UMONS
Université de Mons

Faculté des Sciences

Rapport d'implémentation

Une application client-serveur en Java
"Gestion de Consommation d'énergie"

S-INFO-015 Projet de Génie Logiciel

Réalisé par Roméo IBRAIMOVSKI
& Nicolas SOURNAC
& Clément SAMAIN

Faculté
des Sciences

Supervisé par Tom MENS

2e Bachelier en Sciences Informatiques
Année 2020-2021

ACADÉMIE
UNIVERSITAIRE
WALLONIE-
BRUXELLES

Résumé

Ce rapport d'implémentation est rendu dans le cadre de l'AA-S-INFO-015 "Projet de Génie Logiciel", dispensé par le Prof. Tom Mens en année académique 2020-2021. Ce rapport a pour but d'expliquer et de justifier nos différents choix d'implémentation.

Table des matières

1. Fonctionnalité de base

Vidéo de présentation : `https://youtu.be/EesdVnXq87A`

Notre projet se décline en deux applications distinctes. Pour rappel, l'application de gestion de portefeuille énergétique est appelée application 1 et l'application de facilitation de liens avec le fournisseur d'énergie est appelée application 2. L'application 1 est uniquement accessible pour les consommateurs alors que l'application 2 est accessible à la fois pour les consommateurs et les fournisseurs d'énergie.

Lors du démarrage de l'application 1, le consommateur arrivera directement sur une fenêtre de connexion où il sera invité à choisir entre s'enregistrer ou se connecter. Il pourra également cliquer sur un bouton "mot de passe oublié" et entrer son adresse mail afin de recevoir un mot de passe de secours.

Après une connexion réussie, le consommateur arrive sur une fenêtre où il pourra consulter une liste contenant l'ensemble de ses portefeuilles. Il pourra y créer un nouveau portefeuille et ouvrir une fenêtre via laquelle il pourra gérer l'ensemble des invitations qui le concerne. Il pourra les accepter ou les refuser.

En cliquant sur l'un de ses portefeuilles, le consommateur arrivera sur une fenêtre ou il pourra accéder aux données de celui-ci. Il pourra y observer la liste des compteurs du portefeuille ainsi qu'un tableau et un graphique qui serviront à accueillir les données de consommation du compteur sur lequel il aura cliqué. En fonction de son niveau d'accès au portefeuille, l'utilisateur n'aura pas accès à l'entièreté des fonctionnalités de la fenêtre. Un utilisateur disposant d'un accès en lecture disposera uniquement du menu information où il pourra ouvrir des fenêtres concernant les membres du portefeuille et concernant les compteurs du portefeuille. Dans notre système, un compteur est soit considéré comme ouvert ou fermé. Lorsqu'un compteur est ouvert, il est obligatoirement lié à un contrat qui lie lui-même un fournisseur et un consommateur. Lors de l'ouverture, le compteur est soit lié à un contrat déjà existant, soit lié à un contrat nouvellement créé par le système. Cette action est commanditée par le fournisseur d'énergie dans l'application 2. Un utilisateur disposant d'un accès en écriture disposera en plus de la possibilité d'ajouter des données aux compteurs du portefeuille. Pour ce faire, il devra accéder à l'application 2, soit en la démarrant sur le côté, soit en confirmant son souhait d'ajouter des données depuis l'application 1 ce qui aura pour effet de lancer automatiquement l'application 2. Enfin, un utilisateur disposant d'un accès de manager au portefeuille disposera en plus d'un accès à une fenêtre de gestion du portefeuille à partir de laquelle il pourra ajouter, supprimer ou modifier un compteur ou un utilisateur. L'ajout de compteur ne peut se faire uniquement dans certains cas. Si le compteur n'existe pas encore dans notre système, celui-ci sera créé et sera ajouté au portefeuille. Cependant, ce nouveau compteur sera inutile tant qu'aucun fournisseur d'énergie ne l'aura ouvert. Si le compteur que l'on désire ajouter existe déjà, celui-ci sera ajouté uniquement si le compteur est ouvert et si le contrat qui lui est lié

appartient bien à l'utilisateur désireux de l'ajouter à son portefeuille. La modification d'un compteur consiste uniquement en la modification de son nom. Les autres paramètres d'un compteur étant définitifs, la suppression d'un compteur consiste uniquement en la suppression d'un compteur au niveau du portefeuille et non en la suppression d'un compteur au niveau du système global. L'ajout d'un utilisateur ne se fait pas de manière instantanée mais par le biais d'invitation que l'utilisateur cible recevra et sera libre d'accepter ou de supprimer. La modification d'un utilisateur consiste en la modification de son niveau d'accès au portefeuille. Enfin, n'importe quel utilisateur pourra visualiser la consommation totale du portefeuille en fonction d'un certain type d'énergie via le menu données globales. Le consommateur pourra également filtrer les données à afficher dans le tableau et dans le graphique via une fenêtre de choix de période et d'échelle. Il faut savoir que le consommateur n'aura accès uniquement aux données disponibles à partir de la date d'ouverture de son compteur.

Lors du démarrage de l'application 2, l'utilisateur est invité à se connecter en entrant son adresse mail et son mot de passe. Ensuite, l'application détermine automatiquement si il s'agit d'un consommateur ou d'un fournisseur d'énergie. Dans le premier cas, si il s'agit d'un consommateur, celui-ci aura un accès limité à l'application 2. Après une connexion réussie, celui-ci pourra observer une liste de compteurs actuellement ouverts et qu'il détient dans un de ses portefeuilles où il a un accès en écriture ou supérieur, ainsi qu'une liste contenant un historique de ses compteurs dont il disposait dans un contrat dans le passé. Le consommateur peut choisir d'effectuer deux actions différentes sur ses compteurs actuellement ouverts, il peut saisir manuellement des données via une fenêtre dédiée accessible via le menu en sélectionnant au préalable un compteur et exporter des données vers un fichier via une autre fenêtre également accessible via le menu. Les données que l'utilisateur pourra exporter seront uniquement les données disponibles depuis la date d'ouverture du compteur.

