UFR SEN
Sciences Exactes et Naturelles

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

2022/2023

Système d'Achat de L'énErgie (SALE)

Rapport rédigé à la suite d'une gestion de projet effectuée en :

INFO0503 : Modélisation client-serveur et programmation Web avancée

Sous la supervision de M. BOISSON Jean-Charles

Par
Sami DRIOUCHE & Fayssal SHAITA
S507B

1. Table des matières

1. Introduction

Ce projet vise à développer une application distribuée permettant à un **client** de passer une commande d'énergie et de pouvoir consulter l'état d'avancement de ces dernières. Entre la demande initiale et éventuellement son dénouement positif, différentes étapes ont lieu entre différentes entités. Le client peut choisir le type d'énergie qu'il désire ainsi que le mode d'extraction, l'origine géographique, la quantité en unités d'énergie ainsi que le prix maximum par unité d'énergie.

Une fois la commande finalisée par le client elle est envoyée à un **revendeur** ce dernier contact un **trader** pour lui transmettre la commande.

Le **trader** (TARE) récupère la commande et l'envoie au **Marché de gros,** le trader gère des types d'énergies spécifiques, c'est lui qui sélectionne une énergie du marché correspondant à la commande du client.

Le **Marché de gros** est l'entité qui contient toutes les énergies mises en vente par les **producteurs**, c'est ici que les **traders** consultent les énergies du marché, le **client** peut lui aussi (avant de passer commande) consulter les énergies disponibles. Toutes les énergies du marché sont certifiées par les **autorités**.

Le **producteur** (PONE) produit des énergies et les places sur le marché, il peut produire à la demande tout comme produire des énergies de façon arbitraire.

L'**autorité** (AMI) gère la sécurité du système c'est grâce à elle que les énergies du marché sont certifiées, elle assure aussi le bon déroulement d'une vente d'énergie et le prix que donne les **producteurs** à leur énergie.

2. Prérequis

Voici quelques informations nécessaires à savoir avant d'utiliser notre site **SALE** :

Tout d'abord le site à étais conçu en PHP donc il faut être capable de lancer notre répertoire « **serveur_PHP** » dans un serveur PHP.

Pour tester le site Il faut au minimum disposer de la version **8.0.10** de PHP.

Afin de pouvoir gérer plusieurs clients nous avons mis en place un système de connexions avec 3 utilisateurs par défaut, ce qui permet à un client de ne pas perdre ces commandes déjà passé et de pouvoir les stocker.

Accès à un compte client :

- Utilisateur (Jean-charles BOISSON)
  - Login : **Jean-Charles.Boisson@univ-reims.fr**
  - Password : **tata**

- Utilisateur (Fayssal SHAITA)
  - Login : **fayssal.shaita@etudiant.univ-reims.fr**
  - Password : **titi**

- Utilisateur (Sami DRIOUCHE)
  - Login : **sami.driouche@etudiant.univ-reims.fr**
  - Password : **toto**

Une fois connecté vous pouvez créer une nouvelle commande, cette dernière sera envoyée du coté java, toutes les communications entre nos différents acteurs on était faites dans des classes java.

Il faut disposer au minimum de la version **16** de java.

Afin de faciliter et automatiser le traitement des diffèrent serveur nous avons mis en place une classe de lancement qui se chargera de lancer des threads pour nos diffèrent acteur.

Pour compiler et tester le projet il faut se situer dans le répertoire « **serveur_JAVA** » et taper les commandes suivantes sur un terminal :

- Commande de compilation :

  javac -d cls/ -cp lib/json-20220924.jar -sourcepath src/ src/Lanceur.java

- Commande pour exécuter (sous Windows) :

  java -cp cls/;lib/json-20220924.jar Lanceur config.json

- Commande pour exécuter (sous Linux) :

