MediaTek86 – Application de gestion du personnel

Sommaire

1. Contexte et mission à effectuer

Pour cette activité, nous nous mettons dans la peau d'une technicienne développeuse junior travaillant pour l'ESN fictive InfoTech Services 86, spécialisée dans le développement informatique. Celle-ci vient de remporter le marché pour différentes interventions au sein du réseau MediaTek86, qui gère les médiathèques du département de la Vienne.

La tâche qui nous est confiée est le développement de l'application de bureau qui va permettre de gérer le personnel de chaque médiathèque, leur affectation à un service et leurs absences. Plus précisément, l'application devra permettre au responsable d'effectuer les actions suivantes :
- se connecter,
- afficher la liste du personnel,
- ajouter un membre du personnel,
- modifier un membre du personnel,
- supprimer un membre du personnel,
- afficher les absences d'un membre du personnel,
- ajouter une absence,
- modifier une absence,
- supprimer une absence.

2. Travail réalisé

2.1. Étape 1 : préparation

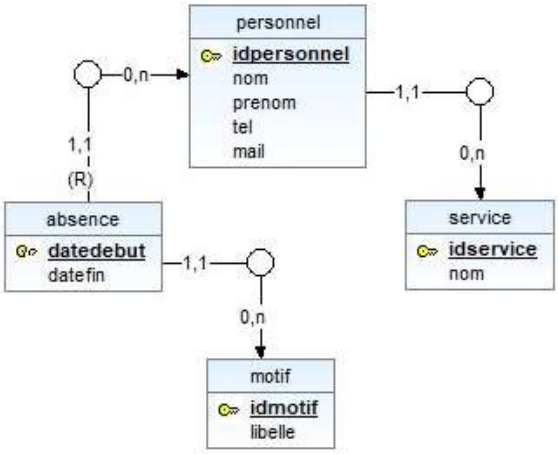
L'étape 1 de la mission consiste à préparer d'une part les outils nécessaires au développement de l'application, et d'autre part la base de données qui y sera liée.

2.1.1. Préparation de l'environnement

Les installations suivantes sont réalisées sur l'ordinateur utilisé pour le développement :
- la dernière implémentation de Java, en l'occurrence la version 17 de Java SE Development Kit, téléchargée depuis la page https://www.oracle.com/java/technologies/downloads,
- Eclipse, l'IDE choisi pour développer en Java,
- Wamp64, qui sera notre serveur de base de données,
- WinDesign, logiciel de modélisation de données.

2.1.2. Création de la base de données

Le schéma conceptuel de données est d'abord récupéré au format WinDesign :



Depuis le logiciel, nous générons le modèle logique, puis le script de la base de données correspondant, dont voici un extrait :

MEDIATEK.SQL

```
DROP DATABASE IF EXISTS MediaTek86;

CREATE DATABASE IF NOT EXISTS MediaTek86;
USE MediaTek86;
# ------------------------------------------------------------------------------
#       TABLE : absence
# ------------------------------------------------------------------------------

CREATE TABLE IF NOT EXISTS absence
 (
   idpersonnel INTEGER NOT NULL  ,
   datedebut DATETIME NOT NULL  ,
   idmotif INTEGER NOT NULL  ,
   datefin DATETIME NULL
   , PRIMARY KEY (idpersonnel,datedebut)
 )
ENGINE=InnoDB;

# ------------------------------------------------------------------------------
#       TABLE : motif
# ------------------------------------------------------------------------------

CREATE TABLE IF NOT EXISTS motif
 (
   idmotif INTEGER NOT NULL AUTO_INCREMENT ,
   libelle VARCHAR(128) NULL
   , PRIMARY KEY (idmotif)
 )
ENGINE=InnoDB;

# ------------------------------------------------------------------------------
#       TABLE : service
# ------------------------------------------------------------------------------

CREATE TABLE IF NOT EXISTS service
 (
   idservice INTEGER NOT NULL AUTO_INCREMENT ,
   nom VARCHAR(50) NULL
   , PRIMARY KEY (idservice)
 )
ENGINE=InnoDB;

# ------------------------------------------------------------------------------
#       TABLE : personnel
```

Structured Query Language file | length : 2 268 lines : 79 | Ln : 1 Col : 1 Pos : 1 | Windows (CR LF)

La base de données **mediatek86** est ensuite créée via l'interface graphique phpMyAdmin depuis Wamp. Dans l'onglet SQL, nous exécutons le script complet de la base de données que nous venons d'obtenir, afin de créer le « squelette » de notre base de données. Les tables *absence*, *motif*, *service* et *personnel* sont créées.

