PPE
Salle de marché

SUK Nathan
BOUAYOUN Ayman
VOULAMANA Johann
NOCETTI Teddy

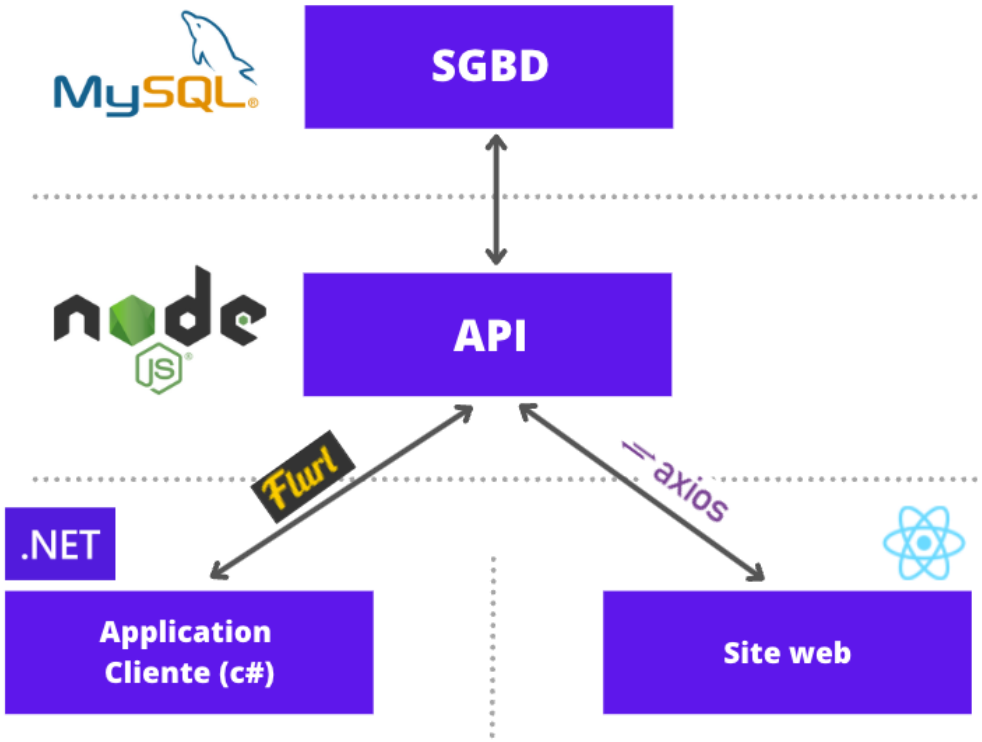
I] Introduction

L'application cliente a été développée avec le langage de programmation orienté objet : C#. Il s'agit d'une application Windows Form développée sous l'environnement Visual Studio. Cette application, devait à l'origine être une application capable de communiquer avec un SBGD traditionnel, récupérer des données, et performer des applications directement dans celui-ci.
Cependant, une application comme telle posait un problème : les informations et les requêtes exécutées auraient été stockées à l'intérieur de l'application cliente. Elles auraient donc été exposées aisément pour tout attaquant ayant des connaissances dans la décompilation d'application en C#, et celui-ci aurait-eu accès aux informations de connexion à la base de données.

Nous avons décidé d'adopter une architecture plus poussée, et de développer une API qui sera l'intermédiaire entre l'application cliente en C# ainsi que l'application web avec le SBGD.
Les applications côté client n'auront qu'à exécuter de manière asynchrone des requêtes en POST et GET, et l'API se chargera d'exécuter les requêtes auprès de la base de données et de retransmettre les informations au format JSON (qui seront traitées côté client grâce aux bibliothèques respectives à chaque langage)

1.1) Avantages

Une seule base de code qui communique avec notre SBGD : Si la base de données venait à changer, les changements à opérer ne seront qu'au niveau de l'API. Sans cela, nous aurions été amenés de maintenir deux applications et d'assurer la cohérence des données et des requêtes sur deux bases de codes distinctes.
Nous nous sommes rapprochés au maximum des standards de développement actuels en utilisant une API. Nous avons donc appris à communiquer avec une API de manière asynchrone en C#, récupérer et traiter une réponse en JSON



II] Documentation technique : Application cliente C#

1.1) Dépendances principales du projet

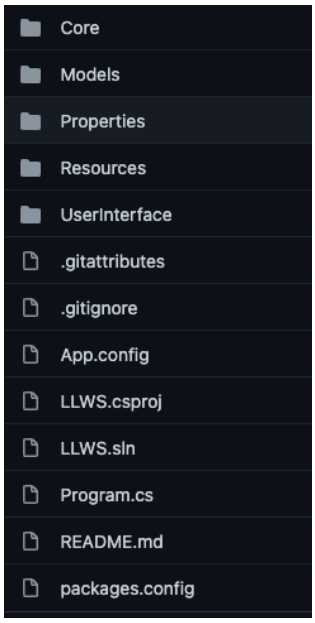
Flurl :
https://flurl.dev

L'application nécessite plusieurs bibliothèques pour fonctionner. La librairie Flurl a été choisie pour consumer l'API. Flurl est open source et se présente comme une surcouche de la librairie native à C# : System.Net.Http.HttpClient. Cette librairie propose de nombreuses fonctions facilitant la communication avec une API via différentes méthodes (GET, POST, PUT, PATCH) avec une syntaxe très lisible.

Newtonsoft.JSON :
https://www.newtonsoft.com/json

Dans un contexte ou la communication avec une API est centrale, il faut pouvoir traiter les données tant pour l'envoi que pour la réception. Newtonsoft.Json est donc un choix judicieux car celle-ci permet de manipuler les objets JSON avec aisance notamment avec la sérialisation et désérialisation de chaînes de caractères.

1.2) Architecture de la base de code



Dossiers :

Core : Contient le(s) fichier(s) permettant le bon fonctionnement global de l'application.

Models : Contient différentes classes qui serviront à structurer et organiser les données envoyées et reçues depuis l'API via la désérialisation de JSON.

Properties : Généré par Visual studio.

Ressources : Contient différentes ressources pour l'application, principalement des images ou d'autres types de médias.

UserInterface : Ce dossier, subdivisé en plusieurs autres, contient tous les fichiers relatifs à l'interface utilisateur.

Racine :

.gitattributes / .gitignore : fichiers permettant le bon fonctionnement du versioning GitHub.
App.config : contient différentes variables globales accessibles au sein de l'application.
LLWS.csproj / LLWS.sln : sont les fichiers de la solution Visual Studio à ouvrir dans l'IDE.
Program.cs : point de départ lors du démarrage de l'application
Packages.config : contient la liste des dépendances du projet, qui seront installées lors d'une première utilisation avant le développement.

1.2.1) Dossier Core

- APIManager.cs : Il s'agit d'un point central de l'application. Ce fichier contient une classe statique appelée APIManager qui contient différentes propriétés et 2 méthodes.

