

Atelier de professionnalisation 2

-

Application C# MediaTek86

-

Compte Rendu

Sommaire

1. Contexte.....	2
2. Mission globale.....	2
3. Les étapes.....	3
3.1 Étape 1 : préparation de l'environnement de travail, création de la base de données.....	3
3.2 Étape 2 : dessiner les interfaces, structurer l'application en MVC, créer un dépôt, coder le visuel.....	4
3.3 Étape 3 : coder le modèle et les outils de connexion, générer la documentation technique.....	7
3.3.1 Connexion avec la base de données.....	7
3.3.2 Data Access Layer : exploitation de la base de données.....	10
3.3.3 Le modèle.....	11
3.3.4 Documentation technique.....	13
3.4 Étape 4 : coder les fonctionnalités de l'application à partir des cas d'utilisation.....	14
3.4.1 Cas d'utilisation n°1 : Se connecter.....	14
3.4.2 Cas d'utilisation n°2 : Ajouter un personnel.....	17
3.4.3 Cas d'utilisation n°3 : Supprimer un personnel.....	19
3.4.4 Cas d'utilisation n°4 : Modifier un personnel.....	19
3.4.5 Cas d'utilisation n°5 : Afficher les absences.....	21
3.4.6 Cas d'utilisation n°6 : Ajouter une absence.....	22
3.4.7 Cas d'utilisation n°7 : Supprimer une absence.....	24
3.4.8 Cas d'utilisation n°8 : Modifier une absence.....	24
3.5 Étape 5 : créer une documentation utilisateur en vidéo.....	26
4. Bilan final.....	26

1. Contexte

Le réseau des médiathèques de la Vienne, MediaTek86, a besoin d'une application interne pour gérer le personnel des médiathèques, leur affectation à un service (Administratif, Médiation culturelle ou Prêt) ainsi que leurs absences.

C'est une application monoposte installé sur un des postes du service Administratif. Les membres du personnel ainsi que leurs absences seront enregistrés dans une base de données locale (si besoin une simple modification d'adresse du serveur permettra à l'application de se connecter à une base de données distante).

2. Mission globale

Après authentification, moyennant un nom d'utilisateur et un mot de passe enregistré sous forme cryptée dans la base de données, l'application affiche une liste de l'ensemble du personnel de la médiathèque. Pour chaque membre la liste comprend son nom, prénom, numéro de téléphone, adresse email, ainsi que le service auquel ce membre est affecté, à choisir dans une liste qui comprend les différents services : Administratif, Médiation culturelle, ou Prêt.

Ensuite on doit pouvoir ajouter un nouveau membre du personnel, modifier les détails d'un membre existant, ou encore supprimer un membre qui ne fait plus partie du personnel.

Pour chaque membre on doit pouvoir afficher une liste de ses absences. Cette liste comprend la date de début et la date de fin de l'absence, ainsi que le motif de l'absence.

On doit pouvoir ajouter une nouvelle absence, modifier une absence enregistrée, ou supprimer une absence. Le motif de chaque absence est à choisir dans une liste comprenant quatre options : vacances, maladie, motif familial ou congé parental.

L'application doit être sécurisé sur différents niveaux. La modification d'une absence ou d'un membre du personnel nécessite la sélection d'un élément de la liste au préalable. L'accès à la base de données doit prévoir une éventuelle erreur de connection, et une protection contre un usage malveillant sous forme d'injection SQL.

Pour coder l'application un choix est proposé entre les langages C# ou Java, pour la base de données un choix est proposé entre MySQL ou MariaDB.

Mon choix s'est porté sur C# pour l'application, MySQL pour la base de données.

3. Les étapes

3.1 Étape 1 : préparation de l'environnement de travail, création de la base de données.

Pour cette première étape il fallait tout d'abord mettre en place (ou vérifier la mise en place de) l'environnement de travail :

- Installation de Wampserver pour l'hébergement et l'accès à la base de données.
- Installation de l'IDE Visual Studio 2019 pour le développement de l'application sous C#.
- Installation de WinDesign pour la création de la base de données.

Un schéma conceptuel de données initial nous a été donné.

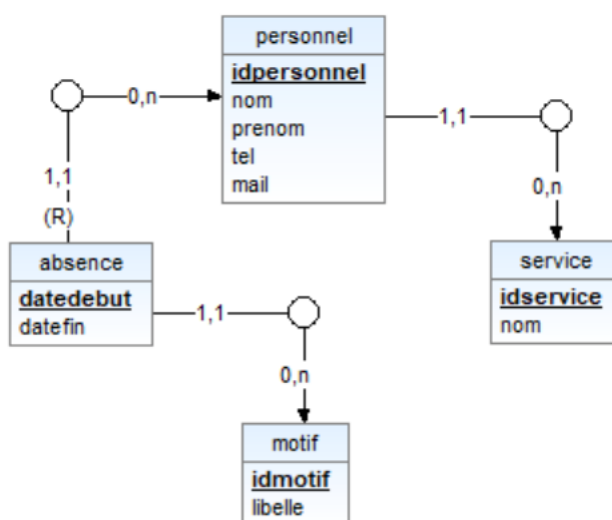


Figure 1 : Schéma conceptuel de la base de données MediaTek86 initiale.

Depuis ce schéma j'ai généré le script SQL permettant de créer la base de données. J'ai ajouté une table 'responsable' qui enregistre le nom d'utilisateur (unique) de l'application, ainsi que son mot de passe (sous forme cryptée). Cette table supplémentaire n'a pas de lien avec les autres tables de la base de données.

La table 'service' permet d'enregistrer les différents services auxquels le personnel de la médiathèque peut être affecté: Administratif, Médiation culturelle, Prêt.

La table 'motif' permet d'enregistrer les différents motifs d'absence possibles: vacances, maladie, motif familial, congé parental.

Afin de pouvoir développer et tester l'application, les tables 'personnel' et 'absence' ont été remplis avec des tuples aléatoires. Ces tuples ont été générés depuis le site web <http://www.generatedata.com/> .

J'ai également créé un utilisateur qui a les droits d'accès à cette base de données.

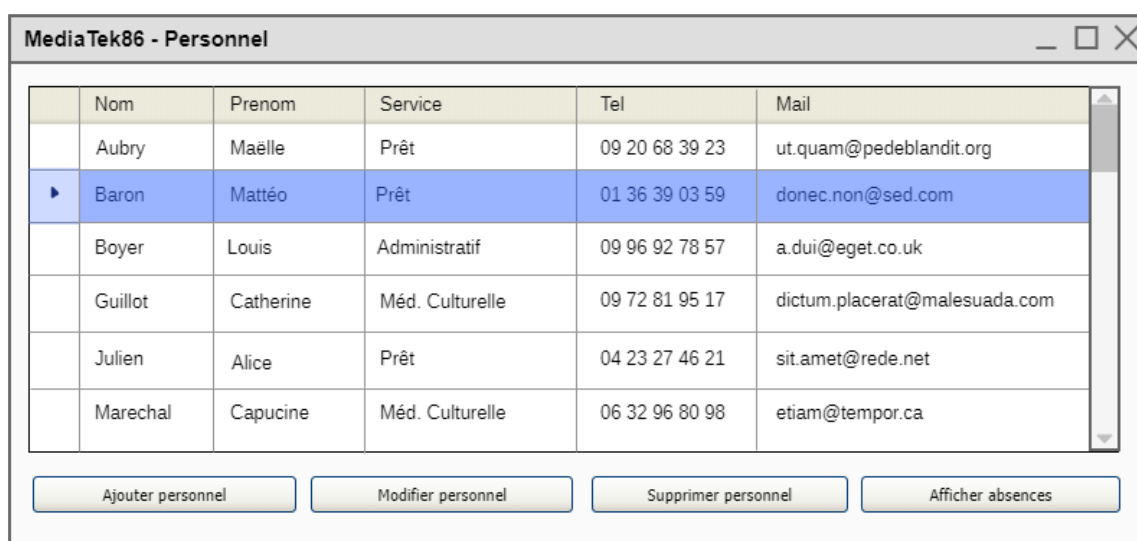
Bilan

À l'issue de cette étape les objectifs spécifiés au préalable ont été atteints:

- Installer un environnement de développement avec les outils nécessaires (pour la conception, la gestion de la base de données et le développement).
- À partir d'un schéma conceptuel de données existant, générer un script SQL et créer une base de données relationnelle.
- Alimenter la base de données pour gérer des tests.

