# PROJET GENIE LOGICIEL (SINFO015) :
## Rapport de modélisation

**Auteurs**:
Adrien Fievet
Claire D'Haene
Julien Ladeuze
Maxime Dupuis

2022-2023

*Titulaire*
Tom Mens
*Enseignants*
Sébastien Bonte
Jeremy Dubrulle
Pierre Hauweele

*Equipe 7*
Adrien (220625) 4
Claire (220323) 1
Julien (220101) 10
Maxime (212107) 7

UMONS
Université de Mons

Faculté
des Sciences

# Contents

# 1 Use cases diagrams

## 1.1 Points communs

Etant donnée la nature des deux applications, de nombreux points communs sont à souligner dans les use cases tels que les logs, gestion des notifications et gestion des paramètres.

Concernant le système de log, cela peut s'apparenter à une application "classique" contenant des utilisateurs avec des fonctionnalités telles que créer un compte, s'authentifier, se déconnecter et réinitialiser le mot de passe.

Pour la gestion des notifications, l'utilisateur aura la possibilité de voir les détails des notifications qu'il aura reçues, de les marquer comme lues et aura le choix de les accepter ou de les refuser. De plus, l'utilisateur pourra rafraichir la page pour voir apparaitre de nouvelles notifications (de même que mentionné dans l'énoncé).

En ce qui concerne la gestion des paramètres, l'utilisateur aura notamment la possibilité de modifier son mot de passe en ayant reçu un mail de confirmation au préalable. Au sujet de la gestion des langues, ce dernier aura le droit d'en ajouter, d'en choisir une qui sera sa langue favorite et de changer la langue actuelle de l'application.

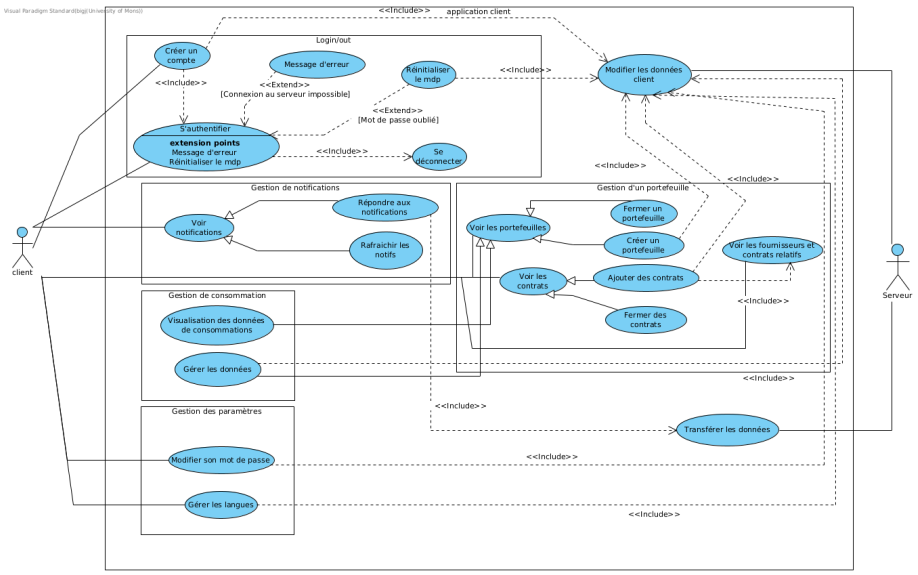
Nous pouvons observer que le serveur est un acteur qui enregistrera toutes les informations utiles et qu'il jouera le rôle de correspondant entre le client et le fournisseur par le biais du use case : "transférer les données". Pour permettre une meilleure lisibilité et compréhension de notre diagramme, nous avons décidé d'impliquer le serveur uniquement dans certaines actions. De ce fait, celui-ci n'est pas relié aux Use cases portant par exemple le nom de ≪ Fermer un portefeuille ≫.

## 1.2 Client

Les deux points principaux à noter sont les Use cases « Voir les portefeuilles » et « Voir les contrats ». En effet, c'est à partir de ces derniers que la logique même des fonctionnalités disponibles pour le client repose.

Lorsque le client est sur la section « Voir les portefeuilles », il pourra créer un portefeuille, fermer un portefeuille, voir les données de consommation et gérer ces données. Il convient de signaler que nous avons décidé de créer une sous-catégorie supplémentaire "Gestion de consommation" afin d'apporter une meilleure lisibilité et séparation des sections.

Quant à la section « Voir les contrats », il aura la capacité d'ajouter ou fermer des contrats. En vue d'ajouter des contrats, le client devra forcément voir les fournisseurs et contrats relatifs. Nous avons choisi que cette option devra également être disponible sans se rendre dans la section mentionnée précédemment.

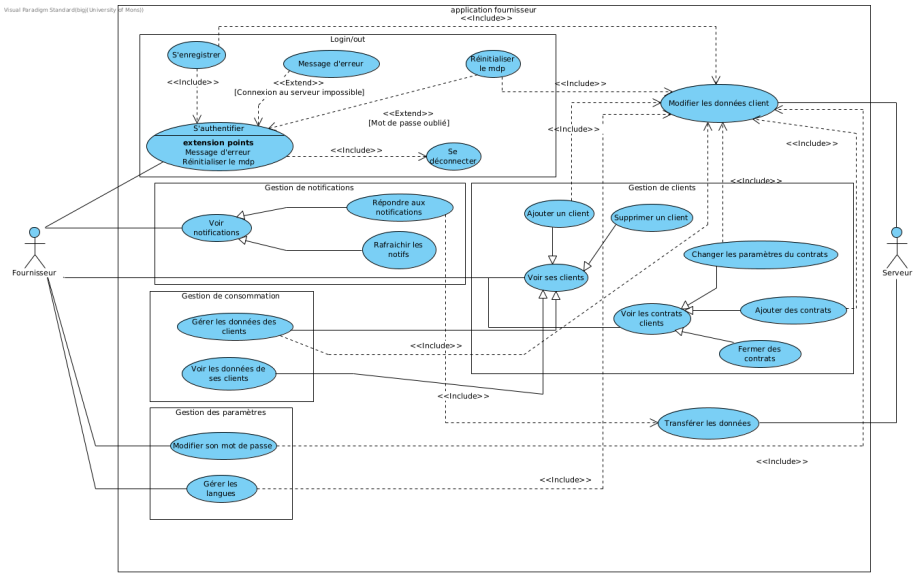


## 1.3 Fournisseur

Pour le fournisseur, les principales différences résident dans le fait qu'il n'y aura plus les sections « Voir les portefeuilles » et « Voir les contrats » mais « Voir ses clients » et « Voir les contrats clients ».

Dans la même optique que pour la gestion d'un portefeuille chez le client, le fournisseur une fois sur la section « Voir ses clients » sera apte à ajouter ou supprimer un client, voir les données de ses clients ainsi que les gérer.

Ce dernier aura, de plus, la possibilité d'ajouter ou fermer des contrats et changer les paramètres des contrats/propositions sur « Voir les contrats clients ».



# 2 interaction overview diagrams

## 2.1 Introduction

Nous avons réalisé nos interaction overview diagrams sur base de nos use cases diagrams. Ce qui nous a permis de détailler la structure de ces diagrammes intuitivement.

Ces derniers permettront d'obtenir une idée globale du parcours des utilisateurs sur l'application ainsi que les fonctionnalités leur étant disponibles.

Nous décrirons donc deux applications, l'application « client » et l'application « fournisseur » et de nouveau, nous verrons apparaître des points communs entre ces dernières.

## 2.2 Points communs

1. **Système de logs**:

   Une fois que l'utilisateur arrivera sur l'application, plusieurs choix s'offriront à lui. Soit ce dernier est nouveau et il peut donc s'enregistrer en respectant les conditions qui lui sont imposées telles qu'une adresse mail valide, vérifiée par une demande de confirmation, et inexistante sur l'application, la mention de son rôle sur l'application (client ou fournisseur), un mot de passe ayant un niveau de sécurité convenable et le choix de sa langue, soit, il possède déjà un compte et se connecte.

   Cependant, celui-ci pourra réinitialiser son mot de passe à l'aide d'un mail de confirmation ou réessayer de se connecter si lors de la vérification des données, les identifiants sont invalides.

   De plus, lorsque l'utilisateur sera connecté sur l'application, il aura la possibilité de se déconnecter s'il le souhaite.

2. **Menu représenté par un « grand » fork :**

   Une fois connecté, l'utilisateur aura accès à un menu reprenant la déconnexion, l'accès aux paramètres, les notifications et les éléments lui étant propres en fonction qu'il soit fournisseur ou client.
   Cette façon de penser nous semblait suffisamment intuitive pour que l'utilisateur puisse naviguer sur l'application facilement. Nous avons également choisi de permettre un « retour » à ce menu lorsque nous sommes dans les sections le composant.

3. **Accès aux paramètres :**

   Une fois sur l'onglet des paramètres, plusieurs choix s'offrent à lui :

   a) Gérer les langues : plus précisément, il pourra en ajouter, en choisir une qui sera sa langue favorite et de changer la langue actuelle de l'application comme expliqué précédemment dans les Use cases diagrams.
   b) Modifier son mot de passe : pour ce faire, l'utilisateur recevra un mail de confirmation permettant de le changer en toute sécurité.

4. **Les notifications :**
   Une fois sur cette section, l'utilisateur pourra répondre à ses notifications, c'est-à-dire, marquer comme lues les contrats, les accepter, les refuser ou encore de voir leurs détails à savoir les données relatives à ces derniers.

   De surcroît, s'il le désire, ce dernier aura le droit de rafraichir ses notifications par lui-même sans attendre que cela se réalise automatiquement.

## 2.3 Accès à l'application pour les clients

Lorsque d'un client voudra **voir ses portefeuilles**,
il retrouvera plusieurs fonctionnalités :

1. Créer un portefeuille
2. Fermer un portefeuille
3. Visualisation des données de consommation
4. Gérer les données

- **Créer un portefeuille :**

  Cette première option permettra au client comme son nom l'indique de créer un portefeuille.
  Dans cette optique, il devra rentrer le nom et l'adresse du portefeuille, le code EAN et ajouter les contrats liés.

- **Fermer un portefeuille :**

  Cela donnera la possibilité au client de clôturer un portefeuille, une fois qu'il n'en aura plus d'utilité.

- **Visualisation des données de consommation :**

  Le client pourra observer les valeurs associées à sa consommation, ainsi que les contrats étant liés à ces dernières.

- **Gérer les données :**
  Il sera également apte à les modifier, c'est-à-dire, renouveler ses données de consommation et ajouter, modifier ou supprimer les contrats étant associés à un portefeuille.
  Grâce à cette option, ce dernier aura pleinement la main sur ses portefeuilles et aura la faculté de les reformer à sa guise.

Le menu offre en outre la possibilité de **voir les contrats** ou plus précisément de voir ses contrats, l'utilisateur sera donc en mesure de, d' :

1. Ajouter des contrats
2. Fermer des contrats

- **Ajouter des contrats :**

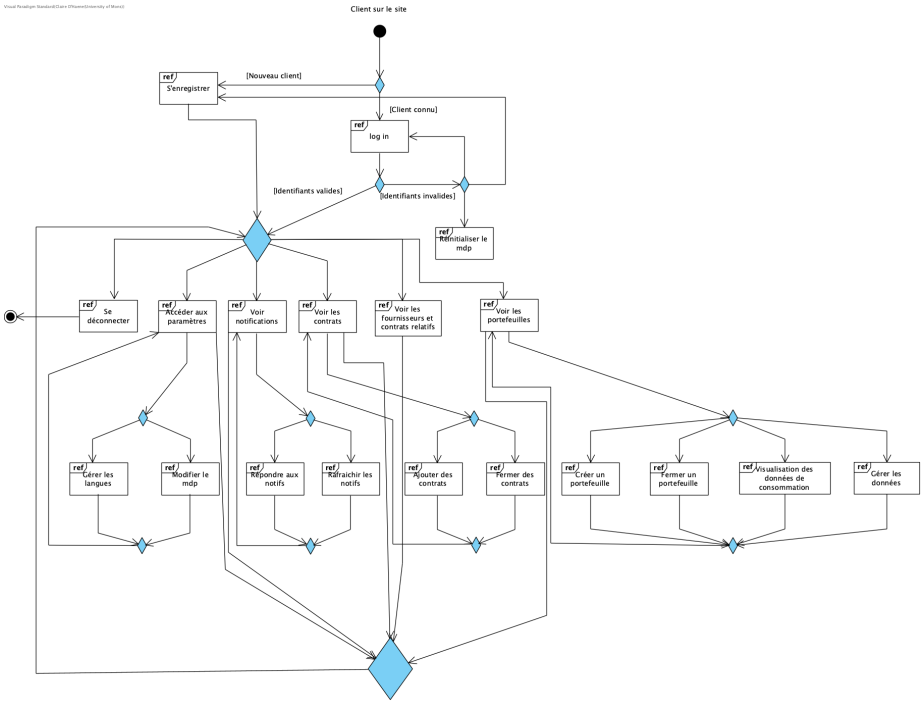
  Le client se retrouvera alors sur la page des **fournisseurs et contrats relatifs** répertoriant tous les contrats qu'il ne possède pas encore et les verra en plus amples détails avant de les ajouter s'il le désire.

