UMONS

Génie Logiciel
Portefeuille énérgétique

Rapport d'implémentation

Élèves :
Nicolas Gatta
Emilien Vancauwenberghe

Enseignant :
Tom Mens
Pierre Hauweele
Sebastien Bonte
Jeremy Dubrulle

26 avril 2023

Table des matières

1 Introduction

Dans ce rapport, nous allons parler des différentes implémentations faites durant la réalisation de ce projet. Cependant, n'ayant pas eu la phase d'analyse, nous allons uniquement présenter la structure globale du site web et la base de données.

2 Technologies utilisées

Une multitude de technologies ont été utilisées tout au long de ce projet aussi bien pour la partie back-end que front-end. Voici la liste des technologies utilisées

- MySql *(back-end)*
- Spring Boot 3.0.2 *(back-end)*
- Java 17 *(back-end)*
- Angular 14 *(front-end)*
- Primeng 14 *(front-end)*

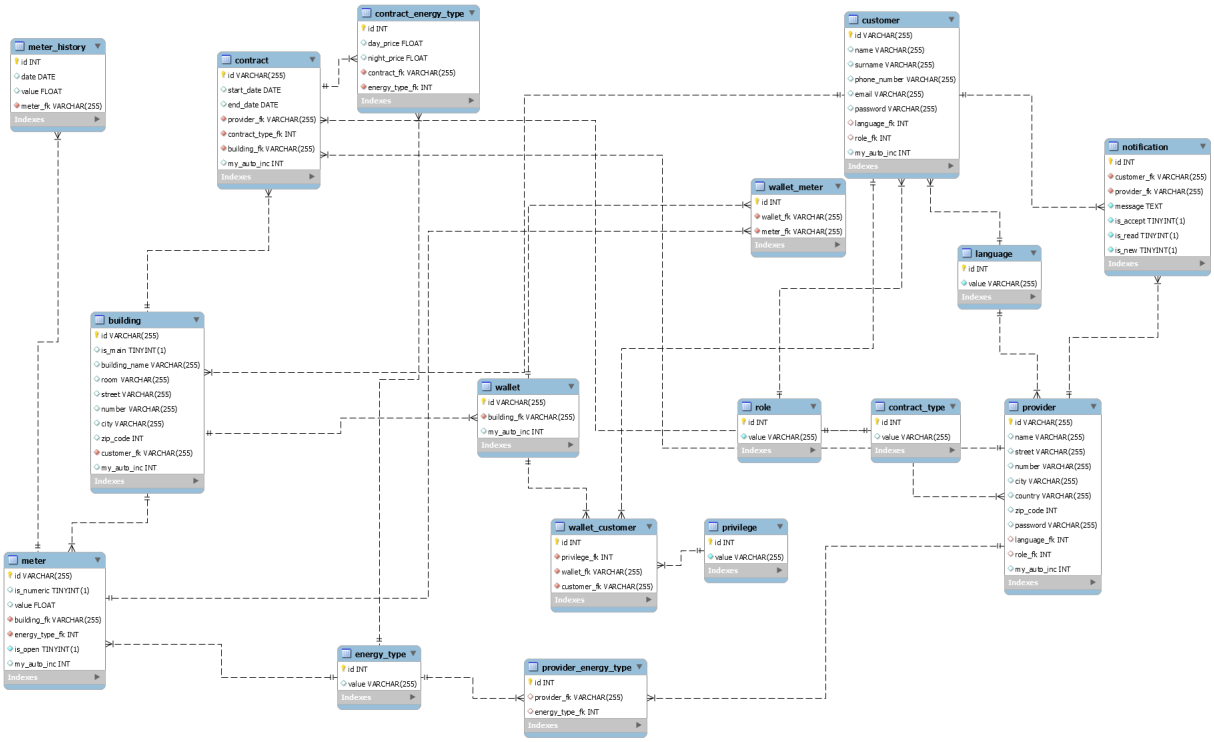
3 Logiciels utilisés

Durant la création de ce projet, il a été question d'utiliser une variété de logiciels allant de l'IDE au gestionnaire de base de données. Voici la liste des logiciels utilisés :

- MySQLWorkbench 8.0
- Intellij
- Visual Studio Code

4 Base de données

Ci-dessous, se trouve le schéma complet de la base de données. Etant donné la taille de cette dernière, nous avons découpé le schéma en plusieurs parties afin de les expliquer plus facilement.