3.2 Étape 2 : dessiner les interfaces, structurer l'application en MVC, créer un dépôt, coder le visuel.

Avant de commencer à coder l'application il convient de faire une maquette pour pouvoir proposer un interface utilisateur au client à partir des différents cas d'utilisation présentés dans le dossier documentaire. J'ai utilisé le logiciel Pencil qui permet non seulement de dessiner les différentes vues de l'application, mais aussi d'exporter un prototype interactif sous forme de pages web.



	Nom	Prenom	Service	Tel	Mail
	Aubry	Maëlle	Prêt	09 20 68 39 23	ut.quam@pedeblandit.org
▶	Baron	Mattéo	Prêt	01 36 39 03 59	donec.non@sed.com
	Boyer	Louis	Administratif	09 96 92 78 57	a.dui@eget.co.uk
	Guillot	Catherine	Méd. Culturelle	09 72 81 95 17	dictum.placerat@malesuada.com
	Julien	Alice	Prêt	04 23 27 46 21	sit.amet@rede.net
	Marechal	Capucine	Méd. Culturelle	06 32 96 80 98	etiam@tempor.ca

Ajouter personnel Modifier personnel Supprimer personnel Afficher absences

Figure 2: Proposition d'interface pour la liste du personnel.

MediaTek86 - Ajouter personnel

Nom Tel

Prénom Mail

Service ▼

Figure 3: Proposition d'interface pour l'ajout ou la modification d'un membre du personnel.

MediaTek86 - Absences Mattéo Baron

	Date début	Date fin	Motif
	25-03-2021	28-03-2021	maladie
▶	02-01-2021	03-01-2021	motif familial
	10-10-2020	15-10-2021	maladie
	08-05-2020	28-05-2020	vacances
	15-04-2020	30-05-2020	congé parental
	28-11-2019	10-12-2019	vacances

Figure 4: Proposition d'interface pour la liste d'absences.

MediaTek86 - Modifier absence

Date début

Date fin

Motif ▼

Figure 5: Proposition d'interface pour l'ajout ou la modification d'une absence.

Une fois la maquette accepté par le client on peut commencer à coder l'application.

D'abord il faut délimiter la structure en créant les packages correspondants au modèle MVC. Il y aura également besoin de deux packages, 'connexion' et 'dal' (data access layer) qui gèrent respectivement la connexion à la base de données et l'interaction de l'application avec cette base de données (principalement la création de requêtes SQL suite aux demandes du contrôleur).

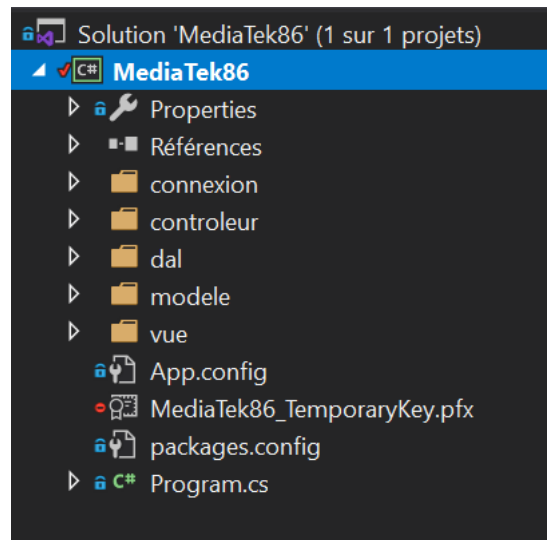
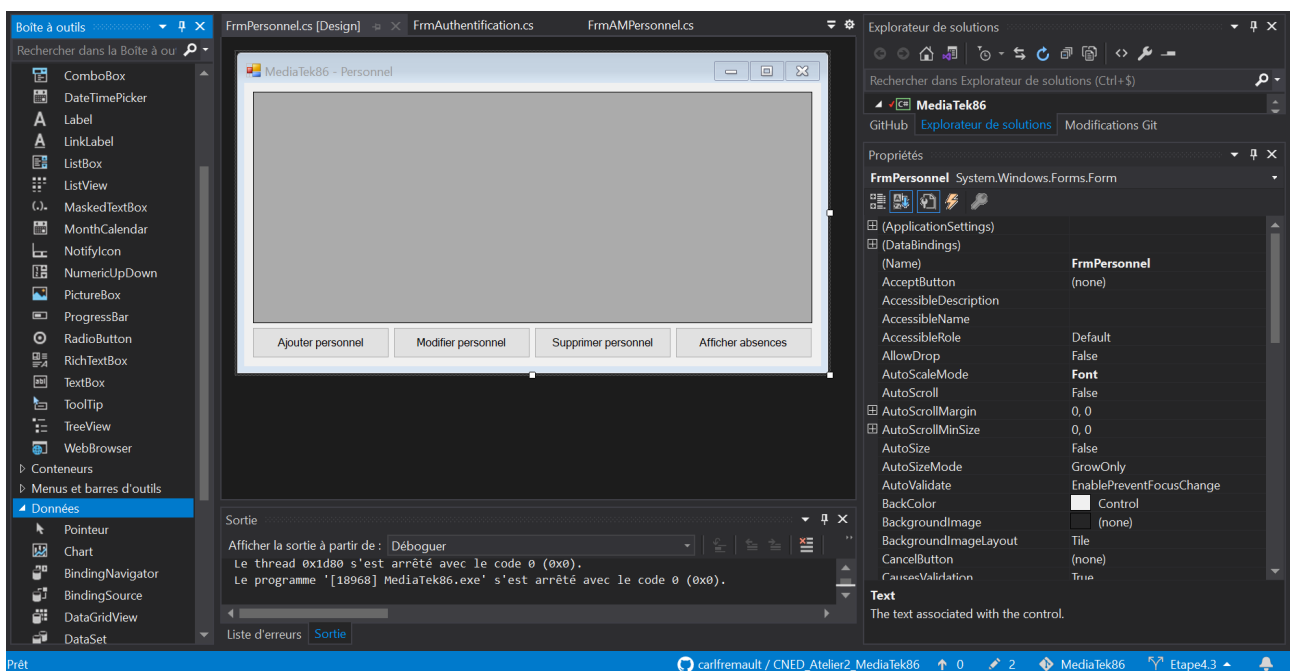


Figure 6: Structure de l'application.

Après la création de la structure j'ai créé un 'repository' sur GitHub afin de pouvoir sauvegarder les différentes étapes du développement et garder des traces des modifications successives.

