Atelier professionnel n°2

Compte rendu de l'activité

FRANCART Jérémy
BTS SIO PREMIERE ANNEE

Sommaire :

Contexte de l'atelier

Cet atelier prend place au sein de l'entreprise InfoTech Services 86 dans laquelle nous travaillons en tant que développeur junior. Cette entreprise a remporté des appels d'offres pour effectuer de multiples interventions pour le réseau de MediaTek86. MediaTek86 est un réseau qui gère les médiathèques de la Vienne, et qui a entre autres pour rôle développer la médiathèque numérique pour l'ensemble des médiathèques du département. C'est dans ce dernier but que MediaTek86 a fait recours aux services de InfoTech Services 86. Parmi ces interventions, nous avons été confiés la création d'une application en qui va permettre de gérer le personnel de chaque médiathèque.

Mission globale

Notre mission consiste à délivrer une application de bureau en C# monoposte fonctionnelle et répondant aux attentes de MediaTek86 : gérer le personnel de chaque médiathèque, leur affectation à un service et leurs absences depuis leur base de données des employés. L'application devra avoir une interface graphique facilitant les manipulations de l'utilisateur.

Nous aurons donc besoin de gérer de multiples aspects qui vont être divisés en plusieurs étapes :

Étape 1 : Préparation de l'environnement de travail, de l'IDE (environnement de développement intégré), de l'application de bureau et de la base de données.

Étape 2 : Conception et codage des interfaces selon le modèle MVC, création du dépôt de l'application.

Étape 3 : Codage du modèle et des outils de connexion à la base de données, gestion de la documentation technique.

Étape 4 : Codage des fonctionnalités de l'application à l'aide du dossier documentaire fourni.

Étape 5 : Ajouts mineurs, correction des problèmes restants, gestion du déploiement de l'application.

Étape 6 : Création de la documentation utilisateur, amélioration de la documentation technique.

Étape 7 : Intégration du projet au portfolio et rédaction du compte rendu.

Réalisation de la mission

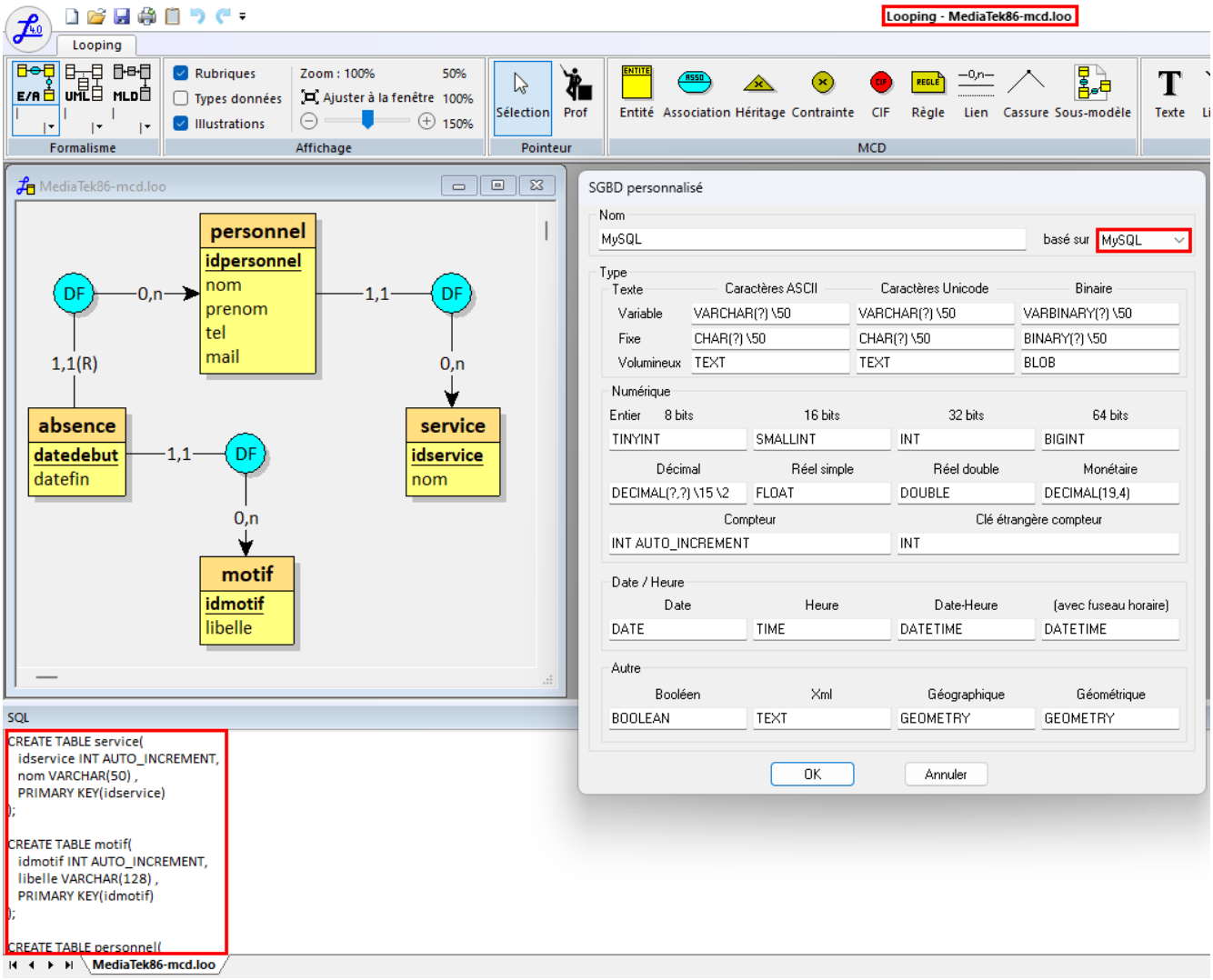
Étape 1 : Préparation de l'environnement de travail, de l'IDE et de la base de données

Le but de cette étape est de préparer et configurer les outils nécessaires à la bonne réalisation de l'application.

Les outils utilisés ici sont :

- Looping pour l'exploitation du modèle conceptuel des données (MCD) fourni
- Wampserver pour sa gestion des systèmes de bases de données MySQL / MariaDB
- Les différents IDEs utilisés :
  - Rider pour le codage en C#
  - DataGrip pour la manipulation et les tests facilitée sur les bases de données
  - Visual Studio pour la gestion de spécificités comme la création des paramètres d'application et la création de l'installateur pour le déploiement

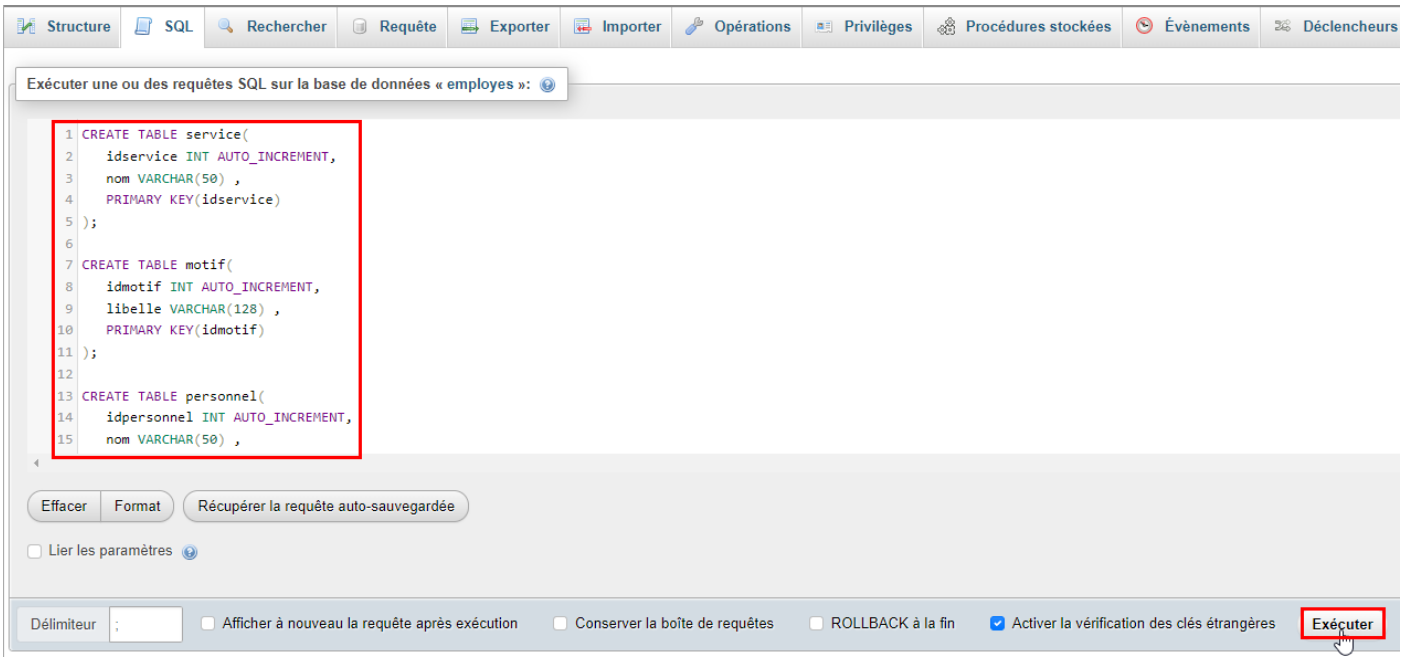
Nous pouvons récupérer le script de création de la base de données à partir du fichier MCD Looping de cette manière après avoir activé la fenêtre « Script SQL » dans le menu ruban en haut à droite :



Nous pouvons maintenant enregistrer le texte contenu dans la fenêtre SQL dans un fichier en .sql.

Depuis l'interface phpMyAdmin du serveur WAMP, nous nous connectons et créons une nouvelle base de données en vérifiant le bon encodage et en validant la création (nous l'appellerons la base « employés »).

