CNED

COMPTE RENDU D'ACTIVITE

BTS SIO option SLAM
1ère année

ATELIER DE PROFESSIONALISATION 2

Damien Verger

Mai 2023

Table des matières

1. Contexte....1
2. Mission....1
3. Projet....2
3.1. Environnement de travail, base de données....2
3.2. Conception des interfaces, structure de l'application, création d'un dépôt distant, et codage du visuel....4
A. Conception des interfaces....4
B. Création et structure de l'application, sauvegarde sur le dépôt distant....6
C. Codage de la partie vue....6
3.3. Codage de la connexion et du modèle, génération de la documentation technique....8
A. Connexion à la base de données....8
B. Classe BddManager....8
C. Package Dal....9
D. Package Modele....9
E. Documentation technique....10
3.4. Codage des fonctionnalités à partir des cas d'utilisation....10
A. Connexion (cas d'utilisation 1)....11
B. Ajout un personnel (cas d'utilisation 2)....13
C. Suppression d'un personnel (cas d'utilisation 3)....16
D. Modification d'un personnel (cas d'utilisation 4)....17
E. Affichage des absences (cas d'utilisation 5)....20
F. Ajout d'une absence (cas d'utilisation 6)....22
G. Suppression d'une absence....24
H. Modification d'une absence (cas d'utilisation 8)....25
3.5. Création d'une documentation utilisateur en vidéo....30
3.6. Déploiement, rédaction du compte rendu et création de la page du portfolio....30
A. Déploiement....30
B. Rédaction du compte-rendu....30
C. Création de la page sur le portfolio....31
4. Bilan....32

ANNEXES....I

1. Contexte

Pour développer son offre numérique et son attractivité, le réseau des médiathèques de la Vienne, MediaTek86 a fait appel à la société InfoTechServices86. Celle-ci a pour mission de gérer le parc informatique de la structure et d'informatiser certaines de ses activités.

MediaTek86 permet déjà aux utilisateurs d'emprunter plus de 200 000 références (livres, journaux, CD, DVD…) à travers un catalogue en ligne et aimerait offrir la possibilité d'emprunter des films en VOD, à travers la création d'un nouveau service.

La plateforme souhaiterait également mettre en place des formations aux outils numériques et des autoformations en ligne.

2. Mission

Dans ce contexte, les services des Ressources Humaines de MediaTek86 ont sollicité la création d'une application de bureau dont le but sera de gérer les personnels d'une médiathèque (leur affectation à un service et un suivi de leurs absences).

Cette application est destinée à être installée sur un unique poste du service administratif, et devra seulement être accessible par un membre du personnel habilité.

Après authentification, la liste des personnels sera affichée, et il devra être possible d'ajouter/modifier/supprimer un personnel, et de demander un visuel sur ses absences.

Il devra également être possible d'ajouter/modifier/supprimer une absence, pour un personnel donné.

L'application sera développée en respectant l'architecture du modèle Modèle/Vue/Contrôleur (MCV), en langage de programmation C#. Les données générées, concernant les personnels, seront sauvegardées et organisées dans une base données, gérée par le système de gestion de bases de données relationnelles (SGBDR) MySql.

3. Projet

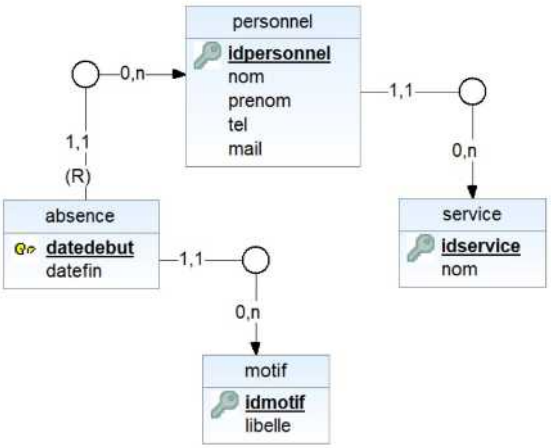
La réalisation de cette mission a été organisée en plusieurs étapes, qui sont détaillées ci-après.

3.1. Environnement de travail, base de données

La première étape consiste en la préparation de l'environnement de travail avec l'installation des outils nécessaires à la réalisation du projet :

- *Pencil* : conception et dessin des différentes fenêtres de l'application,
- *WinDesign* : exploitation du schéma conceptuel de données ,
- *Wampserver* : création d'un serveur local pour héberger la base de données,
- *Visual Studio 2019* : environnement de développement (IDE) permettant d'utiliser le langage C#.

Ensuite, le script de création de la base de données et généré grâce à l'exploitation du schéma conceptuel de données fourni, à l'aide du logiciel *WinDesign*, et exécuté sous *phpMyAdmin* pour créer la base de données.



L'accès à cette base de données doit être sécurisée. Un nouvel utilisateur ayant les droits d'accès est ajouté, à l'aide de l'interface *phpMyAdmin*.

L'accès à l'application doit également être sécurisé, l'identifiant et le mot de passe de la personne habilitée sont stockés dans une table "responsable", crée pour l'occasion, dans cette base.

Elle contient deux champs :

- le login ,
- le mot de passe, chiffré avec la fonction de hashage SHA2.

+ Options

login	pwd
user_responsable	bfd2bb20ff78332082d7c218982157b7a264550374be0e745d...

Les contenus des tables ***motif*** et ***service*** sont remplis à la main avec, pour chaque table, un ***id*** auto-incrémenté qui sert d'identifiant, et les libellés des motifs (*vacances* / *maladie* / *motif familial*/*congé parental*) et des services (*administratif* / *médiation culturel* / *prêt*).

+ Options

	idmotif	libelle
Éditer Copier Supprimer	1	vacances
Éditer Copier Supprimer	2	maladie
Éditer Copier Supprimer	3	motif familial
Éditer Copier Supprimer	4	congé parental

+ Options

	idservice	nom
Éditer Copier Supprimer	1	administratif
Éditer Copier Supprimer	2	médiation culturelle
Éditer Copier Supprimer	3	prêt

Les tables ***personnel*** et ***absence*** sont alimentés à l'aide de données aléatoires générées grâce à un générateur de données en ligne (https://www.generatedata.com/).

+ Options

	idpersonnel	idservice	nom	prenom	tel	mail
Éditer Copier Supprimer	2	1	Powers	Fritz	01 20 20 59 42	fritz.powers@mediatek86.fr
Éditer Copier Supprimer	4	3	Murray	Jarrod	01 20 20 59 56	jarrod.murray@mediatek86.fr
Éditer Copier Supprimer	6	2	Moore	Andrew	01 20 20 59 48	andrew.moore@mediatek86.fr
Éditer Copier Supprimer	7	2	Petersen	Galena	01 20 20 59 47	galena.petersen@mediatek86.fr
Éditer Copier Supprimer	8	2	Clark	Elvis	01 20 20 46 47	elvis.clay@mediatek86.fr
Éditer Copier Supprimer	9	2	Hudson	Arsenio	01 20 20 59 40	arsenio.hudson@mediatek86.fr

+ Options

	idpersonnel	datedebut	idmotif	datefin
Éditer Copier Supprimer	2	2022-06-08 00:00:00	3	2022-06-13 00:00:00
Éditer Copier Supprimer	2	2022-07-14 00:00:00	4	2022-12-22 00:00:00
Éditer Copier Supprimer	2	2022-10-15 00:00:00	3	2023-03-12 00:00:00
Éditer Copier Supprimer	2	2023-03-31 00:00:00	3	2023-04-04 00:00:00
Éditer Copier Supprimer	4	2022-05-28 00:00:00	1	2022-08-10 00:00:00
Éditer Copier Supprimer	4	2022-09-06 00:00:00	4	2022-09-15 00:00:00

A l'issue de cette étape, l'environnement de travail est opérationnel, la base de données est crée, alimentée et sécurisée.