Pour finir cet étape j'ai créé les différentes 'Windows Forms' (les vues) dans l'application à partir des maquettes approuvées par le client



Bien sûr, à la fin de chaque étape il convient de sauvegarder le projet sur le dépôt distant.

Bilan

À l'issue de cette étape les objectifs spécifiés au préalable ont été atteints:

- Dessiner des interfaces répondant aux besoins.
- Structurer une application suivant le modèle MVC.
- Créer un dépôt distant (par exemple sur GitHub).
- Coder la partie Vue d'une application (uniquement le visuel des interfaces).

3.3 Étape 3 : coder le modèle et les outils de connexion, générer la documentation technique.

3.3.1 Connexion avec la base de données

Afin de pouvoir tester l'application au fur et à mesure du développement il est bien entendu nécessaire d'avoir une connexion avec la base de données.

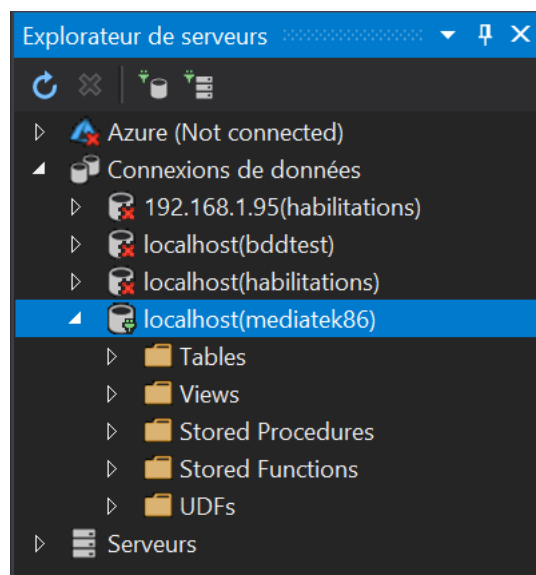


Figure 7: La connexion de l'IDE avec la base de données locale.

Ensuite il faut créer une classe permettant de faire la connexion dans l'application. Pour limiter l'utilisation de ressources cette classe est une classe 'singleton' qui assure qu'il n'y

a qu'une seule connection de faite. Pour y arriver, le constructeur de la classe a un niveau de protection 'privé', l'instanciation se fait à travers une méthode 'GetInstance' qui vérifie si une instance de la classe existe déjà, et la crée si besoin, avant de la retourner en sortie de méthode.

```
private ConnexionBDD(string chaineConnexion)
{
    try
    {
        connection = new MySqlConnection(chaineConnexion);
        connection.Open();
    }
    catch (Exception e)
    {
        MessageBox.Show(e.Message);
        Application.Exit();
    }
}
```

Figure 8: Le constructeur privé.

```
public static ConnexionBDD GetInstance(string chaineConnexion)
{
    if (ConnexionBDD.instance is null)
    {
        ConnexionBDD.instance = new ConnexionBDD(chaineConnexion);
    }
    return ConnexionBDD.instance;
}
```

Figure 9: La méthode GetInstance.

Afin de ne pas bloquer l'application en cas de problème de connexion, le constructeur utilise une structure try/catch et sort de l'application en cas d'erreur.

Une fois la connexion faite, cette classe doit aussi incorporer des méthodes permettant d'interroger ou modifier le contenu de la base de données.


```

public void ReqUpdate(string chaineRequete, Dictionary<string, object> parameters)
{
    try
    {
        command = new MySqlCommand(chaineRequete, connection);
        if (!(parameters is null))
        {
            foreach (KeyValuePair<string, object> parameter in parameters)
            {
                command.Parameters.Add(new MySqlParameter(parameter.Key, parameter.Value));
            }
        }
        command.Prepare();
        command.ExecuteNonQuery();
    }
    catch (Exception e)
    {
        MessageBox.Show(e.Message);
    }
}

```

Figure 10: Méthode qui permet de créer une requête SQL pour toute modification dans la base de données (ajout, suppression, mise à jour de tuples).

```

public void ReqSelect(string chaineRequete, Dictionary<string, object> parameters)
{
    try
    {
        command = new MySqlCommand(chaineRequete, connection);
        if (!(parameters is null))
        {
            foreach (KeyValuePair<string, object> parameter in parameters)
            {
                command.Parameters.Add(new MySqlParameter(parameter.Key, parameter.Value));
            }
        }
        command.Prepare();
        reader = command.ExecuteReader();
    }
    catch (Exception e)
    {
        MessageBox.Show(e.Message);
    }
}

```

Figure 11: Méthode qui permet de créer une requête SQL pour toute interrogation de la base de données.

Pour pouvoir interpréter les données retournées dans le curseur ('reader') on a également besoin d'une méthode pour lire la ligne suivante du curseur (et qui retourne 'false' si la fin a été atteinte), et d'une méthode qui permet de récupérer la valeur contenu dans un des champs.

```

public Boolean Read()
{
    if(reader is null)
    {
        return false;
    }
    try
    {
        return reader.Read();
    }
    catch
    {
        return false;
    }
}

```

Figure 12: Méthode qui lit la ligne suivant du curseur.

```

public object Field(string champ)
{
    if(reader is null)
    {
        return null;
    }
    try
    {
        return reader[champ];
    }
    catch
    {
        return null;
    }
}

```

Figure 13: Méthode qui retourne le contenu d'un champ du curseur.

Finalement, toujours dans l'optique d'optimiser l'utilisation de ressources, il convient de fermer la connection avec la base de données après chaque requête.

```

public void Close()
{
    if (!(reader is null))
    {
        reader.Close();
    }
}

```

Figure 14: Méthode qui ferme la connexion.

3.3.2 Data Access Layer : exploitation de la base de données

La classe AccesDonnees dans le package 'dal' (Data Access Layer) permet de créer les requêtes SQL pour exploiter la base de données, suivant les demandes du contrôleur.

Elle contient une propriété statique de type String pour mémoriser les paramètres d'accès à la base de données.

```

private static string connectionString = "server=localhost;user id=mediatek86; password=motdepasse; database=mediatek86; Sslmode=none";

```

Figure 15: Chaîne de connexion à la base de données.

Ensuite elle contient des différentes méthodes qui construisent une chaîne de requête SQL suivant les besoins, ouvrent une connexion à la base de données, exécutent la commande et, en cas d'interrogation de la base de données, exploitent un curseur pour exploiter les résultats obtenus. Ces méthodes seront écrites lors de l'exécution de la prochaine étape 4, cependant afin de préserver une structure claire dans ce document il me semble judicieux d'insérer les quelques exemples de code ici.

```
public static List<Personnel> GetLePersonnel()
{
    List<Personnel> lePersonnel = new List<Personnel>();
    string req = "select p.idpersonnel as idpersonnel, p.idservice as idservice, p.nom as nom, p.prenom as prenom, s.nom as service, p.tel as tel, p.mail as mail ";
    req += "from personnel p join service s on p.idservice = s.idservice order by nom, prenom;";
    ConnexionBDD curseur = ConnexionBDD.GetInstance(connectionString);
    curseur.RegSelect(req, null);
    while (curseur.Read())
    {
        lePersonnel.Add(new Personnel((int)curseur.Field("idpersonnel"), (int)curseur.Field("idservice"), (string)curseur.Field("nom"),
        (string)curseur.Field("prenom"), (string)curseur.Field("service"), (string)curseur.Field("tel"), (string)curseur.Field("mail")));
    }
    curseur.Close();
    return lePersonnel;
}
```