Les propriétés de cette classe sont des chaînes de caractères qui contiennent chacune une URL de l'API qui sera accessible pour effectuer différentes opérations.

```
public static string API_BASE_URL = ConfigurationManager.AppSettings["lienApi"];
public static HttpClient httpClient;


//Authentification
public static string API_ROUTE_LOGIN = API_BASE_URL + "/login";
public static string API_ROUTE_REGISTER = API_BASE_URL + "/register";


//Cotations
public static string API_ROUTES_GET_ALL_COTATIONS = API_BASE_URL + "/cotations/get";
public static string API_ROUTES_GET_HISTORIQUE = API_BASE_URL + "/cotations/historique/";

//Utilisateur
public static string API_ROUTES_GET_USER_PORTEFEUILLE = API_BASE_URL + "/users/portefeuille/";
public static string API_ROUTES_POST_SELL = API_BASE_URL + "/cotations/sell";
public static string API_ROUTES_POST_BUY = API_BASE_URL + "/cotations/buy";
public static string API_ROUTES_GET_MOUVEMENT_BUY = API_BASE_URL + "/users/" + User.id.ToString() + "/mouvements/1";
public static string API_ROUTES_GET_MOUVEMENT_SELL = API_BASE_URL + "/users/" + User.id.ToString() + "/mouvements/2";

//Admin & responsable
public static string API_ROUTES_GET_USERS = API_BASE_URL + "/admin/users";
public static string API_ROUTES_GET_SPECIFIC_USER = API_BASE_URL + "/admin/user/"; // L'appel devra se faire avec une valeur d'id

public static string API_ROUTES_POST_PROMOTE = API_BASE_URL + "/responsable/user/promote";
public static string API_ROUTES_POST_REVOKE = API_BASE_URL + "/responsable/user/revoke";

public static string API_ROUTES_POST_SUSPENSION = API_BASE_URL + "/responsable/user/suspension";
public static string API_ROUTES_POST_REHABILITATE = API_BASE_URL + "/responsable/user/rehabilitate";

public static string API_ROUTES_POST_ADDBUDGET = API_BASE_URL + "/responsable/user/addBudget";
public static string API_ROUTES_POST_EDITUSER = API_BASE_URL + "/responsable/user/edit";
```

Cette classe est statique. Selon la documentation de Microsoft : « Une classe statique est fondamentalement identique à une classe non statique, à une différence près : une classe statique ne peut pas être instanciée. En d'autres termes, vous ne pouvez pas utiliser l'opérateur new pour créer une variable du type classe. Étant donné qu'il n'y a aucune variable d'instance, vous accédez aux membres d'une classe statique en utilisant le nom de classe lui-même ». De ce fait, lorsque cette classe sera appelée dans d'autres fichiers du projet il n'y aura pas besoin de nouvelle instance de la classe.
Si je veux récupérer la chaîne de caractère qui correspond à l'url de l'API s'occupant de poster des informations pour connecter un utilisateur il suffira d'appeler la classe de cette manière : `string url = APIManager.API_ROUTE_LOGIN`

Cette classe possède également 2 méthodes : `posterData()` et `recevoirData()`, qui sont le cœur de la communication avec l'API. Ces deux méthodes permettent respectivement de poster et récupérer des données sur une route de l'API. `recevoirData()` et `posterData()` prennent en paramètre la route de l'API sur laquelle effectuer une action. posterData() prend également un objet JSON quelconque qui sera posté sur l'URL renseignée.
À l'intérieur de ces méthodes, il s'agit simplement de récupérer la réponse de l'API qui sera traitée dans différents cas de figure.

```
/// <summary>
/// Poste des données en JSON sur une route de l'Api
/// </summary>
/// <param name="route">Route de l'API à appeler (POST uniquement)</param>
/// <param name="json">Objet JSON à poster</param>
/// <returns></returns>
public static async Task<JToken> posterData(string route, object json)
{

    //Les informations sont postéee
    var responseString = await route
    .PostUrlEncodedAsync(json)
    .ReceiveString();

    //On naviague à travers la réponse Json
    JToken token = JToken.Parse(responseString);

    return token;

}


/// <summary>
/// Appelle une route de l'Api et récupère la réponse retournée.
/// </summary>
/// <param name="route">Route de l'API à appeler (GET uniquement)</param>
/// <returns></returns>
public static async Task<JToken> recevoirData(string route)
{

    var responseString = await route
    .GetStringAsync();

    //On naviague à travers la réponse Json
    JToken token = JToken.Parse(responseString);
    return token;

}
```

Il est important de noter que ces deux fonctions sont asynchrones. Selon la documentation de Microsoft : « *Le comportement asynchrone est essentiel pour les activités qui sont potentiellement bloquantes, par exemple l'accès au web. L'accès à une ressource Web est parfois lent ou différé. Si cette activité est bloquée dans un processus synchrone, toute l'application doit attendre. Dans un processus asynchrone, l'application peut poursuivre une autre tâche qui ne dépend pas de la ressource Web jusqu'à ce que la tâche potentiellement bloquante soit terminée.* ». Une fonction asynchrone appelée, ne va donc pas bloquer notre application durant son traitement. L'asynchronicité est le moyen idéal pour communiquer avec une API, car en attendant la réponse de celle-ci, le reste de l'application n'est pas bloqué. Ces méthodes sont déclarées avec le terme « async », et doivent contenir en leur sein une « attente » de quelque chose qui se traduit par le mot clé « await ».

En reprenant le code de `recevoirData()` ci-dessus, on peut facilement comprendre le comportement de la fonction. La réponse de l'API est attendue, celle-ci est récupérée depuis la route passée en paramètre. La librairie Flurl intervient et permet d'appeler sur la route la méthode `GetStringAsync()`, qui signifie que l'on souhaite récupérer la réponse sous forme de chaîne de caractères de manière asynchrone.
Puis, la réponse reçue est analysée et retournée en tant que JToken (grâce à la librairie Newtonsoft.Json).
La fonction posterData() fonctionne de la même manière en utilisant cette fois une autre méthode de Flurl (`PostUrlEncodedAsync()`) qui poste l'objet JSON en paramètre sur l'url de l'api et attend une réponse.

La compréhension du mécanisme ci-dessus, permet alors de comprendre le fonctionnement du projet dans sa globalité. Cette classe est instanciée à chaque endroit de l'application où une action sur l'API est effectuée.
Un exemple concret d'utilisation :

Il y a un bouton de l'interface graphique permettant de rediriger l'utilisateur vers la liste des cotations.
En cliquant sur ce bouton, un appel de APIManager est fait, requérant l'URL de l'API s'occupant de la récupération des cotations, ainsi que la fonction recevoirData() qui prendra cette url en paramètre. Lorsqu'une réponse est reçue, elle est traitée, puis le composant graphique (Windows Form) chargera la réponse. Ce mécanisme de chargement sera explicité dans la partie sur l'interface utilisateur.

1.2.2) Dossier Models

Ce dossier contient différentes classes, qui seront utilisées lors de la désérialisation de JSON.
« Le processus par lequel un format de niveau inférieur (par exemple, qui a été transféré sur un réseau ou stocké dans un magasin de données) est traduit en un objet lisible ou une autre structure de données. ». Par exemple, si je souhaite créer un objet Cotation depuis une réponse JSON, je vais désérialiser la réponse reçue en tant que chaîne de caractères pour qu'elle se « transforme » en objet plus facilement manipulable en c#.
Désavantage : les propriétés de cette classe doivent strictement correspondre aux propriétés du tableau JSON, ce qui explique le non-respect des bonnes pratiques de nommage des propriétés d'une classe.
Une désérialisation s'effectue de la manière suivante (grâce à la librairie Newtonsoft.Json) :

Objet ciblé :

```
class Cotation
{
    public int id { get; set; }
    public string isin_code { get; set; }
    public DateTime stock_date { get; set; }
    public double stock_opening_value { get; set; }
    public double stock_closing_value { get; set; }
    public double stock_highest_value { get; set; }
    public double stock_lowest_value { get; set; }
    public int stock_volume { get; set; }
    public string full_name { get; set; }
    public string ticker_code { get; set; }
}
```

Réponse JSON reçue :