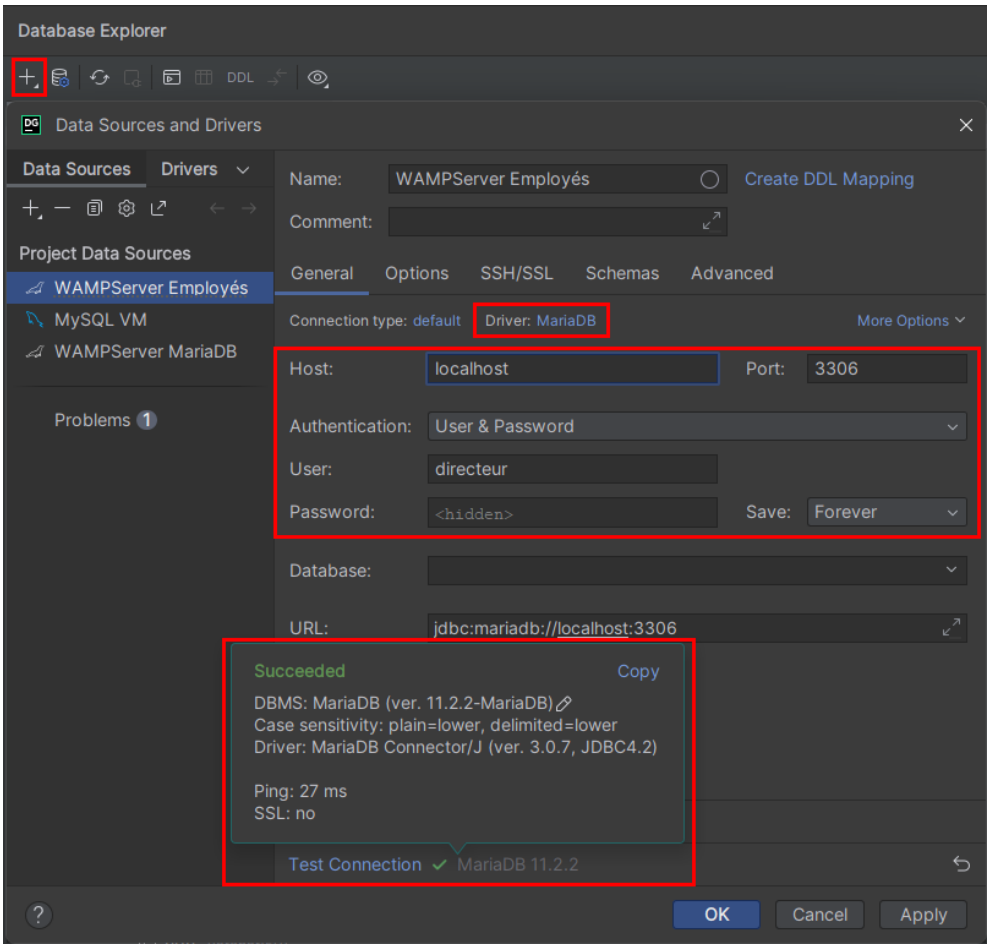
Ensuite, nous pouvons exécuter le script SQL que nous avons récupéré précédemment en allant dans la fenêtre SQL de phpMyAdmin.



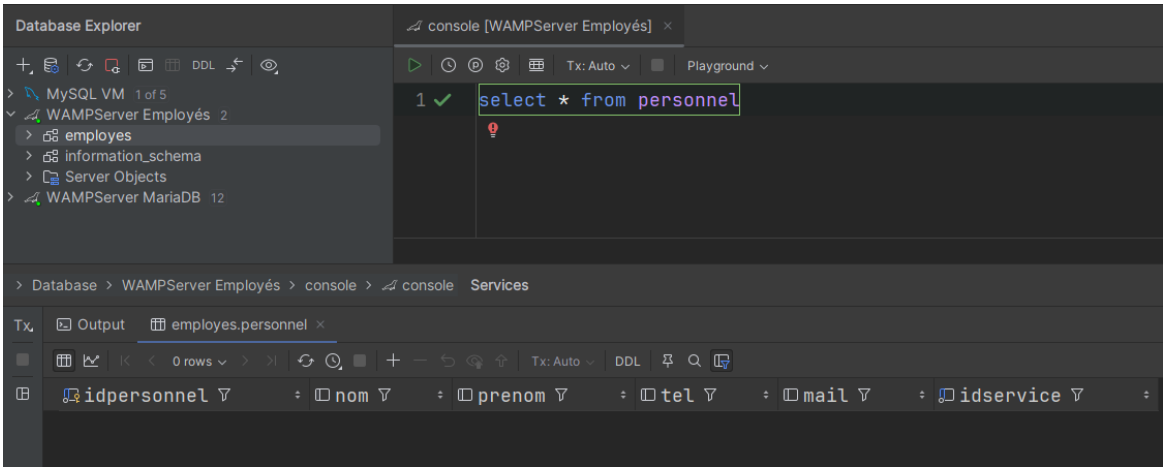
Toujours dans la fenêtre SQL, nous allons exécuter ces requêtes SQL afin de créer l'utilisateur qui aura seulement les privilèges requis afin d'exploiter la base de données employés, on lui change aussi son mot de passe en même temps.

```
CREATE USER IF NOT EXISTS 'directeur'@'%';
SET PASSWORD FOR 'directeur'@'%' = PASSWORD('P@$$word1');
GRANT UPDATE, SELECT, INSERT, DELETE, DROP ON absence TO directeur;
GRANT UPDATE, SELECT, INSERT, DELETE, DROP ON motif TO directeur;
GRANT UPDATE, SELECT, INSERT, DELETE, DROP ON personnel TO directeur;
GRANT UPDATE, SELECT, INSERT, DELETE, DROP ON service TO directeur;
```

Avant de continuer, nous allons configurer DataGrip pour nos futures interactions avec la base de données, il faut créer un projet pour pouvoir accéder aux outils de base de données donc nous créons un projet vide puis, nous pouvons ajouter notre SGBDR MariaDB dans la liste des sources :



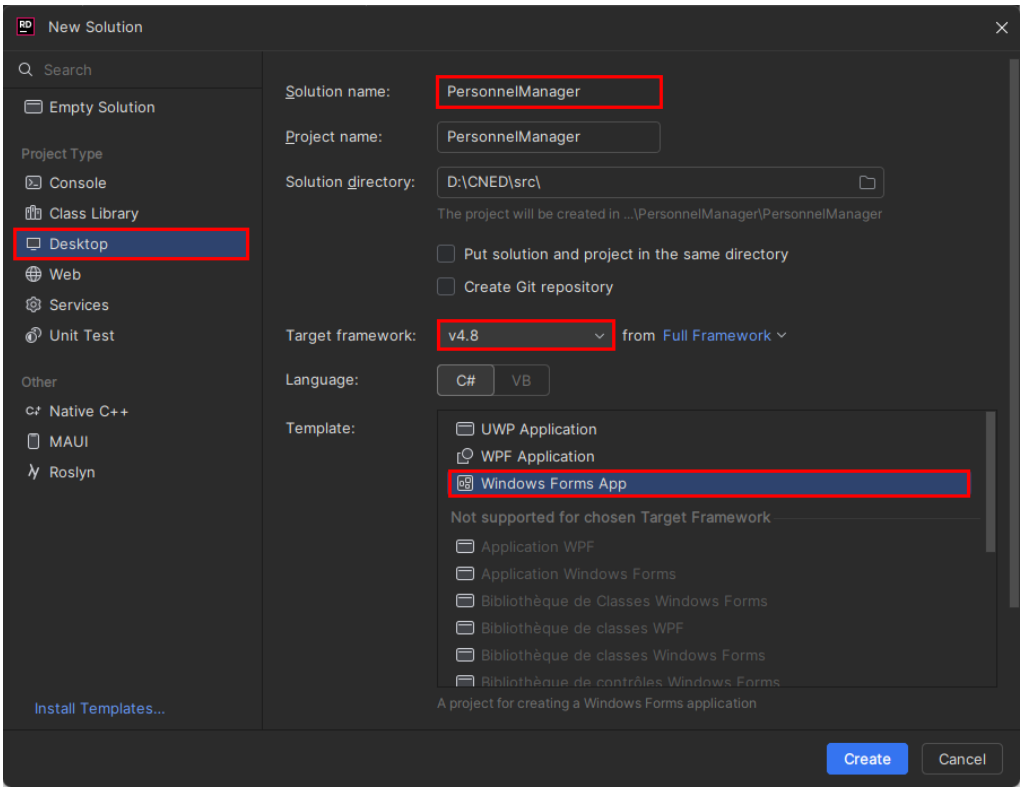
Nous avons maintenant accès aux outils nous permettant d'interagir avec le SGBDR