Figure 16: Exemple de méthode qui crée et exécute une requête SQL de type interrogation. Cette méthode récupère tous les membres du personnel présents dans la base de données sous forme de Liste.

```
public static void AddPersonnel(Personnel personnel)
{
    string req = "insert into personnel(idpersonnel, idservice, nom, prenom, tel, mail) ";
    req += "values(@idpersonnel, @idservice, @nom, @prenom, @tel, @mail);";
    Dictionary<string, object> parameters = new Dictionary<string, object>();
    parameters.Add("@idpersonnel", personnel.IdPersonnel);
    parameters.Add("@idservice", personnel.IdService);
    parameters.Add("@nom", personnel.Nom);
    parameters.Add("@prenom", personnel.Prenom);
    parameters.Add("@tel", personnel.Tel);
    parameters.Add("@mail", personnel.Mail);
    ConnexionBDD connection = ConnexionBDD.GetInstance(connectionString);
    connection.RegUpdate(req, parameters);
    connection.Close();
}
```

Figure 17: Exemple de méthode qui crée et exécute une requête SQL de type insertion. Cette méthode crée un nouveau tuple dans la base de données pour insérer un nouveau membre du personnel.

3.3.3 Le modèle

Il faut également créer les différentes classes métier du modèle. Chaque classe correspond à une table de la base de données, avec pour chaque champ une propriété privée. Le constructeur de la classe valorise ces propriétés, puis afin de faciliter l'exploitation dans le DataGridView il a été choisi d'encapsuler les différentes propriétés.

```

public Personnel(int idPersonnel, int idService, string nom, string prenom, string service, string tel, string mail)
{
    this.idPersonnel = idPersonnel;
    this.idService = idService;
    this.nom = nom;
    this.prenom = prenom;
    this.service = service;
    this.tel = tel;
    this.mail = mail;
}

```

Figure 18: Exemple: constructeur de la classe *Personnel*.

```

public int IdPersonnel { get => idPersonnel; set => idPersonnel = value; }

```

Figure 19: Encapsulation du champ 'idPersonnel' de la classe *Personnel*.

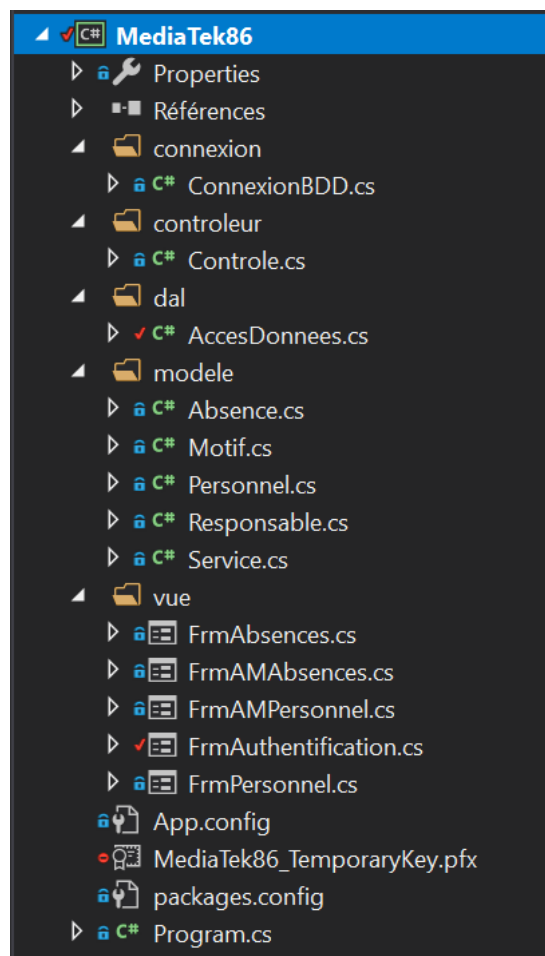


Figure 20: Vue d'ensemble de la structure de l'application, désormais au complet.

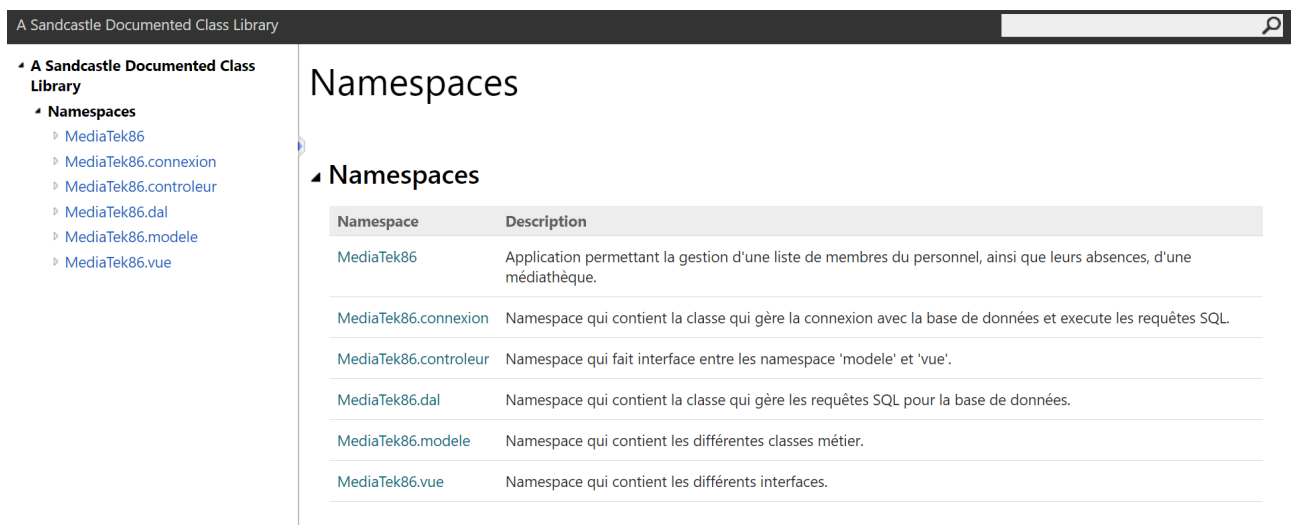
3.3.4 Documentation technique

Afin de permettre la compréhension du fonctionnement du code, et de pouvoir intervenir dessus ultérieurement, il est nécessaire d'avoir une documentation technique qui décrit chaque classe, chaque méthode, et chaque propriété de l'application. En insérant des commentaires 'normalisés' on peut ensuite générer cette documentation sous le format XML.

```
/// <summary>
/// Méthode qui crée une requête SQL puis l'envoie à la classe ConnexionBDD pour modifier une absence de la base de données.
/// </summary>
/// <param name="absenceAModifier">Objet de type absence, correspondant à l'absence initiale qu'on veut modifier.</param>
/// <param name="nouvelleAbsence">Objet de type absence, correspondant à l'absence initiale modifiée.</param>
1 référence | Carl Fremault, il y a 2 jours | 1 auteur, 1 modification
public static void UpdateAbsence(Absence absenceAModifier, Absence nouvelleAbsence)
{
    string ancienneDateDebut = DateTime.Parse(absenceAModifier.DateDebut).ToString("yyyy-MM-dd");
    string req = "update absence set datedebut = @nouveaudatedebut, idmotif = @idmotif, datefin = @datefin ";
    req += "where idpersonnel = @idpersonnel and DATE(datedebut) = @anciennedatedebut;";
    Dictionary<string, object> parameters = new Dictionary<string, object>();
    parameters.Add("@nouveaudatedebut", nouvelleAbsence.DateDebut);
    parameters.Add("idmotif", nouvelleAbsence.IdMotif);
    parameters.Add("datefin", nouvelleAbsence.DateFin);
    parameters.Add("idpersonnel", absenceAModifier.IdPersonnel);
    parameters.Add("@anciennedatedebut", ancienneDateDebut);
    ConnexionBDD connection = ConnexionBDD.GetInstance(connectionString);
    connection.ReqUpdate(req, parameters);
    connection.Close();
}
```

Figure 21: Exemple de méthode avec un commentaire 'normalisé' qui explique le fonctionnement de la classe et les paramètres.