Toujours depuis phpMyAdmin, un nouvel utilisateur est créé, qui disposera des droits d'accès sur la base de données mediatek86. Son login est **mtmanager**, et son mot de passe **Aga,Ajtp86** (« Alouette gentille alouette, Alouette je te plumerai 86 »). Ces identifiants seront plus tard utilisés dans l'application pour se connecter à la base de données.

Ensuite, nous créons une nouvelle table dans la base de données, nommée *responsable*. Elle ne contiendra qu'un enregistrement, et mémorisera le nom et le mot de passe de l'unique responsable autorisé à accéder à la base de données. Nous choisissons **StaCla** comme login, et **Oskour!P1IdMDP** (« Au secours ! Pas 1 idée de mot de passe ») comme en guise de mot de passe, en chiffrant ce dernier avec l'algorithme SHA-2. Il apparaît donc chiffré dans la base de données :

Enfin, il reste à remplir les autres tables. Les tables *motif* et *service* sont remplies manuellement de données précises fournies par le client, tandis que les tables *personnel* et *absence* sont remplies de données générées aléatoirement par le site www.generatedata.com.

La base de données est maintenant prête à être utilisée.

2.2. Étape 2 : structuration de l'application

Dans cette étape, nous passons à l'application proprement dite, avec la structuration générale du programme selon le modèle MVC, puis un maquettage rapide de l'interface principale avant de réaliser son codage.

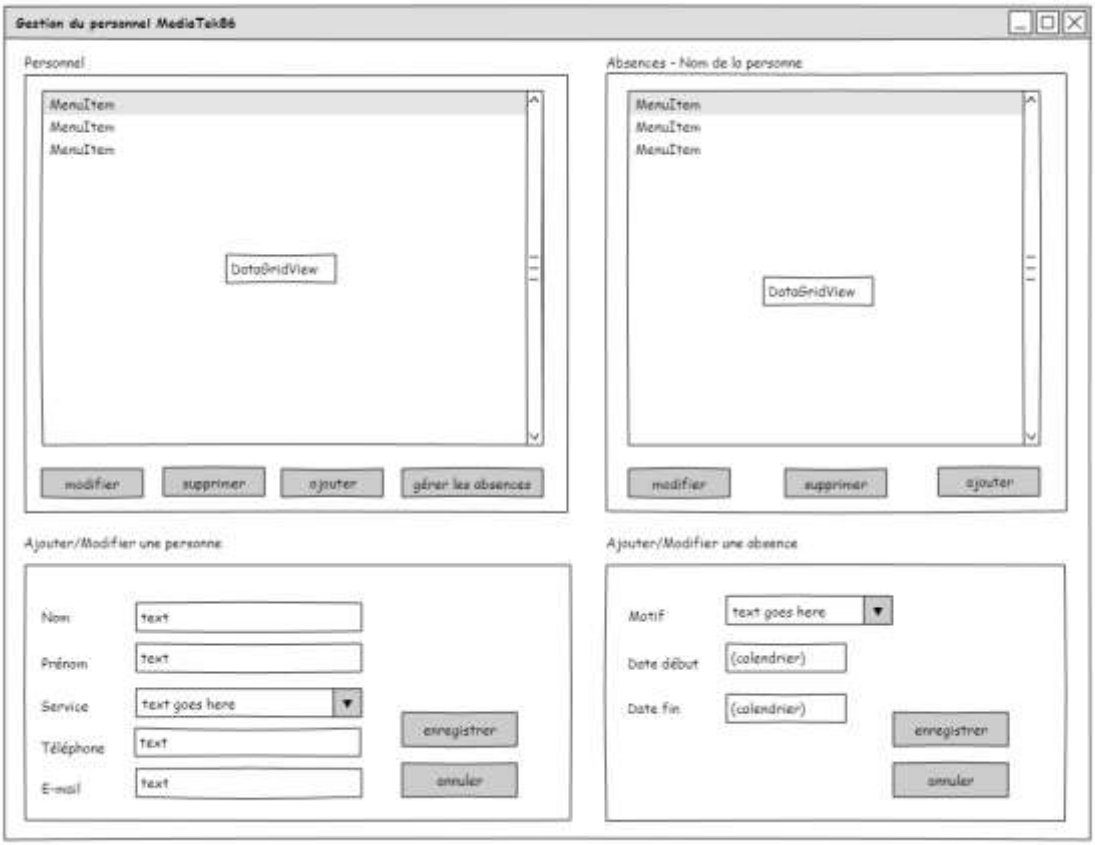
2.2.1. Création de la structure de l'application