Dans le deuxième cas, si l'utilisateur est un fournisseur d'énergie, celui-ci arrivera sur une fenêtre où il pourra observer la liste des compteurs actuellement reliés à un de ses contrats. Il pourra également observer une liste de l'ensemble des compteurs actuellement fermés du système. Depuis cette fenêtre, le fournisseur pourra ouvrir un compteur actuellement fermé en spécifiant un nom d'utilisateur valide ainsi que d'un numéro de contrat déjà existant dans le système. Si il ne spécifie pas de numéro de contrat, un nouveau contrat sera automatiquement créé et associé au compteur. Depuis cette fenêtre, le fournisseur pourra également choisir de fermer un compteur. Lorsqu'il effectue cette action, le système vérifie que le compteur qui vient d'être fermé n'était pas le dernier compteur relié à un contrat auquel cas ce contrat sera supprimé. Pour ré attribuer un compteur ouvert à un autre client, le fournisseur doit donc d'abord effectuer l'action de fermer le compteur et ensuite l'ouvrir pour le nouveau client. Depuis cette fenêtre, le fournisseur peut également, saisir/modifier des données manuellement, importer des données pour compteur depuis un fichier JSON ou csv, exporter des données vers un fichier JSON ou csv et enfin supprimer les données d'un compteur. Si le fournisseur supprime les données d'un compteur, le client associé au

contrat de ce compteur, ce compteur verra apparaitre une notification lui indiquant cette suppression lors de sa prochaine connexion.

Note : lors de l'importation d'un fichier conséquent, l'action peut prendre plusieurs secondes pendant lesquelles le programme est inutilisable. Dans ce cas là, l'utilisateur doit attendre que la fenêtre d'importation disparaisse avant d'entamer une autre action.

La fonctionnalité de support de plusieurs langues par le logiciel n'a pas été implémentée par manque de temps.

Voici les identifiants des comptes déjà existants dans la base de données de la fonctionnalité de base :

1. mail : test@user.com / username : nsournac / mot de passe : umonsapp123 (CONSOMMATEUR)
2. mail : test2@user.com / username : ClemSam / mot de passe : umonsapp123 (CONSOMMATEUR)
3. mail : example@electrabel.com / username : electrabel / mot de passe : umonsapp123 (FOURNISSEUR)

1.1 Choix d'implémentation

Lors de l'implémentation du projet, nous avons décidé de le découper en trois packages principaux. Le package GUI regroupe l'ensemble des classes utiles au fonctionnement de l'interface graphique Java FX. Le package TOOL est composé d'un ensemble de classes utilitaires ainsi que de deux sous-packages sql et restapi. Le premier regroupe l'ensemble des classes utilisées lors de la génération du code sql ainsi que lors de la communication et l'exécution du sql sur notre base de données. Le package restapi regroupe l'ensemble des classes de notre serveur http, on y retrouve notamment l'ensemble des servlets qui composent notre API rest. Le package USER regroupe l'ensemble des classes modélisant la thématique du projet.

Les package TOOL et USER sont communs aux deux applications. Le package GUI est subdivisé en deux sous-package app1 et app2, ces deux packages sont constitués de l'ensemble des classes utiles au fonctionnement de l'interface graphique de l'application qui en dépend.

L'importation et exportation des fichiers peut se faire au format CSV (comma-separated values) ainsi qu'au format JSON (JavaScript Object Notation). Un consommateur ou un fournisseur a la possibilité d'exporter les données associées à un (ou plusieurs) compteurs en fonction d'une période spécifique. Le consommateur ne peut qu'exporter les données d'une période pendant laquelle le compteur lui appartient. Le fournisseur a la possibilité d'importer les données associées à un (ou plusieurs) compteurs à partir de fichier. Pour l'importation et exportation de fichier au format CSV, nous avons utilisé les librairies fournies avec Java 11 pour la lecture/écriture de fichier, java.io et java.nio. Pour l'importation et exportation de fichier au format JSON, nous avons utilisé la librairie Jackson, bien connu pour être la librairie JSON pour Java. (Source : `https://github.com/FasterXML/jackson`)

En ce qui concerne l'hébergement de notre base de données ainsi que de notre serveur http, nous avons choisi l'hébergeur gratuit alwaysdata `https://www.alwaysdata.com/fr/`. Cet hébergeur nous permet de facilement accéder à nos données via une base de données MySQL et de facilement déployer notre serveur (REST API) qui s'occupe de la récupération et de la gestion de requêtes http. Pour la réalisation de ce serveur http, nous avons utilisé Jetty 9.4 `https://www.eclipse.org/jetty/`. C'est via celui-ci que nous avons donc créé notre API rest. Celle-ci permet d'accéder et de modifier les données présentes dans notre base de données via des requêtes http. Pour ce faire, nous avons créé au sein du serveur jetty une application web appelée RestAPI et nous y avons déposé l'ensemble des .class utile au fonctionnement de nos Servlets. On retrouve également au sein de jetty, un fichier web.xml dans lequel l'ensemble de nos Servlets sont définis.