4.1 Partie bâtiment

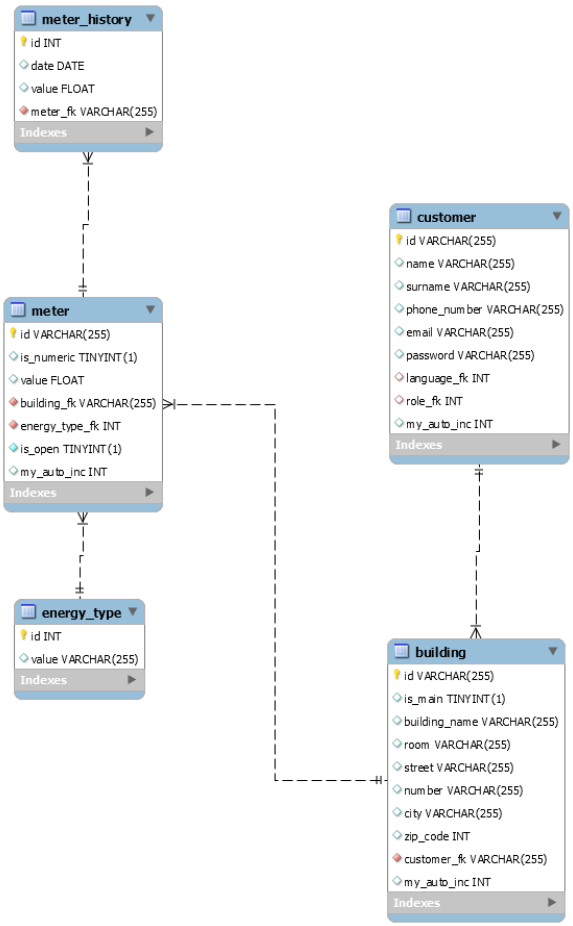
La partie bâtiment est consacrée à la gestion des bâtiments d'un consommateur.
— **meter history :** Représente l'historique des compteurs.
— **meter :** Représente une entité compteur.
— **customer :** Représente une entité client.
— **energy type :** Représente une entité type d'énergie.
— **building :** Représente une entité bâtiment.

4.1.1 Explications

La partie bâtiment possède de multiple relations permettant de lier les données correctement et de pouvoir récupérer de la manière la plus complète possible les informations d'un bâtiment en évitant les redondances. En partant de l'entité **building**, on peut noter que celle-ci est lié à 2 autres entités :

1. Le **customer** permet de savoir à qui appartient le bâtiment.
2. Le **meter** permet de savoir combien de compteur possède un bâtiment ainsi que de savoir quel type de compteur et d'énergies lui a été associé.

En continuant l'exploration de ce schéma, de nouvelles relations apparaissent sur **meter**. Ainsi, le **meter history** va permettre de stocker pour chaque compteur un historique de la consommation passé et **energy type** permettra de savoir quel type d'énergie est géré par le compteur.



4.2 Partie contrat

La partie contrat regroupe les informations d'un contrat qui lie le client au fournisseur d'énergie.