- **Fermer des contrats :**

  Cela lui donnera la possibilité de clôturer un contrat s'il trouve cela plus judicieux.

  L'ajout et la fermeture de contrats impliquera une **notification** sur l'application du fournisseur en question.

  Nous avons décidé de laisser la possibilité au client de voir les fournisseurs et contrats relatifs sans forcément passer par la liste de ses contrats afin de lui permettre un accès plus rapide.


Client sur le site
ref S'enregistrer
[Nouveau client]
[Client connu]
ref log in
[Identifiants valides]
[Identifiants invalides]
ref Réinitialiser le mdp
ref Se déconnecter
ref Accéder aux paramètres
ref Voir notifications
ref Voir les contrats
ref Voir les fournisseurs et contrats relatifs
ref Voir les portefeuilles
ref Gérer les langues
ref Modifier le mdp
ref Répondre aux notifs
ref Rafraichir les notifs
ref Ajouter des contrats
ref Fermer des contrats
ref Créer un portefeuille
ref Fermer un portefeuille
ref Visualisation des données de consommation
ref Gérer les données

## 2.4 Accès à l'application pour les fournisseurs

Au moment où le fournisseur accèdera à l'onglet
« **Voir les contrats clients** », plusieurs choix s'offriront à lui :

1. Ajouter des contrats
2. Fermer des contrats
3. Changer les paramètres des contrats

- **Ajouter des contrats :**

  Cette option permettra au fournisseur d'ajouter de nouveaux contrats qu'il aura créés aux contrats déjà existants en spécifiant les valeurs leur relatives (prix, type de fourniture et compteurs).

- **Fermer des contrats :**

  Cela lui donnera la possibilité de clôturer un contrat, une fois qu'il n'en aura plus d'utilité.

- **Changer des contrats :**

  Le fournisseur sera apte à changer les termes du contrat s'il trouve cela approprié.
  Évidemment comme pour la fermeture de contrats, le changement sera notifié au client.

De plus, le fournisseur sera à même de « **Voir ses clients** » de manière distincte, plus précisément, ce dernier retrouvera une liste de ses clients avec les contrats leur étant associés :

1. Ajouter un client
2. Supprimer un client
3. Voir les données de ses clients
4. Gérer les données de ses clients

- **Ajouter un client :**

  Ce dernier aura la capacité d'ajouter un client en l'associant à un de ses contrats en particulier.

- **Supprimer un client :**

  Cela donnera la possibilité au fournisseur de clôturer tous les contrats actifs avec un client, s'il est d'avis que cela est nécessaire.

- **Visualisation des données de ses clients :**

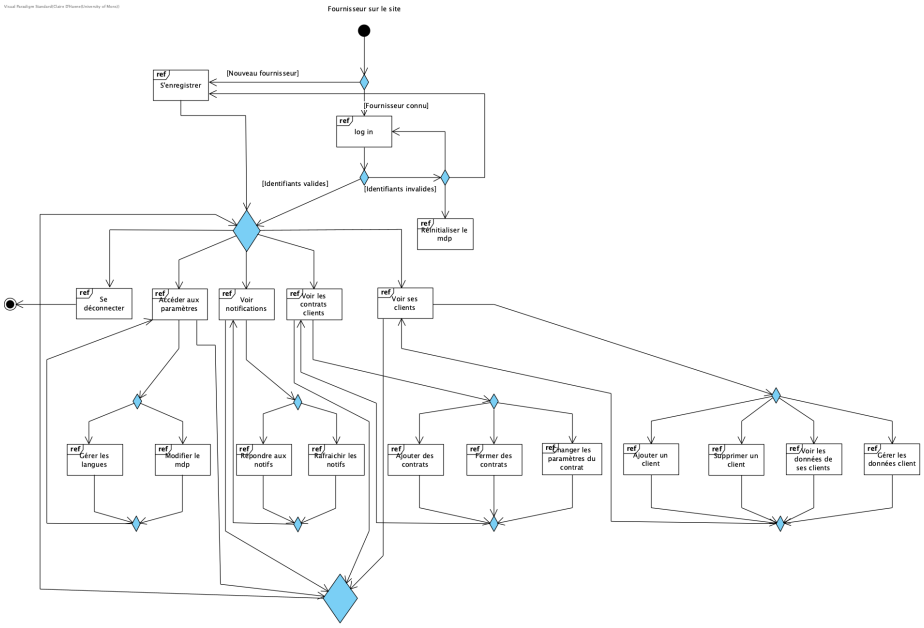
  Celui-ci pourra observer les valeurs associées aux contrats de ses clients, ainsi que leur consommation.

- **Gérer les données de ses clients :**

  Le fournisseur aura également la capacité de les modifier, c'est-à-dire, modifier, ajouter ou supprimer les contrats étant associés à un client.

  Grâce à cette option, ce dernier aura pleinement la main sur ses clients et aura la faculté de les gérer à sa guise.

  L'ajout, la modification et la fermeture de contrats impliquera une notification sur l'application du client en question.


Visual Paradigm Standard(Claire D'Hoore(University of Mons))
Fournisseur sur le site
ref
S'enregistrer
[Nouveau fournisseur]
[Fournisseur connu]
ref
log in
[Identifiants valides]
[Identifiants invalides]
ref
Réinitialiser le mdp
ref
Se déconnecter
ref
Accéder aux paramètres
ref
Voir notifications
ref
Voir les contrats clients
ref
Voir ses clients
ref
Gérer les langues
ref
Modifier le mdp
ref
Répondre aux notifs
ref
Rafraichir les notifs
ref
Ajouter des contrats
ref
Fermer des contrats
ref
Changer les paramètres du contrat
ref
Ajouter un client
ref
Supprimer un client
ref
Voir les données de ses clients
ref
Gérer les données client

# 3 Entity relationship diagram

## 3.1 préambule

Dans cette partie, nous allons parler de la modélisation de la base de donnée. Chaque sous-section parlera d'une relation. Dans ces sous-sections, nous parlerons du fonctionnement de certains attributs et nous définirons nos clés primaires et étrangères.

De manière générale, nous avons fait en sorte d'obtenir le moins de redondance possible entre les différentes relations pour convenir au mieux à la 3NF[1].

## 3.2 utilisateur

Nous partons du principe qu'un fournisseur et un client sont avant tout des utilisateurs de l'application. De ce fait, les deux personnes auront un identifiant "id"("u" + 9 digits). Cet identifiant représente la quantième personne qui a été inscrite. Par exemple, si je suis le premier utilisateur alors j'aurai l'identifiant : u000000000.

A noter que le mot de passe sera stocké dans la base de données après avoir été "hashé" par l'algorithme bcrypt[2] pour éviter toute fuite dans le cas où la base de donnée serait compromise.

PK(utilisateur) = `id`

FK(utilisateur) REFS fournisseur = `id`
FK(utilisateur) REFS client = `id`

[1] `https://en.wikipedia.org/wiki/Third_normal_form`
[2] `https://fr.wikipedia.org/wiki/Bcrypt`

### 3.3 langue

Dans cette relation, nous posons que l'utilisateur peut avoir plusieurs langues. De ce fait, nous avons une clé primaire composée.

De plus, la langue enregistrée peut être une langue préférée (1 si c'est le cas, 0 sinon).

Ensuite, un attribut binaire `langue_actuelle` est stocké. Cet attribut nous permet de savoir quelle langue l'utilisateur est en train d'utiliser ou a utilisé pendant sa dernière session (1 si c'est le cas, 0 sinon).

PK(langue) = `id` et `langue_enregistrée`

FK(langue) REFS utilisateur = `id`

### 3.4 client

Du coté de la relation "client", nous y trouvons un seul attribut. Cela nous permet d'avoir un utilisateur ayant aucun portefeuille.

PK(client) = `id_client`

### 3.5 portefeuille

L'adresse sera définie comme : nomdeville-nomderue-numérodemaison.

PK(portefeuille) = `adresse`

FK(portefeuille) REFS client = `id_client`

### 3.6 contrat du portefeuille

Dans "contrat du portefeuille" et pour toute autre relation, `id_contrat` sera défini comme ceci : "c" + quantième contrat créé.

PK(contrat du portefeuille) = `adresse` et `id_contrat`

FK(contrat du portefeuille) REFS portefeuille = `adresse`

### 3.7 fournisseur

Un fournisseur sera perçu avec son identifiant. De même que le client, un fournisseur aura la possibilité de n'avoir aucun contrat ce qui justifie le fait qu'il n'y ait qu'un seul attribut dans la relation.

PK(fournisseur) = `id_fournisseur`

## 3.8 proposition

Cette section correspondra aux offres du fournisseur avec tous les paramètres qui seront associés. Nous posons qu'un nom d'offre ne peut être unique par rapport à plusieurs fournisseurs. De ce fait, nous avons une clé primaire composée. L'attribut `nom_proposition` sera du type : "p" + nom que le fournisseur aura donné.

A noter que la localisation sera définie par un binaire de taille 3: le premier bit sera la région Wallonne, le second sera la région flamande et le troisième la région Bruxelles-Capitales. Comme exemple, 101 dira que le contrat sera disponible dans la région Wallonne et Bruxelles-Capitales .Du coté des autres types binaires, 1 sera vrai et 0 faux.

De plus, certains paramètres seront dépendants d'autres paramètres. Comme exemple, nous pouvons dire qu'il ne sera pas possible d'avoir un prix différent entre les heures pleines et les heures creuses si le type d'énergie est l'eau ou qu'on a un compteur mono-horaire.

Enfin, si nous avons que le prix est le même pendant les heures creuses et les heures pleines. Il sera donc inutile de conserver une heure bien précise (`début_heures_creuses` et `début_heures_pleines`[3] pourront être nuls)

PK(proposition) = `nom_proposition` et `id_fournisseur`

FK(proposition) REFS fournisseur = `id_fournisseur`

## 3.9 contrat du fournisseur

PK(contrat du fournisseur) = `id_contrat` et `id_fournisseur`

FK(contrat du fournisseur) REFS fournisseur = `id_fournisseur`

## 3.10 contrat

La table contrat contiendra tous les éléments typiques d'un contrat tels que le fournisseur, le client, la proposition,ect,...

A noter que la date de fermeture peut être nulle. Cela conviendra à un contrat à durée indéterminée.

PK(contrat) = `id_contrat`

FK(contrat) REFS contrat du fournisseur = `id_fournisseur` et `id_contrat`
FK(contrat) REFS proposition = `nom_proposition`
FK(contrat) REFS contrat du portefeuille = `adresse` et `id_contrat`
FK(contrat) REFS compteur = `ean`

[3] format: entier tel que les deux premiers digits seront l'heure et les minutes pour les derniers

### 3.11 compteur

Dans cette section, nous représentons le compteur associé à un contrat. Le compteur sera identifié par le code EAN.

On peut aussi noter le fait que la date d'ouverture d'un contrat est différent de la date d'ouverture du compteur(`date_d_ ouverture`).

PK(compteur) = `ean`

### 3.12 consommation

Dans la consommation, nous pouvons noter qu'il est possible qu'un client n'ai rien consommé durant la journée. De ce fait, `consommation_journalière` peut être mis à 0. On a aussi posé qu'un client ne pouvait consommer plus de 30 kw/heure par jour[4] ce qui nous permet de restreindre la taille de `consommation_journalière` à 3 digits.

PK(consommation) = `ean` et `date`

FK(consommation) REFS compteur = `ean`

### 3.13 notification

Les notifications seront représentées par un identifiant("n" + quantième notification créée).