1.1.1 Package gui

Le package GUI regroupe l'ensemble des classes qui composent l'interface graphique. Pour la réaliser, nous avons décidé d'utiliser Java FX 11.0.2. Comme expliqué au point précédent, ce package est lui-même divisé en deux sous-packages contenant les classes propres aux interfaces graphiques de l'application 1 ou de l'application 2. Les deux sous-packages sont tous les deux munis d'une classe Gui, d'une classe GuiHandler ainsi que de différents Controllers. En règle générale, il existe une classe Controller pour une fenêtre présente dans l'interface graphique sauf pour certaines exceptions lorsque plusieurs fenêtres différentes sont similaires ou lorsqu'une fenêtre est commune aux deux applications. Les fenêtres ont été créées grâce au logiciel SceneBuilder, celui-ci permet de créer pas à pas des fenêtres java FX et d'ensuite les récupérer dans un format fxml. Afin de donner vie aux différents composants de chacune des fenêtres, celles-ci sont donc toutes accompagnées par une classe controller qui va permettre à un utilisateur d'interagir avec. En général, chaque Controller dispose d'une méthode initialize() dans laquelle on va pouvoir ordonner l'exécution d'actions avant le chargement de la fenêtre. On y trouvera aussi les méthodes liées aux boutons et menus, c'est via ces controllers que toutes les actions seront initiées. La classe Gui est une classe qui va regrouper l'ensemble des informations nécessaires au bon fonctionnement de la session en cours. La classe GuiHandler est en fait une classe qui regroupe un ensemble de méthodes utilitaires nécessaires au bon fonctionnement du logiciel. Cette classe va également servir de pont pour les Controllers entre-eux, et de pont pour les Controllers et la classe Gui. Les deux applications sont donc indépendamment équipées de cette structure de classe. Ainsi, dans le package de l'application 1, nous retrouvons :

1. **AddDataController :**

Le Controller qui s'occupe de la fenêtre où l'utilisateur est invité à confirmer son souhait à ajouter des données à l'un de ses compteurs et donc de basculer vers l'application 2.

2. **AddMeterController :**

Le Controller de la fenêtre d'ajout de compteur à un portefeuille. C'est via celui-ci que l'application va récupérer l'ensemble des informations saisies par l'utilisateur et créer ou non un nouveau compteur au sein de notre système et dans le portefeuille de cet utilisateur.

3. **AddUserController :**

Le Controller de la fenêtre d'ajout d'utilisateur à un portefeuille. C'est via celui-ci que l'application va récupérer l'ensemble des informations saisies par l'utilisateur et envoyer ou non une invitation à un utilisateur

4. **CreatePController :**

Le Controller de la fenêtre de création de portefeuille. C'est via celui-ci que l'application va récupérer le nom du portefeuille à créer et appeler le méthode nécessaire à la création de ce portefeuille.

5. DashBoardMController :

Le Controller de la fenêtre principale de l'application. Celle ou l'utilisateur va pouvoir accéder aux informations d'un portefeuille. C'est ce controller qui va permettre donc à l'utilisateur d'interagir avec cette fenêtre. Celui-ci pourra y observer les données de consommation relatives aux compteurs du portefeuille par exemple et, en fonction de son niveau d'accès au portefeuille, effectuer diverses actions comme accéder à l'application 2 afin d'y ajouter des données ou même ajouter, modifier ou supprimer un compteur ou un utilisateur.

6. DashBoardPController :

Le Controller de la fenêtre où l'utilisateur pourra visualiser l'ensemble des portefeuilles dans lesquels il dispose d'un accès. Le controller permettra également à l'utilisateur d'accéder à la fenêtre de création de portefeuille ainsi qu'à la fenêtre de gestion de ses notifications.

7. ForgotPwdController :

Le Controller de la fenêtre d'oubli de mot de passe via laquelle l'utilisateur est invité à indiquer une adresse email afin de recevoir un mot de passe de secours.

8. GestionPController :

Le Controller de la fenêtre de gestion du portefeuille, accessible via le menu de la fenêtre de visualisation de données. Ce controller va permettre à l'utilisateur d'effectuer l'ensemble des actions nécessaires à la gestion de son portefeuille. C'est-à-dire les actions d'ajout, de suppression, de modification de compteur et d'utilisateur.

9. Gui :

La classe Gui regroupe des informations nécessaires au bon fonctionnement du logiciel tels que des références vers l'utilisateur courant, le portefeuille courant, le compteur courant. C'est depuis cette classe que l'on démarre l'application 1.

10. GuiHandler :

La classe GuiHandler contient de nombreuses méthodes utilitaires comme par exemple les méthode loadScene() et createScene() qui vont permettre de facilement charger ou créer une nouvelle fenêtre à l'écran. On retrouve également dans cette classe des méthodes utiles aux fonctions de base du logiciel tels que la création de portefeuille, l'ajout de compteur, mais aussi des méthodes qui vont être utiles au bon affichage des informations dans l'interface graphique.

11. InfoCController :

Le Controller de la fenêtre d'information sur les compteurs d'un portefeuille. Ce controller va charger les données dans le tableau qui compose cette fenêtre.

12. InfoPController :

Le Controller de la fenêtre d'information sur les membres d'un portefeuille. Ce Controller va charger les données dans la liste qui compose cette fenêtre.

13. **LoginController :**

Le Controller de la fenêtre de connexion de l'application. C'est via celui-ci que l'application va récupérer les informations saisies par l'utilisateur et effectuer une inscription ou une simple connexion.