  java -cp cls/:lib/json-20220924.jar Lanceur config.json

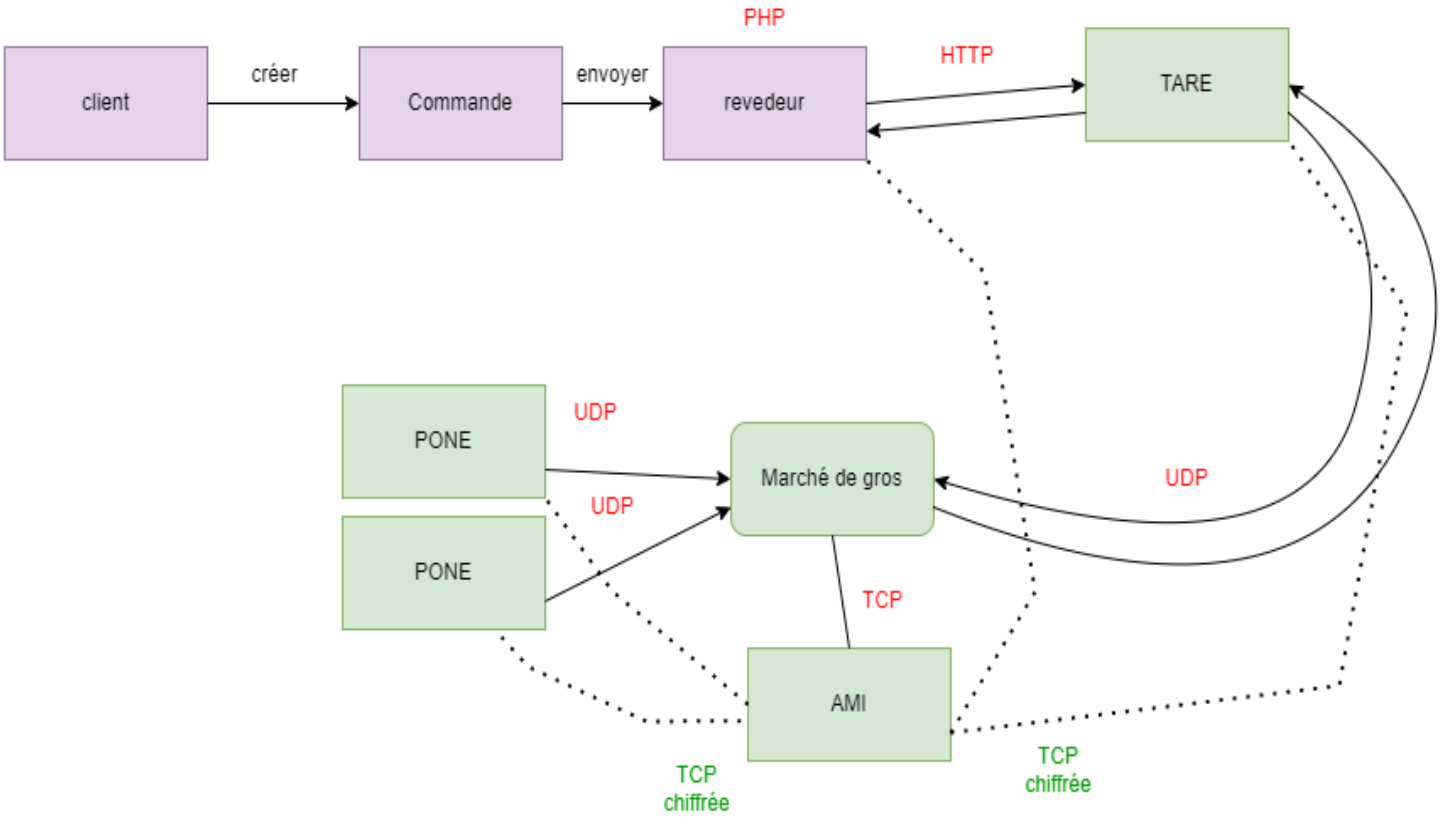
Une fois tous les serveurs démarrés libre à vous de tester notre système, commandé les énergies de demain en toute sécurité. 😊

3. Cahier des charges

Pour mettre a bien notre projet il a fallu réfléchir et mettre en place des communications spécifique entre nos diffèrent acteur.

On souhaite que la commande du client et les énergies produites par les Pone transitent à travers les différentes entités du système

- Schéma de fonctionnement



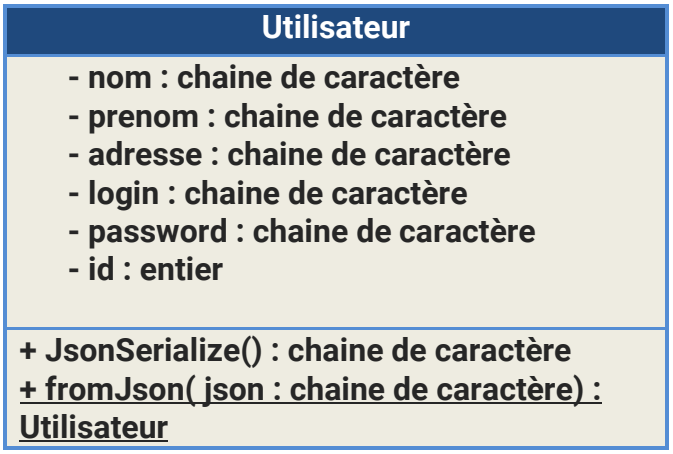
Dans ce schémas les entités représentées de couleur violète seront dans la partie **php** et celle de couleur verte seront des classes **java**.

Dans notre système un client créer une commande elle est ensuite envoyée à un revendeur qui va la communiquer au TARE en la déposant sur l'entête **http**. Ce TARE envoie la commande au marché de gros à l'aide d'une connexion **UDP**. Les PONE communiquent des énergies en parallèle au marché de gros en **UDP,** lorsque le marché reçois des énergies il les communique à l'AMI via une connexion **TCP**. Si notre commande client correspond à une énergie disponible sur le marché et bien cette énergie est envoyée au trader puis remonté au revendeur pour arriver finalement chez le client et conclure la vente.

- Détail des classes

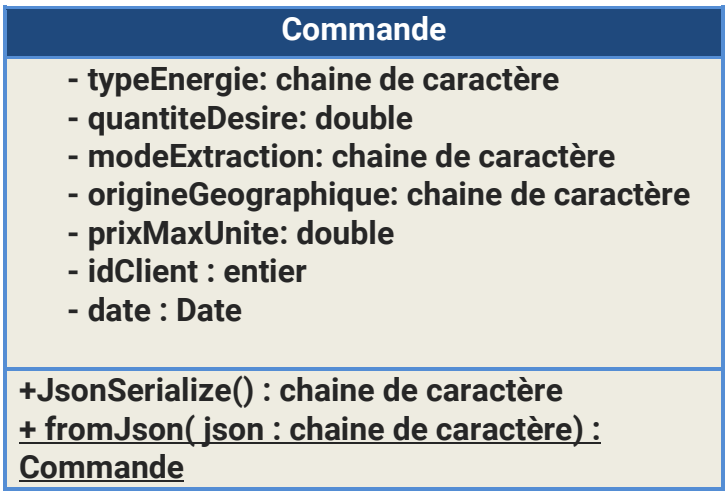
Voici les explications de toute les classes que nous avons implémenter dans ce projet sous forme **d'UML**, pour faciliter la représentation on ne détaillera pas les méthodes liées au **getters setters** ainsi que les méthodes d'affichage **toString**.