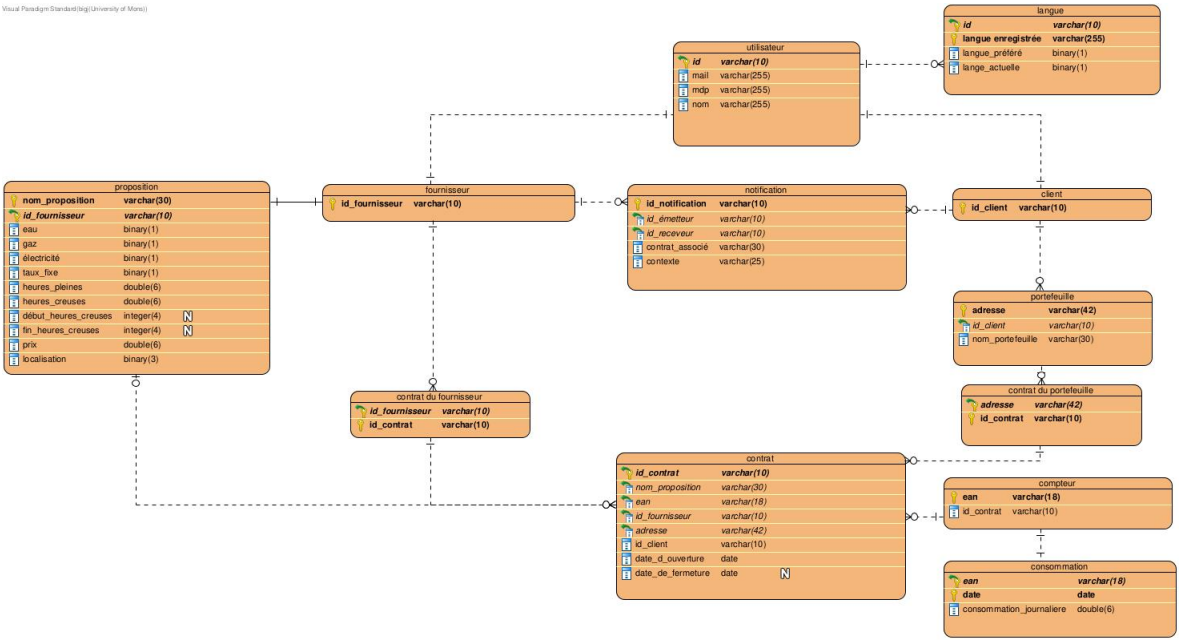
En plus de ça, un contexte relativement court sera posé jusqu'à plus ample information.

PK(notification) = `id_notification`

FK(notification) REFS client = `id_client`
FK(notification) REFS fournisseur = `id_fournisseur`

[4]source : `https://shrinkthatfootprint.com/average-household-electricity-consumption/`

## 3.14 schéma


Visual Paradigm Standard(bigj(University of Mons))
langue
id varchar(10)
langue enregistrée varchar(255)
langue_préféré binary(1)
lange_actuelle binary(1)
utilisateur
id varchar(10)
mail varchar(255)
mdp varchar(255)
nom varchar(255)
proposition
nom_proposition varchar(30)
id_fournisseur varchar(10)
eau binary(1)
gaz binary(1)
électricité binary(1)
faux_fixe binary(1)
heures_pleines double(6)
heures_creuses double(6)
début_heures_creuses integer(4) N
fin_heures_creuses integer(4) N
prix double(6)
localisation binary(3)
fournisseur
id_fournisseur varchar(10)
notification
id_notification varchar(10)
id_émetteur varchar(10)
id_receveur varchar(10)
contrat_associé varchar(30)
contexte varchar(25)
client
id_client varchar(10)
portefeuille
adresse varchar(42)
id_client varchar(10)
nom_portefeuille varchar(30)
contrat du fournisseur
id_fournisseur varchar(10)
id_contrat varchar(10)
contrat du portefeuille
adresse varchar(42)
id_contrat varchar(10)
contrat
id_contrat varchar(10)
nom_proposition varchar(30)
ean varchar(18)
id_fournisseur varchar(10)
adresse varchar(42)
id_client varchar(10)
date_d_ouverture date
date_de_fermeture date N
compteur
ean varchar(18)
id_contrat varchar(10)
consommation
ean varchar(18)
date date
consommation_journaliere double(6)

# 4 Diagramme des classes pour le serveur.

## 4.1 Introduction

Étant donné la taille du projet, nous avons évidemment décidé de séparer le projet en plusieurs parties. Au niveau du serveur, nous pouvons compter 3 grandes parties.

- Le package API

- Le package database

- Le package dataObject

En outre, il y a également la classe de base nommé App. Nous en reparlerons par après.

## 4.2 Le package API:

Ce package contient toute la partie logique de l'ApiRest. Pour créer le serveur API nous avons décidé d'utiliser le framework Vertx[5], nous expliquerons notre choix dans le rapport sur l'implementation. Pour créer une ApiRest, nous avons d'abord besoin d'une classe (MyApi) qui hérite de AbstractVerticle (une classe abstraite venant du package vertx). MyApi permet de lancer le serveur à l'aide de la méthode start.

Avant de la lancer, il faut d'abord configurer les routes de notre API. Notre API étant conséquente, nous avons décidé de la séparer en 4 sous-routes (parties distinctes).

a) **LogApi:**
Celle-ci va être utilisée pour toutes les opérations liées au login. Autrement dit, pouvoir se connecter ou se déconnecter, créer un compte et réinitialiser son mot de passe en cas d'oubli. Nous avons aussi rajouter une méthode getCode pour envoyer un code de vérification par mail lorsque l'utilisateur veut créer un compte ou pour réinitialiser son mot de passe.

b) **ClientApi:**
Celle-ci va être utilisée pour toutes les requêtes liées aux clients lorsqu'ils seront connectés. Ils pourront effectuer toutes les opérations qu'ils souhaitent au niveau du portefeuille ainsi que voir tous leurs contrats et pour finir voir toutes les propositions des fournisseurs ou une en particulier. Le client pourra évidemment proposer de conclure un contrat en fonction des propositions précédemment vue.

c) **ProviderApi:**
Celle-ci va être utilisée pour toutes les requêtes liées aux fournisseurs lorsqu'ils seront connectés. Ils pourront effectuer toutes les opérations qu'ils souhaitent au niveau des leurs propositions mais également voir tous les clients de l'application, tous leurs clients ou un client en particulier. Contrairement au client, il a la possibilité de supprimer une ou toutes les données de consommations liées à un contrat. De plus, tout comme le client, il pourra proposer de conclure un contrat à n'importe quelle client à partir de l'une de ces propositions.

d) **CommonApi:**
Celle-ci va être utilisée pour toutes les requêtes qui sont communes aux clients et aux fournisseurs lorsqu'ils seront connectés. On y trouve les méthodes pour les langues, pour les notifications et pour les données de consommations. Il y a également une méthode pour voir un contrat en particulier et une qui permet de changer de mot de passe.

[5] `https://thierry-leriche-dessirier.developpez.com/tutoriels/java/creer-API-rest-vertx-5-minutes/`

Chacune de ces parties vont donc créer une sous-route grâce à la méthode getSubRouter car elles implémentent toutes l'interface Router. Et c'est donc dans la méthode start de MyApi que toutes ces sous-routes vont être rassemblées pour finalement lancer le serveur API.

Notez que toutes les méthodes qui vont être appelées par l'API sont en privé car seul le serveur API peut les appeler et elles ont le même argument (routingContext) car c'est cette variable qui contient la requête de l'utilisateur ainsi que le corps de la requête. Et c'est également par cette variable que nous pourrons répondre à l'utilisateur.

Nous constatons également que ces quatre parties héritent d'une classe abstraite nommée AbstractToken. Comme son nom l'indique, elle va gérer les tokens. Autrement dit, lorsqu'un utilisateur se connecte, on va créer un token et c'est grâce à ce token qu'il pourra envoyer des requêtes aux classes clientApi, ProviderApi et CommonApi. Le token se supprimera automatiquement après un certain délai (15 minutes) ou lorsque quand l'utilisateur se déconnectera. Notez que si le token est expiré, l'utilisateur devra se reconnecter.

## 4.3 Le package database.

Le package database contient toutes les méthodes qui vont communiquer avec la base de données. Une grande partie de celles-ci ont exactement le même nom que dans le package API puisqu'elles sont contenues dans la suite du programme. L'utilisateur envoie une requête à l'API, en fonction de la requête, la bonne méthode est appelée et appellera la méthode de database portant le même nom pour accéder aux données. Pour séparer le programme, nous avons divisé cette partie en trois. Nous retrouvons donc une classe mère et deux classes filles.

CommonDB est la classe mère de ce package, elle reprend toutes les méthodes communes aux clients et aux fournisseurs comme son nom l'indique. Outre les méthodes de l'API, nous retrouvons également les méthodes createNotification, deleteNotification et createContract car elles devront être appelées à la suite d'une requête et non directement. Nous avons aussi décidé d'ajouter les méthodes createId, getDataOfTable et deleteDataOfTable car ce sont des actions qui seront souvent utilisées et que nous pouvons donc généraliser.

ClientDB et ProviderDB sont les deux classes enfants, elles s'occupent respectivement du côté client et du côté fournisseur comme leurs noms l'indiquent. Elles héritent de la classe CommonDB pour avoir accès à certaines méthodes comme par exemple createId, getDataOfTable et deleteDataOfTable. Notez qu'au niveau du client nous avons rajouté une méthode walletIsEmpty qui permet de savoir si un portefeuille du client est vide.

## 4.4 Le package dataObject.

Le package dataObject contient toutes les classes représentant un objet, c'est-à-dire:

- Les portefeuilles.

- Les contrats.

- Les clients (du point de vue des fournisseurs).

- Les propositions (que les fournisseurs ont créées).

- Les notifications.

Notez également que nous avons rajouter une classe TypeEnergy pour énumerer tous les types d'énergies.

Nous avons décidé de créer ces objets pour transférer facilement les données. En effet toutes les classes ne contiennent que des attributs avec des assesseurs à quelques exceptions près. Notez que nous aurions donc pu mettre les variables en final mais nous avons préféré laisser comme ça car dans le cas contraire nous aurions eu des constructeurs vraiment longs. Ces classes seront donc instanciées lorsque nous recevrons un objet par l'API ou quand nous prendrons des données de la base de données pour facilement les envoyer à l'utilisateur en format json (io.vertx.core.json.Json).

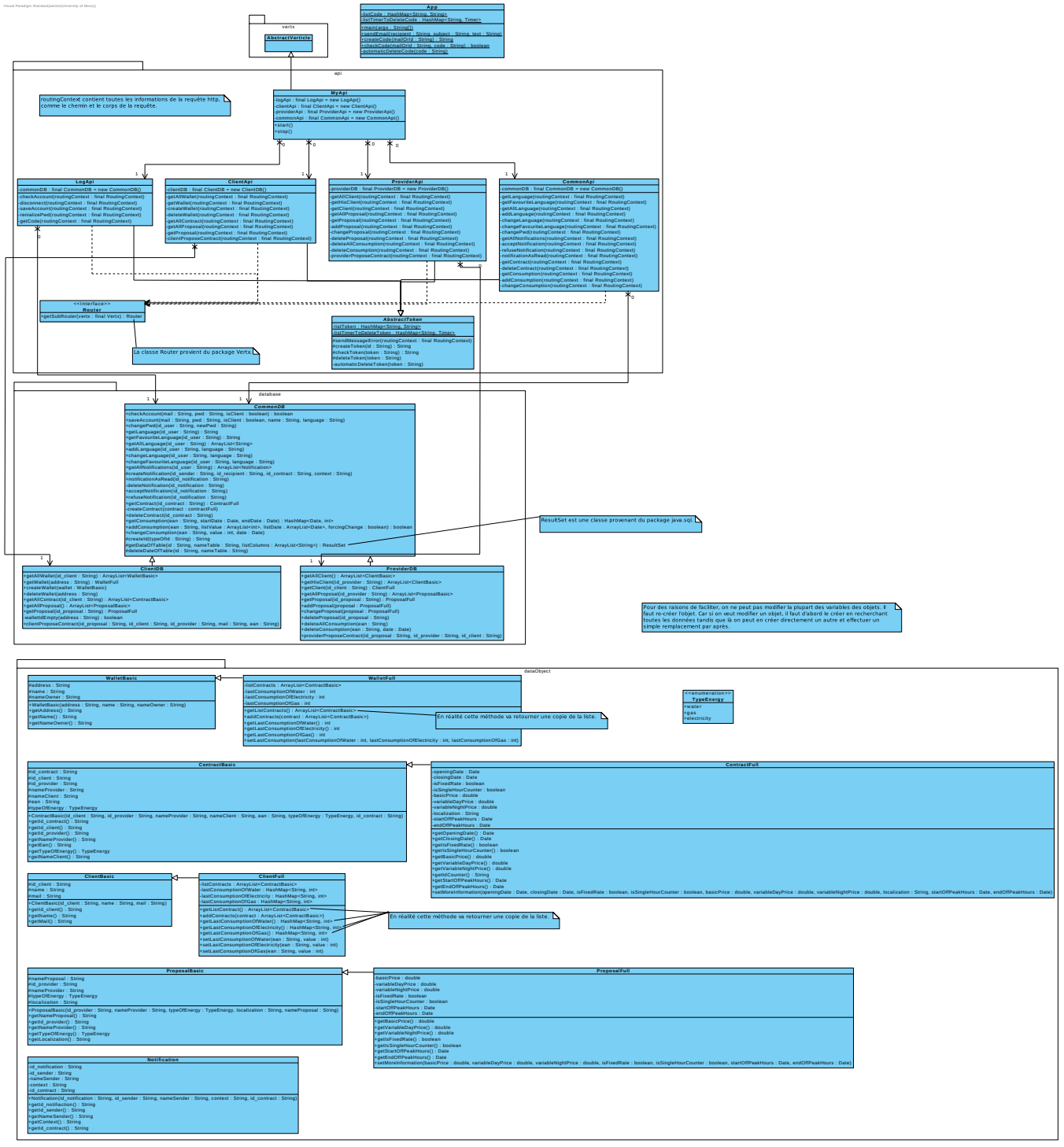
À l'exception de la classe notification, nous avons choisi de séparer chaque sorte d'objets en deux parties, une partie basique avec les informations primaires et une partie complète avec le reste des informations. Cela nous permet, lorsque l'utilisateur souhaite par exemple voir tous ces portefeuilles, d'envoyer seulement le nécessaire au lieu d'envoyer toutes les données en une fois. Ce n'est que lorsque l'utilisateur cliquera sur un portefeuille en particulier que nous enverrons toutes les données correspondant à celui-ci.