14. **ModifMeterController :**

Le Controller de la fenêtre de modification de compteur. C'est via celui-ci que l'application va récupérer les informations saisies par l'utilisateur et effectuer les modifications nécessaires sur le compteur sélectionné.

15. **ModifUserController :**

Le Controller de la fenêtre de modification d'utilisateur pour un portefeuille. C'est via celui-ci que l'application va récupérer les informations saisies par l'utilisateur et effectuer les modifications nécessaires sur l'utilisateur sélectionné.

16. **NotifController :**

Le Controller de la fenêtre de gestion d'invitations. C'est via celui-ci que l'application va permettre à l'utilisateur de visualiser et ensuite d'accepter ou de supprimer des invitations à des portefeuilles.

17. **ScalingController**

Le Controller de la fenêtre de choix d'une échelle et d'une période pour la visualisation de données. Ce controller va exécuter lui-même à partir de l'ensemble des données disponibles pour un compteur, les méthodes qui vont permettre de filtrer ces données afin d'afficher uniquement celles choisies par l'utilisateur.

Dans le package de l'application 2, nous retrouvons :

1. **CloseMeterController :**

Le Controller de la fenêtre de fermeture de compteur. C'est via celui-ci que l'application va récupérer le compteur à fermer que l'utilisateur (en l'occurrence un fournisseur d'énergie) aura choisi et va exécuter les différentes méthodes utiles à cette fermeture.

2. **DashBoardCController :**

Le Controller de la fenêtre principale de l'application 2 côté consommateur. C'est via ce controller que la fenêtre va récupérer les données à afficher dans les tableaux et permettre à l'utilisateur d'y effectuer un ensemble d'actions tel que l'accès à d'autres fenêtres comme la saisie de données ou l'exportation de données vers un fichier.

3. **DashBoardFController :**

Le Controller de la fenêtre principale de l'application 2 côté fournisseur d'énergie. C'est via ce controller que la fenêtre va récupérer les données à afficher dans les tableaux et permettre au fournisseur d'y effectuer l'ensemble de ses actions.

4. **DelDataController :**

Le Controller de la fenêtre de suppression de données. C'est ce Controller qui va récupérer les informations saisies par le fournisseur et exécuter les méthodes nécessaires à la suppression de données d'un certain compteur en fonction d'une période.

5. **ExportDataController :**

Le Controller de la fenêtre d'exportation de données vers un fichier JSON ou CSV.

6. **Gui :**

La classe Gui regroupe des informations nécessaires au bon fonctionnement du logiciel tels que des références vers l'utilisateur courant, le portefeuille courant, le compteur courant, le type de l'utilisateur courant. C'est depuis cette classe que l'on démarre l'application 2.

7. **GuiHandler :**

La classe GuiHandler contient de nombreuses méthodes utilitaires comme par exemple les méthodes loadScene() et createScene() qui vont permettre de facilement charger ou créer une nouvelle fenêtre à l'écran. On retrouve également dans cette classe des variables de classes utiles au fonction de base du logiciel.

8. **ImportDataController :**

Le Controller de la fenêtre d'importation de données depuis un fichier JSON ou CSV.

9. **LoginController :**

Le Controller de la fenêtre de connexion de l'application 2. C'est via ce Controller que l'application va pouvoir vérifier et identifier l'utilisateur courant.

10. **OpenMeterController :**

Le Controller de la fenêtre d'ouverture d'un compteur. C'est via ce Controller que l'application va initier l'ensemble des actions à effectuer afin d'ouvrir un compteur.

11. **WriteDataController :**

Le Controller de la fenêtre d'ajout ou de modification de données manuellement. Cette même fenêtre est disponible à la fois côté consommateur et côté fournisseur.

De plus, le package Gui contient lui-même un certain nombre de classes qui sont indépendantes des deux applications. On y retrouve :

1. **ChangePwdController :**

Le Controller de la fenêtre de changement de mot de passe. C'est via ce controller que l'application va mettre à jour le mot de passe de l'utilisateur dans la base de données.

2. **CustomListCell :**

CustomListCell est une classe qui hérite de la classe ListCell de Java FX. Elle permet de personnaliser l'affichage de nombreux éléments tels que les portefeuilles, les invitations, les consommateurs dans les différentes ListView présentes dans les deux applications.

3. DataSupplyPointTabCell :

DataSupplyPointTabCell est une classe qui hérite de la classe TableCell de Java FX. Elle permet de personnaliser l'affichage des données de consommation dans les différents tableaux des deux applications.

4. GestionCustomListCell :

GestionCustomListCell est une classe qui hérite de la classe ListCell de Java FX. Elle permet de personnaliser l'affichage des utilisateurs et des compteurs dans la fenêtre de gestion du portefeuille de l'application 1.

5. Language :

Language est une énumération reprenant les différents langages que nous voulions que notre projet supporte.

6. MeterListCell :

MeterListCell est une classe qui hérite de la classe ListCell de Java FX. Elle permet de personnaliser l'affichage des compteurs dans les différentes ListView des applications

7. SupplyPointLine :

SupplyPointLine est une classe représentant les données d'un compteur à afficher sur une ligne d'un tableau TableView de l'application.

8. SupplyPointTabCell :

SupplyPointTabCell est une classe qui hérite de la classe TableCell. Elle permet de personnaliser l'affichage des compteurs dans les différentes TableView de l'application 1 et 2.

9. SupplyPointTabCellBox :

SupplyPointTabCellBox est une classe qui hérite de la classe TableCell. Elle permet de personnaliser l'affichage des compteurs dans la fenêtre d'exportation de données de l'application 2 et de surtout rajouter un élément checkbox afin de permettre à l'utilisateur de sélectionner un ou plusieurs compteurs à la fois.