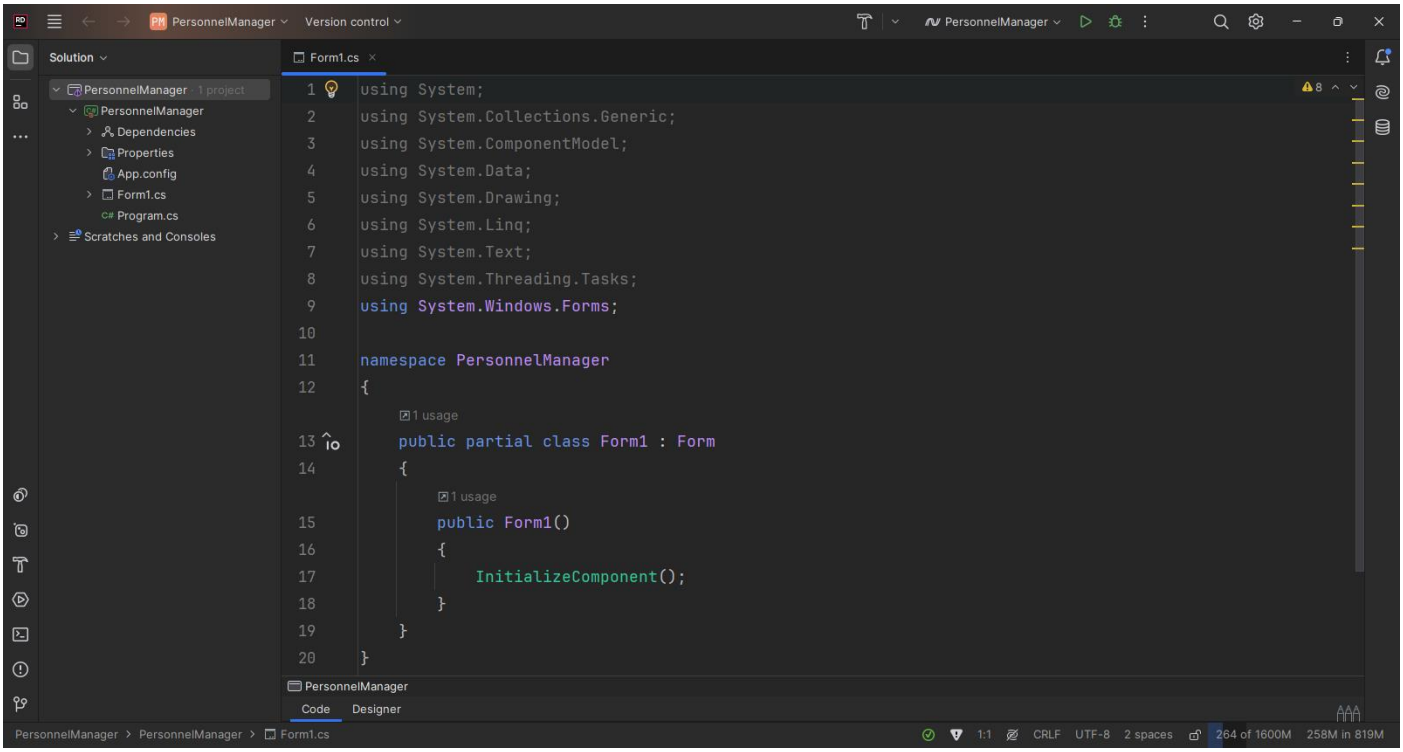
Maintenant, il faut qu'on alimente les tables dans la base de données, les lignes des tables motif et service nous sont données dans les consignes donc nous pouvons juste les insérer avec des requêtes de type INSERT INTO (table) VALUES(lignes).

Pour les tables personnel et absence, on utilisera le site generatedata.com qui permet de générer des données réalistes selon un schéma donné, le site nous donne ensuite un script SQL que nous pouvons exécuter afin d'insérer les personnels et leurs absences.

Enfin, nous allons maintenant créer le projet pour notre application de bureau, pour ce faire, nous nous rendons sous notre IDE Rider, on clique sur « New Solution » et nous créons un projet de type « Desktop » en utilisant .NET Framework version 4.8 et en choisissant l'option Windows Forms App.



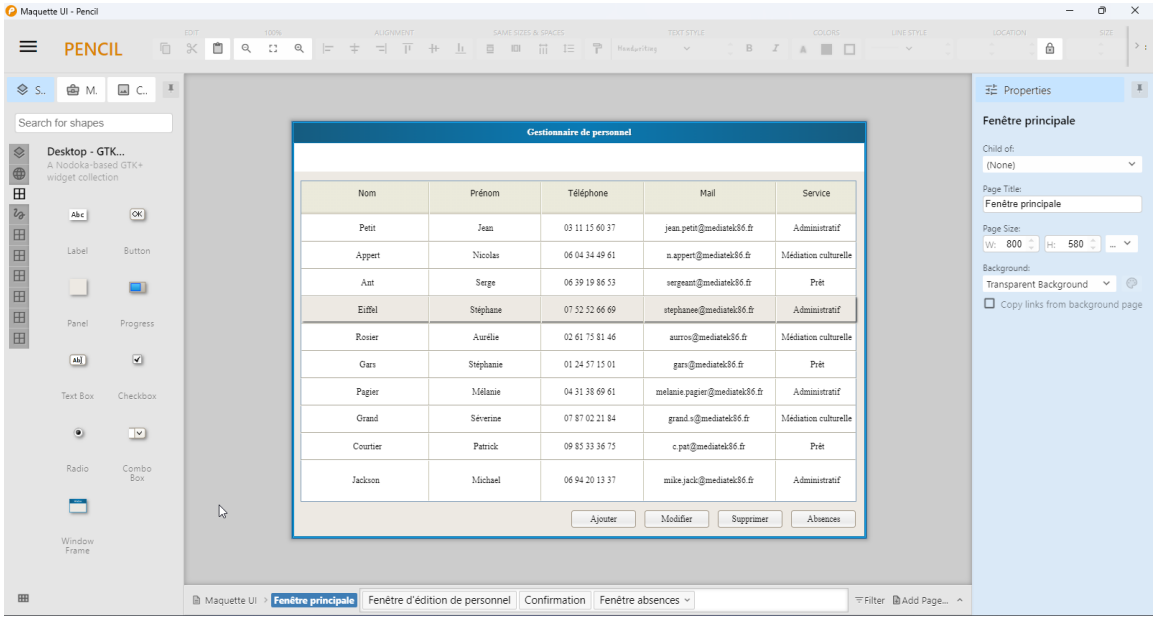
On obtient maintenant un projet vide où l'on va pouvoir créer notre application, Visual Studio nous permet aussi d'ouvrir ce projet avec le fichier en .sln.



Notre environnement est maintenant prêt, nous avons notre serveur de base de données, nos IDEs et notre projet de configuré.

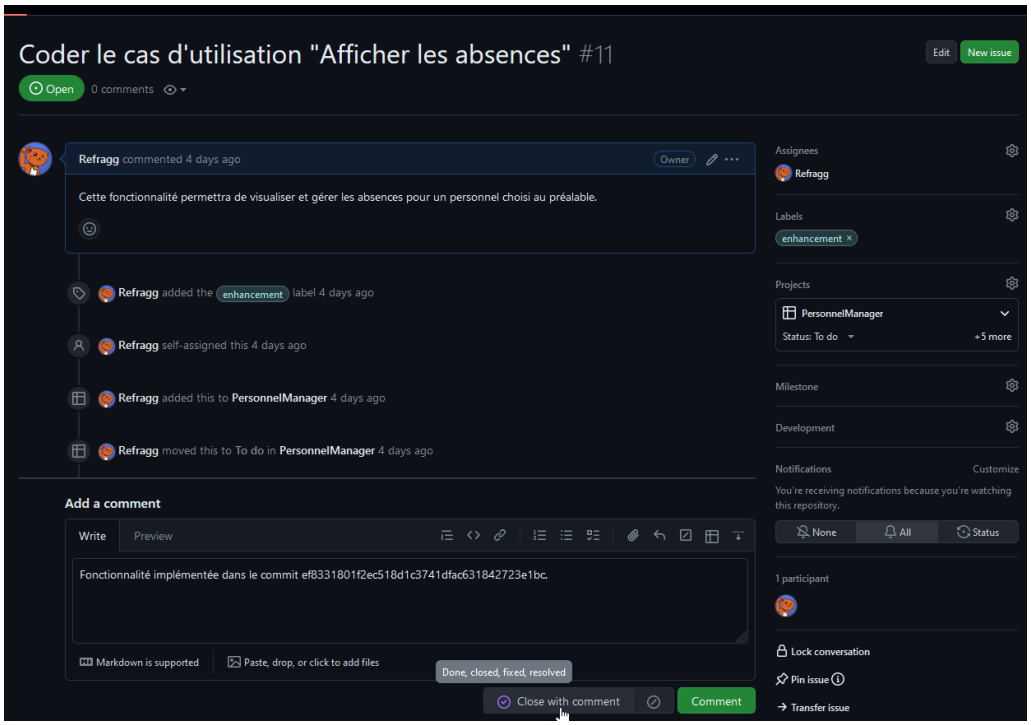
Étape 2 : Conception et codage des interfaces selon le modèle MVC, création du dépôt de l'application.

Dans cette étape, nous allons commencer par dessiner les interfaces avec un outil de maquettage appelé Pencil. Pencil est un outil permettant de dessiner des interfaces simples et de les lier entre elles afin de les faire valider avant de les coder.

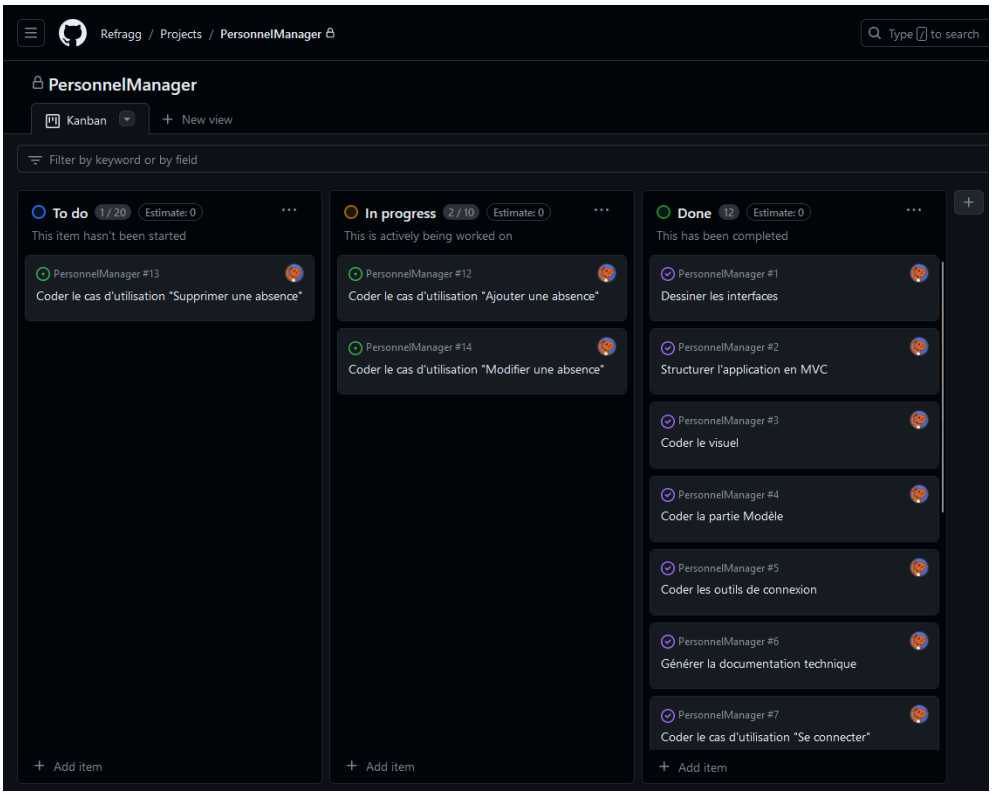


Une fois la maquette réalisée sous Pencil, il est même possible d'en exporter une version interactive en HTML pour avoir un résultat plus parlant.