Utilisateur



Pour la gestion des acteurs nous avons créé une **classe mère utilisateur** qui par default correspond à un simple **client**, lui permettant de s'identifier dans notre site. Chaque client possède un identifiant unique.

Commande



La classe **Commande** permet d'enregistrer la commande d'énergie faite par le client du coté PHP, cette classe a étais réécrite en java pour pouvoir reconstruire la commande.

Revendeur

Revendeur
- Id : entier - commandes : Commande
+ JsonSerialize() : chaine de caractère + fromJson(json : chaine de caractère) : Revendeur + envoyerCommande() : chaine de caractère

La classe **Revendeur** est une classe fille de la classe **Utilisateur** c'est grâce à elle qu'on envoie la commande du php au java et réceptionne l'énergie grâce à la méthode **envoyerCommande()**.

Acteur

Acteur
- id : entier - nom : chaine de caractère - adresse : chaine de caractère
+ afficher() : -

La classe **Acteur** est une classe mère permettant de regrouper les attributs de nos diffèrent acteur pour simplifier l'implémentation, elle se situe uniquement coté java.

Tare (trader)

Tare
- energie : Energie
+ afficher() : -

La classe **Tare** est une classe fille de la classe **Acteur**, elle possède une **Energie** qui correspond enfaite à l'énergie qu'il achète sur le marché.

Pone (producteur)

Pone
- energieProduite : Vector<TypeEnergie> - origineProduction : Vector<OrigineEnergie> - modeExtraction : Vector<ModeExtractionEnergie> -
+ afficher() : - + ajouterEnergieProduite(energie : TypeEnergie) : - + ajouterOrigineProduction(origine : OrigineEnergie) : - + ajouterModeExtraction(mode : ModeExtractionEnergie)

La classe **Pone** est une classe fille de la classe **Acteur**, elle se situe uniquement coté java. Elle possède 3 vecteurs correspondant à la liste des origines de production, des modes d'extraction et des types d'énergie qu'elle est capable de produire.

Ami (autorité)

Ami
-
+ afficher() : - + verifierEnergie(e1 : Energie) : booléen

La classe **Ami** est une classe fille de la classe **Acteur**, elle se situe uniquement coté java, c'est elle qui va contrôler les énergies déposer sur le marché, elle possède une méthode **verifierEnergie()** qui vérifie le prix d'une énergie sur le marché par rapport à la quantité si ce test est validé elle retourne **true** sinon elle retourne **false** et l'énergie est retiré du marché. Elle permet aussi de gérer les certifications **RSA** des énergies que les producteurs produisent en signant le **CRADO**.

RSA

RSA
+ GenererClesRSA(cle_prive : chaine de caractere,cle_publique : chaine de caractere) : - + SignerFichierRSA(cle_prive : chaine de caractere, fichier_signer : chaine de caractere, fichier_sauvegarde : chaine de caractere) : - + VerifierSignatureRSA(fichier_verif : chaine de caractere, fichier_signature : chaine de caractere, fichier_publique : chaine de caractere): chaine de caractere + chiffrerRSA(cle_publique : chaine de caractere, message : chaine de caractere) + dechiffrerRSA(cle_privee : chaine de caractere, message_chiffre : chaine de caractere) : chaine de caractere

La classe **RSA** va permettre à **l'AMI** de pouvoir utiliser les méthodes nécessaires au chiffrement **RSA** ainsi que celle pour la signature de fichier à l'aide d'une clé publique et clé privée.

Marché de gros

MarcheGros
- energies : Vector<Energie>
+ ToJSON() : JSONObject + fromJSON(json : chaine de caractère) : MarcheGros + ajouterEnergie(energie : Energie) : - + supprimerEnergie(energie : Energie) : - + supprimerEnergie(index : entier): - + afficherEnergies(): - + creerFichier(nomFichier : chaine de caractère) : - + recupererFichier(nomFicher : chaine de caractère) : MarcheGros + comparerCommandeEnergie(c1 : Commande) : Energie

La classe **MarcheGros** est la classe qui contrôle tout le marché, elle se situe uniquement coté java. Cette classe possède un vecteur **d'énergie** avec des méthodes qui lui permettent d'ajouter ou de supprimer une énergie du marché, pour que les énergie du marché sois sauvegarder nous avons la méthode **creerFichier()** qui créer un fichier **JSON** le tableaux des énergies du marché, puis une méthode **recupererFichier()** qui va lire ce fichier au lancement du marché de gros pour récupérer les donnée sauvegarder dans le fichier **JSON**.

Energie

Energie
- typeEnergie: TypeEnergie - quantite: double - modeExtraction: ModeExtractionEnergie - origineGeographique: OrigineEnergie - prixUnite: double - idPone : entier - etat : chaine de caractère - code : CodeSuivi
+ ToJSON() : JSONObject + fromJSON(json : chaine de caractère) : MarcheGros + afficher() : -

La classe **Energie** a été créée pour différencier la commande du client des énergie produite par les producteurs, à sa construction une énergie possède un **code de suivi** et un **état** qui indique si l'énergie est toujours en vente ou qu'elle est déjà vendue. Cette classe se situe uniquement du coté java et elle implémente l'interface **Serializable** (ce qui permet la sérialisation de l'objet).