1.1.2 Package tool

Lors de l'implémentation du projet, nous nous sommes rendus compte que certaines tables présentent dans notre modèle ERD initial n'avait pas lieu d'être. C'est pourquoi les deux tables ProvisionPoint et CounterHistory n'ont pas été implémentées lors du développement. En effet, la table ProvisionPoint a été fusionnée avec la table Counter que l'on nomme maintenant SupplyPoints. Le code EAN présent dans l'ancienne table ProvisionPoint sert de clé primaire à notre nouvelle table SupplyPoint. SupplyPoint est une table utilisée pour le stockage de tous les compteurs de notre système. En ce qui concerne la table CounterHistory, celle-ci a été supprimée. SupplyPointHistory contient maintenant directement une colonne spécifiant l'ID de l'utilisateur concerné par l'historique. Voici les tables présentent dans notre base de données :

USERS contient quarte colonnes : USERNAME (nom de l'utilisateur), PWD (mot de passe crypter), TYPE (CONSOMMATEUR/FOURNISSEUR), MAIL(l'adresse mail) PRIMARY KEY (MAIL) UNIQUE (USERNAME)

CONTRACTS contient cinq colonnes : ID (L'identifiant du contrat), ENERGY_SUPPLIER_ID (le nom d'utilisateur de fournisseur), CONSUMER_ID (le nom d'utilisateur du consommateur), TYPE (MONO/BI), START (la date de création du contrat) PRIMARY KEY (ID)

DATA_SUPPLY_POINT contient trois colonnes : EAN18 (le code EAN du compteur), DATE (la date à laquelle la valeur est associée pour le compteur), VALUE (la valeur) PRIMARY KEY (EAN18, DATE)

DELNOTIF contient deux colonnes et représente les notifications de suppression de données : USERNAME (le nom de l'utilisateur à qui appartient la notifications) et EAN (le code ean du compteur où l'on a supprimé des données).

ENERGY_CONSUMER_PORTFOLIO contient trois colonnes : USERNAME (le nom de l'utilisateur), PORTFOLIO_ID (l'identifiant du portefeuille), ACCESS (l'accès de l'utilisateur) PRIMARY KEY (USERNAME, PORTFOLIO_ID)

NOTIFICATION contient trois colonnes : USERNAME(le nom de l'utilisateur qui a reçu l'invitation, ACCESS (le type d'accès qui lui est octroyé), PORTFOLIO_ID (l'identifiant du portefeuille) PRIMARY KEY (USERNAME, PORTFOLIO_ID)

PORTFOLIO contient deux colonnes : ID(l'identifiant du portefeuille), NAME(Le nom du portefeuille) PRIMARY KEY (ID)

PORTFOLIO_SUPPLY_POINT contient deux colonnes : PORTFOLIO_ID(l'identifiant du portefeuille), SUPPLY_POINT_ID (le code EAN du compteur) PRIMARY KEY

(PORTFOLIO_ID, SUPPLY_POINT_ID)

SupplyPoint contient sept colonnes : EAN18 (le code EAN du compteur), ENERGY-TYPE (le type du compteur), NAME (le nom du compteur), STATE (l'état du compteur), CONTRACTID (l'identifiant du contract), DAYSTART (la date d'ouverture du compteur), SUPPLIERID (l'identifiant du fournisseur) PRIMARY KEY (EAN18)

SUPPLY_POINT_HISTORY EAN18 (le code EAN du compteur), USERNAME (le nom de l'utilisateur qui détenait le compteur), OPEN(la date d'ouverture du compteur), CLOSE(la date de fermeture du compteur) PRIMARY KEY (EAN18, USERNAME, OPEN, CLOSE)

package sql : nous nous connectons à la base de données à l'aide de la librairie java.sql (la connexion se fait dans la classe DataBase) nous avons également créé une classe abstraite DB qui s'occupe de générer les requêtes SQL de toutes les classes enfants (pour nous faciliter la tâche vu que les requêtes SQL se ressemblent beaucoup syntaxiquement), ce qui nous permet de créer facilement le code java permettant de récupérer, modifier ou supprimer des éléments d'une table de la base de données. nous avons donc 11 sous-classes de DB (une pour chaque table de la base et une pour la table ADRESSE qui n'est pas utilisée dans le projet)

package restapi : nous avons créé une classe abstraite Servlet, qui est une sorte de "moule" pour tous les servlets que l'on veut créer par la suite, qui nous permet de parser automatiquement les valeurs en JSON pour les envoyer comme résultat de la requête HTTP. nous avons créé beaucoup de servlets et certains se ressemblent trop car nous avons remarqué, tardivement, qu'on pouvait rajouter un paramètre "ACTION" qui nous aurait permis de regrouper plusieurs servlets en un.

1.1.3 Package user

le package user contient les classes représentant les différents objets principaux du projet (USER, SupplyPoint, EnergyPortfolio, ...) seule la classe Home et Adresse ne sont pas utilisées car non présente de manière graphique dans la projet.