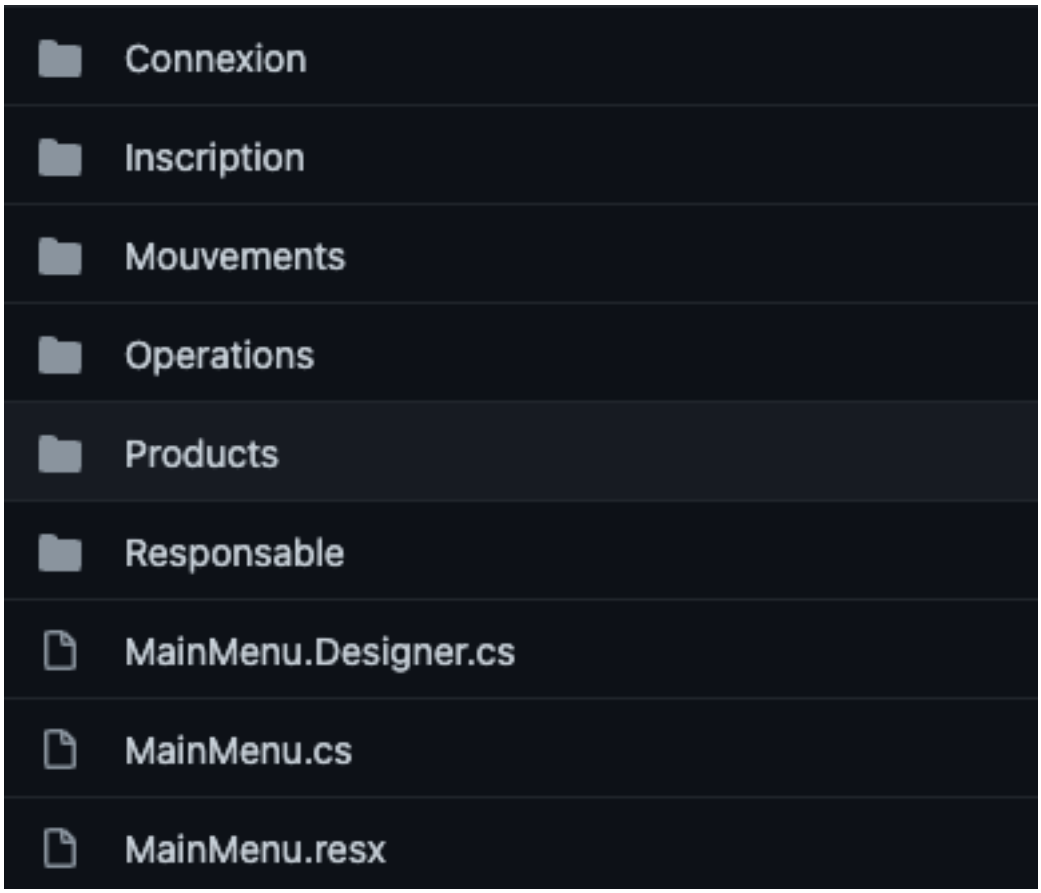
Désérialisation :

```
var historiqueCotation = JsonConvert.DeserializeObject<List<Cotation>>(reponse.SelectToken("result").ToString());
```

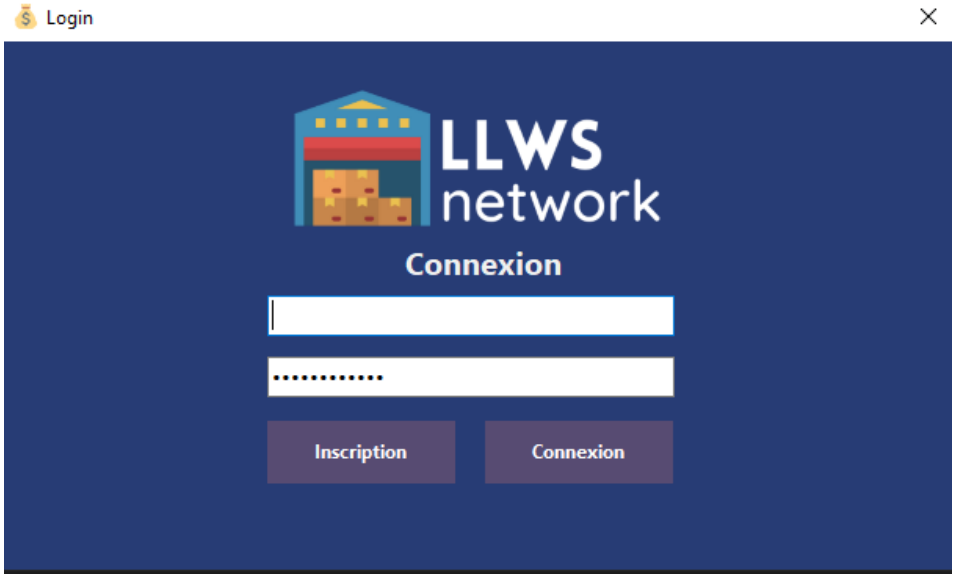
Dans la variable historiqueCotation sera stockée une liste d'objets « Cotation » récupérée depuis la réponse de l'API en chaîne de caractères.
La syntaxe est la suivante :
JsonConvert.DeserializeObjet<Type De Conversion>(réponse API).

1.2.3) Dossier UserInterface

Ce dossier est subdivisé en plusieurs autres dossiers, chacun correspondant à une fenêtre de l'interface utilisateur :

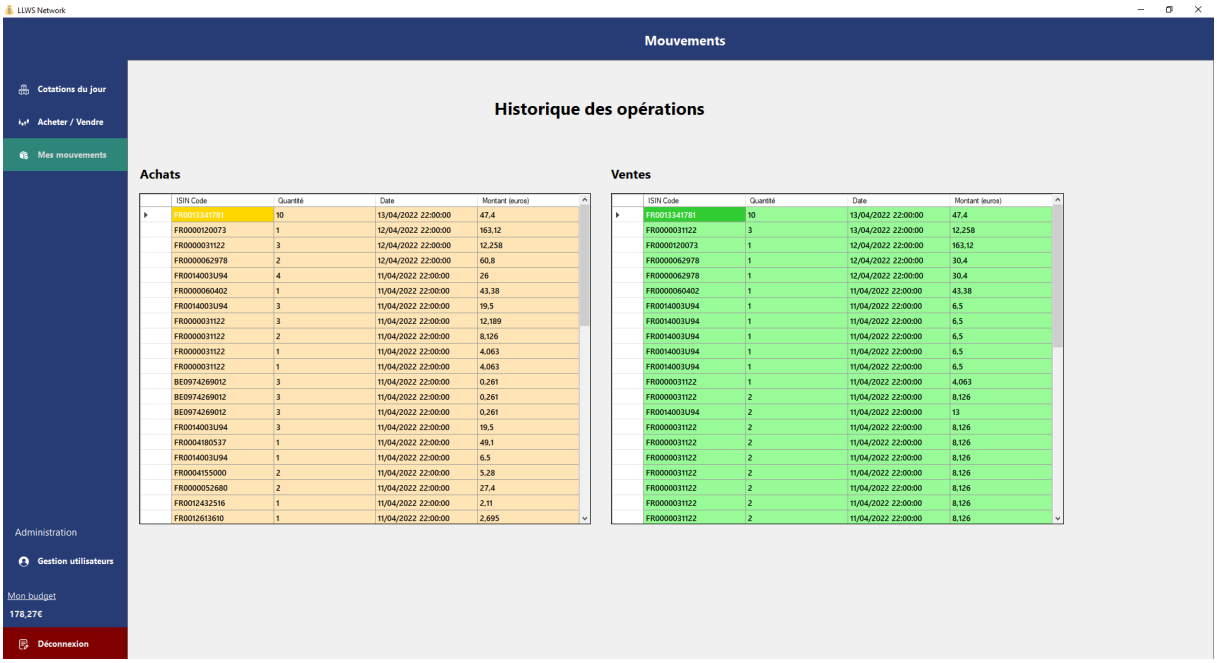


Connexion : Fenêtre de connexion.