Code de Suivi

CodeSuivi
- codeTypeEnergie : chaine de caractère - codeQuantiteDesire : chaine de caractère - codeModeExtraction : chaine de caractère - codeOrigineGeographique : chaine de caractère - codePrixMaxUnite : chaine de caractère - codeIdClient : chaine de caractère - codeEtat : chaine de caractère - codeCommande : chaine de caractère
+ encoderTypeEnergie(typeEnergie : chaine de caractère) : chaine de caractère + encoderOrigineEnergie(origineGeographique : chaine de caractère) : chaine de caractère + encoderModeExtraction(modeExtraction : chaine de caractère) : chaine de caractère + encoderIdClient(idClient : entier) : chaine de caractère + encoderEtat(etat : chaine de caractère) : chaine de caractère + encoderQuantite(quantiteDesire : double) : chaine de caractère + encoderPrix(prixMaxUnite : double) : chaine de caractère + decoderIdClient(codeDeSuivi : CodeSuivi) : entier + decoderEtat(codeDeSuivi : CodeSuivi) : chaine de caractère + decoderTypeEnergie(codeDeSuivi : CodeSuivi) : chaine de caractère + decoderOrigineGeographique(codeDeSuivi : CodeSuivi) : chaine de caractère + decoderModeExtraction(codeDeSuivi : CodeSuivi) : chaine de caractère + decoderQuantiteDesire(codeDeSuivi : CodeSuivi) : double + decoderPrixMaxUnite(codeDeSuivi : CodeSuivi) : double + decoderCodesuivi(codeDeSuivi : CodeSuivi) : Energie

La classe **CodeSuivi** permet d'encoder toutes les caractéristiques d'une énergie, nous avons implémenté pour chaque attribut de notre classe Energie une méthode d'encodage et une méthode statique de décodage, ce qui implique que grâce au code de suivi on peut décoder seulement la partie qui nous intéresse sans pour autant avoir tout l'objet.

Le code s'affiche de la façon suivante : **[000030110002140831 1009]**

Chaque valeur représente un attribut de **l'Energie.**

000030110002140831100 9

La première valeur du code correspond à l'état, lorsque la vente est en cours elle possède la valeur 0, elle peut être à 1 si la vente est finalisée et à 2 si elle est abandonnée.

0**0003**011000214083110 09

Les 4 prochaines valeurs correspondent à l'id du pone, nous avons coder cette id sur 4 chiffres ce qui implique que dans notre système il ne peut y avoir plus de 9999 pone.

00003**01**1000214083110 09

Les 2 prochaines valeurs correspondent au type de l'énergie, et plus précisément a leur ordre dans l'énumération des types d'énergies. Comme nous possédons seulement 5 types d'énergie sa valeur maximum et de 04.

000030**10**0021408311 009

Les 2 prochaines valeurs sont liées à l'énumération des origines géographiques des énergie, cette énumération contient 14 pays donc le nombre maximum pour ce code et de 13.

0000301100 21408311 009

Les 2 prochaines valeurs correspondent à notre dernière énumération, celle des modes d'extraction de l'énergie comme pour les deux précèdent code là aussi il s'agis du positionnement du mode d'extraction de l'énergie dans l'énumération. La valeur maximum est de 04.

00003011000**21408**311 009

Dans cet exemple les 5 prochaines valeurs correspondent à la quantité d'énergie en unité d'énergie. Cependant notre algorithme va d'abord lire les 2 premières valeurs qui vont lui indiquer, pour la première valeur le nombre de chiffre avant la virgule, puis le nombre de chiffre derrière la virgule pour la deuxième valeur, grâce à cette technique nous sommes capables d'encoder n'importe quelle valeur aussi grande soit elle. Ici la quantité est donc de 40,8.

000030110002140 8**311009**

Ici les 5 dernières valeurs indiquent le prix par unité de l'énergie comme nous ne connaissons pas la taille de cette valeur à l'avance nous avons utilisé la même technique de codage que pour la quantité d'énergie. Nous avons donc dans cette exemple 3 chiffres avant la virgule puis 1 chiffre derrière, on peut alors décoder le prix de 100,9 €.

Notre code de suivi fonctionne donc de la même manière qu'un code barre pour la transaction d'énergie.

Mode extraction de l'énergie

ModeExtractionEnergies
- nom : chaine de caractère
+ getValue(s : chaine de caractère) : ModeExtractionEnergies

L'énumération **ModeExtractionEnergies** est une enum permettant de regrouper les diffèrent valeur du mode d'extraction de nos énergies, elle se situe coté java et php.

Valeur possible :

- Eolien
- Nucleaire
- Forage Vertical
- Forage Horizontal
- Solaire

Type de l'énergie

TypeEnergie
- nom : chaine de caractère
+ getValue(s : chaine de caractère) : TypeEnergie

L'énumération **TypeEnergie** est une enum permettant de regrouper les diffèrent valeur des types de nos énergies, elle se situe coté java et php.

Valeur possible :

- Petrole
- Electricite
- Hydrogene
- Biomethane
- Gaz naturel

Origine de l'énergie

OrigineEnergie
- nom : chaine de caractère
+ getValue(s : chaine de caractère) : OrigineEnergie

L'énumération **OrigineEnergie** est une enum permettant de regrouper les diffèrent valeur des origines de nos énergies, elle se situe coté java et php.