1.2 Description d'algorithmes

1.2.1 Ouverture d'un compteur

L'ouverture d'un compteur peut se faire de deux façons différentes. Soit le fournisseur d'énergie désire rajouter le compteur à un contrat déjà existant, dans ce cas le fournisseur doit spécifier un numéro de contrat déjà existant dans notre système. Soit le fournisseur désire créer un nouveau contrat, dans ce cas celui-ci ne doit pas spécifier de numéro de contrat. L'application va en générer un automatiquement. D'un point de vue logiciel, cette action s'exécute au travers de la méthode openCounter() située dans la classe EnergySupplier du package user. Cette méthode prend en paramètre un compteur, un client, un type de contrat, un numéro de contrat potentiellement nul ainsi qu'un booléen newID qui vaut true si l'on veut créer un nouveau contrat et false si on veut ajouter le compteur à un contrat existant. Si newID est égal à true, alors on effectue d'abord une première requête http afin de récupérer l'id du prochain compteur et ensuite une seconde afin de créer ce contrat dans la base de données. Si newID est égal à false, on effectue d'abord une première requête afin que le contrat existe, ensuite on met à jour le contrat avec les dernières données choisies par le fournisseur. Ensuite, on récupère la date d'aujourd'hui et on ouvre le compteur.
Note : Nous avons décelé un potentiel problème dans le fait que la génération d'un id de contrat et la création de contrat se fait en deux requêtes distinctes ce qui pourrait engendrer des problèmes d'atomicité dans le cas où deux fournisseurs effectuent la même action exactement au même moment.

1.2.2 Ajout d'un compteur à un portefeuille

L'ajout de compteur à un portefeuille s'exécute via la méthode addSupplyPoint() situé dans la classe GuiHandler du package gui.app1 . Cette méthode prend en paramètre trois variables de type String : name, ean et type. Tout d'abord, la méthode va vérifier si le compteur n'existe pas déjà dans le portefeuille. Si ce n'est pas le cas, alors la méthode va vérifier si le compteur existe ou non dans notre système en effectuant une requête http. Si le compteur existe déjà dans notre base de donnée, alors la méthode va vérifier si le compteur est bien actuellement ouvert et dispose donc d'un contrat. Si ce n'est pas le cas, le compteur ne pourra pas être ajouter au portefeuille car pour ajouter un compteur à son portefeuille, l'utilisateur doit obligatoirement être le propriétaire du compteur. Un message sera alors affiché lui indiquant qu'il doit attendre l'ouverture de son compteur par son fournisseur. Ensuite, si le compteur est ouvert et qu'il dispose donc d'un contrat, la méthode va donc vérifier via une requête http si l'utilisateur désireux d'ajouter ce compteur au portefeuille en est bien le propriétaire. Si le compteur n'existe pas dans notre système, alors le système va le créer avec les données entrées par l'utilisateur. Enfin, en cas de succès, la méthode va appeler la méthode addSupplyPoint() situé dans la classe EnergyPortofolio. Cette méthode va se charger d'ajouter le compteur au portefeuille au niveau du logiciel et initialiser les champs contrat, date d'ouverture et état de l'objet SupplyPoint représentant le compteur pour le bon fonctionnement du reste de l'application.

1.2.3 Visualisation de données superposées

L'algorithme de visualisation de données superposées est utilisé lorsque l'utilisateur désire visualiser sa consommation globale de données pour un portefeuille en fonction d'un type d'énergie. Lorsque l'utilisateur va cliquer sur un des sous-menus du menu donnée globale, une des trois méthodes globalClicked() va s'exécuter. Depuis cette méthode, la méthode gatherData() qui prend en paramètre un type d'énergie et qui retourne une liste de DataSupplyPoint (une liste de données de consommation) va elle-même s'exécuter. Dans cette méthode, nous allons itérer au travers de l'ensemble des compteurs du portefeuille. Lorsque la méthode rencontre un compteur ayant le même type d'énergie que celui passé en paramètre, nous allons exécuter la méthode superimposeData() qui retourne une List de DataSupplyPoint, en passant en paramètre la liste des données déjà récoltées et la liste des données de consommation du compteur sur lequel la méthode s'est arrêtée. Dans cette méthode superimposeData, nous gérons le cas où la première liste passée en paramètre est vide. Ce qui signifie que l'on peut tout simplement y copier l'ensemble de la deuxième liste. Si la première liste n'est pas vide, l'algorithme va parcourir une seule fois l'ensemble des DataSupplyPoint des deux listes. A chaque itération, l'algorithme va comparer les dates des deux DataSupplyPoint courants. Il va ajouter à la liste à retourner le DataSupplyPoint ayant la plus petite date et incrémenter l'indice de la liste dans laquelle ce DataSupplyPoint a été trouvé. Si les deux DataSupplyPoint courants ont des dates identiques, l'algorithme va superposer les données en les additionnant dans un nouveau DataSupplyPoint, l'ajouter dans la liste à retourner et incrémenter les indices des deux listes. Lorsque l'algorithme a parcouru les deux listes entièrement, il retourne la nouvelle liste composée de la superposition des données des deux listes précédemment passées en paramètre.

1.3 Informations pratiques concernant Gradle et les fichiers auto-exécutables

1. gradle clean : clean tout les .class du projet
2. gradle build : build le projet
3. gradle runAPP1 : lançe l'app 1
4. gradle runAPP2 : lançe l'app 2
5. gradle shadowJar : crée le JAR (avec toute les dépendances) pour l'app1
6. gradle javadoc : génère la javadoc du projet

Note : Pour transformer le jar pour qu'il lance l'app 2, il faut ouvrir le jar, aller dans META-INF et changer dans le fichier MANIFEST.MF la ligne : Main-Class : RunAPP1 par Main-Class : RunAPP2)
Nous avons utilisé la bibliothèque shadowJar car javaFX n'est plus présent depuis java 11 dans les bibliothèques de base, ce qui provoquait des problèmes lors de la création du JAR (shadowJar permet d'inclure toutes les dépendances directement dans notre jar en une seule commande) (link : https ://github.com/johnrengelman/shadow)