3.2. Conception des interfaces, structure de l'application, création d'un dépôt distant, et codage du visuel

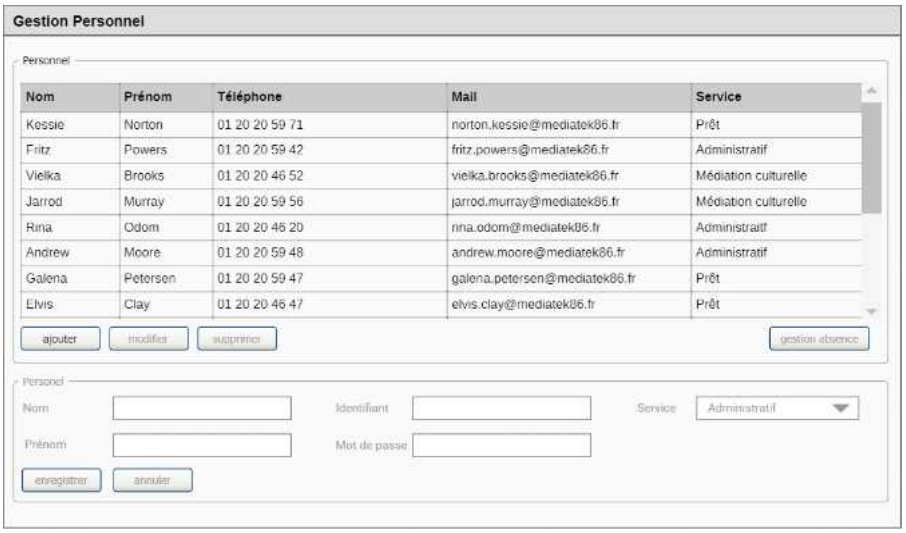
A. Conception des interfaces

A partir du dossier documentaire spécifiant les cas d'utilisations l'application, une maquette des différentes interfaces, permettant de répondre le plus efficacement possible aux besoins émis par le client, est créée. Elle se compose des fenêtres suivantes :

- *connexion*

Il s'agit de la première fenêtre apparaissant lors de l'ouverture de l'application. Elle permet de vérifier que la personne voulant se connecter est bien habilitée, en comparant le couple identifiant/mot de passe saisie avec celui enregistré dans la base de donnée. Des messages apparaissent si le couple identifiant/mot de passe est incorrect, ou si un champ est resté vide.

- *Personnel*



La fenêtre *personnel* s'affiche en cas de connexion réussie. Elle liste tous les personnels d'une médiathèque, classés par ordre alphabétique (nom, prénom) et précise leur nom, prénom, téléphone, adresse mail et leur service. Elle comprend plusieurs boutons permettant demander l'ajout, la modification ou la suppression d'un personnel, où l'affichage des absences relatives au personnel de la ligne sélectionnée.

Des messages apparaissent si aucun personnel n'est sélectionné lors des demandes de modification, suppression et d'affichage des absences.

- Ajout / modification de personnels

En cas de demande d'ajout ou de modification, une nouvelle fenêtre, contenant les champs nom, prénom, téléphone, mail et service, s'ouvre. Les champs sont remplis avec les informations du personnel de la ligne sélectionnée s'il s'agit d'une modification. Ils sont vides s'il s'agit d'un ajout.

Un message apparaît en cas de champs non remplis.

- Absence

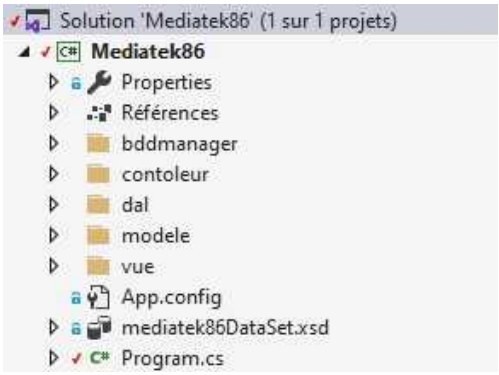
La fenêtre *absence* est appelée par le bouton *gestion des absences*. Elle affiche les absences relatives au personnel sélectionné dans la liste. Des boutons permettent de demander l'ajout, la modification ou la suppression d'une absence.

Des messages apparaissent en cas de champ non rempli, ou si une absence est déjà enregistrée aux dates renseignées.

B. Création et structure de l'application, sauvegarde sur le dépôt distant

L'application est créée sous Visual Studio 2019 et est directement structurée en respectant le MVC.

Les packages (vide dans un premier temps) *modele*, *vue*, *controller* sont créés ainsi que les packages *bddmanager* et *dal*.



Le package *vue* contiendra les formulaires d'affichage, en interaction avec l'utilisateur.

Le package *modele* contiendra les classes métiers (données et traitements liés aux données)

Le package *controller* contiendra les classes servant d'intermédiaire entre la *vue* et le *modele* ou le package *dal*.

Le package *dal (Data Access Layer)* contiendra les classes conçues pour préparer et formater les requêtes destinées à la base de données. Cette classe sera exploitée par les classes du package *controller*.

Le package *bddManager* contiendra la classe outil de connexion à la base de donnée. Cette dernière est isolée du reste de l'application.

Un dépôt distant est ensuite créé sur la plateforme GitHub, et lié à l'application, de manière à pouvoir sauvegarder et partager les versions du code.

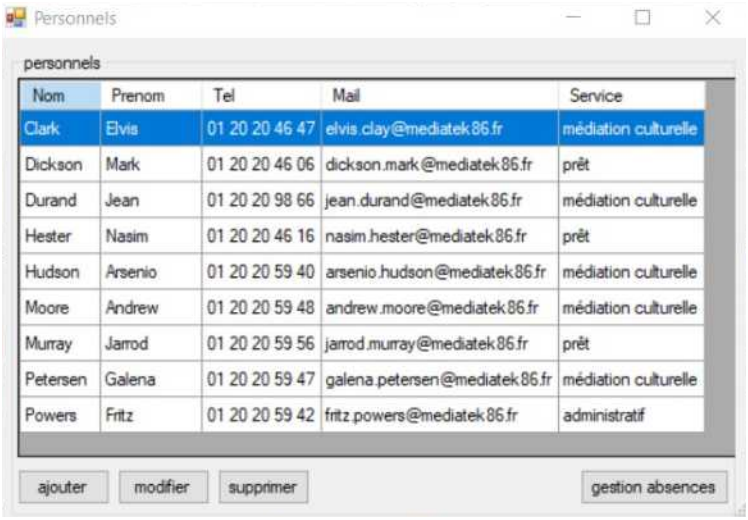
C. Codage de la partie vue

Les différentes vues conçues dans *Pencil* sont créées, à l'aide de l'éditeur graphique de *Visual Studio.*

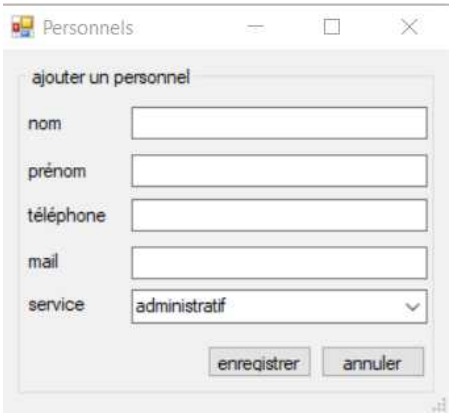
Les différentes vues sont nommées :

- FrmConnection

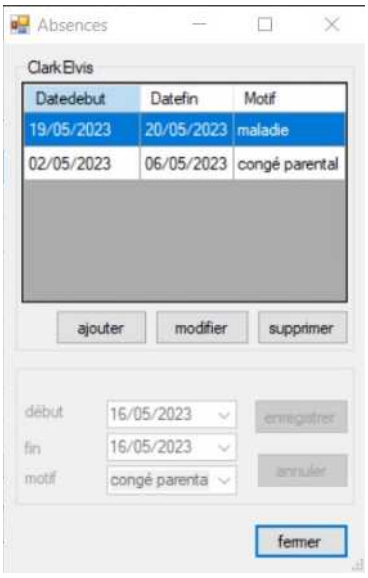
- FrmPersonnels



- FrmAjoutModifPersonnel



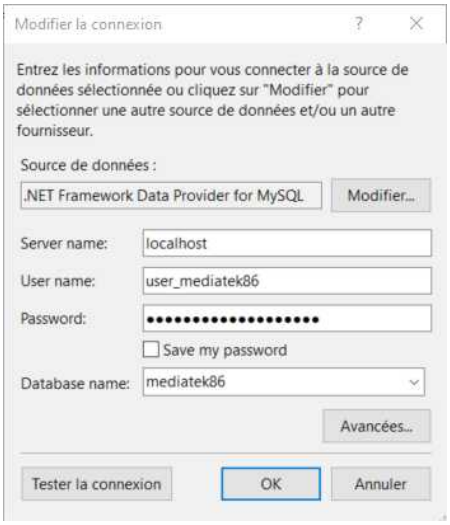
- FrmAbsence



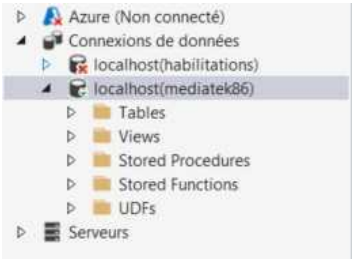
3.3. Codage de la connexion et du modèle, génération de la documentation technique

A. Connexion à la base de données

Visual Studio est configuré de manière à pouvoir se connecter à la base de données nouvellement créée sur le serveur local.