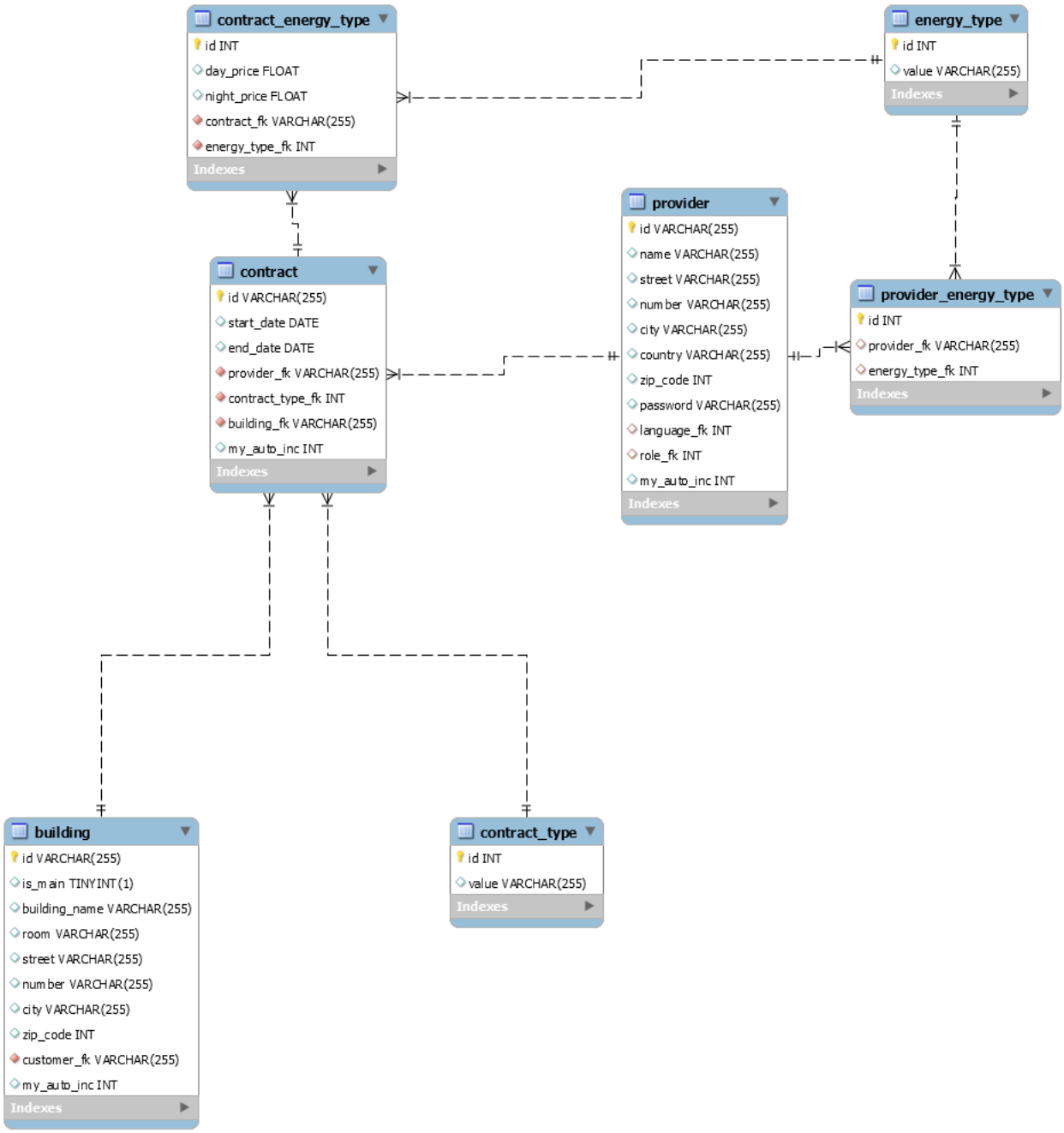
- **contract energy type :** Représente le type d'énergie concerné par le contrat.
- **contract :** Représente une entité contrat.
- **energy type :** Représente une entité type d'énergie.
- **provider :** Représente une entité fournisseur.
- **building :** Représente une entité bâtiment.
- **contract type :** Représente le type du contrat.
- **provider energy type :** Représente quel type d'énergie fournis un fournisseur.

4.2.1 Explications

La partie contrat possède de multiple relations permettant de lier les données correctement ainsi que de pouvoir récupérer de la manière la plus complète possible des informations sur un contrat en évitant les redondances. En partant de l'entité **contract**, on peut noter que celle-ci est lié à 4 autres entités :

1. Le **contract energy type** qui permet de savoir pour chaque contrat, quelle énergie est concernée par celui-ci ainsi que le prix de jour et de nuit pour cette énergie.
2. Le **building** qui va permettre de savoir sur quel bâtiment porte le contrat. Il est alors facile d'identifier de manière unique un client vu qu'un bâtiment ne peut être posséder que par une seule personne
3. Le **contract type** qui permet de savoir si le contrat est fixe ou variable. Cela permettra de savoir si le prix peut être changé ou non.
4. Le **provider** qui va représenter le fournisseur exécutant le contrat.