Le modèle MVC implique de séparer en différents paquetages les classes dédiées au modèle (qui permettent la manipulation des données), celles de la vue (qui s'occupent de l'affichage) et celle du contrôleur (qui fait le lien entre vue et modèle). Par ailleurs, puisque notre application s'appuie sur

une base de données, un paquetage spécialisé dans la connexion à la base de données est créé. Enfin, nous prévoyons un paquetage *dal* pour *data access layer*, qui est un intermédiaire supplémentaire entre le contrôleur et cette classe qui gère la connexion. L'application restera donc jusqu'à la fin structurée selon les paquetages suivants :

2.2.2. Maquettage de la fenêtre principale de l'application

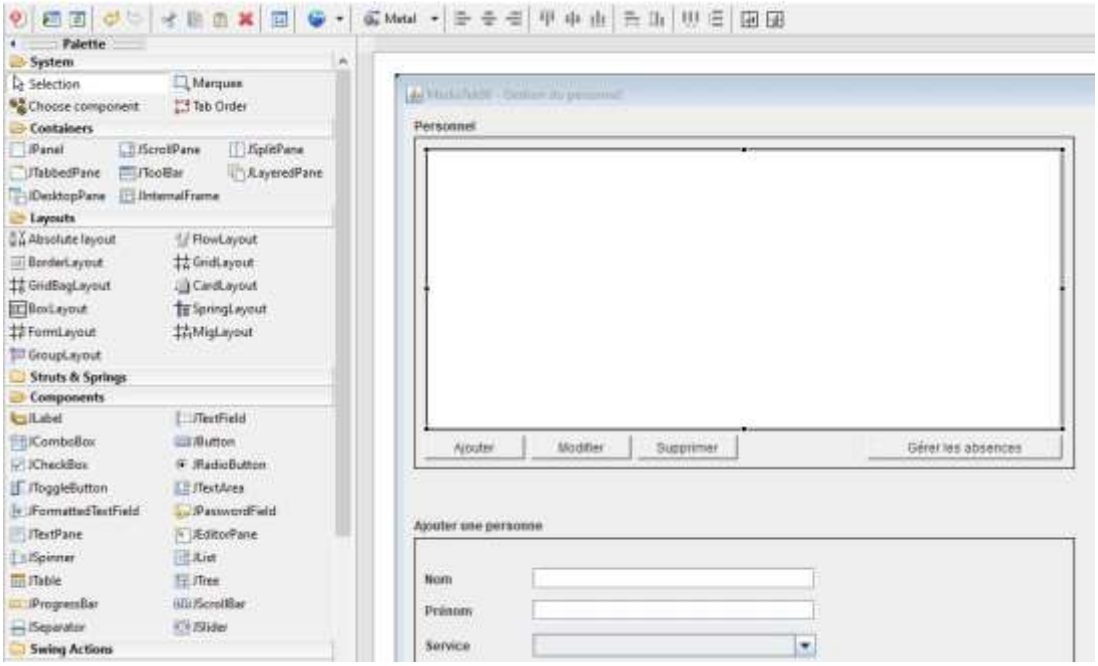
D'après la liste des cas d'utilisation déjà évoqués au point 1, nous élaborons cette maquette de la fenêtre principale, en utilisant le logiciel Pencil :



Notons qu'il est alors prévu de trouver en Java un équivalent à la classe DataGridView du framework .NET, ainsi qu'un calendrier pour sélectionner les dates. Nous verrons dans le point suivant que le codage de ces éléments s'est révélé plus complexe que prévu.

2.2.3. Codage de la fenêtre principale de l'application

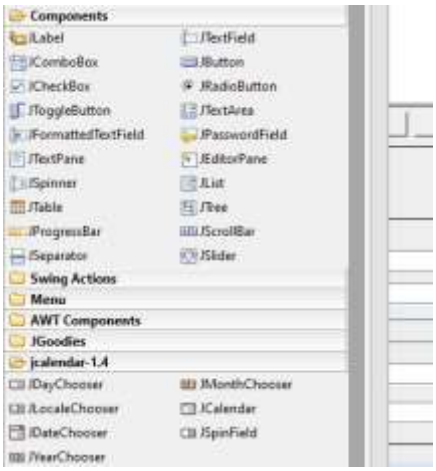
En s'appuyant sur cette maquette, nous créons graphiquement notre fenêtre principale grâce à l'interface WindowBuilder de l'IDE Eclipse.