La connexion est ensuite rattachée au projet. Lors de cette phase, la chaîne de connexion générée est sauvegardée. Elle sera nécessaire dans l'application pour se connecter à la base de données.



B. Classe BddManager

La classe BddManager est créée dans le package *bddmanager*. Elle gère la connexion à la base de données .Cette classe est de type *Singleton*, et ne peut donc être instanciée qu'une seule fois. L'intérêt ici est de limiter le nombre de connexion vers la base de donnée à une seule.

Le constructeur de la classe est donc privé pour que la méthode ne puisse pas être appelée en dehors de la classe.

Classe Bddmanager

```
private BddManager(string stringConnect)
```

Il faut passer par la méthode *GetInstance*, qui contrôle si une instance a déjà été créée : si c'est le cas, elle la retourne, sinon elle l'a créée.

Classe Bddmanager

```
public static BddManager GetInstance(string stringConnect)
```

La classe contient également des méthodes permettant d'interagir avec la base de données :

- une méthode de mise à jour de la base de données recevant en paramètre la requête SQL formatée par la classe du package *dal*, et un dictionnaire contenant les paramètres de cette requête (si il y en a),

Classe Bddmanager

```
public void ReqUpdate(string stringQuery, Dictionary<string, object> parameters = null)
```

- une méthode d'interrogation, avec les mêmes types de paramètres que la méthode de mise à jour. Cette méthode retourne une liste de tableau d'objet contenant les valeurs des colonnes résultantes

Classe Bddmanager

```
public List<Object[]> ReqSelect(string stringQuery, Dictionary<string, object> parameters = null)
```

C. Package Dal

Au cours de cette étape, la classe *Access* du package *Dal* est créée. Elle sert d'interface pour accéder à la classe Bddmanager. Il s'agit d'un Singleton pour l'accès à la classe *BddManager*. Un objet de type *string* est valorisé avec la chaîne de connexion à la base de donnée.

Ensuite, de la même manière que la classe Bddmanager, le constructeur de la classe *Access* est privé, et il faut faut passer par la méthode *GetInstance*

Classe Access

```
private Access()

public static Access GetInstance()
```

D. Package Modele

Dans le package *modele*, les classes métiers correspondantes aux tables de la base de données (personnel, service, absence, motif, responsable), et les propriétés de ces

classes correspondent aux champs. Les constructeurs respectifs des différentes classes valorisent ces propriétés.

E. Documentation technique

La documentation technique d'une application liste ses classes et leurs membres. Les commentaires normalisés dans le code permettent de générer automatiquement une documentation au formal XML avec Visual Studio.

Mediatek86.xml

```xml
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>Mediatek86</name>
    </assembly>
    <members>
        <member name="T:Mediatek86.bddmanager.BddManager">
            <summary>
            Singleton : connexion à la base de données et exécution des requêtes
            </summary>
        </member>
        <member name="F:Mediatek86.bddmanager.BddManager.instance">
            <summary>
            instance unique de la classe
            </summary>
        </member>
        <member name="F:Mediatek86.bddmanager.BddManager.connection">
            <summary>
            objet de connexion à la BDD à partir d'une chaîne de connexion
            </summary>
        </member>
        <member name="M:Mediatek86.bddmanager.BddManager.#ctor(System.String)">
            <summary>
            Constructeur pour créer la connexion à la BDD et l'ouvrir
            </summary>
            <param name="stringConnect">chaine de connexion</param>
        </member>
```

Pour obtenir une version plus facilement lisible, un outil comme SandCastle permet d'obtenir une version HTML, consultable depuis un navigateur, de la documentation.

3.4. Codage des fonctionnalités à partir des cas d'utilisation

Pour l'explication des fonctionnalités de l'application, seules les signatures des différentes méthodes crées sont spécifiées.

Le contenu détaillé des classes et formulaires dans lesquelles se trouvent ces méthodes se trouvent en annexe.

La méthode *Main*, entrée du programme se trouve dans la classe *Program*. Elle contient, entre autres, l'instruction d'ouverture d'une nouvelle fenêtre *FrmAuthentification* en tant que fenêtre principale de l'application.

Classe Program

```
static void Main()
```

A. Connexion (cas d'utilisation 1)

A l'ouverture de *FrmAuthentification*, une méthode d'initialisation est appelée :

Formulaire FrmAuthentification

```
private void Init()
```

La méthode fixe la position de la fenêtre au centre de l'écran (cette instruction sera présente dans les méthode *init* de chaque formulaire), et crée et valorise la propriété *controller* en appelant le constructeur de la classe *FrmAuthentificationController*.

La seule action possible au niveau de cette fenêtre est un clic sur le bouton *se connecter*.

Cette action appelle la méthode *btnSeConnecter_Click :*

Classe FrmAuthentification

```
private void btnSeConnecter_Click(object sender, EventArgs e)
```

La méthode vérifie dans un premier temps que les champs contiennent une valeur. Si ce n'est pas le cas, un message est affiché précisant que tous les champs doivent être remplis.

Si les champs sont remplis, un nouvel objet *responsable* de type *Responsable* est créé et instancié, en appelant le constructeur de la classe *Responsable* (avec en paramètre le contenu des champs de saisie)

Classe Responsable

```
public Responsable(string identifiant, string mdp)
```

Un nouveau test est réalisée en appelant la fonction *ControleAuthentification* du contrôleur *FrmAuthentificationController*, avec en paramètre l'objet *responsable* créé :

Classe FrmAuthentificationController

```
public Boolean ControleAuthentification(Responsable responsable)
```

Cette méthode appelle ensuite la méthode *ControleAuthentification* de la classe *ResponsableAccess,* avec en paramètre le même objet *responsable* :

Classe ResponsableAccess

```
public Boolean ControleAuthentification(Responsable responsable)
```

Cette méthode prépare la requête pour la base de données pour tester si les propriétés de l'objet *responsable* (identifiant et mdp) entrés dans les champs de la fenêtre *FrmAuthentification* correspondent à celle de la personne habilitée à se connecter, enregistrés dans la base de données.

Elle retourne *true* (utilisateur habilité) si les champs correspondent et *false* (mauvais couple identifiant/mdp). Le résultat sous forme de Booléen est donc "renvoyé" jusqu'à la classe *FrmAuthentification*.

A ce moment là, si le Boolean est true, un nouvel objet de type *FrmPersonnels* (correspondant à la fenêtre affichant la liste des personnelles) est créé, instanciée et affichée.

Sinon un message informant que les identifiants de connexion sont incorrects, ou que la personne essayant de se connecter n'est pas habilitée, apparaît.