Valeur possible :

- Allemagne
- Espagne
- Chine
- Italie
- Royaume-Uni
- Pays-Bas
- Portugal
- Suede
- Finlande
- Algerie
- Maroc
- Tunisie
- Turquie

Gestion des classes

Pour faciliter l'organisation de toute ces classes nous avons procédé de la manière suivante, dans notre répertoire de projet il y a 2 dossiers, un pour la partie php « **serveur_php** » et un pour les codes en java « **serveur_java** ».

Lorsque l'on se situe dans la partie php toute nos classes qui instancie un objet son dans le répertoire « **package_php** », de même pour la partie java avec le répertoire « **package_java** ».

4. Réalisation

- Gestion des serveurs
- PHP

C'est du côté php que le client arrive pour s'identifier sur notre site, nous avons donc créé des utilisateurs par default qui ont respectivement leur commande.

Pour réaliser ceci, le gestionnaire d'utilisateurs a besoin de la liste des utilisateurs pour les connecter/déconnecter et vérifier qu'ils existent et sont connectés. Il possédera donc un **fichier JSON** « **Utilisateur.json** » contenue dans le répertoire « **storage** ».

Pour maintenir le client connecté lors de la navigation nous avons stocker l'instance de l'utilisateur connecté dans une variable de **session** que nous détruisons lorsqu'il se déconnecte.

Les commandes du client sont quand a-t-elle aussi stocker dans un **fichier JSON**

« **commande.json** ».

Comme chaque commande possède l'id du client il suffit de parcourir les commandes portant l'id du client connecté et les afficher pour avoir l'historique des commandes clients.

- JAVA

Pour l'interface java nous avons fait le choix d'utiliser qu'un seul terminal pour les diffèrent affichage des communications entre les serveurs, pour se faire nous avons créé une classe de lancement des diffèrent **thread** « **Lanceur.java** ».

Cette classe va d'abord récupérer les informations de chaque port utilisé pour nos communication dans un fichier de configuration en **JSON** « **config.json** », une fois ceci fait nous stockons chaque **thread** de nos acteur dans un tableau **arrayList.**

A l'exécution de notre classe de lancement nous parcourons notre arrayList et démarrons chaque serveur à l'aide de la méthode **start(),** ce qui implique que tous nos serveur implémente l'interface **runnable** et possède une méthode principal **run**.

Pour l'affichage des différents messages et pour mieux s'y retrouver sur le terminal nous avons créé une classe **messenger** qui formalise l'affichage des messages de la façon suivante :

[Marche de Gros] : Lu : prix valide

- Format des messages échangés

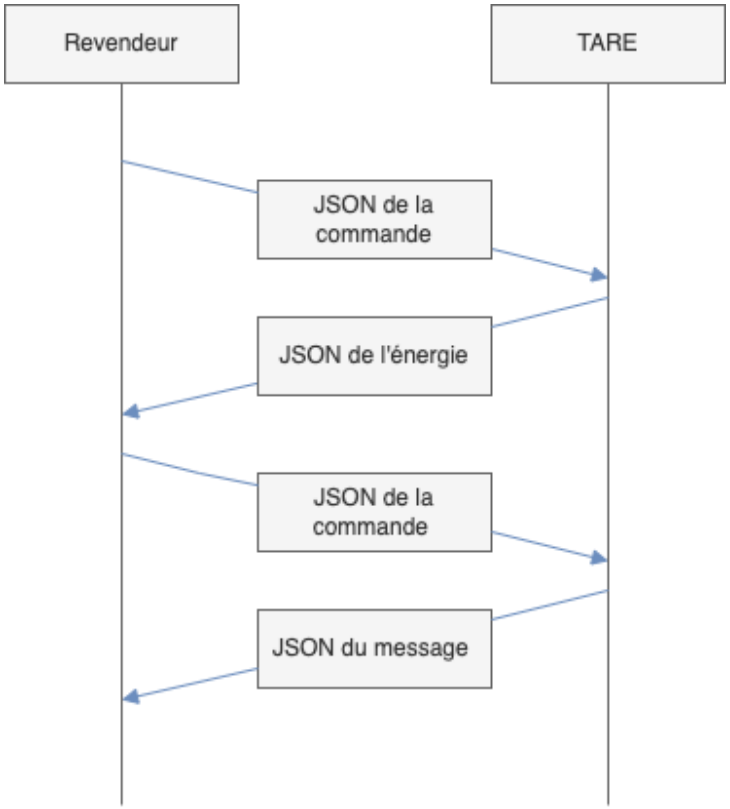
Chaque acteur de notre système communique en envoyant des données voici le format de ces données entre chacun de nos acteurs.

- Dialogue Revendeur/TARE :

Concernant le dialogue entre le revendeur et le TARE, le format **JSON** a été choisi car Il facilite le lien entre le **PHP** et le **JAVA,** c'est un langage qui permet d'avoir un certain formalisme des données stocké. Il y a 2 échanges possible entre le Revendeur et le Tare.

1er échange : Concerne l'envoi de la commande souhaité par le client, le revendeur envoi la commande au Tare. Le Tare lui renvoi l'énergie qu'il a trouvée correspondant à la commande ou une **Energie NULL** pour dire que l'énergie n'est pas disponible.

2e échange : Concerne la vérification de **l'énergie**, le revendeur renvoi le commande au **Tare** afin que le **Tare** puisse l'envoyé au **marché de gros** puis a **l'AMI** pour vérifier les certificats. Une fois vérifier le Tare envoi la réponse au revendeur.