Nous allons maintenant créer le dépôt pour l'application et y déposer notre premier fichier qui sera le fichier maquette de Pencil. Sur GitHub, on crée un « repository » puis, on peut ensuite créer un projet de type Kanban automatisé pour notre dépôt de telle manière que les « issues » de GitHub ajoute et déplace automatiquement des nouvelles cartes dans le Kanban. Voici ce à quoi ca ressemble :



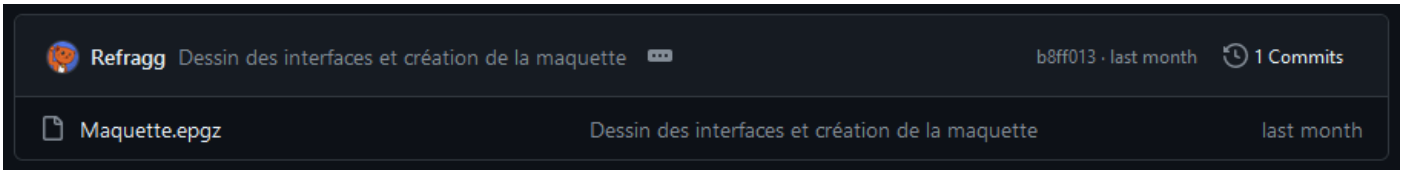
Lors de la création d'une « issue » GitHub, nous pouvons ajouter à notre projet, il sera mis automatiquement dans « To do » par défaut. Quand on clôturera cette demande, la carte sera automatiquement déplacée dans la colonne « Done ». Ce système permet une meilleure organisation des tâches à faire, en cours et finies.



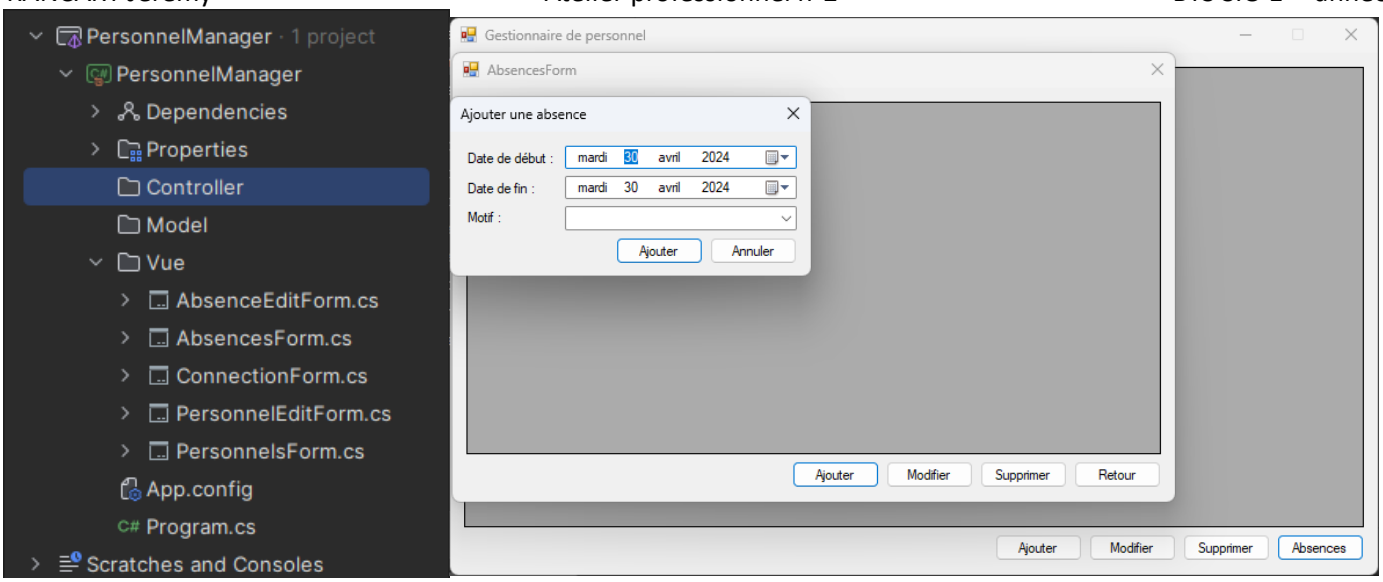
Nous créons donc toutes les issues qui représentent les tâches à effectuer lors du développement de l'application.

Nous allons ensuite utiliser soit notre IDE, soit un autre outil pour pousser nos changements vers notre dépôt distant. J'ai utilisé GitHub Desktop durant la réalisation de ce projet car ce logiciel est bien intégré avec GitHub et facile d'utilisation.

Nous avons maintenant notre premier commit de poussé sur notre dépôt distant GitHub.

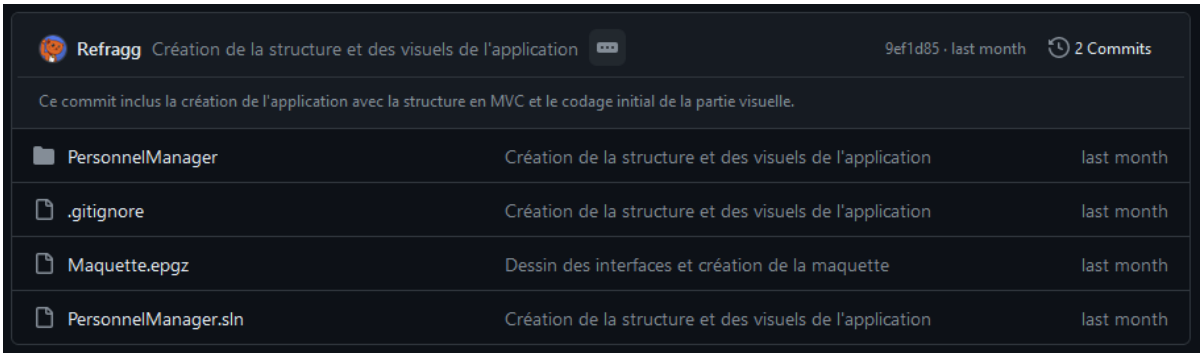


Maintenant que la maquette a été réalisée, validée et sauvegardée sur le dépôt, nous pouvons coder le visuel de l'application depuis notre IDE Rider et grâce à son outil de design d'interface graphique. Il faut respecter le modèle MVC donc nous créons en même temps les dossiers des 3 composantes : Modèle, Vue et Contrôleur. On se retrouve pour le moment avec cette structure de projet et cette interface.



Pour le moment, on a juste une logique pour ouvrir les fenêtres correspondantes aux boutons, nous ajouterons le reste de la logique plus tard.

Une fois cela fait, nous pouvons pousser tout les fichiers sources de l'application sur notre dépôt distant avec un nouveau commit :



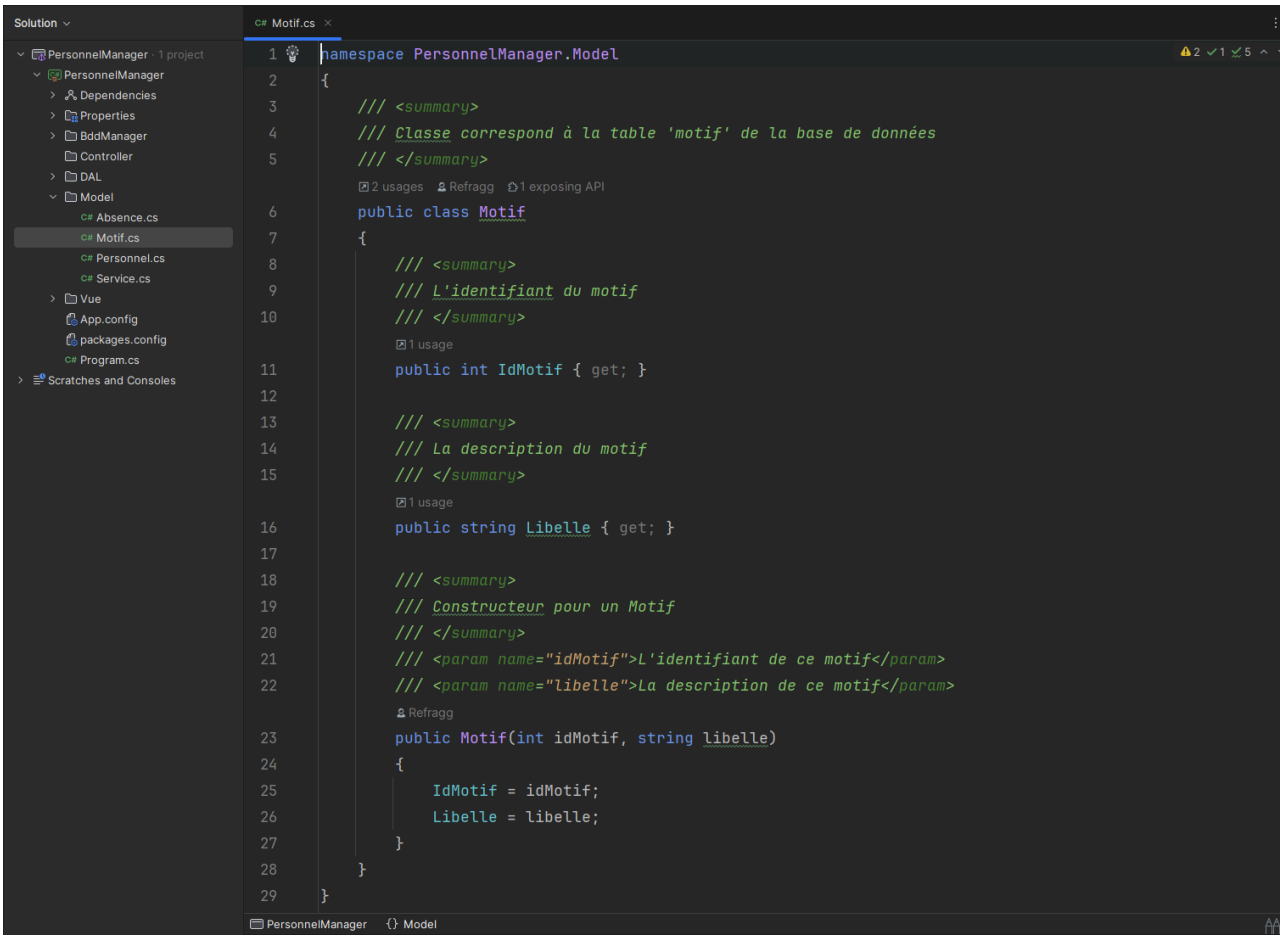
La base de l'application et ses interfaces étant maintenant réalisée et sauvegardée sur GitHub, nous pouvons nous concentrer sur les interactions avec la base de données.