Le constructeur *FrmPersonnels*, recevant en paramètre l'instance de *FrmAuthentification* l'ayant appelé, ferme la fenêtre d'authentification et lance la méthode d'initialisation :

Classe FrmPersonnels

```
public FrmPersonnels(FrmAuthentification frmauthentification)
```

Classe FrmPersonnels

```
private void Init()
```

Cette dernière crée et instancie le contrôleur de type *FrmPersonnelsController* et appelle la méthode *RemplirListePersonnelle*() :

Classe FrmPersonnels

```
public void RemplirListePersonnels()
```

Cette méthode remplit une liste avec des objets de type *Personnel* et l'affiche dans un DataGridView (en passant par une BindingSource). Pour cela, elle fait appel à la méthode *GetLesPersonnels* du contrôleur :

Classe FrmPersonnelsController

```
public List<Personnel> GetLesPersonnels()
```

Qui elle même appelle la méthode du même nom de la classe *PersonnelsAccess* :

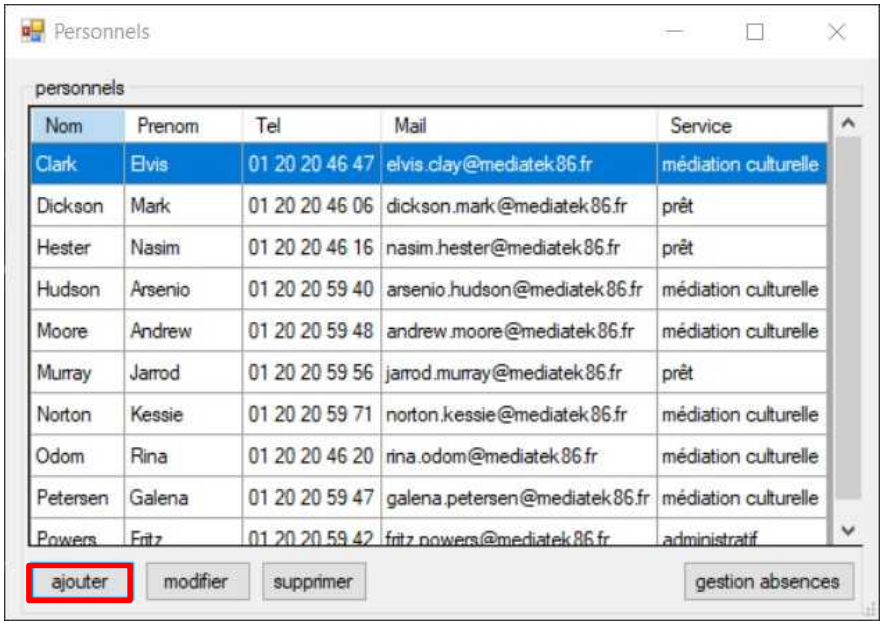
Classe PersonnelsAccess

```
public List<Personnel> GetLesPersonnels()
```

Cette fonction prépare la requête d'interrogation de la base de donnée, exploite la classe Access (qui elle-même exploite la classe de connexion bddManager) pour récupérer la liste des personnels (sous forme d'objet de type Personnels) et la retourner, à travers les méthodes précédentes, à la classe FrmPersonnels.

Ainsi, après le démarrage de l'application, la fenêtre de connexion s'ouvre, pour permettre l'authentification de la personne habilité. Une connexion autorisée permet l'ouverture de la fenêtre Personnels, qui affichent la liste des personnels actifs, récupérés depuis la base de données.

B. Ajout un personnel (cas d'utilisation 2)



La fenêtre *Personnel* précédemment affichée contient un bouton *ajouter*.

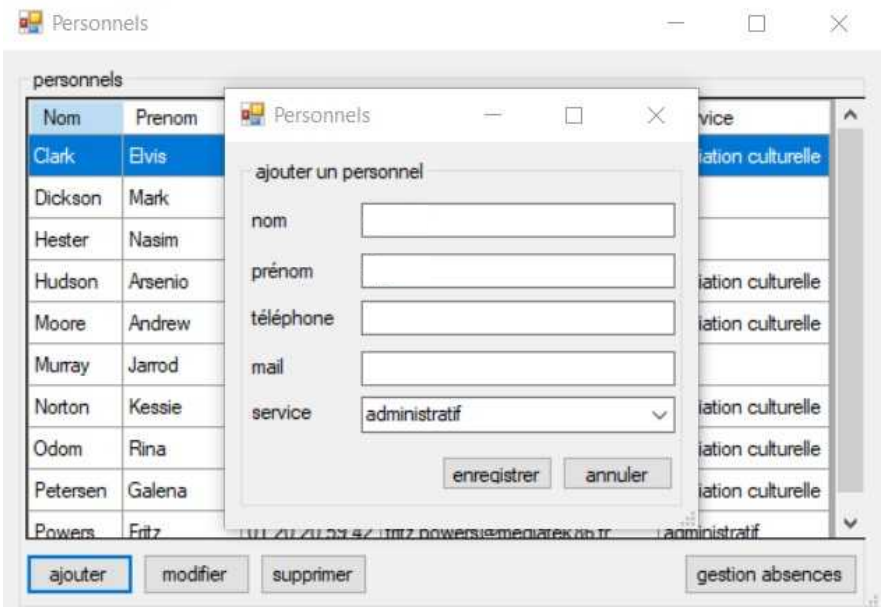
L'action de cliquer sur ce bouton appelle la méthode *btnAjouterPersonnel_Click* du formulaire *FrmPersonnels* qui créer, instancie et affiche une nouvelle fenêtre *FrmAjoutModifPersonnel.*

Classe FrmPersonnels

```
private void btnAjouterPersonnel_Click(object sender, EventArgs e)
```

Cette méthode crée, instancie et affiche une nouvelle fenêtre *FrmAjoutModifPersonnel* avec les paramètres (null, sender, this) :

- *null* : paramètre nécessaire pour le cas d'utilisation "modification personnel"
- *sender* : bouton qui a généré l'action (ici *btnAjouterPersonnel*),
- *this* : instance actuelle de *FrmPersonnel*.



Classe FrmAjoutModifPersonnels

```
public FrmAjoutModifPersonnel(Personnel personnel, Object sender, FrmPersonnels frm)
```

Cette méthode construit les éléments graphiques et appelle l'initialisation (méthode *Init*), avec les paramètres reçus (null, sender, frm) :

Classe FrmAjoutModifPersonnels

```
private void Init(Personnel personnel, Object sender, FrmPersonnels frm)
```

Cette méthode commence par créer et instancier le contrôleur *FrmAjoutModifPersonnelsController.*

La méthode *RemplirListService* de cette même classe est ensuite appelée :

Classe FrmAjoutModifPersonnels

```
private void RemplirListServices()
```

De la même manière que la méthode *RemplirListPersonnels* du formulaire *FrmPersonnels*, cette méthode appelle la méthode *GetLesServices* du contrôleur, qui elle-même appelle la méthode *GetLesServices* de la classe *ServiceAccess*. Cette dernière prépare et formate la requête d'interrogation de la base de donnée (à travers les classes *Access*, puis *bddManager*) et retourne la liste des *Services*.

Ensuite, de retour dans la méthode *Init* un test est réalisé sur l'objet *sender* pour déterminer quel bouton à été cliqué (*btnAjouterPersonnel* ou *btnModifierPersonnel*).

S'il s'agit du bouton a*jouter*, les champs de saisie du formulaire *FrmAjoutModifPersonnel* sont vidés, pour permettre la saisie d'un nouveau personnel.

Un clic sur le bouton *enregistrer* déclenche la méthode associée :

Classe FrmAjoutModifPersonnels

```
private void btnEnregistrerPersonnel_Click(object sender, EventArgs e)
```

Cette méthode commence par vérifier si tous les champs sont remplis, sinon un message d'information apparaît.

Un objet *service* de type *Service* est ensuite créé et valorisé à partir du service sélectionné dans la ComboBox *cbbServices.*

L'étape suivant consiste en un test déterminant si nous sommes dans le cas d'une modification ou d'un ajout de personnel.

Dans le cas d'un ajout, un nouvel objet *personnel* de type *Personnel* est créé et instancié à l'aide de l'appel du constructeur de la classe *Personnel*. (avec comme paramètre, les valeurs insérer dans les champs à remplir)

Classe FrmAjoutModifPersonnels

```
public Personnel(int idpersonnel, string nom, string prenom, string tel, string mail,
                 Service service)
```

La méthode *Addpersonnel* du contrôleur est appelée avec l'objet *personnel*, qui vient d'être créé, en paramètre.