Pour faciliter la lecture de cette documentation j'ai ensuite utilisé le logiciel OpenSource Sandcastle, qui permet de 'transformer' le fichier XML en page web HTML, bien plus facile à lire et à naviguer.



The screenshot shows the Sandcastle Documented Class Library web interface. On the left is a sidebar with a tree view of the project structure: A Sandcastle Documented Class Library > Namespaces > MediaTek86 > MediaTek86.connexion > MediaTek86.controleur > MediaTek86.dal > MediaTek86.modele > MediaTek86.vue. The main content area is titled 'Namespaces' and contains a table with the following data:

Namespace	Description
MediaTek86	Application permettant la gestion d'une liste de membres du personnel, ainsi que leurs absences, d'une médiathèque.
MediaTek86.connexion	Namespace qui contient la classe qui gère la connexion avec la base de données et exécute les requêtes SQL.
MediaTek86.controleur	Namespace qui fait interface entre les namespace 'modele' et 'vue'.
MediaTek86.dal	Namespace qui contient la classe qui gère les requêtes SQL pour la base de données.
MediaTek86.modele	Namespace qui contient les différentes classes métier.
MediaTek86.vue	Namespace qui contient les différents interfaces.

Figure 22: Page d'accueil documentation technique Sandcastle

Bien entendu, comme à chaque fin d'étape, l'ensemble du projet a été sauvegardé sur le dépôt distant.

Bilan

À l'issue de cette étape les objectifs spécifiés au préalable ont été atteints:

- Coder la partie modèle d'une application et les outils pour la connexion à la base de données.
- Générer une documentation technique.
- Gérer les sauvegardes sur un dépôt distant.

3.4 Étape 4 : coder les fonctionnalités de l'application à partir des cas d'utilisation.

Pour cet étape j'ai procédé à coder les différentes classes conforme aux cas d'utilisation décrites dans le dossier documentaire.

3.4.1 Cas d'utilisation n°1 : Se connecter

La classe Program, point d'entrée de l'application, appelle le constructeur de la classe Controle, qui instancie ensuite les différentes vues de l'application, et affiche la vue FrmAuthentification qui permet à l'utilisateur de se connecter.

```
public Controle()
{
    frmPersonnel = new FrmPersonnel(this);
    frmAMPersonnel = new FrmAMPersonnel(this);
    frmAbsences = new FrmAbsences(this);
    frmAMAbsences = new FrmAMAbsences(this);
    frmAuthentification = new FrmAuthentification(this);
    frmAuthentification.ShowDialog();
}
```

Figure 23: Constructeur de la classe Controle.

Lors de l'instanciation des différentes vues celles-ci récupèrent immédiatement les données dont elles ont besoin de la base de données.

```

public void RemplirListePersonnel()
{
    List<Personnel> lePersonnel = controle.GetLePersonnel();
    bdgPersonnel.DataSource = lePersonnel;
    dgvPersonnel.DataSource = bdgPersonnel;
    dgvPersonnel.Columns["idpersonnel"].Visible = false;
    dgvPersonnel.Columns["idservice"].Visible = false;
    dgvPersonnel.AutoSizeColumnsMode = DataGridViewAutoSizeColumnsMode.AllCells;
}

```

Figure 24: Exemple: la méthode *RemplirListePersonnel*, de la vue *FrmPersonnel*, récupère une liste des membres du personnel de la base de données et les insère pour affichage dans un *DataGridView*.

```

public void RemplirServices()
{
    List<Service> lesServices = controle.GetLesServices();
    bdgServices.DataSource = lesServices;
    cboService.DataSource = bdgServices;
    if (cboService.Items.Count > 0)
    {
        cboService.SelectedIndex = 0;
    }
}

```

Figure 25: Exemple: la méthode *RemplirServices*, de la vue *FrmAMPersonnel*, récupère une liste des différents services de la base de données et les insère pour affichage dans une combobox.

Après avoir rempli les champs avec son nom d'utilisateur et mot de passe, l'utilisateur clique sur le bouton 'Connecter' pour accéder à l'application.

```

private void btnConnecter_Click(object sender, EventArgs e)
{
    if (!txtUtilisateur.Text.Equals("") && !txtMotdepasse.Text.Equals(""))
    {
        if (!controle.VerifierAuthentification(txtUtilisateur.Text, txtMotdepasse.Text))
        {
            MessageBox.Show("Utilisateur et/ou mot de passe incorrecte.", "Information");
            txtUtilisateur.Text = "";
            txtMotdepasse.Text = "";
        }
    }
    else
    {
        MessageBox.Show("Tous les champs doivent être remplis.", "Information");
    }
}

```

Figure 26: Méthode événementielle après un clic sur le bouton 'Connecter'. La méthode vérifie si tous les champs sont remplis et appelle la méthode *VerifierAuthentification* du controleur.

La méthode `VérifierAuthentification` du contrôleur appelle la méthode du même nom de la classe `AccesDonnees` qui vérifie l'existence de l'utilisateur dans la base de données (table 'responsable') et si le mot de passe correspond. Cette classe utilise une méthode `GetStringSha256Hash` pour crypter le mot de passe saisi et ainsi pouvoir le comparer avec le mot de passe stocké dans la base de données, sous forme cryptée.

Chaque méthode retourne un booléen à son prédecesseur. Si le nom d'utilisateur et le mot de passe sont correctes l'application ouvre sa fenêtre 'principale' qui affiche la vue des membres du personnel.

```
public static Boolean VerifierAuthentification(string utilisateur, string motdepasse)
{
    string req = "select * from responsable where login=@login and pwd=@pwd;";
    Dictionary<string, object> parameters = new Dictionary<string, object>();
    parameters.Add("@login", utilisateur);
    parameters.Add("@pwd", GetStringSha256Hash(motdepasse));
    ConnexionBDD curseur = ConnexionBDD.GetInstance(connectionString);
    curseur.ReqSelect(req, parameters);

    if (curseur.Read())
    {
        curseur.Close();
        return true;
    }
    else
    {
        curseur.Close();
        return false;
    }
}
```

Figure 27: Méthode `VerifierAuthentification` de la classe `AccesDonnees`.