Étape 3 : Codage du modèle et des outils de connexion à la base de données, gestion de la documentation technique

Avant de récupérer les informations dans la base de données, nous allons d'abord créer les classes modèles qui vont contenir les informations que l'on récupèrera ensuite dans la base de données, ces classes modèles sont juste là pour retenir les informations qu'on leur donne, elles ne savent pas que les données viennent d'une base de données.

Nous ajoutons aussi en même temps les commentaires normalisés qui vont servir à générer la documentation technique des classes plus tard.

Un exemple d'une classe modèle :



Maintenant que nous avons les classes modèles, on va s'occuper des interactions avec la base de données, on récupère la classe BddManager que l'on à codé dans une autre application et on la place dans le dossier BddManager pour bien organiser nos fichiers sources. J'ai légèrement modifié la classe pour utiliser le package NuGet MySqlConnector au lieu d'utiliser MySql.Data, la raison de ce changement est simple, MariaDB semblait ne pas fonctionner sous MySql.Data. Heureusement, les changements à faire étaient très simple puisque MySqlConnector utilise quasiment les mêmes classes et méthodes.

Cette classe BddManager est une simple interface entre la base de données et l'application, elle sert seulement à exécuter les requêtes SQL qu'on lui donne, nous allons donc devoir définir ces requêtes et mapper nos objets à travers une autre interface, la classe Access dans le dossier DAL.

Cette classe va récupérer la chaîne de connexion à la base de données et va obtenir l'instance de la classe BddManager, on pourra ensuite créer d'autres classes qui utilisent la classe Access afin d'obtenir les informations voulues spécifiques aux modèles.

J'ai opté ici au final d'avoir une méthode statique ValidationIdentifiants dans la classe Access qui prend le login et le mot de passe et c'est elle qui va initialiser le BddManager si les identifiants sont valides. L'avantage d'utiliser ce procédé est que l'on n'a pas déjà besoin d'une connexion à la base de données avant de pouvoir vérifier les identifiants, c'est le résultat de la connexion en elle-même qui va pouvoir nous dire si les identifiants sont valides ou pas.

Nous appellerons ensuite cette méthode ValidationIdentifiants depuis le contrôleur de la fenêtre de connexion, si elle retourne false, on sait que les identifiants sont invalides, sinon, on a une connexion valide à la base de données et on pourra continuer.

Voici une partie de la méthode ValidationIdentifiants :

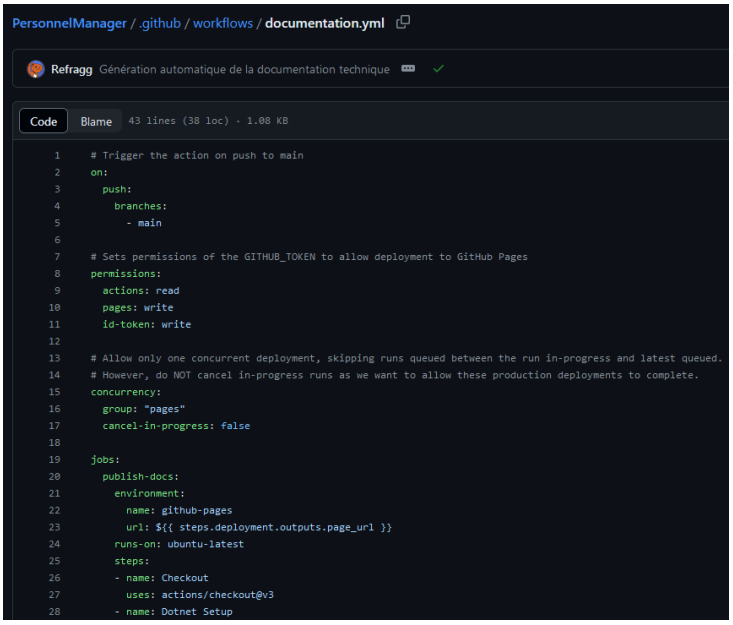
```
37          // Utilisation de la chaîne de connexion comme base et ajout des paramètres 'user id' et 'password' à celle-ci
38          MySqlConnectionStringBuilder connectionStringBuilder = new MySqlConnectionStringBuilder(connectionString);
39
40          connectionStringBuilder.UserID = login;
41          connectionStringBuilder.Password = password;
42
43          try
44          {
45              Instance.Manager = BddManager.BddManager.GetInstance(connectionStringBuilder.ToString());
46          }
47          catch (MySqlException e)
48          {
49              if (e.ErrorCode == MySqlErrorCode.AccessDenied)
50                  return false;
51
52              throw;
53          }
54
55          return true;
```

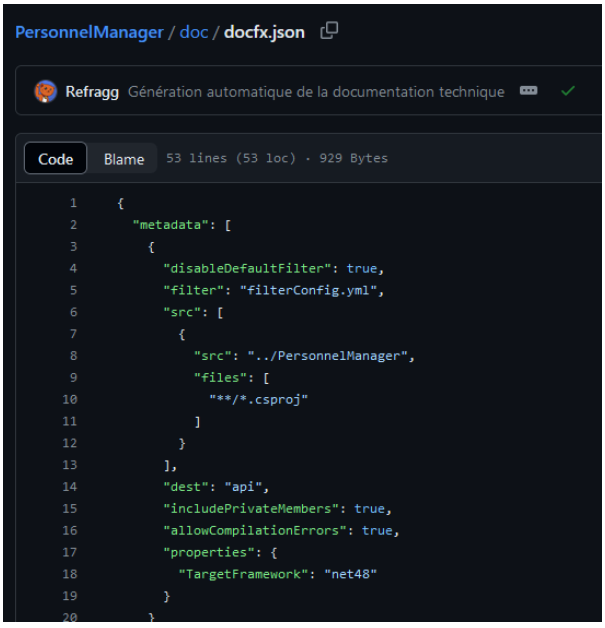
Dernier détail avant de passer à la gestion de la documentation technique, j'ai opté pour une gestion globale des erreurs graves dans l'application, de cette façon, toutes les erreurs non gérées seront interceptées au même endroit et l'on peut afficher un message d'erreur avant de quitter l'application. Cela rend la gestion des erreurs au sein de l'application plus facile.

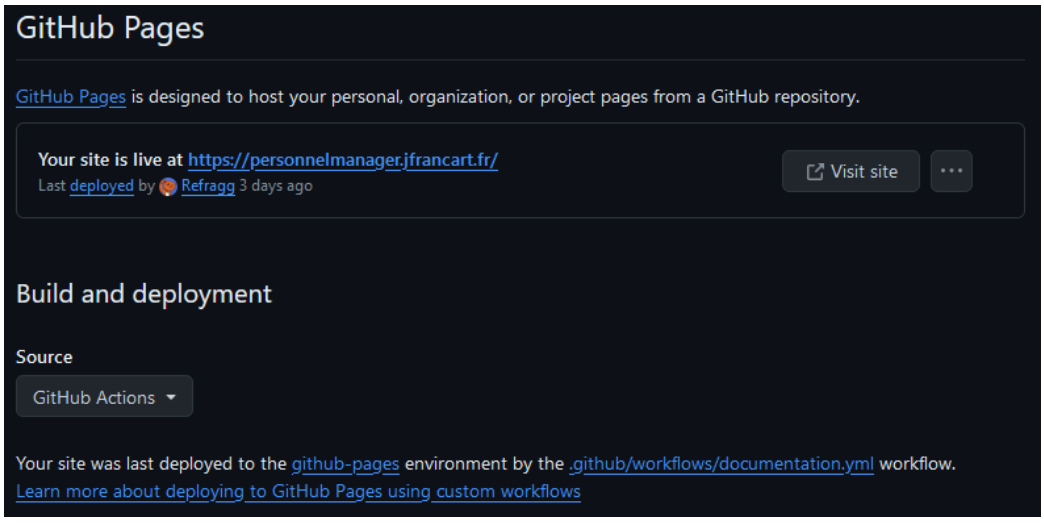
```
Refragg
static void Main()
{
    AppDomain.CurrentDomain.UnhandledException += (o:object, e:UnhandledExceptionEventArgs) => UnhandledException((Exception)e.ExceptionObject);
    Application.ThreadException += (o:object, e:ThreadExceptionEventArgs) => UnhandledException(e.Exception);
    Application.SetUnhandledExceptionMode(UnhandledExceptionMode.CatchException);
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new ConnectionForm());
}

/// <summary>
/// Cette méthode sera invoquée en cas d'erreur grave non gérée dans l'application, les utilisateurs recevront un message d'explication av
/// </summary>
/// <param name="e">L'exception qui a été générée</param>
2 usages  Refragg
private static void UnhandledException(Exception e)
{
    MessageBox.Show(
        text: $"Une erreur irrécupérable est arrivée lors de l'exécution de l'application. L'application va maintenant quitter\r\n\r\nErreur
        caption: "Erreur", MessageBoxButtons.OK, MessageBoxIcon.Error);
    Environment.Exit(1);
}
```

Passons maintenant à la documentation technique, j'ai commencé par rajouter des commentaires normalisés partout où il n'y en avait pas puis, j'ai utilisé l'outil docfx pour permettre la génération de la documentation technique. L'avantage de docfx est sa personnalisation avancée et simple en même temps, il est possible de changer le comportement de beaucoup d'aspects des pages générées de manière simple, il suffit juste de modifier soit le fichier docfx.json ou de modifier la template donnée par défaut. Docfx permet aussi de rédiger des pages annexes comme des instructions pour installer l'application par exemple. Enfin, le site docfx peut être déployé avec GitHub Pages de façon automatique ce qui rend l'actualisation du site très facile.