- Dialogue TARE/Marché :

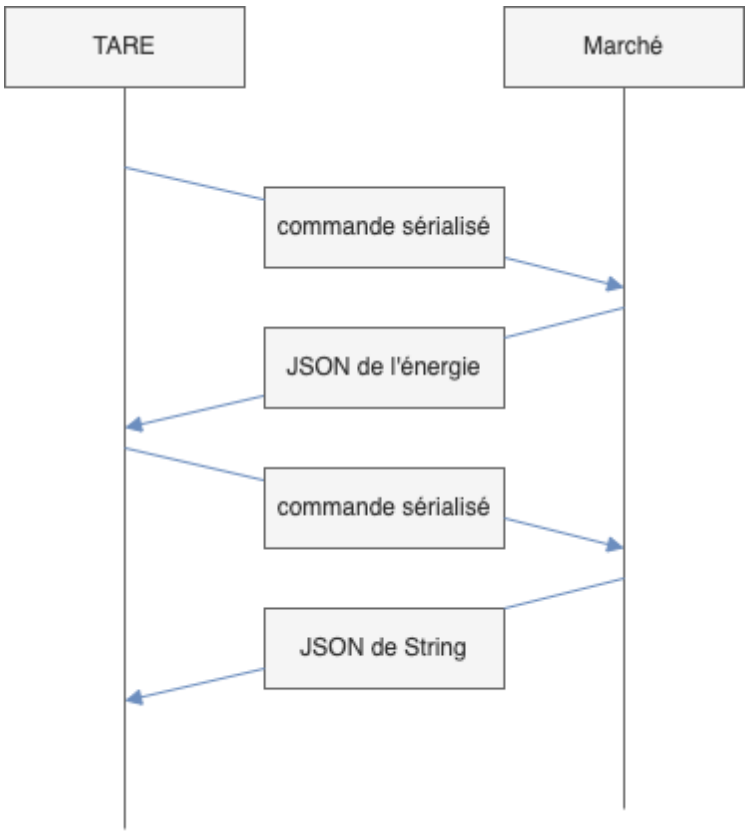
Le **TARE** envoie via **l'UDP** au format d'objet sérialisé en **java** au marché de gros, le marché de gros lui renverra un objet Energie au format **JSON.**

On privilégie La sérialisation d'objet pour les communications en **java** car on récupère exactement le même objet, cependant pour la communication qui part du **marché** au **TARE** nous avons utilisé du **JSON** par choix d'implémentation.

Il y a 2 échanges possible entre le Tare et le Marché.

1er échange : Concernant la commande, le **Tare** envoi la commande du **revendeur** au **Marché**. Le **Marché** la compare avec ces énergies puis lui renvoi un JSON de l'énergie qu'il a trouvée ou un message pour dire que l'énergie n'est pas disponible.

2e échange : Concernant la vérification de l'énergie, le **TARE** renvoi un objet commande au Marché qui correspond enfaite à une Energie que le revendeur a acheté afin que le **Marché** puisse l'envoyer à **l'AMI** pour vérification. Une fois vérifier le Marché envoi la réponse au **TARE**.

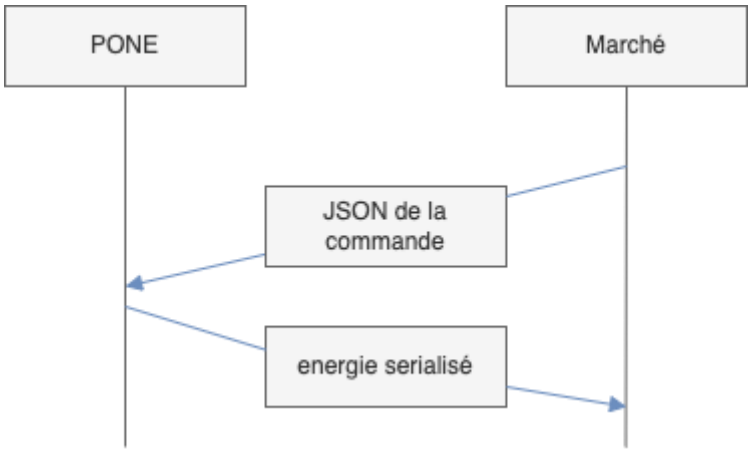


- Dialogue PONE/Marché :

Il y a un échange possible entre le **PONE** et le **Marché**.

Le Marché envoie via **l'UDP** au format d'objet **JSON** une commande au **PONE**, le **PONE** lui renverra un objet **Energie** au format d'objet sérialisé.

Une fois que le **TARE** a envoyé au **Marché** la commande à satisfaire, le **Marché** vérifie si la commande est déjà disponible dans les énergies qu'il dispose, si la commande n'est pas disponible, le **Marché** demande au **PONE** de la produire, le **PONE** envoi l'énergie produite au **Marché**.