En continuant l'exploration de ce schéma, de nouvelles relations apparaissent sur **contract energy type** et **provider**. Ainsi, le **energy type** lié au **contract energy type** va permettre de déterminer le type d'energie et cela se fera aussi pour **provider** en utilisant **provider energy type**


contract_energy_type
id INT
day_price FLOAT
night_price FLOAT
contract_fk VARCHAR(255)
energy_type_fk INT
Indexes
energy_type
id INT
value VARCHAR(255)
Indexes
provider
id VARCHAR(255)
name VARCHAR(255)
street VARCHAR(255)
number VARCHAR(255)
city VARCHAR(255)
country VARCHAR(255)
zip_code INT
password VARCHAR(255)
language_fk INT
role_fk INT
my_auto_inc INT
Indexes
provider_energy_type
id INT
provider_fk VARCHAR(255)
energy_type_fk INT
Indexes
contract
id VARCHAR(255)
start_date DATE
end_date DATE
provider_fk VARCHAR(255)
contract_type_fk INT
building_fk VARCHAR(255)
my_auto_inc INT
Indexes
building
id VARCHAR(255)
is_main TINYINT(1)
building_name VARCHAR(255)
room VARCHAR(255)
street VARCHAR(255)
number VARCHAR(255)
city VARCHAR(255)
zip_code INT
customer_fk VARCHAR(255)
my_auto_inc INT
Indexes
contract_type
id INT
value VARCHAR(255)
Indexes

4.3 Partie notification

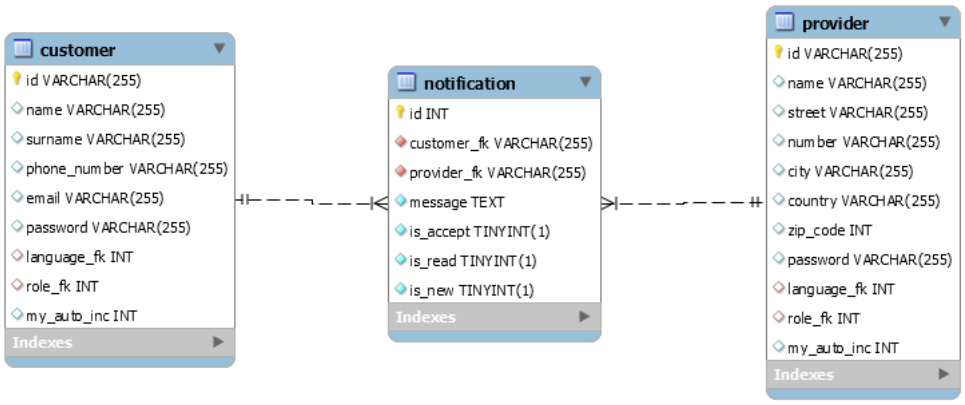
La partie notification est consacrée aux notifications transitant entre le client et son fournisseur.

- **customer :** Représente une entité client.
- **notification** Représente une entité notification
- **provider :** Représente une entité fournisseur.

4.3.1 Explications

La partie notification possède de multiple relations permettant de lier les données correctement et de pouvoir récupérer de manière la plus complète possible des informations sur une notification en évitant les redondances. En partant de l'entité **notification**, on peut noter que celle-ci est liée à uniquement 2 autres entités :

1. Le **customer** qui va représenter un client ayant émis ou reçu une notification.
2. Le **provider** qui va représenter le fournisseur ayant émis ou reçu une notification.