Classe FrmAjoutModifPersonnelsController

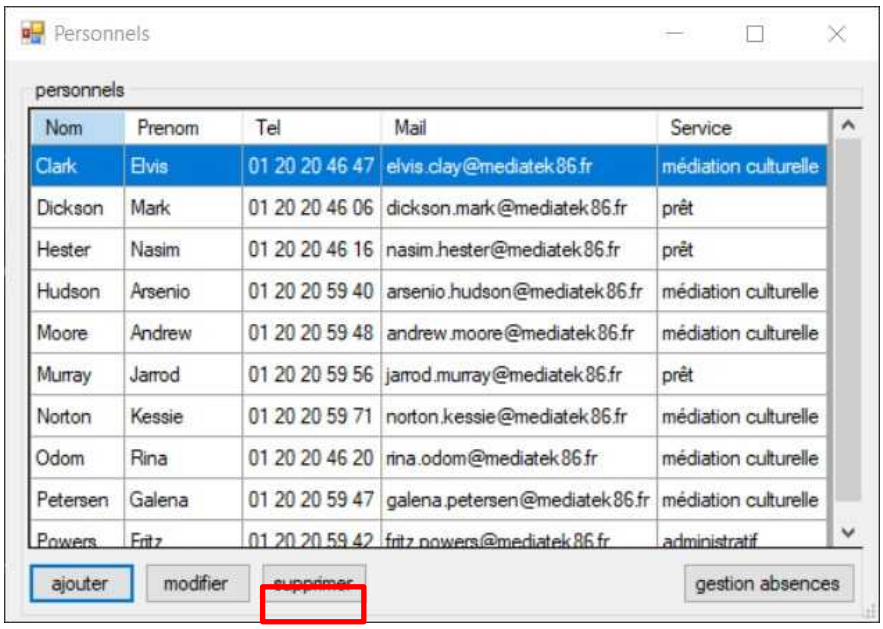
```
public void AddPersonnel(Personnel personnel)
```

Cette dernière appelle la méthode *AddPersonnel* de la classe *personnelAccess*. Celle-ci prépare et formate la requête d'insertion dans la base de données, et demande la mise à jour de cette dernière (à travers les classes *Access*, puis *bddManager*) .

De retour dans la méthode *btnEnregistrerPersonnel_Click*, la méthode *RemplirListePersonnels* du formulaire *FrmPersonnels* est de nouveau appelé pour mettre à jour l'affichage des personnels dans le DataGridView.

c. Suppression d'un personnel (cas d'utilisation 3)

La fenêtre *Personnel* précédemment affichée contient un bouton *supprimer*.



L'action *Clic* sur ce bouton appelle la méthode *btnSupprimerPersonnel_Click* du formulaire *FrmPersonnels*

Formulaire FrmPersonnels

```
private void btnSupprimerPersonnel_Click(object sender, EventArgs e)
```

Cette méthode commence par tester si une ligne est bien selectionnée dans le DataGridView, dans le cas contraire, un message est affiché.

Si une ligne est bien sélectionnée, la méthode génère une MessageBox demandant de confirmer ouy non la suppression du personnel. En cas de confirmation, les méthodes *DelAllAbsence* et *DelPersonnel* du contrôleur sont appelées, avec, pour chaque méthode, l'objet *personnel* issue de la ligne sélectionnée.

Classe FrmAjoutModifPersonnelsController

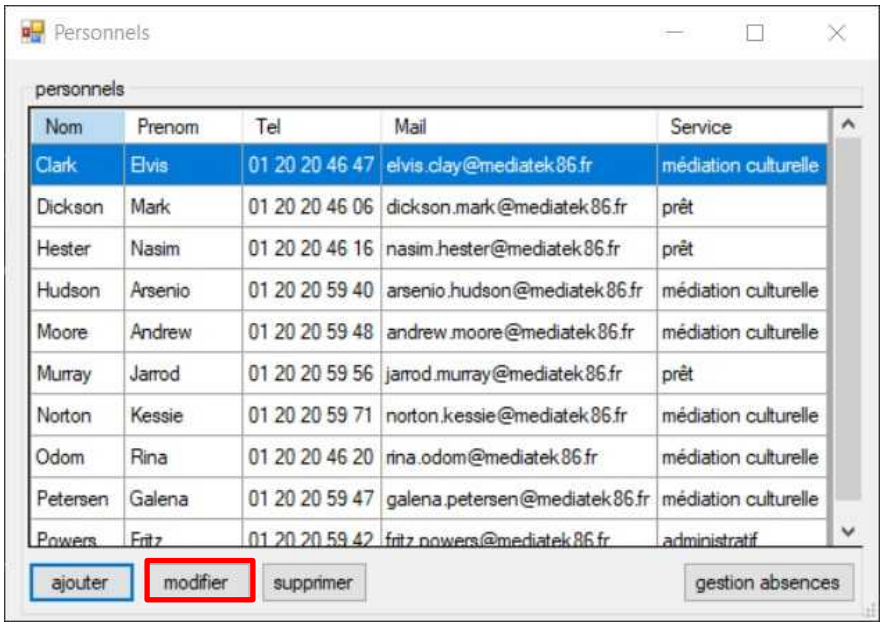
```
public void DelAllAbsence(Personnel personnel)

public void DelPersonnel(Personnel personnel)
```

De part les dépendances entre les tables personnel et absence dans la base de données, il est nécessaire de supprimer les absences liées à un personnel avant de pouvoir supprimer ce dernier.

Le contrôleur appelle ensuite les méthodes *DelAllAbsence* et *DelPersonnel* des classes *absenceAccess* et *personnelAccess*. Ces dernières préparent et formatent la requête de suppression de la base de donnée, et demande la mise à jour de cette dernière (à travers les classes *Access*, puis *bddManager*) .

De retour dans la méthode *btnSupprimerPersonnel_Click*, la méthode *RemplirListePersonnels* du formulaire *FrmPersonnels* est de nouveau appelé pour mettre à jour l'affichage des personnels dans le DataGridView.

D. Modification d'un personnel (cas d'utilisation 4)



La fenêtre *Personnel* précédemment affichée contient un bouton *modifier*.

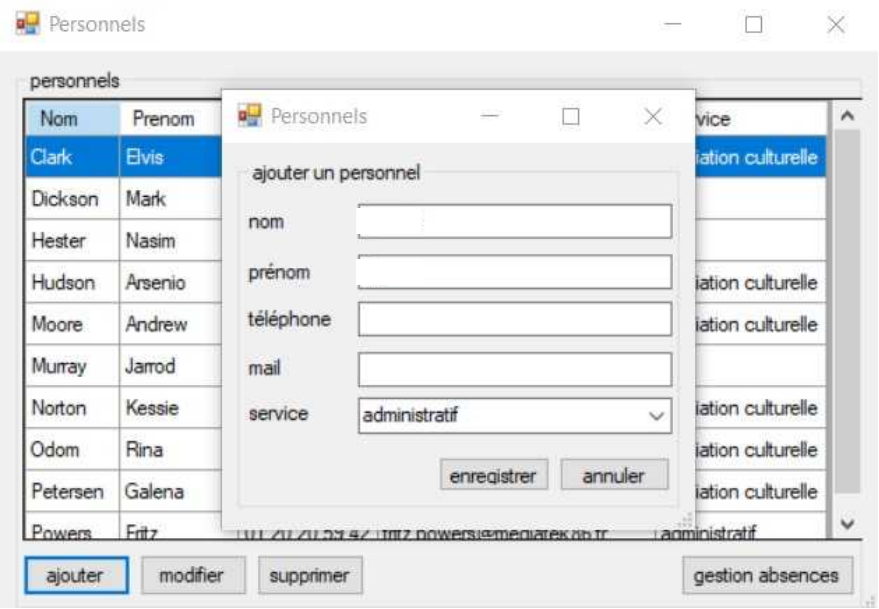
Un clic sur ce bouton appelle la méthode *btnModifierPersonnel_Click* du formulaire *FrmPersonnels* qui commence par tester si une ligne est bien sélectionnée dans le DataGridView, dans le cas contraire, un message est affiché. Si une ligne a bien été sélectionnée, un nouveau formulaire *FrmAjoutModifPersonnel est* crée, instancié et affiché.