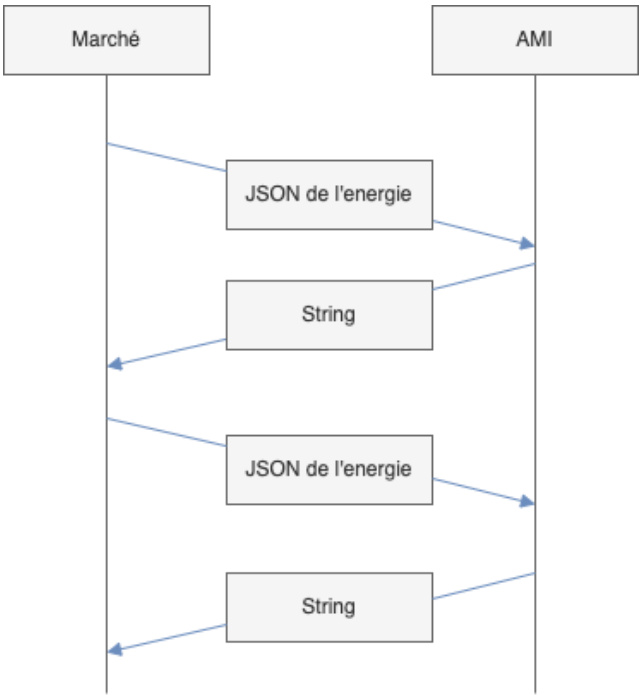
- Dialogue Marché/AMI :

Il y a 2 échanges possible entre le **Marché** et **l'AMI**.

Le **Marché** envoie via le **TCP** au format d'objet **JSON** une commande à **l'AMI**, **l'AMI** lui renverra un JSON de chaine de caractère.

1er échange : Le première échange concerne l'énergie dans le Marché récupérer du **PONE**, le Marché envoi l'énergie a **l'Ami** afin qu'il puisse valider le prix et lui renvoyer sa réponse.

2e échange : Concerne la vérification de l'énergie, le Marché envoi l'énergie qu'il souhaite certifier auprès de **l'Ami**. Une fois vérifier **l'Ami** envoi la réponse au **Marché**.



- La liste des requêtes associées
- Dialogue Revendeur/TARE :

Le revendeur transforme la commande du client en **JSON** puis la transmet au TARE. Le TARE peut sois lui renvoyer au format **JSON** une énergie qui correspond à la commande sois une **énergie vide** sans aucune quantité ni prix ce qui signifie qu'il n'a rien trouvé.

JSON de la commande :

```
{
"date": "11-12-2022",
"typeEnergie": "Electricite",
"quantiteDesire": 100,
"modeExtraction": "Nucleaire",
"origineGeographique": "Turquie",
"prixMaxUnite": 11.4,
"idClient": 1,
"etat": "energie achete"
}
```

JSON d'une énergie :


```
{
"modeExtraction": "FORAGEHORIZONTAL",
"typeEnergie": "PETROLE",
"idPone": 0,
"origineGeographique": "ALGERIE",
"prixUnite": 10.9,
"etat": "en cours",
"quantite": 400
}
```


- Dialogue TARE/Marché :

Le **TARE** enverra toujours au marché de gros la commande du client en **JSON**, le **marché** lui enverra le **JSON** d'une **énergie**.

Si le **JSON** qu'il renvoie est vide cela signifie que le marché n'a pas trouver **d'énergie** correspondante.

- Dialogue PONE/Marché :

Le **Marché** peut envoyer le **JSON** d'une **Commande**, le **PONE** lui renverra sois une **Energie** correspondant à la commande sois une **Energie** quelconque pour simplement la déposer sur le **marché** même si le **TARE** ne la rachète pas derrière.

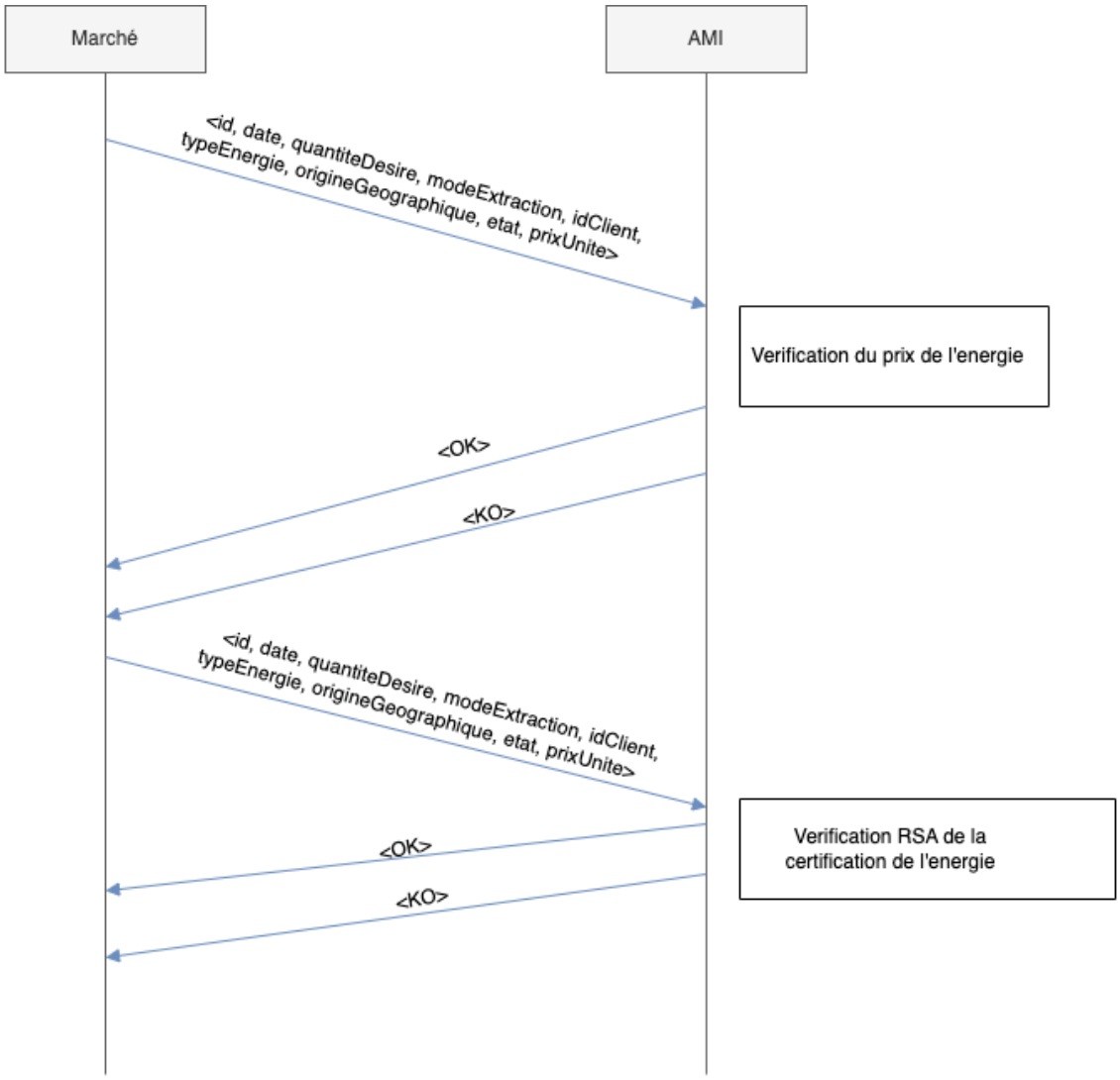
- Dialogue Marché/AMI :