4.4 Partie utilisateur

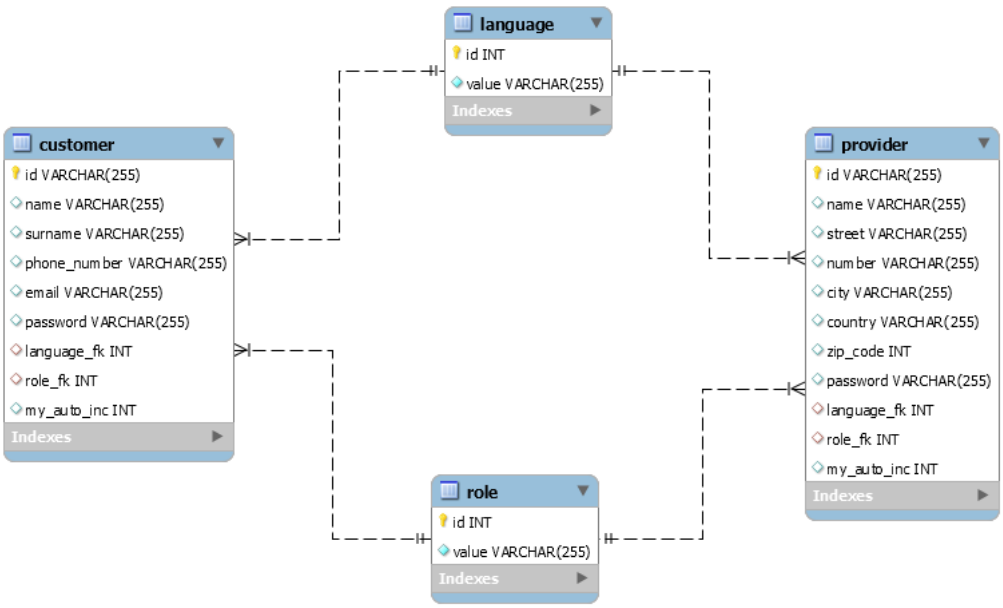
La partie utilisateur est consacrée à la gestion des informations de l'utilisateur, notamment les informations d'authentification et d'internationalisation.

— **customer :** Représente une entité client.
— **language :** Représente une entité langue.
— **provider :** Représente une entité fournisseur.
— **role :** Représente une entité rôle.

4.4.1

Explications La partie utilisateur possède de multiple relations permettant de lier les données correctement et pouvoir récupérer de manière la plus complète possible des informations sur une notification en évitant les redondances. En partant de l'entité **customer** et **provider**, on peut noter que celle-ci est lié à uniquement 2 autres entités :

1. Le **language** qui va représenter la langue utiliser par un client ou un fournisseur.
2. Le **role** qui va représenter le role dans l'application du fournisseur (admin) et du client (user).



5 Design pattern

La majorité des design pattern qui vont être abordés ci-dessus sont des design pattern automatiquement implémenter avec spring boot il sera donc difficile de justifier la motivation d'utilisation de ceux-ci. Les designs patterns concernés vont être notés avec une notation *sb* à coté de leur nom.

5.1 DTO

5.1.1 Explication

Data Transfer Objects (DTO) est un modèle de conception pour une architecture logicielle orientée objet. Son but est de simplifier le transfert de données entre les sous-systèmes d'application.

5.1.2 Motivation

Il est assez commun dans le monde professionnel d'utiliser le DTO pour scinder la partie front-end et back-end d'une application. Cela permet de totalement camoufler la structure de la base de données ainsi que de renvoyer des données modifiés permettant de ne pas afficher toutes les informations.

5.2 Singletion *(sb)*

5.2.1 Explication

Le modèle singleton relève de la catégorie des modèles de création. Son but est d'empêcher une classe d'instancier plusieurs objets. Pour ce faire, nous créons l'objet requis dans sa propre classe, puis nous le récupérons en tant qu'instance statique.

5.2.2 Motivation

L'utilisation du singleton est presque obligatoire dans des projets nécessitant une connexion à une base de données pour plusieurs raisons. La première étant d'éviter une consommation trop importante des ressources d'une base de données. Il faut bien imaginer que de manière générale il va y avoir des millions de connexion dans une application ce qui consomme déjà beaucoup de ressource alors si jamais, les connexions étaient dupliqués voir quadruplés, le serveur n'encaisserai pas la charge. La deuxième raison est la facilité d'accès de la ressource dans le code, vu qu'elle n'est initialisé qu'une fois, on peut y accéder depuis n'importe où dans le code.

5.3 Injection de dépendance *(sb)*

5.3.1 Explication

L'injection de dépendance est un modèle architectural utilisé pour résoudre le problème des dépendances entre objets tout en permettant le découplage des objets liés par indirection.

5.4 Factory *(sb)*

5.4.1 Explication

Le design pattern factory est un pattern de création qui définit une interface pour créer des objets dans une classe mère, mais délègue le choix des types d'objets à créer à des sous-classes.