## 4.5 La classe App.

La classe App est le début du programme. C'est elle qui est chargée de lancer le programme comme toutes applications java grâce à la méthode nommée "main". En effet, cette méthode va lancer l'API de la classe Myapi. Notez que nous n'avons pas directement lié ces deux classes entre elles car l'API se lance indirectement par le framework Vertx.

La classe App contient également quelques méthodes statiques utilisées par les autres parties du projet. Ce sont ces méthodes qui s'occupent de la partie mail du projet, notamment "sendEmail" qui enverra une mail à un utilisateur qui souhaite créer un compte, réinitialiser son mot de passe ou bien le changer. Les autres méthodes s'occupent du "code". En effet, nous avons décidé que lorsqu'un utilisateur souhaite réaliser une des tâches précédentes, nous allons lui envoyer un code par mail avec quelques explications. Nous avons donc besoin d'une méthode pour générer le code aléatoirement, une pour vérifier que l'utilisateur a entré le bon code et pour finir une pour supprimer le code si l'utilisateur a mis trop de temps pour l'envoyer.

Notez que pour envoyer un mail, nous utilisons l'API javaxmail.


Visual Paradigm Standard(admin(University of Mons))
vertx
AbstractVerticle
App
listCode : HashMap<String, String>
listTimerToDeleteCode : HashMap<String, Timer>
main(args : String[])
sendEmail(recipient : String, subject : String, text : String)
createCode(mailOrId : String) : String
checkCode(mailOrId : String, code : String) : boolean
automaticDeleteCode(code : String)
api
routingContext contient toutes les informations de la requête http, comme le chemin et le corps de la requête.
MyApi
logApi : final LogApi = new LogApi()
clientApi : final ClientApi = new ClientApi()
providerApi : final ProviderApi = new ProviderApi()
commonApi : final CommonApi = new CommonApi()
+start()
+stop()
LogApi
-commonDB : final CommonDB = new CommonDB()
-checkAccount(routingContext : final RoutingContext)
-disconnect(routingContext : final RoutingContext)
-saveAccount(routingContext : final RoutingContext)
-serializePwd(routingContext : final RoutingContext)
-getCode(routingContext : final RoutingContext)
ClientApi
-clientDB : final ClientDB = new ClientDB()
-getAllWallet(routingContext : final RoutingContext)
-getWallet(routingContext : final RoutingContext)
-createWallet(routingContext : final RoutingContext)
-deleteWallet(routingContext : final RoutingContext)
-getAllContract(routingContext : final RoutingContext)
-getAllProposal(routingContext : final RoutingContext)
-getProposal(routingContext : final RoutingContext)
-clientProposeContract(routingContext : final RoutingContext)
ProviderApi
-providerDB : final ProviderDB = new ProviderDB()
-getAllClient(routingContext : final RoutingContext)
-getThisClient(routingContext : final RoutingContext)
-getClient(routingContext : final RoutingContext)
-getAllProposal(routingContext : final RoutingContext)
-getProposal(routingContext : final RoutingContext)
-addProposal(routingContext : final RoutingContext)
-changeProposal(routingContext : final RoutingContext)
-deleteProposal(routingContext : final RoutingContext)
-deleteAllConsumption(routingContext : final RoutingContext)
-deleteConsumption(routingContext : final RoutingContext)
-providerProposeContract(routingContext : final RoutingContext)
CommonApi
-commonDB : final CommonDB = new CommonDB()
-getLanguage(routingContext : final RoutingContext)
-getFavouriteLanguage(routingContext : final RoutingContext)
-getAllLanguage(routingContext : final RoutingContext)
-addLanguage(routingContext : final RoutingContext)
-changeLanguage(routingContext : final RoutingContext)
-changeFavouriteLanguage(routingContext : final RoutingContext)
-changePwd(routingContext : final RoutingContext)
-getAllNotifications(routingContext : final RoutingContext)
-acceptNotification(routingContext : final RoutingContext)
-refuseNotification(routingContext : final RoutingContext)
-notificationAsRead(routingContext : final RoutingContext)
-getContract(routingContext : final RoutingContext)
-deleteContract(routingContext : final RoutingContext)
-getConsumption(routingContext : final RoutingContext)
-addConsumption(routingContext : final RoutingContext)
-changeConsumption(routingContext : final RoutingContext)
<<Interface>>
Router
+getSubRouter(verts : final Vertx) : Router
La classe Router provient du package Veritx.
AbstractToken
listToken : HashMap<String, String>
listTimerToDeleteToken : HashMap<String, Timer>
#sendMessageError(routingContext : final RoutingContext)
#createToken(id : String) : String
#checkToken(token : String) : String
#deleteToken(token : String)
-automaticDeleteToken(token : String)
database
CommonDB
+checkAccount(mail : String, pwd : String, isClient : boolean) : boolean
+saveAccount(mail : String, pwd : String, isClient : boolean, name : String, language : String)
+changePwd(id_user : String, newPwd : String)
+getLanguage(id_user : String) : String
+getFavouriteLanguage(id_user : String) : String
+getAllLanguage(id_user : String) : ArrayList<String>
+addLanguage(id_user : String, language : String)
+changeLanguage(id_user : String, language : String)
+changeFavouriteLanguage(id_user : String, language : String)
+getAllNotifications(id_user : String) : ArrayList<Notification>
#createNotification(id_sender : String, id_recipient : String, id_contract : String, context : String)
+notificationAsRead(id_notification : String)
-deleteNotification(id_notification : String)
+acceptNotification(id_notification : String)
+refuseNotification(id_notification : String)
+getContract(id_contract : String) : ContractFull
-createContract(contract : contractFull)
+deleteContract(id_contract : String)
+getConsumption(ean : String, startDate : Date, endDate : Date) : HashMap<Date, int>
+addConsumption(ean : String, listValue : ArrayList<int>, listDate : ArrayList<Date>, forcingChange : boolean) : boolean
+changeConsumption(ean : String, value : int, date : Date)
#createId(typeOfId : String) : String
#getDataOfTable(id : String, nameTable : String, listColumns : ArrayList<String>) : ResultSet
#deleteDateOfTable(id : String, nameTable : String)
ResultSet est une classe provenant du package java.sql.
ClientDB
+getAllWallet(id_client : String) : ArrayList<WalletBasic>
+getWallet(address : String) : WalletFull
+createWallet(wallet : WalletBasic)
+deleteWallet(address : String)
+getAllContract(id_client : String) : ArrayList<ContractBasic>
+getAllProposal() : ArrayList<ProposalBasic>
+getProposal(id_proposal : String) : ProposalFull
-walletIdEmpty(address : String) : boolean
+clientProposeContract(id_proposal : String, id_client : String, id_provider : String, mail : String, ean : String)
ProviderDB
+getAllClient() : ArrayList<ClientBasic>
+getThisClient(id_provider : String) : ArrayList<ClientBasic>
+getClient(id_client : String) : ClientFull
+getAllProposal(id_provider : String) : ArrayList<ProposalBasic>
+getProposal(id_proposal : String) : ProposalFull
+addProposal(proposal : ProposalFull)
+changeProposal(proposal : ProposalFull)
+deleteProposal(id_proposal : String)
+deleteAllConsumption(ean : String)
+deleteConsumption(ean : String, date : Date)
-providerProposeContract(id_proposal : String, id_provider : String, id_client : String)
Pour des raisons de facilité, on ne peut pas modifier la plupart des variables des objets. Il faut re-créer l'objet. Car si on veut modifier un objet, il faut d'abord le créer en recherchant toutes les données tandis que là on peut en créer directement un autre et effectuer un simple remplacement par après.
dataObject
WalletBasic
#address : String
#name : String
#nameOwner : String
+WalletBasic(address : String, name : String, nameOwner : String)
+getAddress() : String
+getName() : String
+getNameOwner() : String
WalletFull
-listContracts : ArrayList<ContractBasic>
-lastConsumptionOfWater : int
-lastConsumptionOfElectricity : int
-lastConsumptionOfGas : int
+getListContracts() : ArrayList<ContractBasic>
+addContracts(contract : ArrayList<ContractBasic>)
+getLastConsumptionOfWater() : int
+getLastConsumptionOfElectricity() : int
+getLastConsumptionOfGas() : int
+setLastConsumption(lastConsumptionOfWater : int, lastConsumptionOfElectricity : int, lastConsumptionOfGas : int)
En réalité cette méthode va retourner une copie de la liste.
<<enumeration>>
TypeEnergy
+water
+gaz
+electricity
ContractBasic
#id_contract : String
#id_client : String
#id_provider : String
#nameProvider : String
#nameClient : String
#ean : String
#typeOfEnergy : TypeEnergy
+ContractBasic(id_client : String, id_provider : String, nameProvider : String, nameClient : String, ean : String, typeOfEnergy : TypeEnergy, id_contract : String)
+getId_contract() : String
+getId_client() : String
+getId_provider() : String
+getNameProvider() : String
+getEan() : String
+getTypeOfEnergy() : TypeEnergy
+getNameClient() : String
ContractFull
-openingDate : Date
-closingDate : Date
-isFixedRate : boolean
-isSingleHourCounter : boolean
-basicPrice : double
-variableDayPrice : double
-variableNightPrice : double
-localization : String
-startOffPeakHours : Date
-endOffPeakHours : Date
+getOpeningDate() : Date
+getClosingDate() : Date
+getIsFixedRate() : boolean
+getIsSingleHourCounter() : boolean
+getBasicPrice() : double
+getVariableDayPrice() : double
+getVariableNightPrice() : double
+getIdCounter() : String
+getStartOffPeakHours() : Date
+getEndOffPeakHours() : Date
+setMoreInformation(openingDate : Date, closingDate : Date, isFixedRate : boolean, isSingleHourCounter : boolean, basicPrice : double, variableDayPrice : double, variableNightPrice : double, localization : String, startOffPeakHours : Date, endOffPeakHours : Date)
ClientBasic
#id_client : String
#name : String
#mail : String
+ClientBasic(id_client : String, name : String, mail : String)
+getId_client() : String
+getName() : String
+getMail() : String
ClientFull
-listContracts : ArrayList<ContractBasic>
-lastConsumptionOfWater : HashMap<String, int>
-lastConsumptionOfElectricity : HashMap<String, int>
-lastConsumptionOfGas : HashMap<String, int>
+getListContract() : ArrayList<ContractBasic>
+addContracts(contract : ArrayList<ContractBasic>)
+getLastConsumptionOfWater() : HashMap<String, int>
+getLastConsumptionOfElectricity() : HashMap<String, int>
+getLastConsumptionOfGas() : HashMap<String, int>
+setLastConsumptionOfWater(ean : String, value : int)
+setLastConsumptionOfElectricity(ean : String, value : int)
+setLastConsumptionOfGas(ean : String, value : int)
En réalité cette méthode va retourner une copie de la liste.
ProposalBasic
#nameProposal : String
#id_provider : String
#nameProvider : String
#typeOfEnergy : TypeEnergy
#localization : String
+ProposalBasic(id_provider : String, nameProvider : String, typeOfEnergy : TypeEnergy, localization : String, nameProposal : String)
+getNameProposal() : String
+getId_provider() : String
+getNameProvider() : String
+getTypeOfEnergy() : TypeEnergy
+getLocalization() : String
ProposalFull
-basicPrice : double
-variableDayPrice : double
-variableNightPrice : double
-isFixedRate : boolean
-isSingleHourCounter : boolean
-startOffPeakHours : Date
-endOffPeakHours : Date
+getBasicPrice() : double
+getVariableDayPrice() : double
+getVariableNightPrice() : double
+getIsFixedRate() : boolean
+getIsSingleHourCounter() : boolean
+getStartOffPeakHours() : Date
+getEndOffPeakHours() : Date
+setMoreInformation(basicPrice : double, variableDayPrice : double, variableNightPrice : double, isFixedRate : boolean, isSingleHourCounter : boolean, startOffPeakHours : Date, endOffPeakHours : Date)
Notification
-id_notification : String
-id_sender : String
-nameSender : String
-context : String
-id_contract : String
+Notification(id_notification : String, id_sender : String, nameSender : String, context : String, id_contract : String)
+getId_notification() : String
+getId_sender() : String
+getNameSender() : String
+getContext() : String
+getId_contract() : String

# 5 Sequence diagrams

## 5.1 Introduction

Les sequences diagrams nous permettent de détailler le chemin effectué lorsqu'un utilisateur souhaite intéragir avec l'application.