Le **Marché** envoie donc le **JSON** d'une Energie produite par le PONE pour que **l'AMI** puisse la vérifier en comparant son prix. Lorsque l'Energie est validé il envoie sous forme de JSON un message « Prix valide » si tout est OK sinon « prix trop élevé ».

Le **Marché** pourra lors d'une vérification d'Energie envoyer une **energie** au format **JSON** avec comme valeur pour l'état : « verification » ce qui va permettre à **l'AMI** de tester sa certification **RSA** à l'aide du **CRADO**. L'AMI renverra un message pour valider ou non la certification.

Le détail de cet échange pour illustrer son fonctionnement :

- Chronogramme



5. Les scénarios, établir les échanges complets

- Scénario A :

Pour le scénario A le client fait une simple commande sans contrainte (mode d'extraction, origine géographique) que le revendeur envoie au TARE, puis que le tare envoie au marché de gros pour qu'il puisse renvoyer une énergie correspondante qu'il dispose. Comme le pone à produit exactement le même type d'énergie la commande est tout de suite satisfaite.

Exemple de commande :

Commandé par exemple du pétrole sans mode d'extraction et origine géographique spécifié.

- ## Scénario B :

Le scénario B se réalise lorsque le client commande une énergie qui n'est pas tout de suite disponible, mais après un certain temps d'attente le producteur produit le type d'énergie désiré.

Afin de simuler ce scénario nous avons mis en place une fonction **creerEnergie()** qui lorsqu'elle reçoit une certaine commande se met en attente pendant 5 seconde et renvoie une énergie correspondant à la commande client.

Exemple de commande :

Commandé de l'électricité provenant de France avec un mode d'extraction solaire.

- ## Scénario C :

Faute de temps nous n'avons malheureusement pas pu implémenter ce scénario pour projet, ce scenario devait se réaliser lorsque le client commande une énergie qui n'est pas réalisable par le Pone de base, mais après un certain temps un nouveau Pone est rajouté et il peut fournir l'énergie demandée.

Afin de simuler ce scénario nous aurions pu ajouter à la classe Pone un extends thread pour lancer deux Pone.

- ## Scénario D :

Faute de temps nous n'avons malheureusement pas pu implémenter ce scénario a notre projet, ce scenario devais se réaliser avec deux Pones et deux Tares, le client devais se baser sur la somme des énergies fournies par les 2 Pones. Cependant, les tares ne pouvaient acheter qu'à un seul Pone à la fois. C'était donc au revendeur de gérer le rassemblement des 2 retours de Tare.

Afin de simuler ce scénario nous aurions pu ajouter à la classe Pone un extends thread pour lancer deux Tare.

- Scénario A2 :

Le scenario A2 est le même que le A, on a simplement ajouté un bouton pour faire vérifier la certification de l'énergie achetée.

Pour réaliser cela, nous avons choisis de comparer le fichier **json** de **l'Energie** que le **Pone** a produit avec la signature **RSA** réaliser par **l'AMI** de **l'Energie** correspondante.

6. Conclusion

Pour finir ce projet a été long à réaliser tant au niveau de la modélisation que de la production.

En effet, nous ne nous rendions pas compte de sa complexité et de son timing de réalisation.

Cependant, c'est notre premier projet client-serveur avec l'utilisation du multi thread dont nous sommes assez fiers, bien sûr le projet n'est pas parfait mais il est fonctionnel et nous pensons que c'est le plus important.

Finalement, ce projet nous a permis de prendre en main l'utilisation des API JSON et Http en Java et PHP, et de découvrir certaines des problématiques propres au développement d'une application distribuée.

7. Webographie

https://stackoverflow.com/

https://getbootstrap.com/

https://www.php.net/manual/fr/book.json.php

https://www.tutorialspoint.com/json/json_java_example.htm

https://tutowebdesign.com/extraction-chaine-php.php