. Utilisation de l'IA

- Recherche d'API adaptées (CoinGecko, alternatives).
- Reformulation des User Stories.
- Assistance technique pour LINQ et Windows Forms(un des source d'information **théorique**).

Toutes les suggestions ont été vérifiées, adaptées et comprises avant intégration.
IA a servi à structurer le contenu, corriger les fautes d'orthographe et vérifier la cohérence des User Stories, donc les fautes d'orthographe des stories.

8. bilan technique

Le bilan technique permet de faire le point sur les choix technologiques, les méthodes de développement et l'efficacité du code dans le projet Plot Those Lines! (Crypto Edition).

8.1 Choix technologiques

- Langage et framework : C# avec Windows Forms
    - Avantage : simplicité pour créer une interface graphique rapide et intégration facile avec ScottPlot.
- Bibliothèques et outils :
    - ScottPlot pour les graphiques financiers.
    - System.Text.Json pour la désérialisation JSON.
- API externe : CoinGecko
    - Fournit des données fiables et à jour sur les cryptomonnaies.

- Gestion des données : LINQ pour filtrer, transformer et manipuler les collections.
    - Avantage : remplace les boucles for et foreach, offrant un code plus lisible et maintenable.

8.2 Architecture et organisation du code

- Classes principales :
    - **Form1** : point d'entrée de l'application et interface principale.
        - Gère les événements utilisateur (boutons, ComboBox, CheckedListBox).
        - Appelle les classes `ApiClient`, `FileClient`, et `PlotManager` pour charger et afficher les données.
        - Contient la logique de persistance d'état via les méthodes `SaveAppState()` et `LoadAppState()`.
        - Met à jour dynamiquement le graphique selon les cryptomonnaies visibles et les fichiers locaux.
    - **AppState** : modèle simple stockant l'état de l'application entre les sessions.
        - Contient deux listes : `LoadedCryptos` (cryptos chargées) et `VisibleCryptos` (cryptos affichées).
        - Sérialisé au format JSON dans `local_data/state.json` pour permettre la restauration automatique.
    - **ApiClient.cs**
      Contient les appels HTTP asynchrones vers l'API CoinGecko.
        - Chaque requête est précédée d'un appel à `EnforceRateLimitAsync()` pour éviter les erreurs 429.
        - Le parsing des réponses JSON se fait uniquement via LINQ (`Select`, `Where`, `ToList`).
    - **FileClient.cs**
        - Fournit une méthode `LoadPricesFromFile()` pour lire les historiques de prix stockés localement.
        - Utilise également LINQ pour transformer les tableaux JSON en objets `CryptoPrice`.
    - **PlotManager.cs**
        - Centralise les opérations de tracé sur `ScottPlot`.
        - Gère la mise à jour dynamique du graphique en fonction des cryptos visibles.
    - **CryptoPrice** : modèle de données représentant le prix d'une crypto à un instant donné.
    - **CoinInfo** : informations de base sur une crypto (Id, Nom, Symbole).

- Méthodologie LINQ :
    - Filtrage des données (périodes 7, 30, 90 jours).
    - Transformation des données JSON en séries temporelles.
    - Sélection des cryptos visibles dans le graphique.
    - Réduction des boucles classiques pour un code plus concis et fonctionnel.
    - stockage l'état de l'application entre les sessions.

8.3 Gestion des erreurs et robustesse

- Prévoir des fallbacks : si l'API n'est pas disponible, les fichiers JSON locaux sont utilisés.

- Validation des données avant affichage : les données vides ou incorrectes déclenchent des messages utilisateurs.
- Gestion des exceptions dans tous les appels API et fichiers locaux.

8.4 Tests et validation

- Unitaires : vérification de la désérialisation, de la conversion timestamp → DateTime, du filtrage LINQ, et de la cohérence des séries pour ScottPlot.
- D'acceptation : validation des interactions utilisateur (sélection crypto, période, visibilité sur le graphique).