Dans le but de permettre une meilleure compréhension de ces diagrammes, nous les avons séparés en trois parties.

1. Common
2. Client
3. Provider

Afin de les réaliser convenablement, nous sommes repartis de notre class diagram et nos Use cases diagrams et avons représenté l'interaction des méthodes avec l'API et la base de données.

Avant de commencer, il est important de signaler dans le but d'éviter toute répétition, que la vérification du token s'effectuera pour chaque action lorsqu'un utilisateur est connecté.

Si le token est bon, on continue le programme, sinon, on envoie un message d'erreur à l'aide de la méthode "sendMessageError".

Pour rappel, l'utilisateur n'appelle pas directement les méthodes partant de sa lifeline mais utilise des requêtes de l'API. De plus, vous pouvez constater que dans le diagramme des classes, nous n'avons pas mis de type de retour aux méthodes de l'API. Cependant, dans les diagrammes de séquences, nous avons quand-même représenté une flèche de retour. Cela est dû au fait que pour retourner quelque chose à l'utilisateur, nous utilisons une méthode de la variable routingContext.

## 5.2 Common

Pour la partie commune aux clients et fournisseurs, nous retrouvons plusieurs Use Cases à décrire.

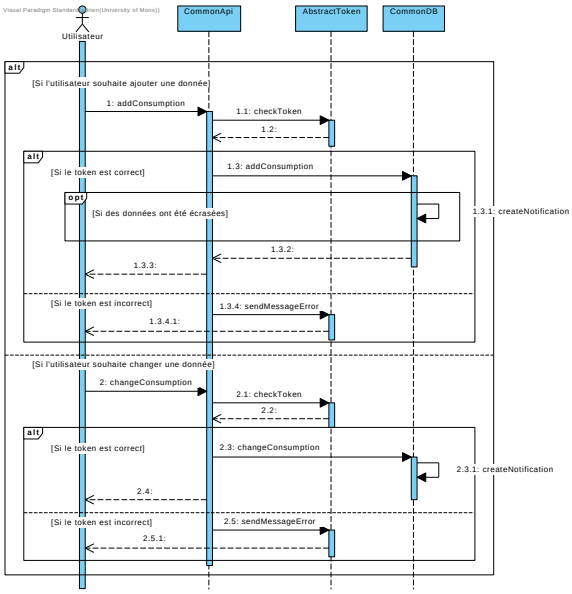
1. Gérer les données
2. Gérer les langues
3. Modifier son mot de passe
4. Répondre aux notifications
5. S'authentifier
6. Visualisation des données de consommations
7. Voir les contrats
8. Voir notifications

### 5.2.1 Gérer les données

Nous pouvons gérer les données de deux manières. L'utilisateur peut ajouter une donnée de consommation ou en changer une.

Pour ajouter une donnée, il suffit d'appeler la méthode addConsumption. Cette dernière peut écraser des données selon les paramètres utilisés. Lorsque des données sont écrasées, nous devons donc créer une notification pour prévenir l'autre utilisateur concerné par ce changement.

Pour changer une donnée, il suffit d'appeler la méthode changeConsumption. Celle-ci change automatiquement une donnée, nous devons donc créer une notification pour prévenir l'autre utilisateur.

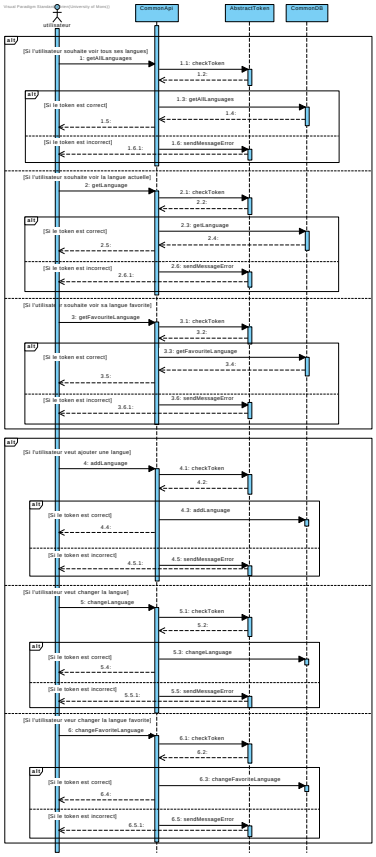


### 5.2.2 Gérer les langues

Nous pouvons gérer les langues à plusieurs niveaux. L'utilisateur peut voir toutes ses langues, sa langue actuelle et sa langue préférée. En plus de cela, il peut ajouter une langue, changer sa langue actuelle ou encore changer sa langue favorite.

Pour les trois premières fonctionnalités, le principe est le même. On appelle la méthode adéquate et celle-ci va t'être rappelée dans la partie base de donnée pour rechercher l'information désirée.

Concernant les autres fonctionnalités, on appelle également la méthode du même nom dans la partie base de données. Notez que ces méthodes ne retournent rien.

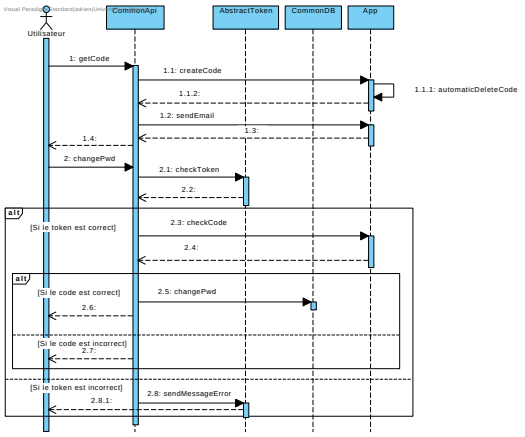


### 5.2.3 Modifier son mot de passe

Pour modifier son mot de passe de manière sécurisée, nous passons par l'adresse mail de l'utilisateur pour être certains que ce soit bien celui-ci qui veuille changer le mot de passe. Pour se faire, il faut tout d'abord faire appel à la méthode getCode, cette dernière va appeler les méthodes createCode et sendEmail de la classe App.

À noter que createCode va appeler automaticDeleteCode afin que le code se supprime automatiquement au bout d'un laps de temps au cas où l'utilisateur prendrait trop de temps ou n'utiliserait jamais ce code.

L'utilisateur recevra donc par mail un code de vérification. Il pourra ensuite entrer le code et son nouveau mot de passe sur le site pour finalement appeler la méthode changePwd de l'API. Avant de le changer, nous utilisons la méthode checkCode d'App pour vérifier que le code est correct. Si c'est bien le cas, alors nous pouvons appeler changePwd de CommonDB pour changer le mot de passe. Dans le cas contraire, nous renvoyons évidemment un message d'erreur.

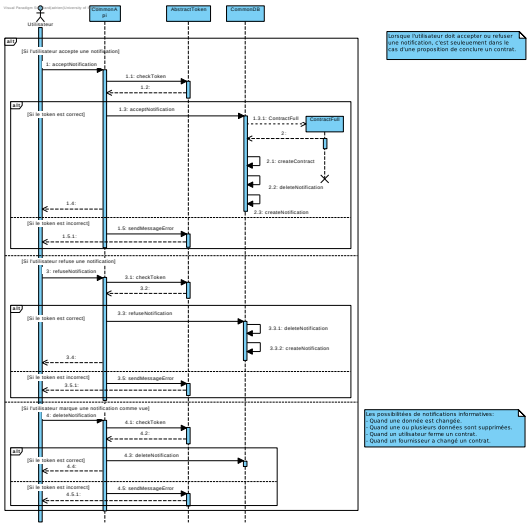


### 5.2.4 Répondre aux notifications

Répondre aux notifications implique trois possibilités. L'utilisateur peut l'accepter ou la refuser dans le cas d'une demande de contrat ou bien dans les autres cas, noter la notification comme lu.

Lorsqu'un utilisateur accepte une notification (une demande de contrat). Dans la partie base de données, nous commençons par créer un objet contractFull à partir de la proposition acceptée, ensuite nous faisons appel à la méthode createContract pour enregistrer le contrat. Nous pouvons donc supprimer la notification actuelle étant donné qu'elle vient d'être acceptée. Pour finir, nous créons une nouvelle notification pour informer à l'autre utilisateur que sa demande a été acceptée.

Dans le cas d'une notification informative, l'utilisateur peut la marquer comme lue. Ce qui implique de la supprimer en appelant la méthode deleteNotification.



### 5.2.5 S'authentifier

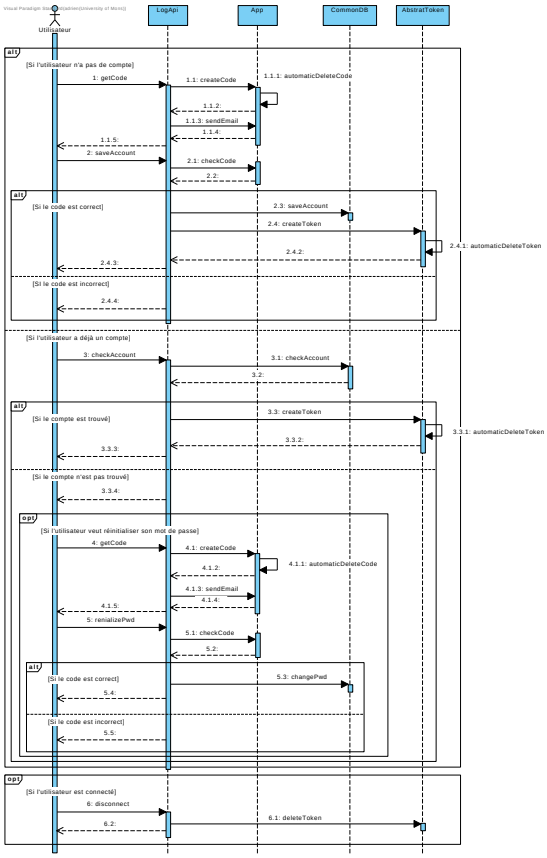
Lorsque l'utilisateur arrive sur la fenêtre de connexion. Deux cas de base sont possibles. Si l'utilisateur n'a pas de compte ou au contraire s'il en possède déjà un.

Pour créer un compte, suivant le même principe que le changement de mot de passe. Un code va être crée et envoyé à l'adresse mail que l'utilisateur vient d'entrer grâce aux méthodes déjà expliquées ci-dessus. Quand l'utilisateur souhaite sauvegarder son compte, la méthode saveAcccount est invoquée. Après vérification du code, cette dernière est ré-appelée dans CommonDB pour sauvegarder le compte.
Suite à cela, Nous pouvons créer un token de connexion grâce à la méthode createToken de la classe abstraite AbstractToken et l'envoyer à l'utilisateur pour ses prochaines requêtes car en créant son compte, il s'est également connecté. Notez que lors de la création du token, nous appelons automaticDeleteToken pour limiter la durée de vie du token.

Si l'utilisateur possède déjà un compte, nous vérifions les données de connexion grâce à checkAccount. Si les données sont correctes, nous pouvons créer un token et l'envoyer comme expliqué précédemment.

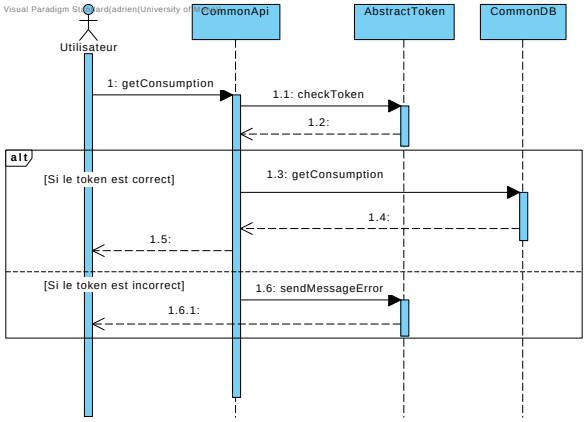
Si l'utilisateur possède un compte mais ne parvient pas à se connecter. Il a la possibilité de réinitialiser son mot de passe. Le processus utilisé est le même que lors du changement de mot de passe expliqué précédemment.

Une fois connecté, l'utilisateur peut de toute évidence se déconnecter en faisant appel à la méthode disconnect. Celle-ci servant juste à appeler deleteToken pour supprimer le token.