2. Application android

L'IDE utilisé pour le projet est Android Studio. l'application android a le même fonctionnement que l'application 1 sauf qu'elle fonctionne sous android. J'ai rencontré des soucis au niveau de l'envoi de requêtes HTTP car il faut ajouter la permission d'accès à internet du téléphone. De plus, l'utilisation de la librairie java.net et de la classe HttpRequest provoquait des problèmes car il faut absolument les lancer dans un Thread à part(d'où l'utilisation de la méthode doInBackground de la classe abstraite AsyncTask de la librairie android.os).
Android studio et l'os d'android gère directement l'orientation du téléphone. En effet, je pensais, lors de la partie modélisation, qu'il fallait bouger et modifier la disposition de mes éléments dans la code java pour replacer correctement les éléments et mieux les répartir. Cependant il ne fallait en réalité que créer à chaque fois deux fichiers xml, un pour l'orientation paysage et l'autre pour l'orientation portrait, ce qui permet par exemple de passer d'une listView(qui permet d'afficher les éléments d'une liste dans une liste déroulante) à un gridView (qui a le même fonctionnement que la listView mais pouvant afficher plusieurs colonnes d'éléments en fonction de la taille d'écran, ce qui est utile dans le mode paysage) car elles ont toutes deux la même classe parent AbsListView (liste abstraite, donc font la même chose mais ne diffèrent que dans leur fonctionnement interne.
Pour la représentation des éléments dans la listView, j'ai utilisé à chaque fois un adapter (qui permet d'afficher une View spécifique pour chaque élément de la liste), j'ai aussi donc créé des xml pour chaque liste d'éléments à représenter dans l'app 1 (invitations, données de compteur, compteurs d'un portefeuille, portefeuilles et utilisateurs).
J'ai également créé des bords personnalisés pour certains éléments afin de rendre le tout plus joli. les objets représentant l'utilisateur et ce qu'il possède se charge au fur et à mesure de l'avancement dans l'application, c'est-à-dire que par exemple la liste des compteur d'un portefeuille est vide avant que l'utilisateur ne clique sur le portefeuille, je ne pense pas que ça soit le meilleur moyen pour récuperer les données mais par manque de temps j'ai préféré reprendre le même mode de fonctionnement que l'app sur ordinateur (Pour moi le meilleur moyen aurait été de charger, juste après le login, l'entièreté des éléments de la DB et construire tous les objets à ce moment-là et ensuite lorsque l'utilisateur effectue une tâche qui a besoin d'envoyer des requêtes http, là, j'aurais appelé une fonction update(User user) qui aurait mis à jour les objets).
Également lors de la phase de conception je n'avais pas vraiment pris attention à la la taille des écrans androids, je pensais que l'utilisation de la mesure abstraite dp (Density independant pixel) allait gérer pour moi toutes les tailles d'écran. Sauf que ce n'est pas le cas, il fallait aussi que je fasse des xml (portrait et paysage) pour chaque catégorie différente de densité (il y en à 6). Par manque de temps, je n'ai fait le cas que pour une seule taille d'écran (cela veut dire que cela fonctionnera pour les écrans de cette taille et ceux de taille supérieure, mais pour ceux de taille inférieure il peut y avoir des éléments qui disparaissent).
J'ai créé également une classe SwipeDetector qui a pour but de faire une action quand l'utilisateur glisse son doigt sur l'écran ce qui permet de naviguer plus facilement et de manière

plus intuitive. Mon extention est découpée en 4 packages : handler, qui contient toutes les classes qui gèrent les fenêtres xml modelGUI.adapter qui contient tous les adapteurs utilisés tool qui contient l'ensemble de classes utilitaires user contient les classes représentants les différents objets principaux du projet (USER, SupplyPoint, EnergyPortfolio, ...)
il y a également des dossiers ressources importants : drawable-V24 qui contient tous les éléments drawable (les bords personnalisés et les images) layout qui contient les layouts représentant les items (invitation, portfolio, compteur, ...) layout-land et layout-port qui contiennent respectivement les layouts principaux pour le mode paysage et pour le mode portrait
Les deux commandes gradles (pour les lancer il faut cependant le SDK d'android disponible sur android studio et se placer dans le dossier AppGLandroid) disponibles sont : gradle assembleRelease qui permet de générer l'apk de l'application (application apk signée, l'apk est généré dans AppGLandroid/app/build/outputs/apk/release) gradle myJavadoc qui permet de générer la javadoc de l'application android. NB : il y a eu pas mal de problème lorsqu'on génère la javadoc d'android car la javadoc ne trouvais pas la classe "R", la classe R.java est «une classe générée dynamiquement, créée pendant le processus de construction pour identifier dynamiquement tous les actifs (des chaînes aux widgets Android en passant par les mises en page), pour une utilisation dans les classes Java dans l'application Android. Notez que ceci R.javaest spécifique à Android (bien que vous puissiez le dupliquer pour d'autres plates-formes, c'est très pratique), donc cela n'a pas grand-chose à voir avec les constructions du langage Java» (voir : `https://stackoverflow.com/questions/6804053/understand-the-r-class-in-android`), j'ai donc du "bidouiller" avec les paramètres de gradle mais finalement j'ai réussis à créer la commande et à la faire fonctionner.
j'ai également utilisé la librairie externe AnyChart et multidex pour les graphiques de données (voir `https://github.com/AnyChart/AnyChart-Android`)
Etant donné que la partie traduction n'a pas été finie par le collègue qui devait la terminer dans les derniers jours, cette fonctionnalité n'est également pas présente dans l'application android
Pour finir, l'application android ne ressemble pas en tout point aux maquettes faites à l'aide de Adobe XD mais la structure reste la même
VIDEO : `https://youtu.be/q4Zk6CpVdPk`
NB : les images utilisées dans mon extensions sont toutes libres de droit et se trouve sur le site `https://pixabay.com/`