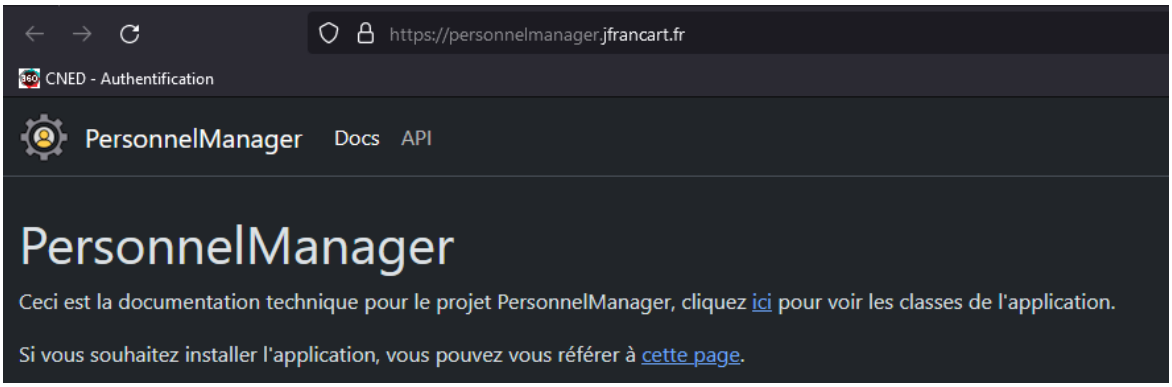
On initialise docfx en créant un dossier doc et en définissant un fichier docfx.json. Puis, on configure une « GitHub Action » et la « GitHub Pages » afin de rendre cela automatique.







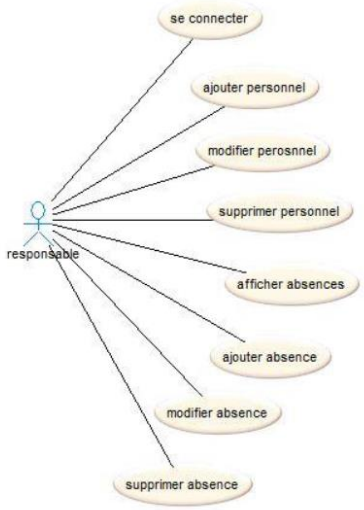
Après avoir configuré mon domaine personnalisé, nous avons la documentation technique disponible en ligne :



Étape 4 : Codage des fonctionnalités de l'application à l'aide du dossier documentaire fourni

Maintenant que nous avons les interfaces pour accéder à la base de données de prêtes, nous pouvons nous concentrer sur le codage des fonctionnalités de l'application. Un dossier documentaire contenant le descriptif des cas d'utilisation était fourni, cela a permis de développer les fonctionnalités comme elles étaient attendues.

Diagramme de cas d'utilisation



Remarque : tous les cas d'utilisation ne sont accessibles qu'après s'être connecté.

Descriptif des cas d'utilisation

Remarque importante :
Il n'est pas demandé de faire forcément une fenêtre par cas d'utilisation.
Le développeur a le choix de l'organisation des fenêtres (en nombre et en contenu) dans le respect des besoins au niveau des fonctionnalités et avec l'objectif de faciliter les manipulations.

Cas d'utilisation n°1	Se connecter
Acteur	Responsable du personnel
Événement déclencheur	aucun
Intérêt	Accéder aux fonctionnalités de gestion du personnel
Précondition	aucune
Scénario nominal	1. Le système affiche les zones de saisie du login et du mot de passe. 2. Le responsable saisit son login, son mot de passe et valide. 3. Le système affiche la liste des personnels et des boutons de commande pour pouvoir ajouter un nouveau personnel, supprimer un personnel sélectionné dans la liste, modifier un personnel sélectionné dans la liste, accéder à la gestion des absences d'un personnel sélectionné dans la liste.
Scénario alternatif	3a. Les champs ne sont pas tous remplis : retour au point 2. 3b. Le login et/ou le mot de passe ne correspondent pas à ceux enregistrés dans la base de données, le système affiche une alerte : retour au point 2.

Il était conseillé de ne pas utiliser une fenêtre par cas d'utilisation mais j'ai trouvé que d'avoir seulement une ou deux fenêtres rendait l'interface difficile à lire et comprendre, c'est pour ça que j'ai utilisé cette approche d'une fenêtre par cas d'utilisation, cela facilite en même temps la gestion de l'interface.

On commence par la connexion à l'application, dans le respect du MVC, nous allons créer un contrôleur qui va gérer les requêtes de la fenêtre de connexion, on peut ajouter à ce contrôleur la méthode ValidationIdentifiants qui va transférer la requête à la classe Access.

```
/// <summary>
/// Controlleur gérant les requêtes de la classe ConnectionForm
/// </summary>
2 usages  Refragg
public class ConnectionFormController
{
    /// <summary>
    /// Méthode permettant de valider les identifiants d'un utilisateur de l'application
    /// </summary>
    /// <param name="login">Le nom d'utilisateur de la personne à vérifier</param>
    /// <param name="password">Le mot de passe de l'utilisateur à vérifier</param>
    /// <returns>'true' si les identifiants sont valides, 'false' sinon</returns>
    1 usage  Refragg
    public bool ValidationIdentifiants(string login, string password) => Access.ValidationIdentifiants(login, password);
}
```

Nous pouvons maintenant utiliser cette méthode dans notre fenêtre de connexion quand on clique sur le bouton « Se connecter ». Si les identifiants sont invalides, on affiche un message d'erreur. Sinon, on ouvre la fenêtre principale de l'application et on cache la fenêtre de connexion.

```
/// <summary>                                                                    3 ✓1
/// Méthode événementielle quand on clique sur le bouton 'Se connecter'
/// </summary>
/// <param name="sender">Paramètre inutilisé</param>
/// <param name="e">Paramètre inutilisé</param>
1 usage  Refragg
private void btnConnection_Click(object sender, EventArgs e)
{
    if (!_controller.ValidationIdentifiants(login: txtUsername.Text, txtPassword.Text))
    {
        MessageBox.Show(text: "Les identifiants donnés sont invalides, veuillez les revérifier avant de recommencer",
            caption: "Identifiants invalides", MessageBoxButtons.OK, MessageBoxIcon.Warning);
        txtPassword.Focus();
        return;
    }

    var mainForm = new PersonnelsForm();
    mainForm.Closed += (o:object, ev:EventArgs) => Close();
    mainForm.Show();
    Hide();
}
```

Nous ajoutons enfin une vérification pour que l'utilisateur ne puisse pas cliquer sur se connecter avant d'avoir rempli tous les champs.

```
/// <summary>
/// Méthode événementielle déclenchée quand on édite du texte dans la case 'Nom d'utilisateur :'
/// </summary>
/// <param name="sender">Paramètre inutilisé</param>
/// <param name="e">Paramètre inutilisé</param>
1 usage  Refragg
private void txtUsername_TextChanged(object sender, EventArgs e) => RefreshConnectionButtonStatus();

/// <summary>
/// Méthode événementielle déclenchée quand on édite du texte dans la case 'Mot de passe :'
/// </summary>
/// <param name="sender">Paramètre inutilisé</param>
/// <param name="e">Paramètre inutilisé</param>
1 usage  Refragg
private void txtPassword_TextChanged(object sender, EventArgs e) => RefreshConnectionButtonStatus();

/// <summary>
/// Méthode qui met à jour l'état d'activation du bouton 'Se connecter' en fonction de l'état de remplissage des cases
/// </summary>
2 usages  Refragg
private void RefreshConnectionButtonStatus()
{
    btnConnection.Enabled = !string.IsNullOrWhiteSpace(txtUsername.Text) && !string.IsNullOrEmpty(txtPassword.Text);
}
```

Nous allons maintenant nous occuper de l'affichage des personnels dans la fenêtre principale. Nous devons donc récupérer les informations des personnels depuis la base de données, nous allons donc créer nos premières classes spécifiques d'accès aux informations dans les tables de la base de données. On commence par la classe d'accès aux services car la récupération des services est nécessaire avant de récupérer les personnels. On crée donc la classe ServiceAccess qui va utiliser la classe Access pour communiquer avec la base de données.

```
public static class ServiceAccess
{
    /// <summary>
    /// L'instance unique pour accéder à la base de données
    /// </summary>
    private static Access _access = Access.Instance;

    /// <summary>
    /// Méthode permettant de récupérer les services depuis la base de données
    /// </summary>
    /// <returns>La liste complète des services</returns>
    /// <exception cref="MySqlConnector.MySqlException">La requête vers la base de données a échoué</exception>
    1 usage  Refragg
    public static List<Service> GetServices()
    {
        var result:List<object[]> = _access.Manager.ReqSelect(stringQuery: "select * from service");

        var services = new List<Service>();
        foreach (object[] service in result)
            services.Add(item: new Service((int)service[0], (string)service[1]));

        return services;
    }
```

C'est ici qu'on définit notre requête SQL, on récupère ensuite une liste d'objets qui représente nos lignes et nos colonnes, on parcourt toutes les lignes et on crée les services en castant les champs récupérés.