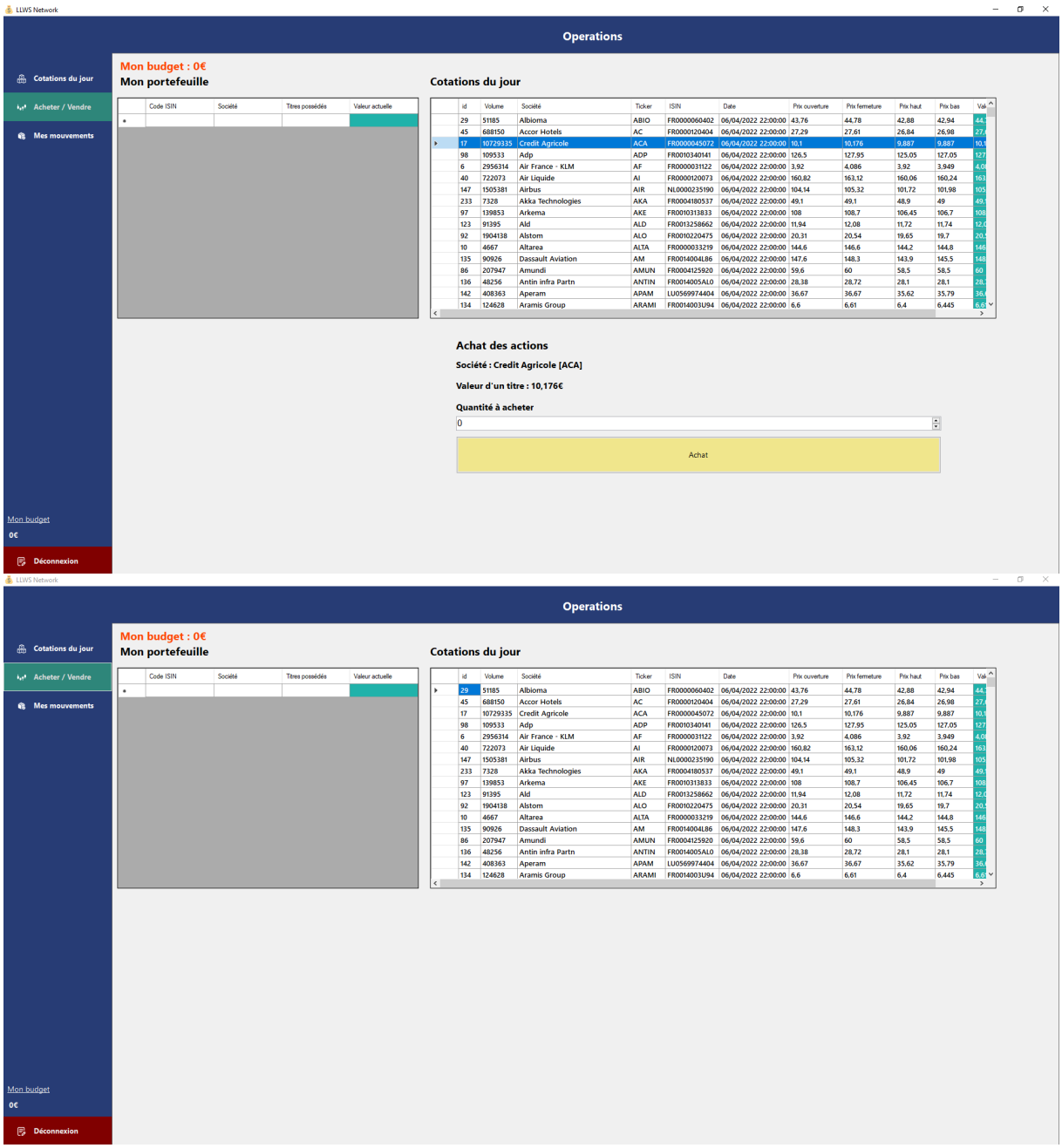


Inscription : Fenêtre d'inscription

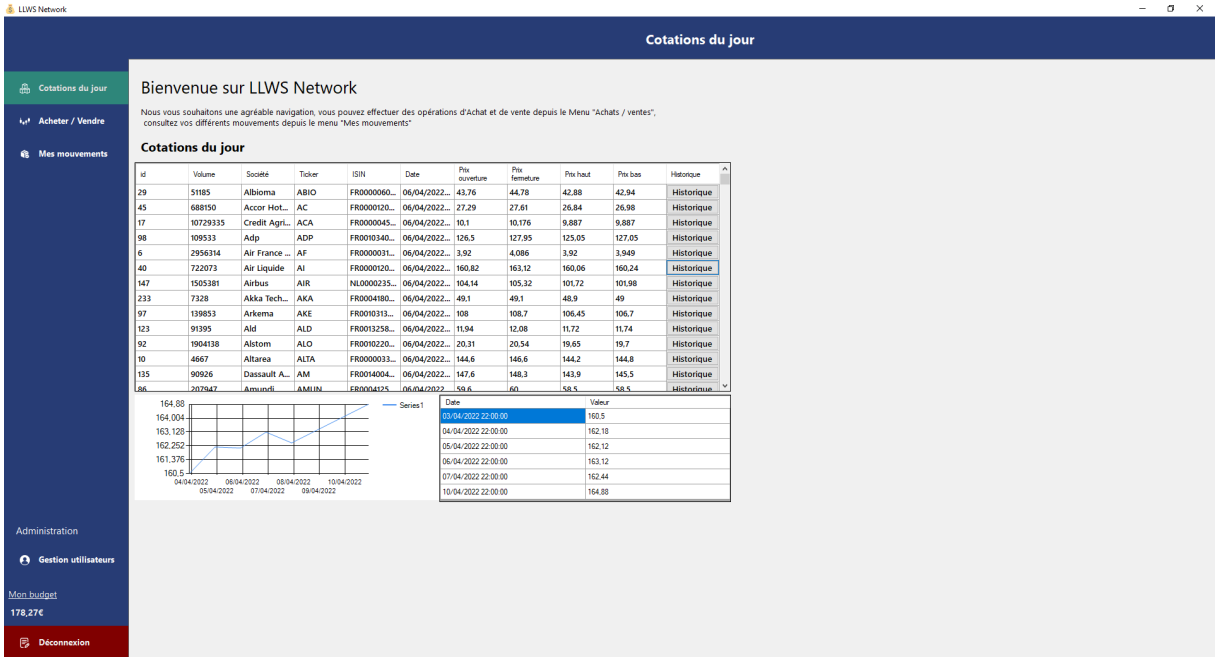
Mouvements : Fenêtre des mouvements (achats/ventes) de l'utilisateur.