Utilisateur
LogApi
App
CommonDB
AbstratToken
alt
[Si l'utilisateur n'a pas de compte]
1: getCode
1.1: createCode
1.1.1: automaticDeleteCode
1.1.2:
1.1.3: sendEmail
1.1.4:
1.1.5:
2: saveAccount
2.1: checkCode
2.2:
alt
[Si le code est correct]
2.3: saveAccount
2.4: createToken
2.4.1: automaticDeleteToken
2.4.2:
2.4.3:
[Si le code est incorrect]
2.4.4:
[Si l'utilisateur a déjà un compte]
3: checkAccount
3.1: checkAccount
3.2:
alt
[Si le compte est trouvé]
3.3: createToken
3.3.1: automaticDeleteToken
3.3.2:
3.3.3:
[Si le compte n'est pas trouvé]
3.3.4:
opt
[Si l'utilisateur veut réinitialiser son mot de passe]
4: getCode
4.1: createCode
4.1.1: automaticDeleteCode
4.1.2:
4.1.3: sendEmail
4.1.4:
4.1.5:
5: renializePwd
5.1: checkCode
5.2:
alt
[Si le code est correct]
5.3: changePwd
5.4:
[Si le code est incorrect]
5.5:
opt
[Si l'utilisateur est connecté]
6: disconnect
6.1: deleteToken
6.2:

### 5.2.6 Visualisation des données de consommations

Rien de plus simple. Nous invoquons la méthode getConsumption de l'API, ensuite elle est ré-appelée dans CommonDB pour récupérer les valeurs dans la base de données pour finalement les renvoyer.

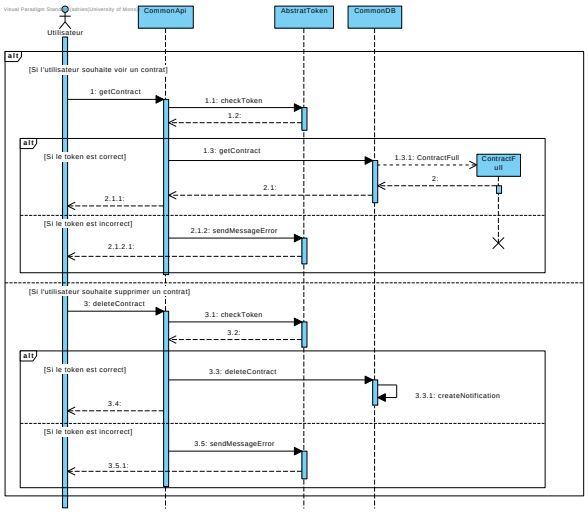


### 5.2.7 Voir les contrats

Au niveau des contrats dans la partie commune, il est possible de voir un contrat en particulier ainsi que d'en supprimer un.

Il suffit d'appeler la méthode getContract jusqu'à la base de données, nous pouvons ensuite créer un objet ContractFull pour facilement renvoyer les valeurs liées à cet objet. Notez qu'une fois envoyé, l'objet est détruit.

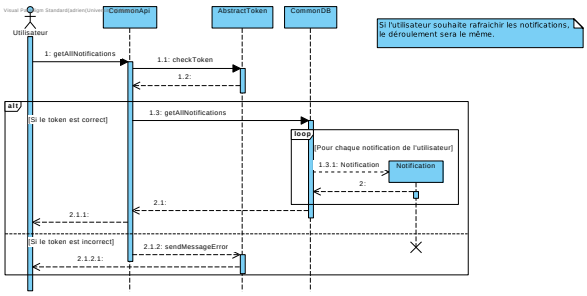
Comme son nom l'indique, deleteContract permet de supprimer un contrat. Une fois son devoir accompli, cette méthode doit également créer une notification pour avertir l'autre utilisateur impliqué par cette suppression. Il n'est pas utile que cette méthode renvoie quelque chose.



### 5.2.8 Voir notifications

Pour finir, l'utilisateur peut voir ses notifications. Pour cela, nous suivons toujours le même principe en appelant d'abord la méthode getAllNotifications au niveau de l'API et ensuite dans la partie base de données. Nous créons un objet Notification pour chaque notification stockée pour les envoyer dans une liste vers l'utilisateur. Tous les objets précédemment créés seront évidement supprimés à la suite de l'envoi de la liste.

Notez que lorsque l'utilisateur rafraîchit les notifications, c'est cette même méthode qui est appelée.



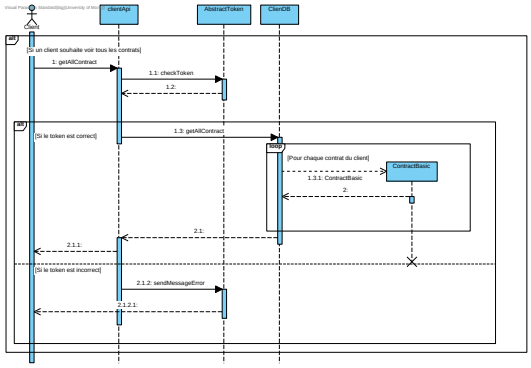
## 5.3 Client

### 5.3.1 Contrats

Commençons par le Use Case **"Voir les contrats"**, nous appelons la méthode "getAllContract" de clientApi et ensuite, la méthode du même nom de ClientDB.

Le but étant d'obtenir une ArrayList de "contractBasic", nous devons effecter une boucle afin de récupérer tous les "contractBasic".

Une fois effectuée, nous renvoyons cette liste.



### 5.3.2 Fournisseurs et contrats relatifs

Continuons avec l'Use Case **"Voir les fournisseurs et contrats relatifs"**. A cet effet, il y aura deux méthodes :

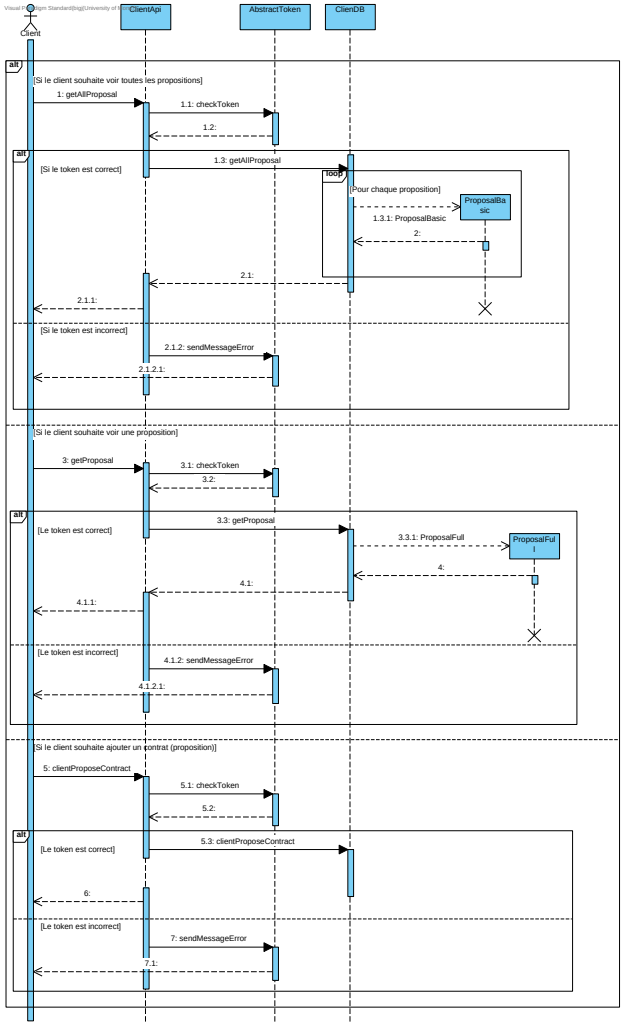
1. getAllProposal si le client souhaite voir toutes les propositions.
2. getProposal si le client souhaite voir une proposition en particulier.

Ces deux méthodes sont appelées une fois de clientApi et ensuite, une fois de ClientDB.

Dans le cas de "getAllProposal", la valeur de retour est une ArrayList de "proposalBasic" donc nous utiliserons une boucle et dans le cas de "getProposal", nous devons simplement retourner une instance de "proposalFull".

Concernant le Use Case **"Ajouter des contrats"**, nous appelons la méthode "clientProposeContract" de clientApi et ensuite, la méthode du même nom de ClientDB.

Etant donné que cette dernière ne doit rien retourner, il n'y aura pas de valeur de retour.


Client
ClientApi
AbstractToken
ClienDB
alt
[Si le client souhaite voir toutes les propositions]
1: getAllProposal
1.1: checkToken
1.2:
alt
[Si le token est correct]
1.3: getAllProposal
loop
[Pour chaque proposition]
ProposalBasic
1.3.1: ProposalBasic
2:
2.1:
2.1.1:
[Si le token est incorrect]
2.1.2: sendMessageError
2.1.2.1:
[Si le client souhaite voir une proposition]
3: getProposal
3.1: checkToken
3.2:
alt
[Le token est correct]
3.3: getProposal
3.3.1: ProposalFull
ProposalFull
4:
4.1:
4.1.1:
[Le token est incorrect]
4.1.2: sendMessageError
4.1.2.1:
[Si le client souhaite ajouter un contrat (proposition)]
5: clientProposeContract
5.1: checkToken
5.2:
alt
[Le token est correct]
5.3: clientProposeContract
6:
[Le token est incorrect]
7: sendMessageError
7.1:

### 5.3.3 Portefeuilles

L'Use case **"Voir les portefeuilles"** fonctionne de la même manière qu'expliqué précédemment pour "Voir les fournisseurs et contrats relatifs". En effet, il y aura deux méthodes :

1. getAllWallet si le client souhaite voir tous les portefeuilles. On doit donc retourner une ArrayList de "walletBasic".

2. getWallet si le client souhaite voir un portefeuille en particulier. On doit donc retourner une instance de "walletFull".

Passons maintenant aux Use Cases **"Créer un portefeuille"** et **"Fermer un portefeuille"**.

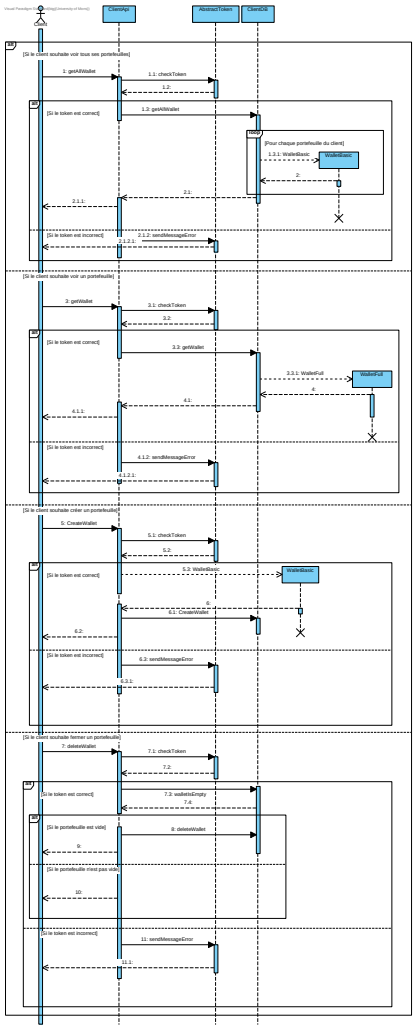
Pour **créer un portefeuille**, nous appelons donc la méthode "createWallet" de clientApi et ensuite, la méthode du même nom de ClientDB.

Nous créons une instance de "walletBasic", que nous devons enregistrer dans "ClientDB" avant de clôturer l'exécution de la méthode car il s'agit d'un nouveau portefeuille à stocker.

Pour **supprimer un portefeuille**, nous appelons la méthode "deleteWallet" de clientApi.

Cependant, avant d'appeler la méthode du même nom de ClientDB, nous devons vérifier que le portefeuille est vide.

Si c'est le cas le portefeuille sera bien supprimé, sinon, on renverra une erreur.


ClientApi
AbstractToken
ClientCtrl
[Si le client souhaite voir tous ses portefeuilles]
1: getAllWallet
1.1: checkToken
1.2:
[Si le token est correct]
1.3: getAllWallet
loop
[Pour chaque portefeuille du client]
1.3.1: WalletBasic
WalletBasic
2:
2.1:
2.1.1:
[Si le token est incorrect]
2.1.2: sendMessageError
2.1.2.1:
[Si le client souhaite voir un portefeuille]
3: getWallet
3.1: checkToken
3.2:
[Si le token est correct]
3.3: getWallet
3.3.1: WalletFull
WalletFull
4:
4.1:
4.1.1:
[Si le token est incorrect]
4.1.2: sendMessageError
4.1.2.1:
[Si le client souhaite créer un portefeuille]
5: CreateWallet
5.1: checkToken
5.2:
[Si le token est correct]
5.3: WalletBasic
WalletBasic
6:
6.1: CreateWallet
6.2:
[Si le token est incorrect]
6.3: sendMessageError
6.3.1:
[Si le client souhaite fermer un portefeuille]
7: deleteWallet
7.1: checkToken
7.2:
[Si le token est correct]
7.3: walletIsEmpty
7.4:
[Si le portefeuille est vide]
8: deleteWallet
9:
[Si le portefeuille n'est pas vide]
10:
[Si le token est incorrect]
11: sendMessageError
11.1:

## 5.4 Provider

Tout d'abord, nous avons fait en sorte qu'un use case peut être représenté via un choix alternatif. Aussi, pour éviter de répéter la même chose dans le rapport, nous posons que toutes les méthodes liées à l'API appèleront la méthode **checkToken** de **AbstractToken** ce qui impliquera deux choix alternatifs : "Si le token est correct" et "Si le token est incorrect". Dans le dernier cas, on renverra un message d'erreur au fournisseur.