8.5 Performances et maintenabilité

- Utilisation de HashSet pour la gestion des cryptos visibles → accès rapide et filtrage efficace.
- LINQ pour manipuler les collections → code plus lisible, moins d'erreurs liées aux boucles.
- Modularité des classes → facile à maintenir et à étendre (ajout d'une nouvelle crypto ou source de données).

8.6 Difficultés techniques rencontrées

- **Problèmes liés à l'API gratuite CoinGecko**
  La principale difficulté rencontrée a été la **limitation de taux ("429 Too Many Requests")** imposée par l'API gratuite de CoinGecko.
  Comme chaque requête HTTP consomme un quota, l'application dépassait parfois la limite en effectuant plusieurs appels consécutifs (par exemple, lors du rechargement automatique ou de la comparaison de plusieurs cryptomonnaies).
  Pour contourner ce problème, j'ai ajouté une méthode EnforceRateLimitAsync(), qui insère un délai entre chaque appel API. Cette approche garantit le respect du quota sans bloquer l'interface utilisateur, tout en maintenant un comportement asynchrone fluide.

- **Sauvegarde et restauration de l'état utilisateur**
  Une autre difficulté a concerné la gestion du fichier state.json.
  Le principal défi était d'assurer une **sauvegarde fiable** (éviter les corruptions de fichiers en cas de fermeture brutale) et une restauration cohérente des données (vérification de la présence des fichiers locaux avant de les recharger).
  J'ai implémenté la méthode LoadAppState() dans Form1.cs pour charger automatiquement l'état au démarrage, et SaveAppState() pour enregistrer les paramètres lors de la fermeture.

- Conversion correcte des timestamps Unix en DateTime pour l'affichage chronologique.

- Gestion des fichiers JSON importés par l'utilisateur et des séries multiples dans ScottPlot.

- **Compatibilité LINQ complète sans boucles classiques pour toutes les opérations sur les collections**
  J'ai choisi de remplacer toutes les boucles for et foreach par des requêtes LINQ, ce qui a nécessité une réécriture importante du code.
  La difficulté principale a été d'exprimer certaines logiques complexes (comme la mise à jour des listes ou dictionnaires) de manière fonctionnelle, tout en gardant un code lisible et performant.

9. bilan personnel

Ce projet m'a permis de :

- Développer une application complète de visualisation de données.
- Travailler avec une API externe (CoinGecko) et traitement de données JSON.
- Maîtriser LINQ et WinForms pour des traitements de données complexes.
- Mettre en pratique les tests unitaires et d'acceptation, renforçant ma rigueur.
- Générer et afficher des séries temporelles avec ScottPlot.
- Implémentation de tests unitaires et tests d'acceptation réels.

Les principales difficultés ont été :

- Comprendre et appliquer LINQ pour filtrage et transformation des séries.
- Synchroniser les User Stories avec les tests d'acceptation et issues GitHub.

Grâce à la planification et à l'usage de JDT, j'ai pu résoudre ces difficultés progressivement. Ce projet a renforcé mes compétences en programmation et en gestion de projet, tout en améliorant ma capacité à documenter et présenter un projet de manière professionnelle.

10. Conclusion

L'application Plot Those Lines! (Crypto Edition) est complète, intuitive et fiable.
Elle repose sur une architecture claire : chaque classe a un rôle précis et communique avec les autres via des interfaces bien définies.
Le code est 100 % LINQ, ce qui le rend concis, maintenable et conforme aux objectifs pédagogiques du module.
J'ai pu lier théorie et pratique, produire une documentation structurée et assurer la traçabilité entre User Stories, tests et issues GitHub.

11. Références

- CoinGecko API: https://www.coingecko.com/en/api
- Exemple d'interface: https://www.investing.com/charts/live-charts
- ScottPlot: https://scottplot.net
- Documentation Microsoft LINQ: https://learn.microsoft.com/dotnet/csharp/linq
- CoinMarketCap: https://coinmarketcap.com