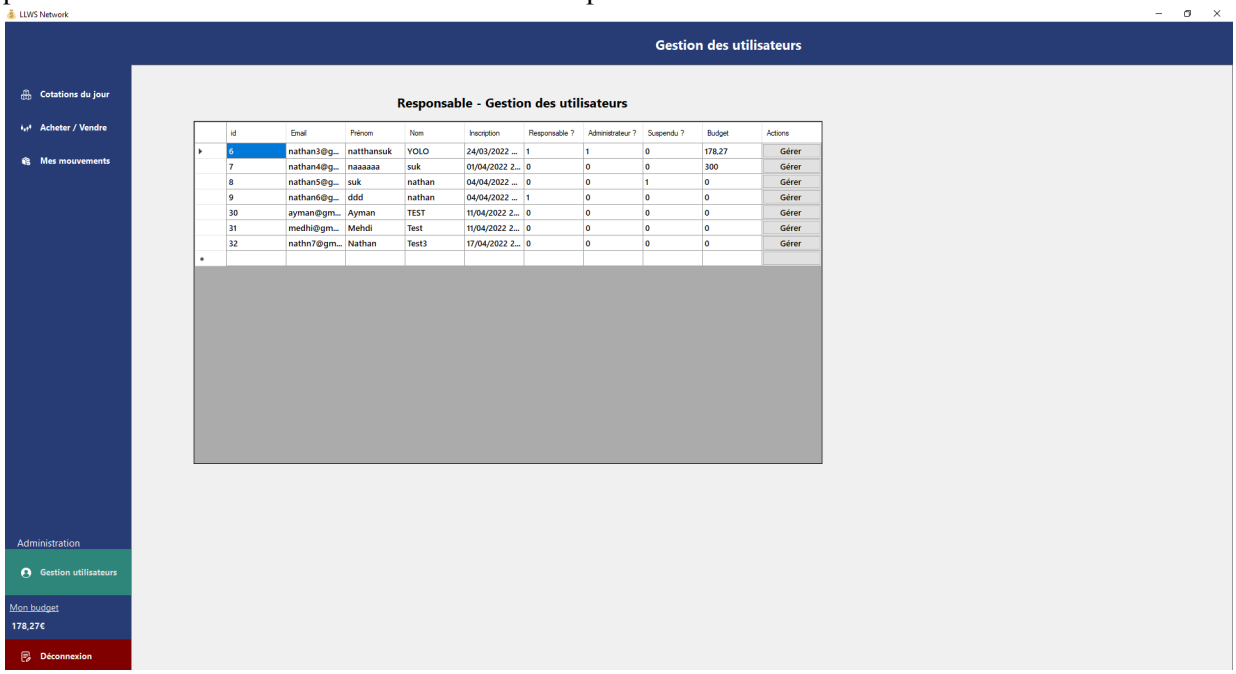
Opérations : Fenêtre affichant le portefeuille d'actif de l'utilisateur (à gauche) qu'il peut revendre au cours actuel, et à droite la liste des cotations du jour qu'il peut choisir d'acheter.

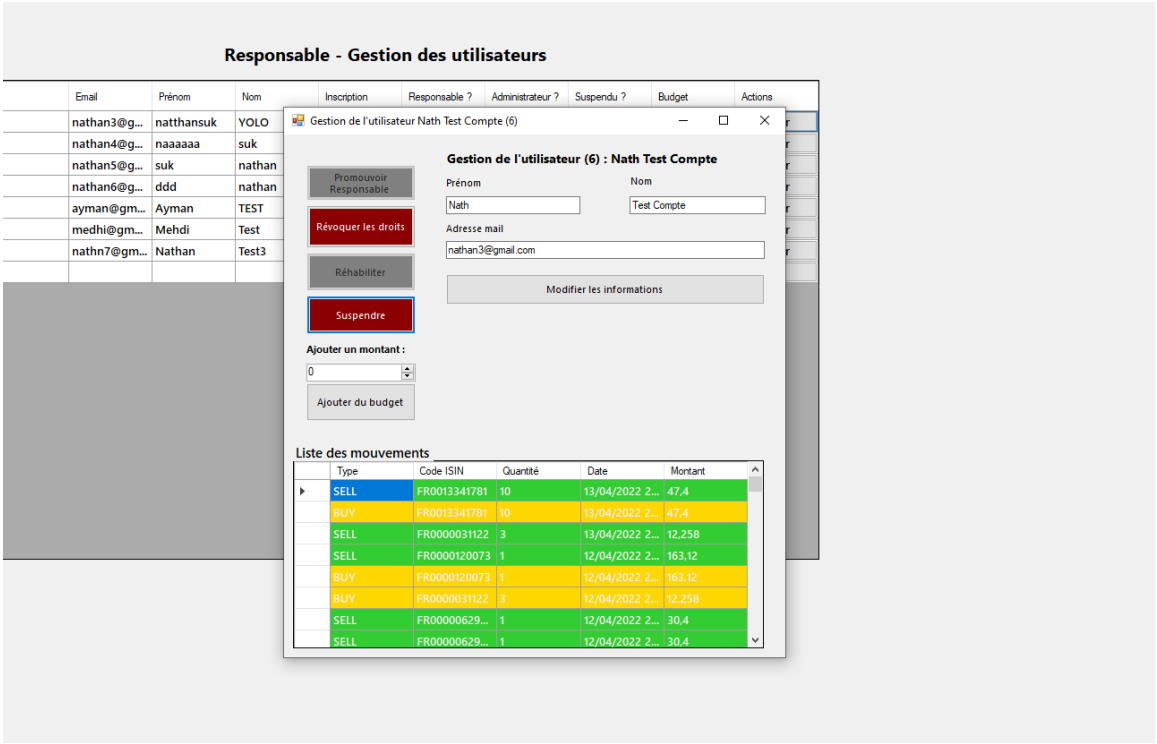


Products (page d'accueil après connexion) : Fenêtre affichant les cotations du jour, et un historique de l'évolution du prix de l'action depuis les 5 derniers jours.



Responsable : Fenêtre affichant la liste de tous les utilisateurs enregistrés, avec un bouton permettant d'effectuer différentes actions sur son profil :

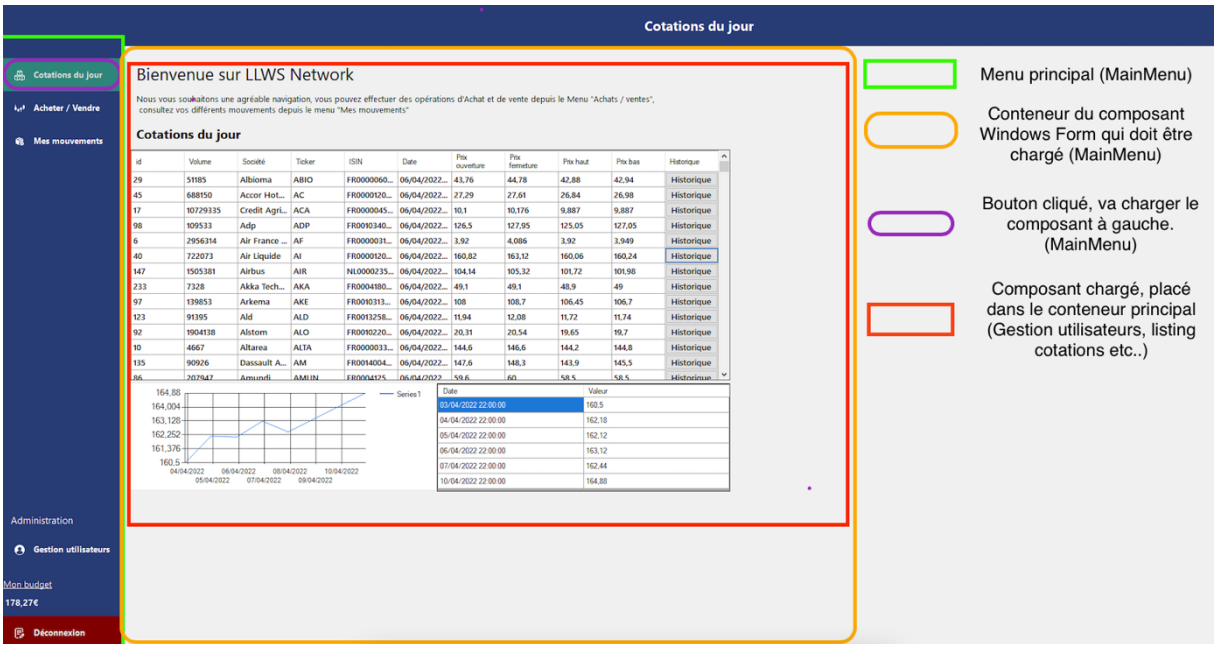




Conception globale de l'interface utilisateur :

Ici, les fenêtre d'inscription et de connexion seront mises de côtés car elles sont indépendantes de la fenêtre de navigation principale pour les utilisateurs connectés.