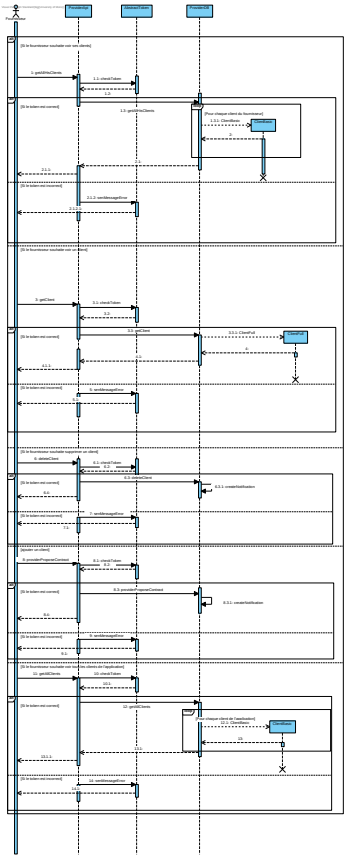
### 5.4.1 Gestion des clients

Commençons par le use case "voir ses clients". Afin qu'un fournisseur puisse voir ses clients, il lui suffit d'appeler la méthode **getAllHisClients** dans la classe **ProviderApi** et **ProviderDB**. Etant donné qu'on retourne une collection d'objet **ClientBasic**. Nous devons utiliser une boucle. Pour voir un seul de ses clients, il lui suffit d'appeler la méthode **getClient**.

Passons maintenant au use case "supprimer un client". Pour faire cela, la méthode **deleteClient** sera utilisé. En concéquence de ça, une notification sera envoyé au client ce qui nous oblige à faire appel à la méthode **createNotification**.

Pour ajouter un client, cela se fera par le biais d'une proposition du fournisseur au client. Ceci faisant appel à la méthode **providerProposeContract**. Nous posons comme hypothèse que le client n'a aucun contrat en cours avec le fournisseur.

Pour qu'un fournisseur puisse ajouter des clients. Il doit d'abord avoir accès à tous les clients de l'application. Ceci se fera par la méthode **getAllClients**. De la même manière que voir "ses clients", nous utiliserons une boucle.



### 5.4.2 Gestion des propositions

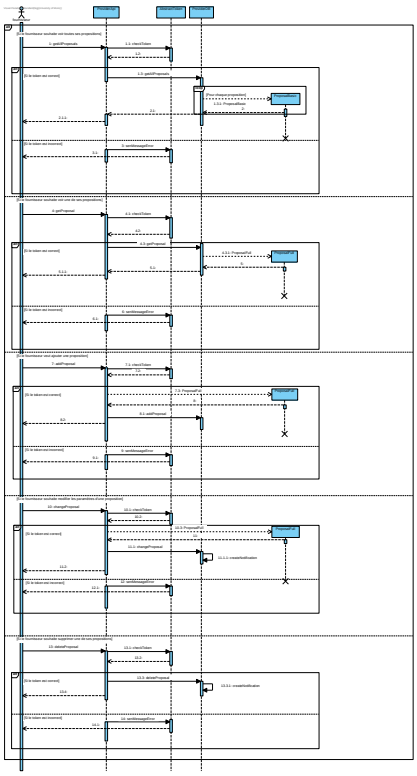
Avant de parler des diagrammes de séquence de cette partie. Nous avons choisis de mettre les contrats d'un fournisseur et ses propositions dans les use cases parlant des contrats. De plus, Seules les propositions seront abordés dans cette partie étant qu'un contrat sera commun au fournisseur et au client(*ie:* **??**).

Pour commencer, nous allons regarder comment le fournisseur peut avoir accès à toutes ses propositions. Pour cela, il devra utiliser la méthode **getAllProposals**. De plus, une boucle permettra de récupérer tous les objets. Si le fournisseur veut avoir plus de précision sur une de ses propositions, alors il aura juste besoin d'apeller la méthode **getProposal** en ayant l'identifiant de la proposition au préalable.

Après ça, Si le fournisseur veut ajouter une proposition. La méthode **addProposal** sera utilisé. Il faudra néanmoins créer un objet **ProposalFull** qui sera ensuite ajouté par la précédente méthode à la base de données.

Ensuite, si le fournisseur souhaite modifier les paramètres d'une de ses propositions. Il n'aura qu'à faire un nouvel objet qui servira de "remplacant" à l'ancienne proposition. La méthode **changeProposal** sera appelée par la suite pour faire le changement des paramètres directement dans la base de données. A noter que modifier les paramètres d'une proposition revient à changer les contrats avec les clients qui ont pris cette proposition. De ce fait, une notification sera envoyé via la méthode **createNotification**.

Pour finir cette partie, nous allons parler de comment fonctionne la procédure pour supprimer une proposition. Nous avons juste à faire appel à la méthode **deleteProposal**. De manière similaire au fait de modifier, supprimer implique la supression des contrats des clients ayant comme base cette proposition ce qui implique l'envoie d'une notification par la méthode **createNotification**.

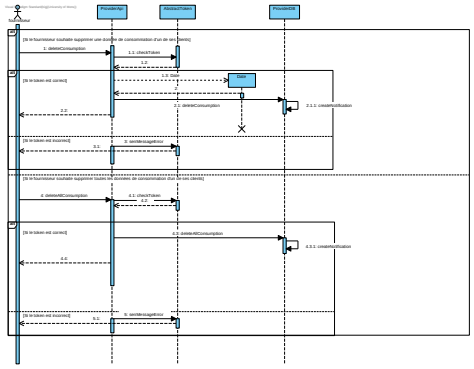


### 5.4.3 Gestion de la consommation

Cette sous-sous-section parlera de la gestion de la consommation d'un client par le fournisseur. A noter que le fait pour un fournisseur de voir la consommation de ses clients est la même procédure qu'un client qui va voir ses consommations à l'exception qu'un fournisseur ne peut voir la consommation créée par le client liée à son contrat.

Tout d'abord, le fournisseur souhaitant supprimer une donnée de consommation devra passer par la méthode **deleteConsumption** de l'API. En plus de ça, il aura besoin d'un objet date ne contenant que l'année, le mois et le jour. Après ça il pourra supprimer la donnée qu'il souhaite avec la méthode portant le même nom que celle de l'API. Une notification sera envoyé au client avec la méthode **createNotification**.

Ensuite, si le fournisseur souhaite supprimer toutes les données de consommation d'un client. Il aura juste besoin de la méthode **deleteAllConsumption**. De même que le paragraphe du dessus, une notification sera créée.



# 6 Interface

## 6.1 Introduction

Nous avons réalisé notre interface principalement sur base de nos usecases et overview diagrams ainsi que les choix effectués concernant la base de données et le class diagram.

Cette manière de procéder nous a permis de structurer nos idées afin de fournir l'interface la plus intuitive possible.

De plus, nous avons décidé de séparer l'interface en **trois parties** :

1. Le système de logs
2. L'interface client
3. L'interface fournisseur

ce qui nous semblait plus approprié étant donné que nous devons obtenir deux applications distinctes, une pour le fournisseur et l'autre pour le client.

Il est important de noter que nous avons utilisé le logiciel "Figma" pour réaliser ces maquettes.

## 6.2 Système de logs

Sur la page principale de connexion, l'utilisateur devra préciser s'il est client ou fournisseur en cochant la case apppropriée et pourra également changer de langue s'il le souhaite.

Il pourra dès lors entrer ses informations et **se connecter**.

Cependant s'il a **oublié son mot de passe**, un bouton sera prévu à cet effet. Ce dernier enverra l'utilisateur sur une nouvelle page où il devra inscrire le code qu'il aura reçu par mail et son nouveau mot de passe.

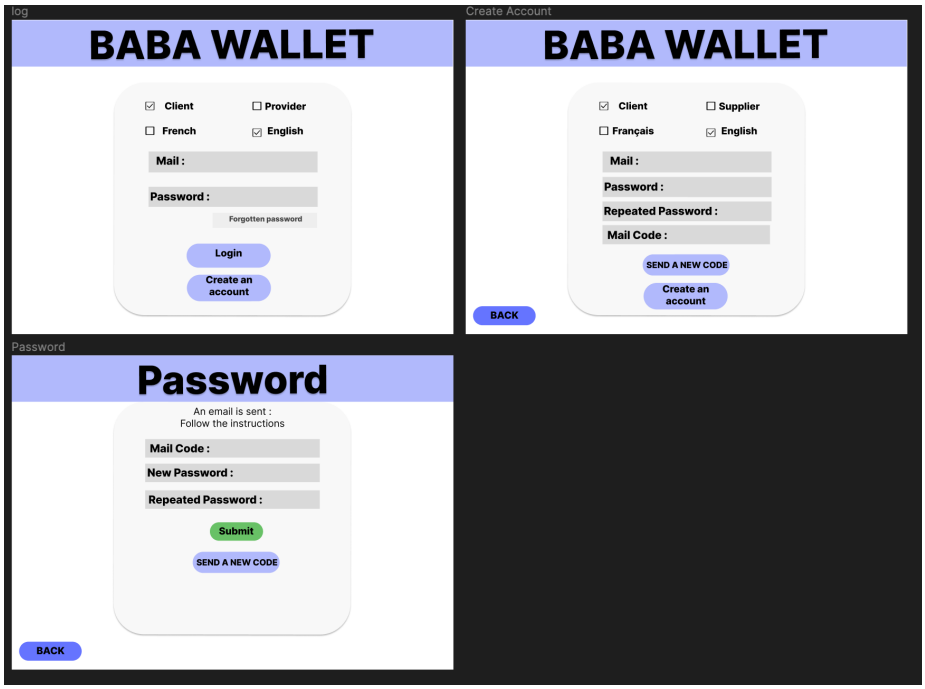
Un bouton "Submit" aura pour but de confirmer ce que l'utilisateur a entré et un bouton "Back" sera présent si jamais ce dernier veut retourner en arrière.

S'il n'a pas encore de compte, l'utilisateur aura la possibilité d'en **créer un nouveau**. Le sytème fonctionnera comme la connexion expliquée précédemment.

La seule différence réside lorsque ce dernier appuiera sur le bouton "Create an account". L'utilisateur sera alors envoyé sur une nouvelle page où on lui demandera de confirmer la création de son compte en entrant le code qu'il aura reçu par mail pour plus de sécurité.

Pour finir, de la même manière que pour le changement de mot de passe, un bouton "Finish" aura pour but de confirmer ce que l'utilisateur a entré et un bouton "Back" sera présent si jamais ce dernier veut retourner en arrière.

En outre, l'utilisateur pourra demander qu'on lui renvoie un mail si le précédent n'est pas reçu.




## 6.3 Points communs

Les points communs entre l'interface client et fournisseur sont

1. **Les notifications :**
   L'utilisateur verra le fournisseur ou client en question, le contexte et le contrat lié à la notification.
   Il pourra également observer plus en détail le contrat avec le bouton "See", marquer commme lue les notifications grâce au bouton "OK" et les accepter ou refuser dans le cadre d'une demande.
   En outre, la possibilité d'actualiser lui-même s'il le désire lui est laissée.

2. **Les paramètres :**
   Concernant le changement de mot de passe, cela se déroule exactement de la même manière que dans le système de logs. Pour la langue, l'utilisateur pourra en ajouter, choisir sa préférée et changer celle étant active.
   Dans tous les cas, ce dernier sera conduit sur une page où il aura la possibilité de chercher une langue et ensuite soit de la télécharger, soit de la choisir en fonction du contexte.

## 6.4 Interface client

Lorsqu'un client se connectera, il arrivera sur la page d'acceuil ou plus précisément "Home".

Ce dernier aura alors les possibilités suivantes :

1. Voir ses portefeuilles ("Wallets")
2. Voir ses contrats ("Your contracts")
3. Voir ses notifications
4. Voir les fournisseurs et contrats relatifs ("See new contracts")
5. Se déconnecter
6. Aller sur la page des paramètres

Il est important de noter que ce n'est qu'à partir de ces six pages que le client pourra revenir sur la page "Home" étant donné le choix effectué dans notre overview diagram.

Néanmoins, ce dernier aura la possibilité de revenir en arrière à l'aide du bouton "Back" lorsqu'il sera sur des "sous-pages" de ces six sections principales.

Nous pouvons maintenant commencer par l'option **"Voir ses portefeuilles" ("Wallets")** :

Le client pourra à partir d'ici voir une liste reprenant tous les portefeuilles qu'il aura créé précédement.