Formulaire FrmPersonnels

```
private void btnModifierPersonnel_Click(object sender, EventArgs e)
```

La méthode crée et instancie un objet personnel de type *Personnel* à partir de l'élément sélectionné dans le DataGridView. Elle crée, instancie et affiche un nouveau formulaire *FrmAjoutModifPersonnel* avec les paramètres (*personnel*, *sender*, *this*) :

- *personnel* : objet qui servira à remplir le formulaire *FrmAjoutModifPersonnel* avec les informations du personnel à modifier
- *sender* : bouton qui a généré l'action (ici *btnAjouterPersonnel*),
- *this* : instance actuelle de *FrmPersonnel*.



Formulaire FrmAjoutModifPersonnels

```
public FrmAjoutModifPersonnel(Personnel personnel, Object sender, FrmPersonnels frm)
```

Cette méthode construit les éléments graphiques et appelle la méthode d'initialisation, avec les paramètres reçus (*personnel*, *sender*, *frm*) :

Formulaire FrmAjoutModifPersonnels

```
private void Init(Personnel personnel, Object sender, FrmPersonnels frm)
```

La méthode d'initialisation commence par instancier le contrôleur *FrmAjoutModifPersonnelsController.*

La méthode *RemplirListService* de cette même classe est ensuite appelée :

Formulaire FrmAjoutModifPersonnels

```
private void RemplirListServices()
```

De la même façon que la méthode *RemplirListPersonnels* du formulaire *FrmPersonnels*, cette méthode appelle la méthode *GetLesServices* du contrôleur, qui elle-même appelle la méthode *GetLesServices* de la classe *ServiceAccess*. Cette dernière prépare et formate la requête d'interrogation dans la base de données (à travers les classes *Access*, puis *bddManager*) et retourne la liste des *Services*.

De retour dans la méthode *Init* un test est réalisé sur l'objet *sender* pour déterminer quel bouton à été cliqué (*btnAjouterPersonnel* ou *btnModifierPersonnel*).

S'il s'agit du bouton *modifier*, les champs de saisie du formulaire *FrmAjoutModifPersonnel* sont remplis avec les propriétés de l'objet personnel reçu en paramètre, pour permettre la modification.

Un clic sur le bouton *enregistrer* déclenche la méthode associée :

Formulaire FrmAjoutModifPersonnels

```
private void btnEnregistrerPersonnel_Click(object sender, EventArgs e)
```

Cette méthode commence par vérifier si tous les champs sont remplis, sinon un message d'information apparaît.

Un objet service de type *Service* est ensuite créé et valorisé à partir du service sélectionné dans le ComboBox *cbbServices.*

L'étape suivant consiste en un test déterminant si nous sommes dans le cas d'une modification ou d'un ajout de personnel.

Dans le cas d'une modification, les paramètres de l'objet *personnel* reçu en paramètre sont valorisés avec les valeurs des champs de saisie (modifié, ou non)

La méthode *Update*P*ersonnel* du contrôleur est appelée avec l'objet *personnel*, qui vient d'être modifié, en paramètre.

Classe FrmAjoutModifPersonnelsController

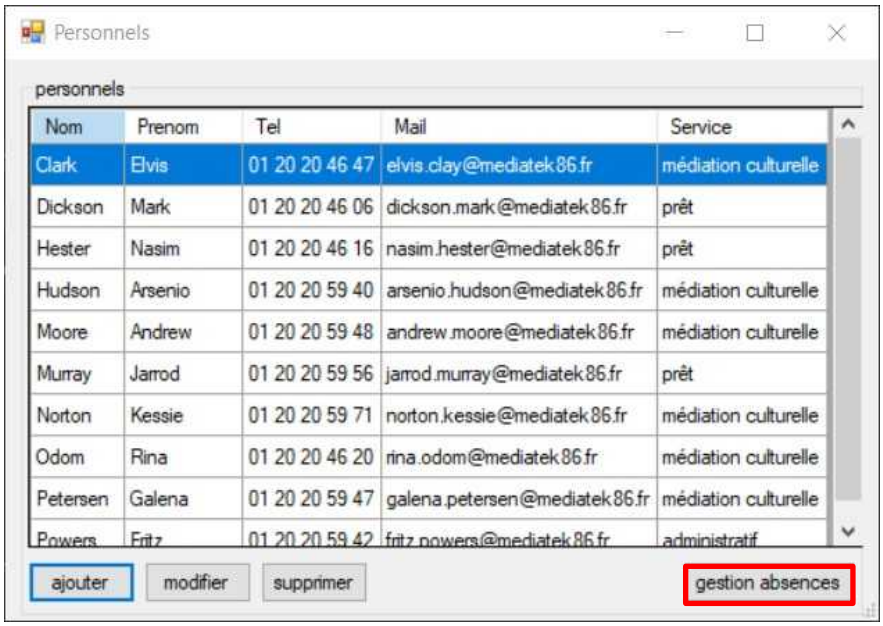
```
public void UpdatePersonnel(Personnel personnel)
```

Elle-même appelle la méthode *UpdatePersonnel* de la classe *personnelAccess*. Cette dernière prépare et formate la requête de mise à jour dans la base de données, et demande la mise à jour de cette dernière (à travers les classes *Access*, puis *bddManager*).

De retour dans la méthode *btnEnregistrerPersonnel_Click*, la méthode *RemplirListePersonnels* du formulaire *FrmPersonnels* est de nouveau appelé pour mettre à jour l'affichage des personnels dans le DataGridView.

E. Affichage des absences (cas d'utilisation 5)

La fenêtre *Personnel* précédemment affichée contient un bouton *gestion absences*.



L'action de cliquer sur ce bouton appelle la méthode *btnAbsence_Click* du formulaire *FrmPersonnels*

Formulaire FrmPersonnels

```
private void btnAbsence_Click(object sender, EventArgs e)
```

Cette méthode commence par tester si une ligne est bien sélectionnée dans le *DataGridView*, dans le cas contraire, une MessageBox affiche un message d'information.

Si une ligne est sélectionnée, la méthode crée et instancie un objet *personnel* de type *Personnel* à partir de la ligne sélectionnée dans le DataGridView. Elle crée, instancie et affiche un nouveau formulaire *FrmAbsence* avec le paramètre (*personnel*) :

- *personnel* : objet qui servira à déterminer le personnel pour lequel les absences doivent être affichées.

Le constructeur du formulaire *FrmAbsence* initialise les composants graphiques et appelle sa méthode *Init* en lui envoyant l'objet *personnel* en paramètre.

Formulaire FrmAbsence

```
public FrmAbsence(Personnel personnel)

private void Init(Personnel personnel)
```

Cette dernière crée le contrôleur (*frmAbsenceController*), et appelle ses méthodes *RemplirListeAbsences et RemplirListeMotif*. La méthode *RemplirListeAbsences* est appelé, avec l'objet *personnel,* dont on veut afficher les absences, en paramètre.

Formulaire FrmAbsence

```
private void RemplirListeAbsences(Personnel personnel)

private void RemplirListeMotifs()
```

La méthode *RemplirListeAbsences* remplit une liste avec des objets de type *Absence,* associés au *personnel* reçu en paramètre et l'affiche dans un DataGridView (en passant par une BindingSource). Pour cela, elle fait appel à la méthode *GetLesAbsences* du contrôleur (toujours avec l'objet *personnel* en paramètre) :

La méthode *RemplirListeMotifs* remplit une liste avec des objets de type *Motif et l'insère dans une ComboBox* (en passant par une BindingSource). Pour cela, elle fait appel à la méthode *GetLesMotifs* du contrôleur :

Classe FrmAbsenceController

```
public List<Absence> GetLesAbsences(personnel)

public List<Motif> GetLesMotifs()
```

Ces deux méthodes appellent les méthodes du même nom de la classe *PersonnelsAccess* :

Classe AbsenceAccess

```
public List<Absence> GetLesAbsences(Personnel personnel)
```

Classe MotifAccess

```
public List<Motif> GetLesMotifs()
```

Ces fonctions préparnt les requêtes d'interrogation de la base de donnée, exploite la classe *Access* (qui elle-même exploite la classe de connexion *bddManager*) pour récupérer les listes des absences (sous forme d'objet de type *Absence*) et des motifs (sous forme d'objet de type *Motif*) et les retourner, à travers les méthodes précédentes, au formulaire *FrmAbsence*.