5.5 Template *(sb)*

5.5.1 Explication

Le design pattern template est basé sur une classe abstraite exposant des méthodes ou des modèles définis pour l'exécution de ses méthodes. Ses sous-classes peuvent surcharger l'implémentation de la méthode en fonction des besoins, mais l'invocation doit se faire de la même manière que celle définie par la classe abstraite.

5.5.2 Motivation

Le pattern Template fournit une implémentation d'un squelette d'un algorithme dans une super classe et laisse les sous-classes surcharger un partie précise de l'algorithme. Spring Boot fournit plusieurs classes de modèles, telles que JdbcTemplate et RestTemplate, qui peuvent être étendues pour créer des modèles personnalisés.

5.6 Strategy *(sb)*

5.6.1 Explication

Dans le pattern de stratégie, le comportement d'une classe ou son algorithme peut être modifié au moment de l'exécution. Ce type de design pattern fait partie des patterns de comportement.

5.6.2 Motivation

Le pattern Strategy peut fournir des algorithmes enfichables. Spring Boot propose plusieurs stratégies, telles que la mise en cache et la sécurité, qui peuvent être personnalisés pour répondre aux besoins de l'application.

5.7 Observer *(sb)*

5.7.1 Explication

Le pattern d'observateur est utilisé lorsqu'il existe une relation de type "un pour plusieurs" entre des objets.

5.7.2 Motivation

Le pattern Observer peut utiliser dans la gestion des événements. Spring Boot fournit un modèle d'événement qui permet aux composants de publier des événements et de s'y abonner. De même dans pour Angular qui utilise des Observables pour récupérer des objets lors d'un appel à l'API spring boot

5.8 Decorator *(sb)*

5.9 Explication

Le pattern décorateur permet à l'utilisateur d'ajouter de nouvelles fonctionnalités à un objet existant sans en modifier la structure. Ce type de modèle de conception fait partie des patterns structurel, car il agit comme une enveloppe pour une classe existante. Ce pattern crée une classe décoratrice qui enveloppe la classe d'origine et fournit des fonctionnalités supplémentaires tout en conservant la signature des méthodes de la classe intacte.

5.9.1 Motivation

De manière général, les décorateurs sont utiliés en sping boot pour signaler une action spécifique à effectuer. On retrouve ceux-ci pour signaler au framework qu'il doit utiliser une injection de dépendance ou encore qu'une requête vers une base de données est une transaction par exemple.

6 Commande Gradle

6.1 Exécuter les tests

Pour exécuter les tests avec gradle, il suffit simplement d'utiliser la commande suivante : **gradle test**

6.2 Exécuter le programme

Pour exécuter le programme avec gradle, il suffit simplement d'utiliser la commande suivante : **gradle bootRun**

Pour exécuter le code angular, il faut se rendre sur le dossier Site Web et faire les deux commandes suivantes : `npm install --force` pour installer tous les modules suivis de **ng serve**

(La connexion à la base de donnée se fait sur un serveur distant donc aucun besoin d'exécuter le code de la base de données)

6.3 Compiler l'application

Pour exécuter le programme avec gradle, il suffit simplement d'utiliser la commande suivante : **gradle build**.

Cependant, il nous a été impossible de combiner la compilation d'angular et spring boot par manque de temps mais il est possible de le faire.

7 Login et Mot de passe

Username	Password
Aspiravi Energy	Provider
DATS 24	Provider
Ebem	Provider
Eneco	Provider
Energie.be	Provider
Engie	Provider
Luminus	Provider
Mega	Provider
Octa	Provider
TotalEnergies	Provider
AIEC	Provider
AIEM	Provider
CIESAC	Provider
CILE	Provider
IDEN	Provider
IEG	Provider
INASEP	Provider
SWDE	Provider
Falafel.Thierry@JeSuisNul.com	Customer
Monsieur.Inspecteru@JeParleTrop.com	Customer
Gatta.Nicolas@TuMeTrouverasPas.com	Customer
Vancauwenberghe.Emelien@JeSuisCaché.com	Customer

8 URL de la vidéo

https ://youtu.be/uA8XNadRcqo