3.4.2 Cas d'utilisation n°2 : Ajouter un personnel

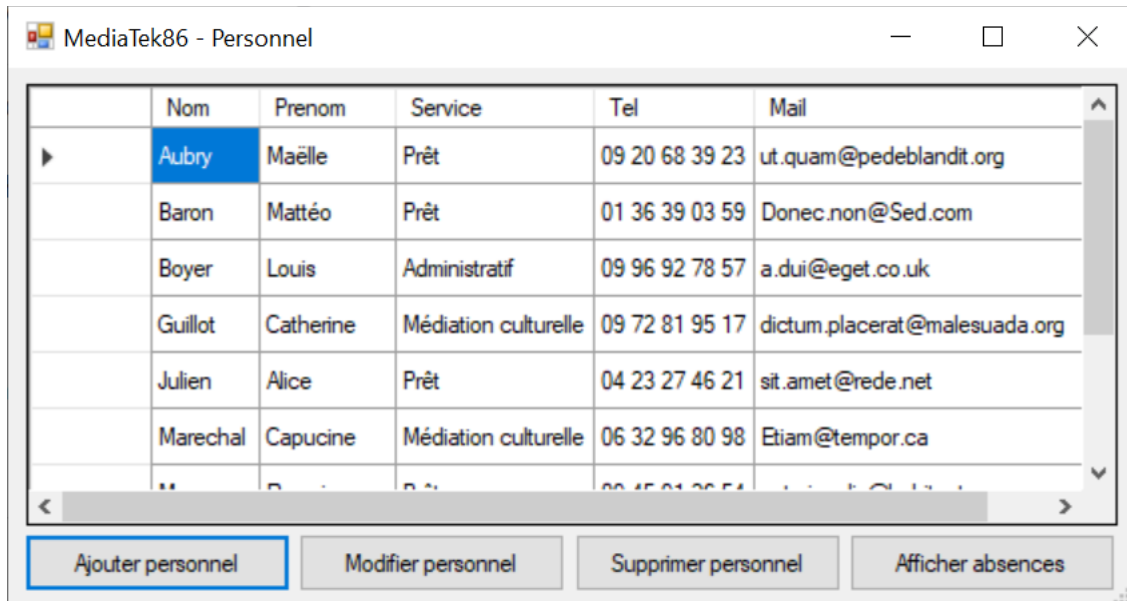


Figure 28: Fenêtre principale de l'application : vue FrmPersonnel.

Un clic sur le bouton 'Ajouter personnel' permet à l'utilisateur d'ajouter un nouveau membre du personnel dans la base de données.

```
private void btnAjoutPersonnel_Click(object sender, EventArgs e)
{
    controle.AjouterPersonnel();
}
```

Figure 29: La méthode appelle la méthode du même nom du contrôleur...

```
public void AjouterPersonnel()
{
    frmAMPersonnel.Text = "MediaTek86 - Ajouter personnel";
    frmAMPersonnel.ShowDialog();
}
```

Figure 30: ... qui ouvre ensuite la vue FrmAMPersonnel.

Figure 31: La vue FrmAMPersonnel qui permet d'ajouter ou modifier un membre du personnel.

Après avoir rempli tous les champs, et avoir choisi parmi les différents services, l'utilisateur peut enregistrer le nouvel employé en cliquant 'Enregistrer'. L'application vérifie si tous les champs sont bien remplis, mais aussi si on est en train de modifier ou ajouter un membre du personnel (voir point 3.4.4).

Après confirmation du souhait d'enregistrer de la part de l'utilisateur la méthode envoie le nouveau membre du personnel au contrôleur, qui la fait suivre à la classe AccesDonnees vu précédemment.

Un clic sur le bouton 'Annuler' permet de revenir sur la liste du personnel.

```
private void btnEnregistrer_Click(object sender, EventArgs e)
{
    if (!txtNom.Text.Equals("") && !txtPrenom.Text.Equals("") && !txtMail.Text.Equals("") && !txtTel.Text.Equals("") && cboService.SelectedIndex != -1)
    {
        Service leService = (Service)bdgServices.List[bdgServices.Position];
        if (modification)
        {
            Personnel personnelAModifier = new Personnel(idPersonnelAModifier, leService.IdService, txtNom.Text, txtPrenom.Text, leService.Nom, txtTel.Text, txtMail.Text);

            if (MessageBox.Show("Souhaitez-vous confirmer la modification?", "Confirmation de modification", MessageBoxButtons.YesNo) == DialogResult.Yes)
            {
                controle.UpdatePersonnel(personnelAModifier);
                modification = false;
            }
            else
            {
                controle.FermerAMPersonnel();
            }
        }
        else
        {
            int idPersonnel = 0;
            Personnel lePersonnel = new Personnel(idPersonnel, leService.IdService, txtNom.Text, txtPrenom.Text, leService.Nom, txtTel.Text, txtMail.Text);
            controle.AddPersonnel(lePersonnel);
        }
    }
    else
    {
        MessageBox.Show("Tous les champs doivent être remplis.", "Information");
    }
}
```

Figure 32: Méthode événementielle après un clic sur le bouton 'Enregistrer'.

3.4.3 Cas d'utilisation n°3 : Supprimer un personnel

Après un clic sur le bouton 'Supprimer personnel' (cf. Fig. 28) l'application vérifie si l'utilisateur a bien sélectionné un membre du personnel. Après une demande de confirmation de suppression, la méthode appelée récupère d'abord la liste des absences liés à ce membre du personnel. Effectivement, dans le tableau 'absences' de la base de données, la clé primaire est composé de deux attributs: la date de début d'absence, mais aussi l'id du membre du personnel. Cet attribut ne peut donc pas être null, ce qui nous oblige de supprimer les absences avant de pouvoir supprimer le personnel.

Ensuite le membre du personnel est supprimé aussi (appel du contrôleur en lui envoyant le membre du personnel à supprimer) et la liste des membres du personnel est rafraîchi.

```
private void btnSuppPersonnel_Click(object sender, EventArgs e)
{
    if(dgvPersonnel.SelectedRows.Count > 0)
    {
        Personnel personnel = (Personnel)bdgPersonnel.List[bdgPersonnel.Position];
        if (MessageBox.Show("Confirmez-vous la suppression de " + personnel.Prenom + " " + personnel.Nom + " de la liste?", "Confirmation de suppression",
            MessageBoxButtons.YesNo) == DialogResult.Yes)
        {
            List<Absence> absences = controle.GetLesAbsences(personnel);
            foreach(Absence absence in absences)
            {
                controle.DelAbsence(absence, personnel);
            }
            controle.DelPersonnel(personnel);
            RemplirListePersonnel();
        }
    }
    else
    {
        MessageBox.Show("Veuillez sélectionner un membre du personnel.", "Information");
    }
}
```

Figure 33: Méthode événementielle après un clic sur le bouton 'Supprimer personnel'.

3.4.4 Cas d'utilisation n°4 : Modifier un personnel

Dans les grandes lignes ce cas d'utilisation se déroule de façon similaire à l'ajout d'un personnel, si ce n'est que, tout comme le cas de suppression de personnel, l'application vérifie d'abord si un membre du personnel a bien été sélectionné avant de l'envoyer au contrôleur.

```
private void btnModifPersonnel_Click(object sender, EventArgs e)
{
    if (dgvPersonnel.SelectedRows.Count > 0)
    {
        Personnel personnel = (Personnel)bdgPersonnel.List[bdgPersonnel.Position];
        controle.ModifierPersonnel(personnel);
    }
    else
    {
        MessageBox.Show("Veuillez sélectionner un membre du personnel.", "Information");
    }
}
```

Figure 34: Méthode événementielle après un clic sur le bouton 'Modifier personnel'.