On fait aussi une version pour récupérer un service selon son identifiant que l'on obtiendra en même temps que le personnel.

```
/// <summary>
/// Méthode permettant de récupérer un service précis depuis son ID dans la base de données
/// </summary>
/// <param name="idService">L'ID du service à récupérer</param>
/// <returns>Le service en question</returns>
/// <exception cref="MySqlConnector.MySqlException">La requête vers la base de données a échoué</exception>
1 usage  Refragg
public static Service GetService(int idService)
{
    var result:List<object[]> = _access.Manager.ReqSelect(
        stringQuery:"select idservice, nom from service where idservice = @idservice",
        parameters:new Dictionary<string, object> { { "idservice", idService } });

    var rawService:object[] = result[0];

    return new Service((int)rawService[0], (string)rawService[1]);
}
```

Cette requête utilise la paramétrisation de l'identifiant, cela sera géré par la classe BddManager qui validera la saisie afin d'éviter les injections SQL.

On fait la même chose pour les personnels et on utilise la méthode GetService afin d'obtenir le service du personnel :

```
/// <summary>
/// Méthode permettant de récupérer les personnels depuis la base de données
/// </summary>
/// <returns>La liste complète des personnels</returns>
/// <exception cref="MySqlConnector.MySqlException">La requête vers la base de données a échoué</exception>
1 usage  Refragg
public static List<Personnel> GetPersonnels()
{
    var result:List<object[]> = _access.Manager.ReqSelect(
        stringQuery:"select idpersonnel, personnel.nom, prenom, tel, mail, idservice from personnel");

    var personnels = new List<Personnel>(result.Count);
    foreach (object[] ligne in result)
        personnels.Add(item:new Personnel(
            (int)ligne[0],
            nom:(string)ligne[1],
            prenom:(string)ligne[2],
            telephone:(string)ligne[3],
            mail:(string)ligne[4],
            ServiceAccess.GetService((int)ligne[5])));

    return personnels;
}
```

Nous pouvons maintenant ajouter et utiliser ces méthodes depuis le contrôleur de la fenêtre principale.

```
/// <summary>
/// Constructeur de la fenêtre
/// </summary>
1 usage  Refragg
public PersonnelsForm()
{
    InitializeComponent();
    RefreshPersonnels();
}

/// <summary>
/// Méthode qui rafraichit la liste des personnels
/// </summary>
4 usages  Refragg
public void RefreshPersonnels()
{
    lstPersonnels.DataSource = new BindingSource
    {
        DataSource = _controller.GetPersonnels() // List<Personnel>
            .OrderBy(x :Personnel => x.Nom)
            .ThenBy(x :Personnel => x.Prenom) // IOrderedEnumerable<Personnel>
    };
}
```

Nous pouvons maintenant nous concentrer sur l'addition et la modification d'un personnel, nous allons pour cela utiliser la fenêtre d'édition de personnel, on va passer en paramètre si l'on édite ou si l'on ajoute un personnel et l'on va modifier la fenêtre selon ce critère. Donc si l'on ajoute un personnel, on affiche juste la fenêtre. Sinon, on rempli les informations avec le personnel donné puis on affiche la fenêtre.

```
/// <summary>
/// Constructeur de la fenêtre
/// </summary>
/// <param name="personnel">Le personnel à éditer, s'il faut ajouter un personnel, utilisez null</param>
2 usages  Refragg
public PersonnelEditForm(Personnel personnel = null)
{
    InitializeComponent();
    RefreshServices();
    Personnel = personnel;

    if (Personnel is null)
    {
        Text = "Ajouter un personnel";
        btnConfirm.Text = "Ajouter";
        cbxService.SelectedIndex = 0;
        return;
    }

    Text = "Modifier un personnel";
    btnConfirm.Text = "Modifier";

    txtNom.Text = Personnel.Nom;
    txtPrenom.Text = Personnel.Prenom;
    txtTel.Text = Personnel.Telephone;
```

Du côté de la fenêtre principale, voila ce qu'il se passe :

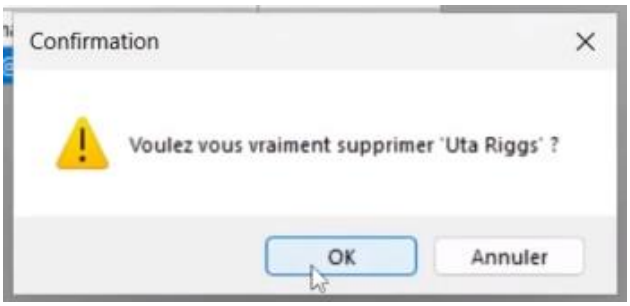
```
/// <summary>
/// Méthode qui gère l'addition d'un personnel
/// </summary>
1 usage  Refragg
private void AdditionPersonnel()
{
    var form = new PersonnelEditForm();
    if (form.ShowDialog() == DialogResult.OK)
    {
        _controller.AddPersonnel(form.Personnel);
        RefreshPersonnels();
    }
}
```

```
public Personnel Personnel { get; private set; }
 in class PersonnelManager.Vue.PersonnelEditForm

Le personnel en cours d'édition ou d'addition, à récupérer à la fin de l'ajout
```

```
/// <summary>
/// Méthode qui gère l'édition d'un personnel
/// </summary>
2 usages  Refragg
private void EditionPersonnel()
{
    if (lstPersonnels.SelectedRows.Count == 0)
    {
        MessageBox.Show(text: "Veuillez sélectionner une ligne");
        return;
    }

    Personnel personnel = (Personnel)((BindingSource)lstPersonnels.DataSource)[lstPersonnels.SelectedRows[0].Index];

    if (new PersonnelEditForm(personnel).ShowDialog() == DialogResult.OK)
    {
        _controller.UpdatePersonnel(personnel);
        RefreshPersonnels();
    }
}
```

Nous avons bien évidemment entre temps ajouté les méthodes nécessaires à PersonnelAccess et au contrôleur afin de pouvoir ajouter et mettre à jour les personnels.

Enfin, on gère la suppression d'un personnel de la même manière après avoir affiché une confirmation :



Enfin pour les absences, on affiche une fenêtre très similaire à la fenêtre principale, elle a aussi un bouton ajouter, modifier et supprimer…

Étape 5 : Ajouts mineurs, correction des problèmes restants, gestion du déploiement de l'application

Pour rendre la navigation plus facile et rapide, j'ai décidé d'ajouter des interactions spécifiques au clavier, on peut donc maintenant appuyer sur une touche pour ajouter, supprimer, modifier ou voir les absences d'un personnel.

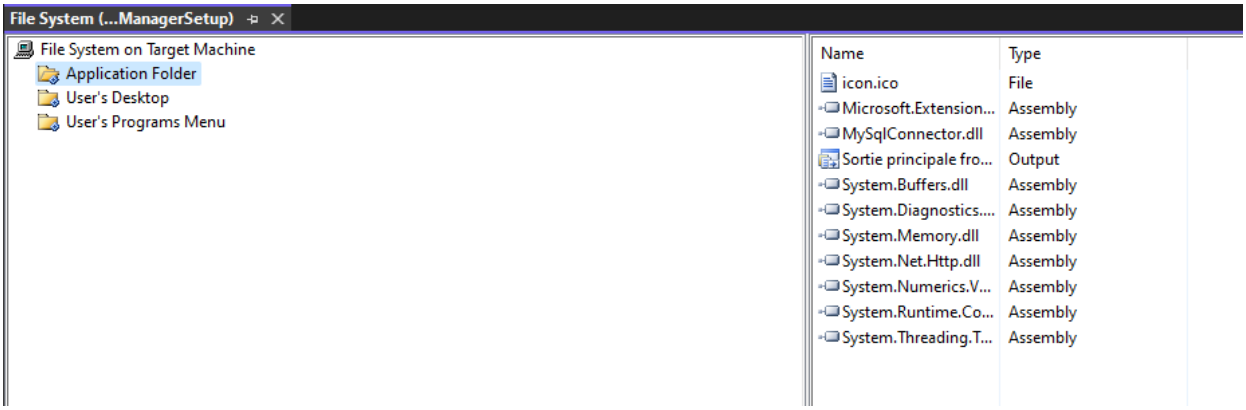
```
/// <summary>
/// Méthode événementielle qui gère la pression d'une touche à l'intérieur de la liste des personnels
/// </summary>
/// <param name="sender">Paramètre inutilisé</param>
/// <param name="e">Paramètre inutilisé</param>
1 usage  Refragg
private void lstPersonnels_KeyDown(object sender, KeyEventArgs e)
{
    switch (e.KeyCode)
    {
        case Keys.Enter:
        case Keys.E:
            e.Handled = true;
            EditionPersonnel();
            break;
        case Keys.Delete:
        case Keys.Subtract:
            e.Handled = true;
            SuppressionPersonnel();
            break;
        case Keys.Tab:
            e.Handled = true;
            btnAdd.Focus();
            break;
```

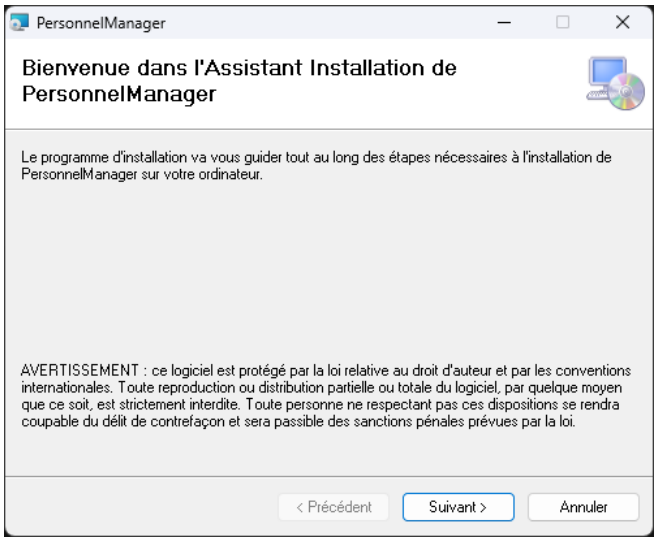
J'ai aussi ajouté le fait de pouvoir double cliquer sur un champ et d'être pris directement sur l'édition de ce champ dans la fenêtre d'édition. Pour cela, on regarde le champ sélectionné dans la liste puis on le passe à la fenêtre d'édition

```
PersonnelEditForm.PersonnelField selectedField = PersonnelEditForm.PersonnelField.None;

switch (lstPersonnels.CurrentCell.ColumnIndex)
{
    case 1:
        selectedField = PersonnelEditForm.PersonnelField.Nom;
        break;
    case 2:
        selectedField = PersonnelEditForm.PersonnelField.Prenom;
        break;
    case 3:
        selectedField = PersonnelEditForm.PersonnelField.Tel;
        break;
    case 4:
        selectedField = PersonnelEditForm.PersonnelField.Mail;
        break;
    case 5:
        selectedField = PersonnelEditForm.PersonnelField.Service;
        break;
}

Personnel personnel = (Personnel)((BindingSource)lstPersonnels.DataSource)[lstPersonnels.SelectedRows[0].Index];

if (new PersonnelEditForm(personnel, selectedField).ShowDialog() == DialogResult.OK)
```