De retour dans la méthode *Init* du formulaire *FrmAbsence*, une dernière méthode est appelée :

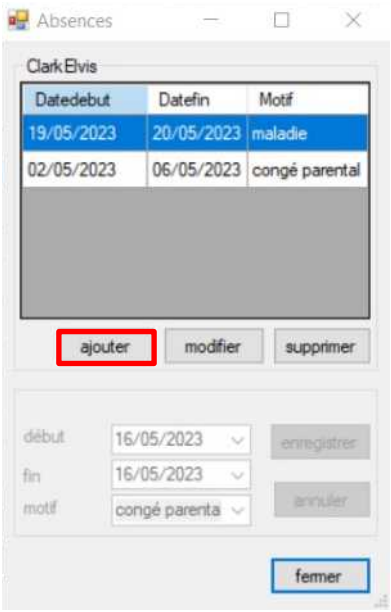
Formulaire FrmAbsence

```
private void InitialisationActionAbsence()
```

Cette méthode rend accessible la sélection d'une absence dans le *DataGridView*, rend inaccessible la zone d'ajout / modification, et initialise les dates des DateTimePicker début et fin à la date du jour.

F. Ajout d'une absence (cas d'utilisation 6)

Le formulaire *Absence* précédemment affiché contient un bouton *ajouter*.



L'action de cliquer sur ce bouton appelle la méthode *btnAjouter_Click* du formulaire *FrmAbsence* qui rend disponible la partie du formulaire contenant les zones de saisie.

La méthode *EnCoursAjoutAbsence* est ensuite appelée, avec comme paramètre, le booléen *true*

Formulaire FrmAbsence

```
private void EnCoursAjoutAbsence(Boolean modif)
```

Cette méthode prépare le formulaire pour un ajout d'absence, en rendant accessible la zone de saisie et en initialisant les DateTimePicker à la date du jour, et en affichant le premier élément de la liste dans la ComboBox des motifs.

Un clic sur le bouton *enregistrer* déclenche la méthode associée :

Formulaire FrmAbsence

```
private void btnEnregistrerAbsence_Click(object sender, EventArgs e)
```

Dans cette méthode, pour les tests qui vont être énumérés ci-dessous, si ils ne remplissent pas la condition, un message d'information est à chaque fois affiché dans une MessageBox.

Un premier test commence par vérifier si un élément est bien sélectionné dans la ComboBox des motifs (les DateTimePicker ne peuvent en effet pas être "vide").

Vient ensuite un deuxième test qui vérifie que la date de départ est bien antérieure à la date de fin de l'absence à ajouter.

Un troisième test vérifie que la méthode *ControleAbsenceSimultanées* renvoie bien le booléen *false*.

Formulaire FrmAbsence

```
private Boolean ControleAbsencesSimultanees()
```

Cette méthode renvoie *true* lorsque l'intervalle entre les dates de l'absence à ajouter chevauche un intervalle de date d'une absence déjà enregistrée. Il ne peut en effet pas y avoir deux absences sur la même période, pour un même personnel. Pour cela, pour chaque absence, elle vérifie que la date de début sélectionnée n'est pas antérieure ou égale à la date de début enregistrée. Elle vérifie également que la date de fin n'est pas postérieure ou égale à la date de début enregistrée.

L'étape suivant consiste en un test déterminant si nous sommes dans le cas d'une modification ou d'un ajout d'absence.

Dans le cas d'un ajout, un nouvel objet *absence* de type *Absence* est créé et instancié à l'aide de l'appel du constructeur de la classe *Absence*. (avec comme paramètre, les valeurs sélectionnées dans les champs de date et de motif)

Classe Absence

```
public Absence (int idpersonnel, DateTime datedebut, DateTime datefin, Motif motif)
```

La méthode *AddAbsence* du contrôleur est appelée avec l'objet *absence*, qui vient d'être créé, en paramètre.

Classe FrmAbsenceController

```
public void AddAbsence(Absence absence)
```

Elle-même appelle la méthode *AddAbsence* de la classe *absenceAccess*.

Classe absenceAccess

```
public void AddAbsence(Absence absence)
```

Cette dernière prépare et formate la requête d'insertion dans la base de donnés,et demande la mise à jour de cette dernière (à travers les classes *Access*, puis *bddManager*)
.

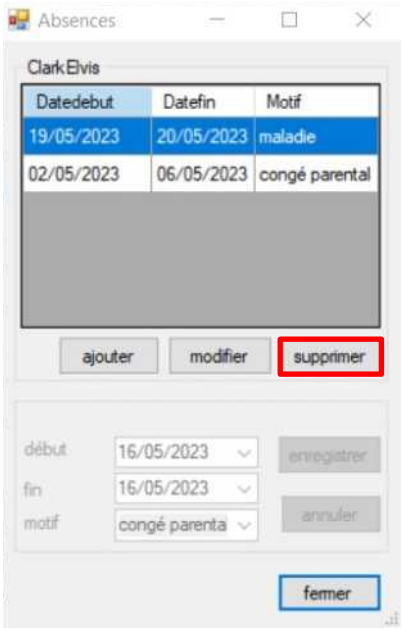
De retour dans la méthode *btnEnregistrerAbsence_Click*, la méthode *EnCoursAjoutAbsence* est appelée avec le booléen *false* en paramètre, pour valoriser le paramètre de classe *enCoursAjoutAbsence* à *false* (phase d'ajout terminée)

La méthode *RemplirListeAbsence* du formulaire *FrmAbsence* est ensuite de nouveau appelée avec comme paramètre, le personnel concerné par les absences, pour mettre à jour l'affichage des absences dans le DataGridView.

Enfin, la méthode *InitialisationActionAbsence* est appelée pour rendre inaccessible la zone de saisie, rendre accessible le DataGridView et réinitialiser les DateTimePicker et la ComboBox.

G. Suppression d'une absence

Le formulaire *Absence* précédemment affichée contient un bouton *supprimer*.



L'action de cliquer sur ce bouton appelle la méthode *btnSupprimerAbsence_Click* du formulaire *FrmAbsence*

Formulaire FrmAbsence

```
private void btnSupprimerAbsence_Click(object sender, EventArgs e)
```

Cette méthode commence par tester si une ligne est bien sélectionnée dans le DataGridView, dans le cas contraire, un message est affiché.

Si une ligne est bien sélectionnée, un objet absence de type *Absence* est créé, et valorisé avec l'objet *absence* récupéré grace à la sélection de la ligne du DataGridView. La méthode génère ensuite une MessageBox demandant de confirmer la suppression de l'absence.

Si la suppression est confirmée, la méthode *DelAbsence* du contrôleur est appelée, avec l'objet *absence* créé précédement, en paramètre.

Classe FrmAjoutModifPersonnelsController

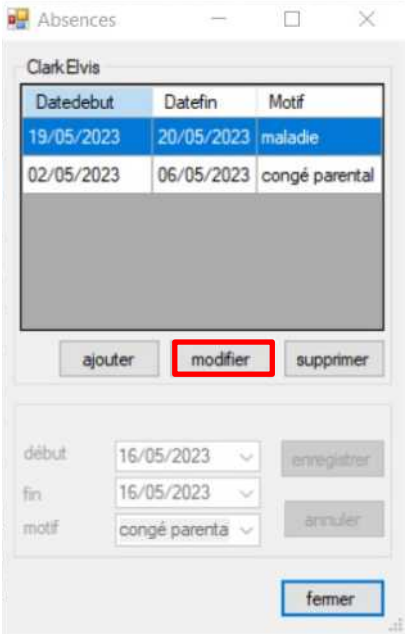
```
public void DelAbsence(Absence absence)
```

Le contôleur appelle ensuite la méthode *DelAbsence* de la classe *absenceAccess*. Cette dernière prépare et formate la requête de suppression dans la base de données, demande la mise à jour de cette dernière (à travers les classes *Access*, puis B*ddmanager*) .