La fenêtre principale se déconstruit en 2 parties :
La partie de gauche, qui est le menu, composé de différents boutons.
La partie de gauche qui affiche le composant Windows Form correspondant au bouton cliqué.



L'affichage d'un composant se fait de la manière suivante :

Lorsqu'un clic est fait sur un des boutons des menu, une fonction OpenActiveForm() est appelée :

```
private void btnProducts_Click(object sender, EventArgs e)
{

    OpenActiveForm(new Products(), sender);

}
```

Cette fonction prend en paramètre la nouvelle instance d'une fenêtre que l'on souhaite ouvrir (ici Products) ainsi que le bouton qui envoie l'information (Sender).
Cette fonction OpenActiveForm, va chercher le composant demandé (dans le dossier UserInterface) puis va l'instancier en tant qu'« enfant » de la fenêtre principale, et va afficher ce composant dans le conteneur au milieu.

Utilité : Si l'on souhaite ajouter une nouvelle fonctionnalité qui requiert le développement d'une nouvelle fenêtre d'interface, il n'est pas nécessaire de recréer le menu à chaque fois.

Ajout d'une nouvelle fenêtre : Il suffit de créer une Form classique (ajout de code, logique, et composants graphiques), de le placer dans UserInterface dans un dossier nommé correctement. Puis, créer un bouton sur la fenêtre principale (MainMenu), et d'implémenter l'évènement « Click » qui appellera la fonction OpenActiveForm() et qui prendra en paramètre la nouvelle fenêtre créée.

Boutons avec évènement asynchrone

Certains boutons du Menu ont une implémentation de logique exécutée de manière asynchrone lorsque l'évènement « Click » est déclenché. Ceci se réfère à la partie sur la communication asynchrone avec l'API.

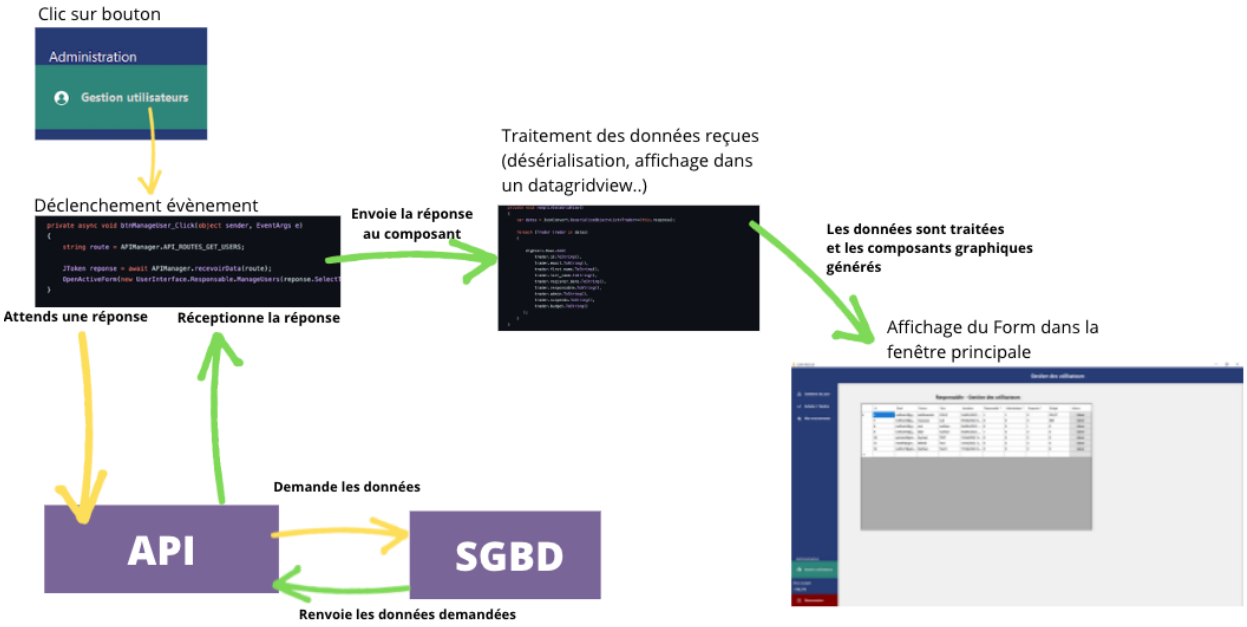
Exemple : Pour un administrateur, un bouton « Gestion des utilisateurs » est affiché.
En cliquant sur ce bouton un appel est fait à l'API.

```
private async void btnManageUser_Click(object sender, EventArgs e)
{
    string route = APIManager.API_ROUTES_GET_USERS;

    JToken reponse = await APIManager.recevoirData(route);
    OpenActiveForm(new UserInterface.Responsable.ManageUsers(reponse.SelectToken("result").ToString()), sender);
}
```

On souhaite recevoir les données depuis la route « API_ROUTES_GET_USER » qui retourne la liste de tous les utilisateurs en base de données.
Une fois la réponse récupérée, on appelle la fonction d'affichage du composant, puis nous passons en paramètre du constructeur la réponse reçue par l'API (qui sera traitée au sein de la logique du composant)

Voici un schéma du cycle d'interaction entre un clic sur un bouton du menu et la consommation de l'API :



Système d'authentification

L'authentification se fait par adresse électronique et mot de passe.
Le mot de passe est encrypté à l'aide de BCrypt dans la base de données.
Le système d'authentification a nécessité un travail de recherche afin de trouver une solution qui vient combler l'absence de système de session natif à C#. Contrairement à PHP, C# ne stocke pas de session ni de cookies à proprement parler. La solution la plus viable était de stocker un utilisateur en mémoire. Une nouvelle fois, une classe statique a été créée et celle-ci va stocker l'utilisateur de la session en cours si la connexion est réussie.
Cette classe statique est instanciée avec des propriétés nulles dès l'ouverture de l'application.

```
/// <summary>
/// Cette classe stocke l'utilisateur récupéré lors de la connexion (session)
/// Elle est statique dont accessible tout au long du cycle de vie de l'application
/// Cette classe statique permet de créer une "Session" ou les informations de l'utilisateur de l'application est stockée.
/// </summary>
public static class User
    {
        public static int id { get; set; }
        public static string email { get; set; }
        public static string password { get; set; }
        public static string first_name { get; set; }
        public static string last_name { get; set; }
        public static DateTime register_date { get; set; }
        public static int responsable { get; set; }
        public static int admin { get; set; }
        public static int suspendu { get; set; }
        public static object dateDebut { get; set; }
        public static object dateFin { get; set; }
        public static double budget { get; set; }

        /*
         * Le token de connexion est central, et permettra de vérifier qu'il est bien connecté mais également
         * de voir si il peut performer certaines actions.
         */
        public static string userToken { get; set; }

    }
```

Problématique et approche

Au départ, et comme dans beaucoup de système, un utilisateur est reconnu selon plusieurs critères :

- Son adresse électronique qui est considérée comme unique (propriété « email »).
- Son id considéré comme unique, puisqu'il est attribué automatiquement lors de la création d'un nouvel utilisateur en base de données (propriété « id »).

On suppose que seulement ces deux propriétés sont uniques, puisqu'une autre personne peut porter le même nom ou dans un cas très rare avoir le même mot de passe par exemple.
Si l'on revient au fonctionnement de l'application et la communication avec l'API, on sait que chaque requête est faite avec une méthode différente selon les besoins. Mais lors de l'envoi d'une requête il faut identifier qui en est l'auteur ou qui est la cible ;
Exemple : lors de l'achat d'un titre en bourse, il faut pouvoir déterminer qui est l'acheteur, quel est son portefeuille d'actif dans la base de données.