Une autre différence est que le contrôleur, avant d'ouvrir la vue 'FrmAMPersonnel' (cf. Fig. 31), appelle la méthode `ModifierPersonnel` de cette classe pour actionner le booléen 'modification' (pour que la méthode qui enrégistre un personnel (cf. Fig. 32) peut appeler la méthode correspondante) et remplir les champs avec les détails du membre du personnel sélectionné.

```
public void ModifierPersonnel(Personnel personnel)
{
    frmAMPersonnel.Text = "MediaTek86 - Modifier personnel";
    frmAMPersonnel.ModifierPersonnel(personnel);
    frmAMPersonnel.ShowDialog();
}
```

Figure 35: Méthode `ModifierPersonnel` du contrôleur.

```
public void ModifierPersonnel(Personnel personnel)
{
    if (!(personnel is null))
    {
        modification = true;
        idPersonnelAModifier = personnel.IdPersonnel;
        txtNom.Text = personnel.Nom;
        txtPrenom.Text = personnel.Prenom;
        txtTel.Text = personnel.Tel;
        txtMail.Text = personnel.Mail;
        cboService.SelectedIndex = cboService.FindStringExact(personnel.Service);
    }
}
```

Figure 36: Méthode `ModifierPersonnel` de la vue `FrmAMPersonnel`.

3.4.5 Cas d'utilisation n°5 : Afficher les absences

Après un clic sur le bouton 'Afficher absences', l'application vérifie si un membre du personnel a bien été sélectionné, récupère les absences lui concernant de la base de données, puis affiche ces absences dans la vue FrmAbsences.

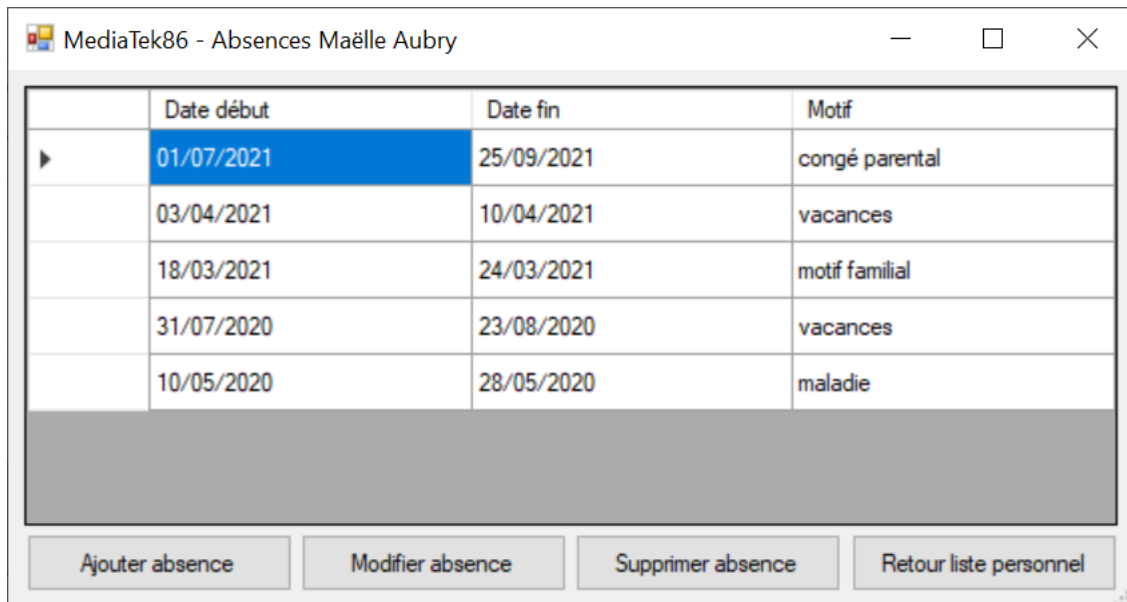


Figure 37: Vue FrmAbsences

Bien évidemment cette fenêtre permet de revenir sur la liste du personnel en cliquant sur le bouton 'Retour liste personnel'.

Une petite difficulté dans cette partie de l'application consiste au traitement du format des dates. SQL nécessite l'enregistrement sous forme 'yyyy-MM-dd', dans l'application on préfère un format 'Européen' plus lisible 'dd/MM/yyyy'.

Ainsi sur différents endroits on doit procéder à un formatage des dates.

```
while(curseur.Read())
{
    string dateDebut = ((DateTime)curseur.Field("datedebut")).ToString("dd/MM/yyyy");
    string dateFin = ((DateTime)curseur.Field("datefin")).ToString("dd/MM/yyyy");
    Absence absence = new Absence((int)personnel.IdPersonnel, dateDebut, (int)curseur.Field("idmotif"), (string)curseur.Field("motif"), dateFin);
    lesAbsences.Add(absence);
}
```

Figure 38: Extrait de la méthode GetLesAbsences de la classe AccesDonnees qui récupère les dates sous format 'SQL' de la base de données puis les formate pour usage dans l'application.

```

Absence nouvelleAbsence = new Absence((int)personnelAbsence.IdPersonnel, ((DateTime)ntpDebut.Value).ToString("yyyy-MM-dd"), (int)motif.IdMotif, (string)
motif.Libelle, ((DateTime)ntpFin.Value).ToString("yyyy-MM-dd"));
if (modificationAbsence)
{
    if ((MessageBox.Show("Souhaitez-vous confirmer la modification?", "Confirmation de modification", MessageBoxButtons.YesNo)) == DialogResult.Yes)
    {
        modificationAbsence = false;
        controle.UpdateAbsence(absenceAModifier, nouvelleAbsence, personnelAbsence);
    }
    else
    {
        controle.FermerAMAbsences(personnelAbsence);
    }
}
else
{
    controle.AddAbsence(nouvelleAbsence, personnelAbsence);
}

```

Figure 39: Extrait de la méthode `btnEnregistrer_Click` de la vue `FrmAMAbsences` qui transforme le format des dates récupérés dans les champs 'DateTimePicker' avant de les envoyer dans la base de données (par intermédiaire du contrôleur et la classe `AccesDonnees`).

```

public static void DelAbsence(Absence absence, Personnel personnelAbsence)
{
    string dateDebut = DateTime.Parse(absence.DateDebut).ToString("yyyy-MM-dd");
    string req = "delete from absence where idpersonnel = @idpersonnel and DATE(datedebut) = @datedebut;";
    Dictionary<string, object> parameters = new Dictionary<string, object>();
    parameters.Add("@idpersonnel", personnelAbsence.IdPersonnel);
    parameters.Add("@datedebut", dateDebut);
    ConnexionBDD connection = ConnexionBDD.GetInstance(connectionString);
    connection.ReqUpdate(req, parameters);
    connection.Close();
}

```

Figure 40: La méthode `DelAbsence` de la classe `AccesDonnees` (cf point 3.4.7) récupère les dates non pas depuis les champs 'DateTimePicker' mais directement depuis l'objet `Absence` concerné. Un traitement différent était donc nécessaire.

3.4.6 Cas d'utilisation n°6 : Ajouter une absence

Dans une même approche que le cas d'utilisation n°2 (Ajouter un personnel, cf. point 3.4.2) un clic sur le bouton 'Ajouter absence' permet à l'utilisateur d'ajouter une nouvelle absence concernant un membre du personnel dans la base de données.

Figure 41: La vue *FrmAMAbsences* qui permet d'ajouter ou modifier une absence.