De retour dans la méthode *btnSupprimerAbsence_Click*, la méthode *RemplirListeAbsence* du formulaire *FrmAbsence* est de nouveau appelé pour mettre à jour l'affichage des absences dans le *DataGridView*.

H. Modification d'une absence (cas d'utilisation 8)

Le formulaire *Absence* précédemment affiché contient un bouton *modifier*.



Un clic sur le bouton sur ce bouton appelle la méthode *btnModifierAbsence_Click* du formulaire *FrmAbsence.*

Formulaire FrmAbsence

```
private void btnModifierAbsence_Click
```

Cette méthode commence par rendre disponible la partie du formulaire contenant les zones de saisie.

Elle appelle la méthode *EnCoursModifAbsence*, avec comme paramètre, le booléen *true*.

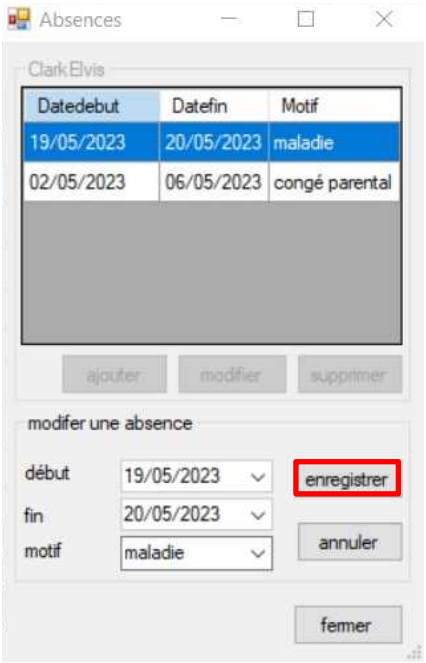
Formulaire FrmAbsence

```
private void EnCoursModifAbsence(Boolean modif)
```

Cette méthode prépare le formulaire pour une modification d'absence, en rendant accessible la zone de saisie.

Un nouvel objet *absence* de type *Absence* est créé à partir de la ligne sélectionnée dans le DataGridView.

Les champs date sont ensuite initialisés avec les valeurs de l'absence choisie dans le DataGridView.



Un clic sur le bouton *enregistrer* déclenche la méthode associée :

Formulaire FrmAbsence

```
private void btnEnregistrerAbsence_Click(object sender, EventArgs e)
```

Dans cette méthode, pour les tests qui vont être énumérés ci-dessous, si ils ne remplissent pas la condition, un message d'information est à chaque foi affiché dans une MessageBox.

Un premier test commence par vérifier si un élément est bien sélectionné dans la ComboBox des motifs (les DateTimePicker ne peuvent en effet pas être "vide").

Vient ensuite un deuxième test qui vérifie que la date de départ est bien antérieure à la date de fin de l'absence à ajouter.

Un troisième test vérifie que la méthode *ControleAbsenceSimultanées* renvoie bien le booléen *false*.

Formulaire FrmAbsence

```
private Boolean ControleAbsencesSimultanees()
```

Cette méthode renvoie *true* lorsque l'intervalle entre les dates de l'absence à ajouter chevauche un intervalle de date d'une absence déjà enregistrée (cf F. Ajout d'une absence (cas d'utilisation 6) , p23)

L'étape suivant consiste en un test déterminant si nous sommes dans le cas d'une modification ou d'un ajout d'absence.

Dans le cas d'une modification, un nouvel objet *absence* de type *Absence* est créé à partir de la ligne selectionné dans le *DataGridView*, et ses paramètres sont valorisés avec les nouvelles valeurs *DateTimePicker* et de la *ComboBox.*

La méthode *UpdateAbsence* du contrôleur est appelée avec l'objet *absence* (qui vient d'être créé), l'objet *personnel* (concerné par l'absence) et la propriété *dateDebut* (date de début initial de l'absence à modifier, avant modification, pour permettre d'identifier la bonne absence dans la base de donnée) en paramètre .

Classe FrmAbsenceController

```
public void UpdateAbsence(Absence absence, Personnel personnel, DateTime dateDebut)
```

Elle-même appelle la méthode *UpdateAbsence* de la classe *absenceAccess*.

Classe absenceAccess

```
public void UpdateAbsence(Absence absence, Personnel personnel, DateTime dateDebut )
```

Cette dernière prépare et formate la requête de mise à jour dans la base de données, et demande la mise à jour de cette dernière (à travers les classes *Access*, puis *bddManager*).

De retour dans la méthode *btnEnregistrerAbsence_Click*, la méthode *EnCoursModifAbsence* est appelée avec le booléen *false* en paramètre, pour valoriser le paramètre de classe *enCoursModifAbsence* à *false* (phase de modification terminée)

La méthode RemplirListeAbsence du formulaire *FrmAbsence* est ensuite de nouveau appelée avec comme paramètre, le personnel concerné par les absences, pour mettre à jour l'affichage des absences dans le DataGridView.

Enfin, la méthode *InitialisationActionAbsence* est appelée pour rendre inaccessible la zone de saisie, rendre accessible le DataGridView et réinitialiser les DateTimePicker et la ComboBox.

3.5. Création d'une documentation utilisateur en vidéo

Une documentation utilisateur de l'application sous forme de vidéo a été crée à l'aide des logiciels open-source suivants :

- OBS Studio : capture vidéo,
- Audacity : enregistrement audio,
- OpenShot Video Editor : montage vidéo.

La vidéo décrit tous les cas d'utilisation, avec à chaque fois, une démonstration du scénario nominal, et des scenarii alternatifs.

3.6. Déploiement, rédaction du compte rendu et création de la page du portfolio

A. Déploiement

Un installateur de l'application est généré sous Visual Studio (à l'aide de l'extension Microsoft Visual Studio Installer Project), de manière à pouvoir installer le logiciel sur un ordinateur tiers.

Lors de cette étape, il est possible :

- d'ajouter une icône pour les différents raccourcis qui seront créés,
- de définir l'identité de l'auteur de l'application
- de définir le nom qui sera utilisé pour créer le dossier lors de l'installation.

Un dossier *Setup* est ensuite généré dans le dossier du projet, contenant, entre autre l'installeur *.msi* (installateur indépendant contenant tous les fichiers nécessaires à l'installation et l'éxécution de l'application, hormis la base de données).

Le script complet de la base de données exploitée par l'application est généré sous *phpMyAdmin*. Les instructions pour la création du compte utilisateur sont rajouter manuellement.

```SQL
CREATE USER 'user_mediatek86'@'%' IDENTIFIED BY 'user_mediatek86_pwd';
GRANT ALL PRIVILEGES ON `mediatek86`.* TO 'mediatek86'@'%';
```

B. Rédaction du compte-rendu

Ce compte-rendu d'activité, détaillant tous les étapes du projet, de la préparation de l'environnement de travail, jusqu'à la publication de la page regroupant les informations du projet du le portfolio, a été conçu avec LibreOffice Writer.

Il reprend et détaille toutes les étapes suivies pour la réalisation de se projet.

C. Création de la page sur le portfolio

Une publication dédiée au projet à été ajouter sur le portfolio, dans la catégorie *Projets*, disponible en suivant le lien suivant : https://damienverger.com

Cette publication contient un les liens suivant :

- dépôt GitHub du projet,
- liens des différents document au format *.pdf* (contexte, cas d'utilisation, compte-rendu),
- lien du script de la base de données,
- lien vers la page HTML de la maquette réalisée sous Pencil,
- lien vers la page HTML de la documentation,
- lien de téléchargement de l'installateur,
- la vidéo de démonstration de l'application.

4. Bilan

Cet atelier a permis de gérer un projet dans sa globalité et en autonomie, de l'installation des outils nécessaires à la conception et la réalisation de l'application, jusqu'à l'alimentation du portfolio avec un compte-rendu d'activité résumant toutes les étapes de conception et de réalisation du projet.

Il a permis de concevoir une application fonctionnelle dans un langage défini, tant au niveau du design que de l'organisation du code. Cette application devait également exploiter une base de données et répondre à un certains nombres de cas d'utilisation.

Une documentation technique à partir des commentaires normalisées ainsi qu'une documentation utilisateur sous format vidéo ont été réalisés.