On pourrait donc envoyer à l'API l'identifiant de l'utilisateur ou son courriel afin de reconnaître qui est à l'origine de l'envoi.
Verbeusement cela donnerait : « *Je suis l'utilisateur numéro **1** dans la table utilisateur, achète **1** titre chez Boursorama* ».

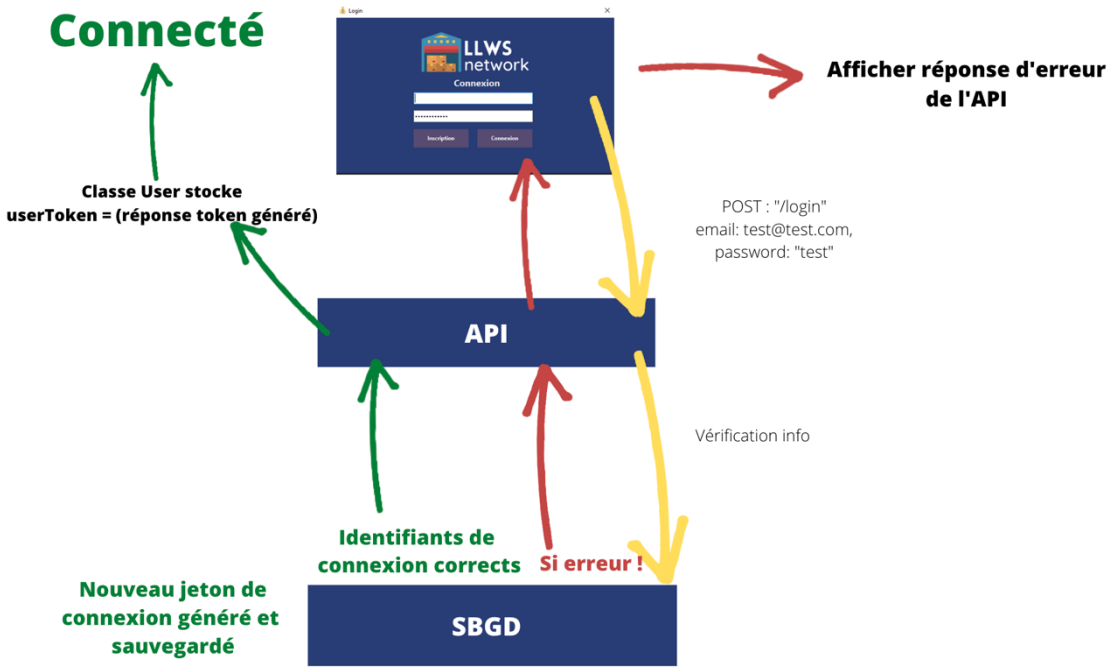
Cette méthode n'est pas fiable : Si un attaquant effectuait une requête à l'API en remplaçant l'id par un entier aléatoire, il y en aurait supposément 1 qui correspondrait à un utilisateur dans la base de données, et par conséquent il pourrait effectuer une action sans permission à la place d'un utilisateur.

La méthode la plus efficace fut de créer un moyen de communiquer avec l'API de manière sécurisée, avec une propriété unique d'un utilisateur et qui peut être regénérée. Nous avons donc développé au sein de l'API un « générateur » de jeton de connexion, haché à l'aide de BCRYPT qui, à chaque nouvelle instance de connexion réussie, va générer un nouveau jeton de connexion (chaîne de caractère composé de symboles, lettres, chiffres et salage) et va stocker celui-ci dans la table utilisateur dans « loginToken » au sein du SGBD.

Côté client, ce jeton de connexion sera également stocké dans l'utilisateur de session (classe statique User) dans la propriété « userToken », et lorsque la session est active, le jeton de connexion sera utilisé pour identifier l'utilisateur lors d'une requête (il sera envoyé à l'API dans le corps de la requête).

Le système d'authentification peut être retrouvé côté client dans le fichier UserInterface/Login.cs.

À noter : d'autres propriétés sont stockées dans l'utilisateur de session comme son budget, son id, ses permissions (admin / responsable), notamment pour afficher les menus cachés de gestion.
Cependant, nous sommes conscients que si un utilisateur arrive à changer la classe statique et donc changer les propriétés en admin = 1 ou responsable = 1, il aurait potentiellement accès aux menus cachés. C'est pourquoi nous avons également sécurisé les routes de l'API et ceci sera expliqué au cours de la documentation de l'API.



Consigne d'envoi de requête

Il faut **toujours** envoyer, lors d'une requête vers l'API, le jeton de session en cours de l'utilisateur en appelant la classe statique : User.userToken (chaîne de caractères), l'API se chargera du traitement et de vérifier la session en cours, l'utilisateur correspondant et si besoins les droits d'accès.

III] Documentation technique : API NodeJS

« Une API, ou interface de programmation d'application, est un ensemble de définitions et de protocoles qui facilite la création et l'intégration de logiciels d'applications. »

Afin de compléter la documentation technique sur l'application cliente, il est nécessaire d'expliquer l'intermédiaire de communication entre le SGBD et l'application cliente (Windows Form et site web).

« Les API sont parfois considérées comme des contrats, avec une documentation qui constitue un accord entre les parties : si la partie 1 [client] envoie une requête à distance selon une structure particulière, le logiciel de la partie 2 [Api] devra répondre selon les conditions définies. »

Pourquoi NodeJS ?

Node.js est un environnement serveur qui permet d'exécuter des serveurs Web développés en JavaScript. Node.js utilise le même moteur JavaScript que Chrome et Edge, le moteur V8, développé par Google.

Node.js utilise certes du JavaScript, mais l'environnement est très différent du Front-End : il n'y a pas de DOM et de Web API, en revanche il utilise de nombreuses librairies développées en C très performantes comme libuv pour la gestion des événements et des modules pour la cryptographie, l'accès au système de fichiers.

Le choix de NodeJS s'est donc fait sur deux critères principaux :

- La popularité : à ne pas négliger, plus une technologie est populaire, plus les librairies tierces et la communauté sont développées et de ce fait, les réponses aux différents problèmes pendant la phase de développement sont trouvées plus rapidement.
- Javascript : NodeJS utilise le Javascript, ce qui permet d'avoir un langage simple d'utilisation, et qui peut très facilement s'adapter à de nouveaux ajouts dans l'écosystème de notre projet (site web, applications mobiles etc..)

Dépendances principales de l'API

Afin de créer une API performante, nous avons utilisé différentes librairies qui vont permettre la gestion de différentes composantes essentielles.