3. Auto-production d'électricité

Vidéo de présentation : `https://youtu.be/dOeR73FSHDA`

L'extension Auto-production d'électricité permet au consommateur d'énergie de devenir producteur en ajoutant des compteurs de production d'électricité reliés à une installation de panneaux solaires ou à une installation de type éolienne. Le consommateur, via cette extension, pourra également monitorer sa production d'électricité et le rendement de ses installations. Aussi, lorsqu'un de ses compteurs aura produit 1000 kwh d'énergie depuis la dernière demande de certificat vert, le consommateur qui détient ce compteur recevra un email le notifiant qu'il peut effectuer une nouvelle demande de certificat vert. Cette demande se fait via l'application 1, dans le menu surplombant la fenêtre de visualisation de données. Pour cette extension , un nouvel utilisateur est introduit : l'autorité régionale. Lors de sa connexion à l'application 1, celui-ci arrivera sur une fenêtre qui lui est totalement dédiée et où il pourra observer l'ensemble des demandes de certificats verts en attente. En cliquant sur l'une d'elles, il pourra ainsi visualiser les données de productions du compteur lié à la demande de certificat et approuver ou non le certificat si celui-ci est valide. En ce qui concerne la visualisation des données de production pour un utilisateur classique, celle-ci se déroule exactement comme lors de la visualisation de données de consommation. L'extension permet de visualiser l'ensemble de la production du portefeuille ainsi que la superposition de la production et de la consommation du portefeuille via le menu Données globales.

Lorsque l'utilisateur clique sur un portefeuille de production, celui-ci peut accéder à une fenêtre de rendement via le menu Rendement. Cette fenêtre varie en fonction du type d'installation, si il s'agit d'une installation de panneaux solaires, l'utilisateur va pouvoir observer le rendement théorique de son installation ainsi que le rendement réel actuel. Il pourra également y observer les données de production et modifier l'échelle et la période afin de calculer son rendement sur une certaine période uniquement. Le calcul du rendement théorique d'une installation de panneaux solaires se fait via les données introduites dans le système par l'utilisateur à la création de son compteur lié à l'installation. Lorsque l'utilisateur ajoute un compteur à un portefeuille et que celui-ci est inconnu par le système, celui-ci pourra spécifier si il s'agit d'un compteur de production ou de consommation. Si l'utilisateur spécifie qu'il s'agit d'une installation de panneaux solaires, celui-ci sera invité à indiquer la puissance des panneaux solaires, la superficie d'un panneau ainsi que le nombre de panneaux.
Les formules utilisées pour le calcul du rendement théorique et réel sont tirées du site : https ://www.panneausolaire-info.be/rendement .La formule du calcul du rendement théorique utilisée est : puissance/(superficie x 1000). Vu que l'on dispose de la quantité réelle d'énergie produite, une quantité totale d'énergie moyenne produite sur une année va être calculée à partir des données de production disponible sur la période choisie pas l'utilisateur. Grâce à cette valeur de production annuelle de l'installation, on va pouvoir calculer la puissance réelle d'un panneau. Le rendement réel de l'installation est donc déterminé

par la formule : (puissance réelle x 0.9)/(superifice x 1000). Le coefficient 0.9 est dû au fait qu'un panneau n'est pas toujours forcément utilisé à 100%.

S'il s'agit d'une installation éolienne, la fenêtre de rendement est simplifiée et ne permet que d'observer une moyenne représentant le production en kwh par jour de l'installation.

Au niveau de la base de données, 3 nouvelles tables sont introduites et une table a été modifiée. Les tables GreenCertificate, ProductionType et SolarPanelInfo apparaissent. GreenCertificate sert à stocker les certificats vers qu'ils soient en attente de validation ou validés. Elle contient une colonne ID représentant l'id unique du certificat, une colone meterID représentant le code ean du compteur relié au certificat, une colonne regionalid représentant l'id de l'autorité régionale en charge de ce certificat, une colonne value représentant la quantité d'énergie attachée à ce certificat ainsi que les colonnes askingDate et validationDate qui représentent les dates de demande et de validation du certificat. ProductionType va permettre de différencier les différents compteurs de type production en fonction du type d'installation relié. Notre système permet donc le support d'installation éolienne ainsi que d'installation solaire. Cette table contient deux colonnes : ean (le code ean du compteur) et type (le type d'installation : SOLAR ou WIND). Enfin, la table SolarPaenInfo contient l'ensemble des informations nécessaires au calcul du rendement d'un compteur de production solaire. On y retrouve donc les colonnes Puissance, superficie, nombre (de panneaux solaires) ainsi qu'une colonne ean qui représente le code ean du compteur associé.

Évidemment, pour la réalisation de cette extension, un deuxième serveur a été développé. Celui-ci est en tout point identique à celui de la fonctionnalité de base, il dispose simplement des servlets supplémentaires propres à l'extension.

Voici les identifiants des comptes déjà existants dans la base de données de l'extension :

1. mail : test@user.com / username : nsournac / mot de passe : azertyu (CONSOMMATEUR)
2. mail : example@engie.com / username : engie / mot de passe : azertyu (FOURNISSEUR)
3. mail : example@authority.com / username : authority / mot de passe : azertyu (AUTORITE REGIONALE)