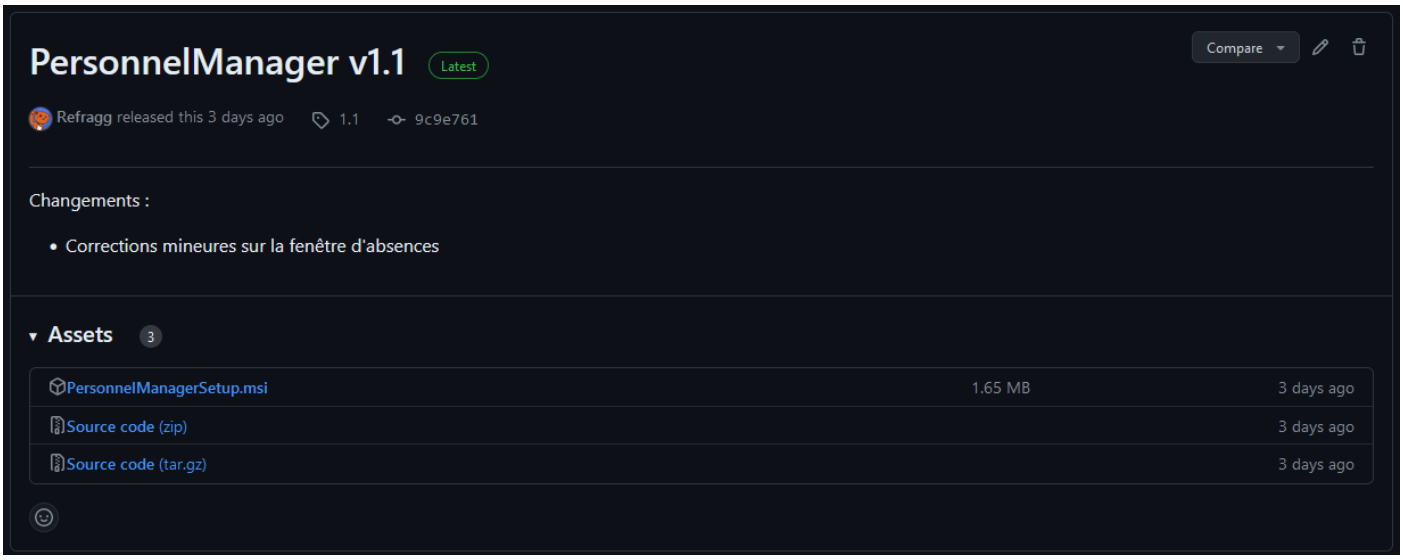
Enfin, il y avait un problème où l'application crashait quand un personnel n'avait aucune absence et, double cliquer dans la fenêtre d'absence ne faisait rien alors que ça devrait ouvrir la fenêtre d'édition d'absences.

Pour le déploiement, j'ai créé un deuxième projet dans la solution, ce projet est de type installateur automatisé, on peut définir les fichiers installés, les raccourcis… Puis, on génère un fichier installateur en .msi qui pourra être exécuté sur n'importe quel ordinateur. Ce type de projet n'étant pas pris en charge sur Rider, j'ai utilisé Visual Studio pour ceci.





On peut maintenant créer une « Release » sur GitHub afin d'y déposer notre installateur téléchargeable par tout le monde.

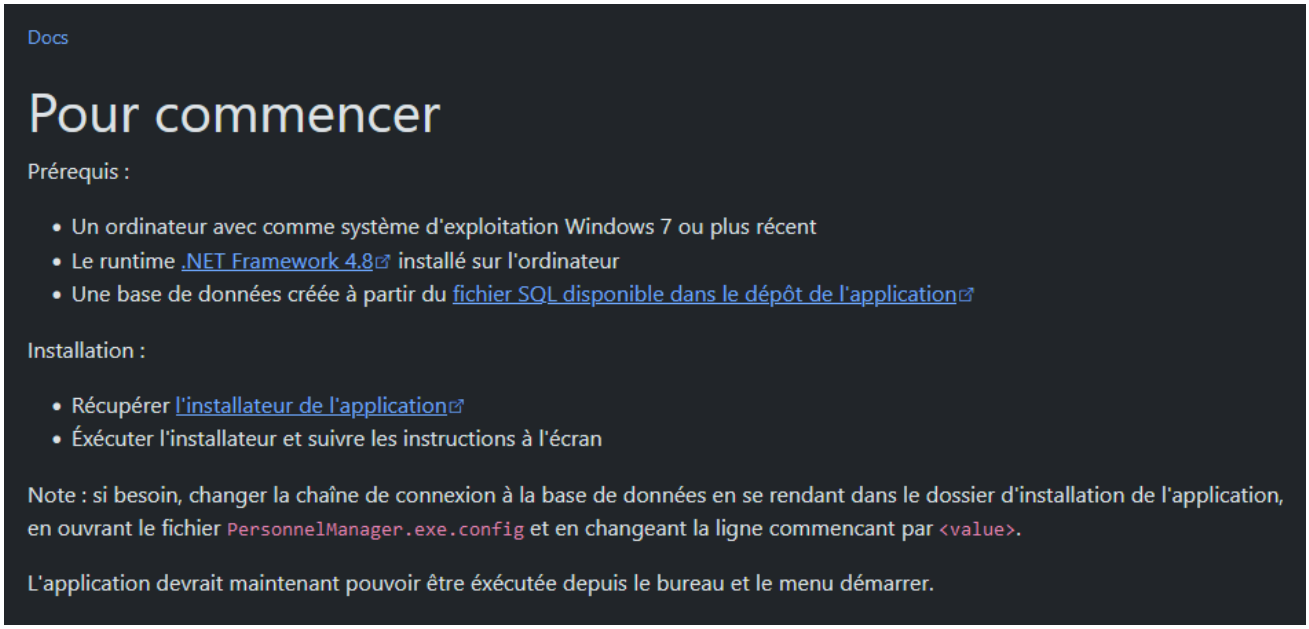


Étape 6 : Création de la documentation utilisateur, amélioration de la documentation technique

Pour la documentation utilisateur, il était demandé de réaliser une vidéo expliquant toutes les fonctionnalités de l'application, la vidéo devait être d'au moins 5 minutes. Pour ce faire, j'ai utilisé le logiciel OBS Studio qui permet d'enregistrer des vidéos mais bien plus encore, étant déjà plutôt familier avec ce logiciel, il ne restait plus qu'a appuyer sur le bouton pour enregistrer et de montrer les fonctionnalités en vidéo. J'ai utilisé une machine virtuelle Windows 11 vierge afin d'éviter de potentielles distractions présentes sur mon système hôte.

Une fois la vidéo enregistrée, j'ai édité quelques détails avec le logiciel de montage vidéo Shotcut puis j'ai exporté la vidéo finale, elle est retrouvable sur la page dédiée dans le portfolio.

J'ai aussi peaufiné quelques pages sur la documentation technique, on peut y retrouver un guide pour installer l'application ainsi que la page d'accueil présentée précédemment.



Étape 7 : Intégration du projet au portfolio et rédaction du compte rendu

J'ai codé mon portfolio de telle manière à faciliter l'ajout de projets, de ce fait, tout ce que j'ai eu à faire est d'ajouter un objet projet à la liste des projets

```
private List<Projet> GetProjects() =>
    [new Projet(Name: "PersonnelManager",
        Description: "Une application pour gérer le personnel et leurs absences dans une entreprise",
        ThumbnailPath: "img/personnelmanager-icon.ico",
        Year: "BTS SIO\n1ère année",
        Language: "C#",
        Link: "/projets/PersonnelManager"),

    new Projet(Name: "Portfolio",
        Description: "Le portfolio que vous visitez actuellement",
        ThumbnailPath: "favicon.png",
        Year: "BTS SIO\n1ère année",
        Language: "ASP.NET Core",
        Link: "/projets/Portfolio")];
```

… Il ne reste plus maintenant qu'à rédiger la page du projet (en cours de rédaction au moment où j'écris ces lignes) :

```
@page "/projets/PersonnelManager"

<PageTitle>Portfolio - PersonnelManager</PageTitle>

<a class="location-changer btn btn-outline-primary" role="button" href="projets">
    <span class="bi-arrow-return-left" style="..."></span>
    Retour aux projets
</a>

<h3 style="...">PersonnelManager - Gestionnaire de personnel</h3>

Une application pour gérer le personnel et leurs absences dans une entreprise

<div style="...">
    <a class="btn btn-outline-primary" role="button" href="https://github.com/Refragg/PersonnelManager" target="_blank">
        <span class="bi-github" style="..."></span>
        Voir sur GitHub
    </a>
</div>
```

Pour la rédaction de ce compte rendu, j'ai repris le code source à différents moments du développement en utilisant Git et les commits que j'ai réalisés, puis j'ai établi un plan et commencé à rédiger les pages que vous êtes en train de lire.

Bilan final

La réalisation de cette application selon un contexte réaliste a permis de voir des aspects de réalisation d'un projet parfois oubliés, la documentation technique, utilisateur, le respect d'un cahier des charges précis et la réalisation d'un projet en entier. Cela a été très enrichissant tout en ayant découvert et appris beaucoup de nouvelles choses.