Express.js
https://expressjs.com/fr/

« Express est une infrastructure d'applications Web Node.js minimaliste et flexible qui fournit un ensemble de fonctionnalités robuste pour les applications Web et mobiles. »

Moment.js
https://momentjs.com/

« MomentJS est un framework JavaScript autonome et open-source qui enveloppe les objets de type date et élimine les objets de type date natifs de JavaScript, qui sont lourds à utiliser. Moment. js facilite l'affichage, le formatage, l'analyse, la validation et la manipulation des dates et de l'heure grâce à une API propre et concise. »

MySQL
https://www.npmjs.com/package/mysql

« Mysql est un driver MySQL basique pour Node.js écrit en javascript et ne nécessitant pas de compilation. Il s'agit de la solution la plus simple et rapide à mettre en place pour interagir avec une base de données MySQL en Node. »

Dotenv
https://www.npmjs.com/package/dotenv

« Dotenv est un module à dépendance zéro qui charge les variables d'environnement depuis un fichier .env dans une variable globale : process.env. Le stockage de la configuration dans l'environnement séparément du code est basé sur la méthodologie des douze facteurs de l'application. »

Bcrypt
https://www.npmjs.com/package/bcrypt

« bcrypt est une fonction de hachage créée par Niels Provos et David Mazières. Elle est basée sur l'algorithme de chiffrement Blowfish et a été présentée lors de USENIX en 19991. En plus de l'utilisation d'un sel pour se protéger des attaques par table arc-en-ciel (rainbow table), bcrypt est une fonction adaptative, c'est-à-dire que l'on peut augmenter le nombre d'itérations pour la rendre plus lente. Ainsi elle continue à être résistante aux attaques par force brute malgré l'augmentation de la puissance de calcul. »

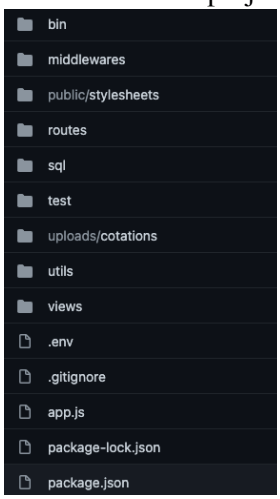
Mocha
https://mochajs.org/

« Mocha est un cadre de test JavaScript riche en fonctionnalités fonctionnant sur Node.js et dans le navigateur, rendant les tests asynchrones simples et amusants. Les tests Mocha s'exécutent en série, ce qui permet d'établir des rapports flexibles et précis, tout en faisant correspondre les exceptions non capturées aux bons cas de test.»

Supertest
https://www.npmjs.com/package/supertest

« La motivation de ce module est de fournir une abstraction de haut niveau pour tester des requêtes HTTP, tout en permettant de descendre au niveau inférieur de l'API grâce à superagent. »

Arborescence du projet



Middlewares : Contient la liste des middlewares utiles pour les routes de l'application

Routes : contenues dans plusieurs fichiers, les routes ouvrent une porte aux requêtes à exécuter depuis un logiciel externe.

SQL : Contient les différents scripts SQL pour mettre à jour la base de données.

Test : contient les fichiers utilisant mocha / supertest pour tester l'API

Utils : contient un ensemble de fonctions réutilisables à travers l'application (calculs, autres utilitaires …)

- **Dossier middlewares :** « Les fonctions de middleware sont des fonctions qui peuvent accéder à l'objet Request (req), l'objet response (res) et à la fonction middleware suivant dans le cycle

demande-réponse de l'application. La fonction middleware suivant est couramment désignée par une variable nommée next.
Les fonctions middleware effectuent les tâches suivantes :

Exécuter tout type de code.
Apporter des modifications aux objets de demande et de réponse.
Terminer le cycle de demande-réponse.
Appeler la fonction middleware suivant dans la pile.
Si la fonction middleware en cours ne termine pas le cycle de demande-réponse, elle doit appeler la fonction next() pour transmettre le contrôle à la fonction middleware suivant. Sinon, la demande restera bloquée.

Ici, les middlewares contiennent notamment des fonctions permettant de reconnaître les droits et permissions d'un utilisateur avant d'accéder aux fonctions associées à une route.
Ces middlewares seront « attachés » aux routes en ayant besoin.
Par exemple, accéder à la liste des utilisateurs en base de données n'est accessible qu'aux responsables ou administrateurs.
Nous avons donc codé un middleware de cette forme :

```
const userHasRole = (role) => {

    return function(req, res, next){
        //Avant de passer à cette étape, on vérifie que le middleware précédent est valide
        isSessionTokenValid(req, res, function (){
            //Si le middleware nous envoie bien à la suite (next) on continue
            console.log('ARRIVE DANS MIDDLEWARE 2')
            //On recherche de nouveau l'utilisateur avec le login token
            db.query("SELECT * FROM user WHERE loginToken = ?", [req.token], (err, result) =>{

                if(result.length > 0){
                    //On regarde la permission demandée dans le paramètre 'role'
                    if(role === 'admin' && result[0]['admin'] === 1 || role === 'responsable' && result[0]['responsable'] === 1){
                        return next()
                    } else {
                        //Sinon on retourne une erreur
                        res.json({
                            status: "ERROR",
                            message: "Vous n'avez pas la permission pour cela."
                        })
                    }
                }
            })
        })
    }
};
```

Premièrement, le middleware vérifie que le jeton de session est valide (correspondance avec un utilisateur en base de données). Puis grâce à ce jeton (valide uniquement), nous récupérons l'utilisateur en table puis nous vérifions son rôle. Si le rôle correspond au rôle demandé en paramètre de la fonction alors les droits lui sont accordés (next())

- **Dossier routes :**
Une route au sein d'une API est un chemin spécifique à emprunter pour obtenir des informations ou des données spécifiques. Traditionnellement, il est recommandé de créer un fichier .js par catégorie de routes, par exemple, toutes les routes correspondant aux actions faisables par un utilisateur sont définies dans le fichier routes/user.js.

Cotations.js : contient les routes permettant d'obtenir les cotations du jour, de les renouveler.
Index.js : Contient les routes liées à l'authentification (inscription, connexion)
Responsable.js : routes accessibles avec les utilisateurs possédant le rôle « responsable » (routes protégées par le middleware userHasRole('responsable).

Admin.js : routes accessibles avec les utilisateurs possédant le rôle « admin » (routes protégées par le middleware userHasRole('admin').

- **Dossier test :**
  Contient tous les fichiers de tests de l'API. De la même manière que pour les routes, il faut créer un fichier par catégorie de routes afin de ne pas avoir de grands fichiers de tests.
  Pour créer un test d'une route, il faut utiliser les librairies Mocha et SuperTest de cette manière :

```
describe('POST /login', function() {
    it('Réponds avec une erreur en format json (Aucun utilisateur trouvé)', function(done) {
        request(app)
            .post('/login')
            .send({email: 'ddddd', password: 'ddddd'})
            .set('Accept', 'application/json')
            .expect(200, {
                status: 'ERROR',
                message: "Aucun utilisateur n'a été trouvé."
            }, done)
    });
});
```

  Il faut donner un titre explicite au test effectué, définir le type de requête, les données à envoyer (si post/patch/put) et donner le résultat escompté.
  Pour lancer un test, il suffit simplement de taper la commande « Mocha » dans l'invite de commande du dossier, les tests s'exécuteront un à un.

- **Dossier uploads**

  Le dossier uploads contient les différents médias téléchargés sur le serveur. Dans notre cas, l'API se charge de stocker les fichiers .txt (ou .csv) des cotations journalières afin d'en extraire le résultat et de sauvegarder une copie.

- **Dossier utils**
  Contient un ensemble de fonction exportables au sein de toute l'API. Il y a notamment databaseConnection.js qui contient les informations de connexion à la base de données (exportée sous le nom « db »)

Installation de l'API

Premièrement il faut installer NodeJS depuis :
https://nodejs.org/en/

Il faut également posséder un serveur MySQL actif, sur un hébergeur quelconque ou bien via XAMPP, MAMP, WAMP etc..
https://www.wampserver.com/

Pour installer l'API, il suffit de télécharger le projet sur Github (ou le cloner) :
https://github.com/LLWSPPE/node-api/

Puis d'ouvrir le projet au sein d'un IDE, et via l'invite de commande se rendre à la racine du projet.

Il suffira ensuite d'exécuter la commande « npm install » qui lancera un processus d'installation automatique des dépendances dont le projet à besoins (qui seront stockées dans le fichier node_modules).
Lorsque l'installation est terminée, il faut exécuter la commande « npm update » afin de s'assurer que tous les modules soient bien à jour.
Enfin pour démarrer le serveur, il faut exécuter la commande : npm start (ou npm run dev en phase de développement)