Chaque rectangle représentant un portefeuille comprend :

1. Le nom du portefeuille
2. Le nom du propriétaire du portefeuille
3. L'adresse associée au portefeuille

Le bouton "Go" permettra de voir plus en détail un portefeuille et le bouton "+" sur le rectangle vide donnera la possibilité au client d'ajouter un nouveau portefeuille.

Lorsqu'il souhaitera **voir plus en détail un portefeuille**, le client aura accès au nom du portefeuille, au nom du propriétaire du portefeuille et à l'adresse associée au porteuille comme sur la page précédente.

Toutefois d'autres actions lui seront possibles :

1. Voir ses dernières consommations
2. Voir les contrats associés
3. Voir les contrats associés en détail (Bouton "Go")
4. Ajouter des consommations (Bouton "Add consumptions")
5. Fermer définitivement le portefeuille (Bouton "Close the Wallet")

Le bouton "Go" concernant les contrats associés l'amènera sur une page montrant le contrat que nous expliquerons prochainement.

Le bouton **"Add consumptions"** l'entraînera vers une page où il pourra voir ses données de consommations en graphique ou en tableau à l'aide des deux boutons placés en haut de la page.

Le réctangle gris les représentent et il pourra également séléctionner dans ce dernier s'il souhaite voir les données mensuellement, hebdomadairement,...

Le client pourra également s'il le désire exporter ses données. Nous pouvons noter que ce dernier exportera les données présentent dans le tableau gris au moment où il appuiera sur le bouton "Export".

En ce qui concerne le changement et l'ajout de consommation, celui-ci devra entrer la date correspondante, le type d'énergie et la valeur associée et devra ensuite confirmer ses modifications grâce aux boutons "Change" et "Add".

**Le bouton "+"** mentionné précédemment servant à ajouter un nouveau portefeuilles est connecté à une page qui lui demandera d'écrire le nom et l'adresse du nouveau portefeuille.

Cette manière de procéder est liée au fait que nous avons décidé qu'un portefeuille doit être créé avant d'associer un contrat à ce dernier.

La deuxième option est la suivante : **"Voir ses contrats" ("Your contracts")**

Une fois sur cette page, le client aura accès à sa liste de contrats et il pourra rechercher un contrat spécifique en tapant le code "EAN" ou le nom du fournisseur.

Les informations basiques qu'il verra sur chaque ligne de la liste sont le fournisseur et le type d'énergie.

Lorsqu'il cliquera sur le **bouton "Go"** la nouvelle page lui offrira les informations suivantes : le type d'énergie et le fournisseur de la même manière que sur la page précédente.

Les nouvelles informations disponibles sont les suivantes :

1. Le nom du contrat
2. Le portefeuille associé (Le code "EAN" ainsi que l'adresse)
3. La localisation (du contrat)
4. Le prix basique
5. Le prix dépendant du jour et la nuit
6. Le taux fixe ou variable
7. Les heures creuses
8. La date d'ouverture du contrat
9. La date de fermeture du contrat

Il pourra aussi fermer le contrat par le bouton "Close the contract".

La troisième et dernière section est **"Voir les fournisseurs et contrats relatifs ("See new contracts")"**

Sur cette page, le client pourra voir une liste reprenant les noms des fournisseurs, les types d'énergies et la localisation pour chaque contrat.

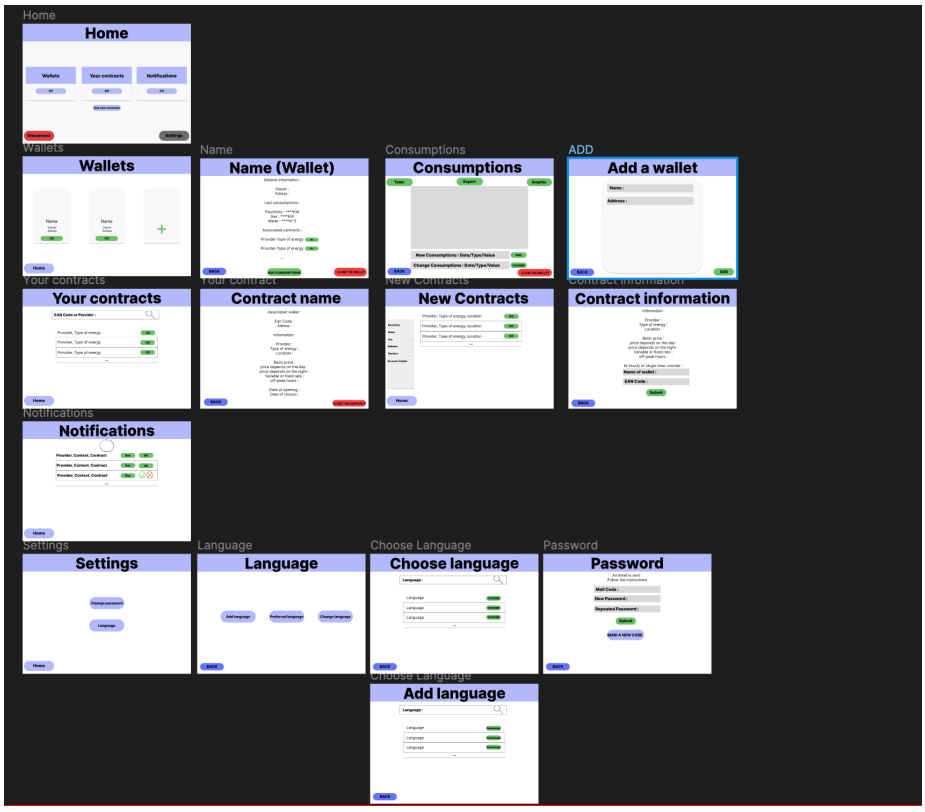
Pour lui rendre la tâche de recherche plus facile, nous avons mis à sa disposition une sélection sur le côté droit. Cette dernière permettra de choisir le type d'énergie souhaité (Electricité, gaz et eau) ainsi que la localisation.

Le **bouton "Go"** l'amènera sur les informations liées à un nouveau contrat :

1. Le fournisseur
2. Le type d'énergie
3. La localisation (du contrat)
4. Le prix basique
5. Le prix dépendant du jour et la nuit
6. Le taux fixe ou variable
7. Les heures creuses
8. Le type de compteur

Pour confirmer sa demande de contrat, le client devra entrer le nom du portefeuille auquel il souhaite l'associer ainsi que le code "EAN" et appuyer sur le bouton "Submit".

Nous ne reviendrons pas sur les points "Voir les notifications" et "Les paramètres", ces derniers étant expliqués dans les points communs.


Home
Home
Wallets
Wallets
Name
Name (Wallet)
Consumptions
Consumptions
ADD
Add a wallet
Your contracts
Your contracts
Contract name
New Contracts
Contract information
Notifications
Notifications
Settings
Settings
Language
Language
Choose Language
Choose language
Password
Password
Choose Language
Add language

## 6.5 Interface fournisseur

Lorsqu'un fournisseur se connectera, il arrivera sur la page d'acceuil ou plus précisément "Home".

Ce dernier aura alors les possibilités suivantes :

1. Voir ses clients ("Your clients")
2. Voir ses contrats ("Your contracts")
3. Voir ses notifications
4. Se déconnecter
5. Aller sur la page des paramètres

Tout comme pour l'interface client, il est important de noter que ce n'est qu'à partir de ces six pages que le fournisseur pourra revenir sur la page "Home" étant donné le choix effectué dans notre overview diagram.

Néanmoins, ce dernier aura la possibilité de revenir en arrière à l'aide du bouton "Back" lorsqu'il sera sur des "sous-pages" de ces six sections principales.

La première section est **Voir ses clients ("Your clients")**

Le founisseur pourra à partir d'ici voir une liste reprenant tous les noms de ses clients ainsi que leurs adresses mails.

**Le bouton "Go"** lui permettra de voir plus en détail un client, il aura accès à :

1. Son nom
2. Son adresse mail
3. Les contrats lui étant associés (Nom du contrat, code "EAN", le type d'énergie, la dernière consommation) :
   Le bouton "Go" lui donnera la possibilité de voir plus en détail le contrat de la même manière que lorsqu'un client souhaite voir ses contrats.
   Il aura la faculté de supprimer un contrat et voir les consommations.
   Contrairement au client, il ne pourra pas ajouter de nouvelles données de consommations mais simplement les modifier et les supprimer et ne pourra pas exporter les données mais en importer.
4. La suppresion d'un client (Bouton "Delete Client")

Le **bouton "Add clients"** le mènera sur une page où il pourra sélectionner un contrat à envoyer à un possible nouveau client.

Passons à la deuxième section : **Voir ses contrats ("Your contracts")**

Le founisseur pourra à partir d'ici voir une liste reprenant tous les noms de ses contrats, leurs types d'énergie ainsi que leurs localisations.

**Le bouton "Go"** lui donnera la possibilité de voir plus en détails ses contrats avec :

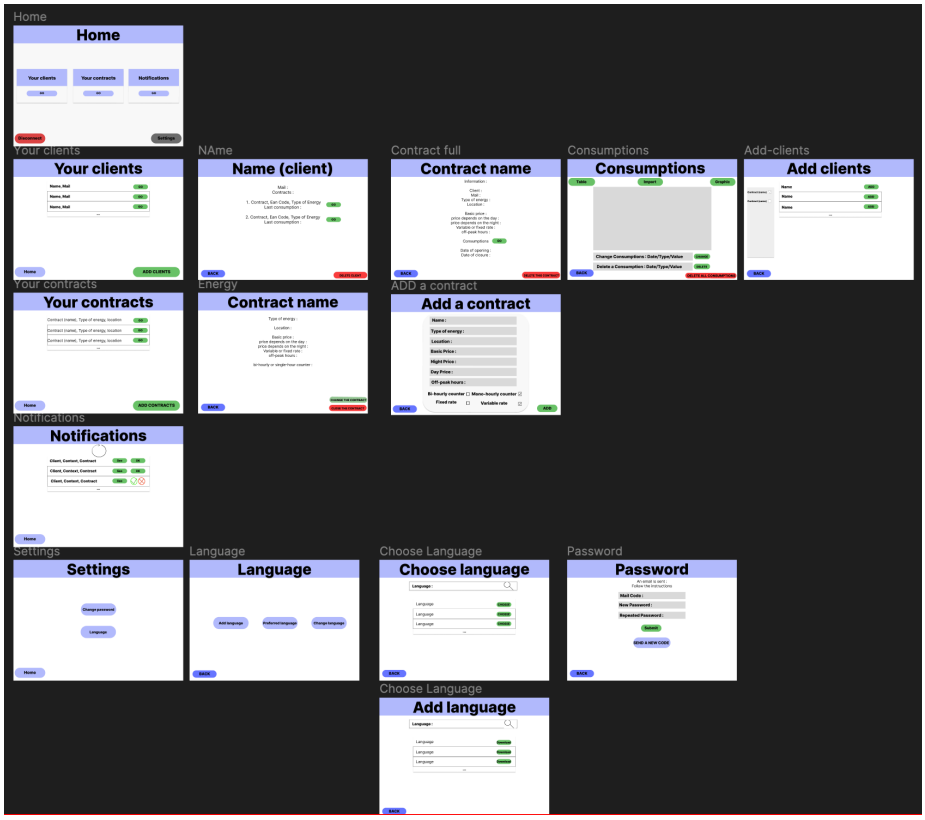
1. Le nom du contrat
2. Le type d'énergie
3. La localisation (du contrat)
4. Le prix basique
5. Le prix dépendant du jour et la nuit
6. Le taux fixe ou variable
7. Les heures creuses
8. Le type de compteur

Ce dernier aura également accès à la suppression du contrat et au changement du contrat fonctionnant comme **l'ajout de contrats (Bouton "Add contracts")** que nous allons expliquer ci-dessous.

Lorsqu'un fournisseur souhaitera ajouter un contrat il devra entrer :

1. Le nom du contrat
2. Le type d'énergie
3. La localisation (du contrat)
4. Le prix basique
5. Le prix dépendant du jour et la nuit
6. Les heures creuses
7. Le taux fixe ou variable
8. Le type de compteur

Le bouton "Add" lui servira à valider la création du nouveau contrat.


Home
Home
Your clients
Your clients
NAme
Name (client)
Contract full
Contract name
Consumptions
Consumptions
Add-clients
Add clients
Your contracts
Your contracts
Energy
Contract name
ADD a contract
Add a contract
Notifications
Notifications
Settings
Settings
Language
Language
Choose Language
Choose language
Password
Password
Choose Language
Add language

# 7 Introduction-extension

Pour plus de clarté, nous avons décidé de proposer un schéma de la base de données et de l'API pour les fonctionnalités de base.

Ensuite, nous expliquerons comment ces schémas seront étendus respectivement pour chaque extension.

Notez donc que lors de l'implémentation, nous rassemblerons tout en une seule base de données et une seule API.