Après un clic sur le bouton 'Enregistrer' l'application vérifie si tous les champs ont été remplis, si on est en train d'ajouter ou modifier une absence (avec, à l'identique des méthodes concernant le personnel, un booléen 'modificationAbsance' qui est initialisé à 'false' puis passe à 'true' quand l'utilisateur choisit de modifier une absence (cf. point 3.4.8)) puis elle vérifie aussi si la date de fin est bien postérieure à la date de début, avant d'envoyer l'absence au contrôleur (puis à la classe *AccesDonnees*) pour enregistrement dans la base de données.

```
private void btnEnregistrer_Click(object sender, EventArgs e)
{
    if (!dtpDebut.Value.Equals("") && !dtpFin.Value.Equals("") && cboMotif.SelectedIndex != -1)
    {
        if (dtpDebut.Value.Date <= dtpFin.Value.Date)
        {
            Motif motif = (Motif)bdgMotifs.List[bdgMotifs.Position];
            Absence nouvelleAbsence = new Absence((int)personnelAbsence.IdPersonnel, ((DateTime)dtpDebut.Value).ToString("yyyy-MM-dd"), (int)motif.IdMotif, (string)
            motif.Libelle, ((DateTime)dtpFin.Value).ToString("yyyy-MM-dd"));
            if (modificationAbsance)
            {
                if ((MessageBox.Show("Souhaitez-vous confirmer la modification?", "Confirmation de modification", MessageBoxButtons.YesNo)) == DialogResult.Yes)
                {
                    modificationAbsance = false;
                    controle.UpdateAbsence(absenceAModifier, nouvelleAbsence, personnelAbsence);
                }
                else
                {
                    controle.FermerAMAbsences(personnelAbsence);
                }
            }
            else
            {
                controle.AddAbsence(nouvelleAbsence, personnelAbsence);
            }
        }
        else
        {
            MessageBox.Show("La date de fin de l'absence ne peut être inférieure à la date de début.", "Information");
        }
    }
}
```

Figure 42: Extrait de la méthode événementielle *btnEnregistrer_Click*.

3.4.7 Cas d'utilisation n°7 : Supprimer une absence

Toujours dans la même approche que le traitement des membres du personnel, lors d'un clic sur le bouton 'Supprimer absence' l'application vérifie si l'utilisateur a bien sélectionné une absence et demande une confirmation de suppression de la part de l'utilisateur.

Ensuite, la liste des absences est rafraîchi.

```
private void btnSuppAbsence_Click(object sender, EventArgs e)
{
    if (dgvAbsences.SelectedRows.Count > 0)
    {
        Absence absence = (Absence)bdgAbsences.List[bdgAbsences.Position];
        if (MessageBox.Show("Confirmez-vous la suppression de l'absence de la liste?", "Confirmation de suppression", MessageBoxButtons.YesNo) == DialogResult.Yes)
        {
            controle.DelAbsence(absence, personnelAbsence);
            RemplirListeAbsences(personnelAbsence);
        }
    }
    else
    {
        MessageBox.Show("Veuillez sélectionner une absence.", "Information");
    }
}
```

Figure 43: La méthode `btnSuppAbsence_Click` de la vue `FrmAbsences`, appelée après un clic sur le bouton 'Supprimer absence'.

3.4.8 Cas d'utilisation n°8 : Modifier une absence

Encore une fois on suit la même logique, non seulement du cas d'utilisation de modification d'un membre du personnel, mais aussi du cas d'ajout d'une absence.

Un clic sur le bouton 'Modifier absence' vérifie si une absence a été sélectionnée, ouvre la vue `FrmAMAbsences` en y insérant les valeurs de l'absence sélectionnée.

```
private void btnModifAbsence_Click(object sender, EventArgs e)
{
    if (dgvAbsences.SelectedRows.Count > 0)
    {
        Absence absence = (Absence)bdgAbsences.List[bdgAbsences.Position];
        controle.ModifierAbsence(absence, personnelAbsence);
    }
    else
    {
        MessageBox.Show("Veuillez sélectionner une absence.", "Information");
    }
}
```

Figure 44: La méthode événementielle `btnModifAbsence` après un clic sur le bouton 'Modifier absence'.


```

public void ModifierAbsence (Absence absence, Personnel personnel)
{
    if (!(absence is null))
    {
        modificationAbsence = true;
        absenceAModifier = absence;
        personnelAbsence = personnel;
        dtpDebut.Value = DateTime.Parse(absence.DateDebut);
        dtpFin.Value = DateTime.Parse(absence.DateFin);
        cboMotif.SelectedIndex = cboMotif.FindStringExact(absence.Motif);
    }
}

```

Figure 45: La méthode 'ModifierAbsence' de la vue FrmAMAbsences qui actionne le booléen 'modificationAbsence' et récupère les valeurs de l'absence à modifier pour mettre à jour les différents champs.

Après le bouton 'Enregistrer' appelle la même méthode 'btnEnregistrer_Click' vue précédemment (Fig. 42).

Bien entendu chaque implémentation d'un cas d'utilisation est suivi d'une phase de tests pour vérifier le bon fonctionnement. Aussi, les commentaires normalisés pour chaque méthode ou propriété sont mises à jour au fur et à mesure, puis à chaque lancement de l'application l'IDE met automatiquement à jour le fichier XML de la documentation technique. Il suffit ensuite (de préférence à la fin de l'étape du développement) de régénérer la page web créée par Sandcastle.

Bilan

À l'issue de cette étape les objectifs spécifiés au préalable ont été atteints:

- Coder les fonctionnalités d'une application.
- Gérer les sauvegardes sur un dépôt distant.
- Tester une application.
- Mettre à jour une documentation technique.

3.5 Étape 5 : créer une documentation utilisateur en vidéo

Pour cette étape il fallait créer une documentation utilisateur sous forme de vidéo. Suite aux recommandations j'ai utilisé le logiciel Wink.

Bilan

À l'issue de cette étape l'objectif spécifié au préalable a été atteint:

- Créer une documentation utilisateur (démonstration des fonctionnalités de l'application) sous forme de vidéo.

3.6 Étape 6 : gérer le déploiement, rédiger le compte rendu, créer la page du portfolio dédiée à la mission

Pour l'étape finale il fallait tout d'abord exporter l'application sous forme de fichier permettant son installation sur un poste de travail.

Ensuite j'ai exporté le script de la base de données, aussi bien pour la création de la structure que pour l'insertion des données dans les tables. Ce script a été rajouté dans le dépôt distant sur GitHub.

Après j'ai écrit ce compte rendu détaillant pour chaque étape les travaux effectués ainsi qu'un bilan pour vérifier si les objectifs demandés sont bien atteints.

Puis pour compléter le tout j'ai créé une publication sur mon portfolio avec les liens vers le dépôt GitHub, la documentation technique, ce pdf. La vidéo de démonstration a également été incorporé.

Bilan

À l'issue de cette étape l'objectif spécifié au préalable a été atteint:

- Préparer les fichiers nécessaires pour le déploiement d'une application.
- Rédiger un compte rendu d'activité.
- Créer une page dans le portfolio pour présenter l'activité et donner tous les liens nécessaires.

N.B. : Compte tenu de la structure de mon portfolio, sous forme approximative de blog, plutôt que de créer une page séparée j'ai préféré insérer le projet dans un nouveau 'post' sur la page principale des projets.

4. Bilan final

À l'issue de ce projet j'estime que les objectifs spécifiés ont bien été atteints. L'application est fonctionnelle, les demandes des différents cas d'utilisation ont été respectées. La documentation technique est au complet puis l'ensemble du projet, y compris ce compte rendu, est disponible sur mon portfolio.

Le projet a été très intéressant et formateur. Il m'a permis d'approfondir mes connaissances du langage C#, mais aussi du modèle MVC, de l'exploitation d'une base de données dans une application et du travail en mode projet.

Quelques difficultés ont été rencontrés en route (utilisation de la classe DataGridView, formatage de dates, ...) mais grâce aux nombreux ressources disponibles en ligne rien n'était insurmontable.