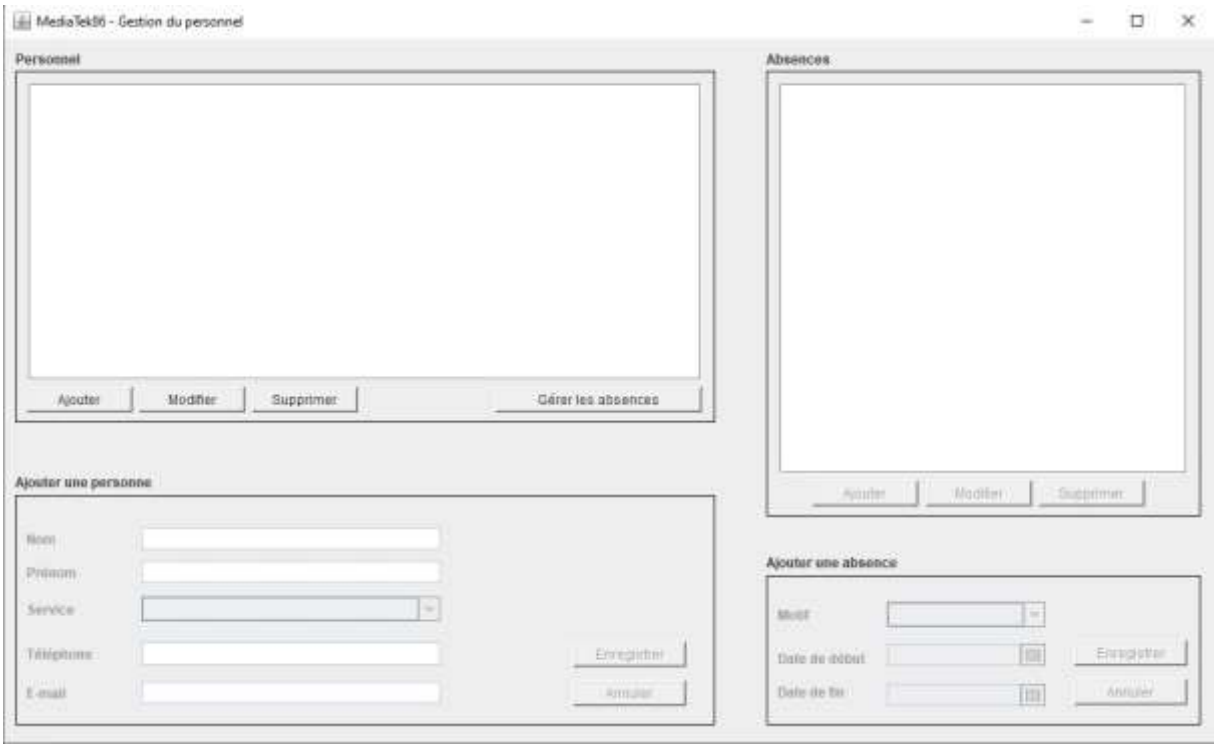
Si les boutons, champs de texte et labels ne posent pas de difficulté particulière, nous hésitons quant au choix de l'élément graphique destiné à accueillir les données du personnel. Nous choisissons temporairement la classe JList, en prévoyant de revenir sur la question au moment de l'importation des données depuis la base de données.

En revanche, il est plus difficile d'écarter la question du sélecteur de dates. Il apparaît que la palette de WindowBuilder ne propose pas d'élément ayant rapport de près ou de loin avec la gestion de dates. Différentes recherches sur le web montrent que la question est récurrente, et que plusieurs solutions existent. Nous choisissons simplement celle qui nous paraît la plus simple à mettre en œuvre sur le moment : JCalendar, disponible sur https://toedter.com/jcalendar.

Nous importons donc la bibliothèque *jcalendar-1.4.jar* dans le projet, et ajoutons ses composants directement dans la palette de WindowBuilder.



Nous optons donc pour deux éléments de type JDateChooser pour servir de sélecteurs de dates de début et de fin d'absence. À la fin de cette étape, la fenêtre principale a donc l'aspect suivant :



Par ailleurs, nous créons également une fenêtre plus modeste qui s'ouvrira à terme au lancement de l'application, afin de demander à l'utilisateur son nom et son mot de passe :

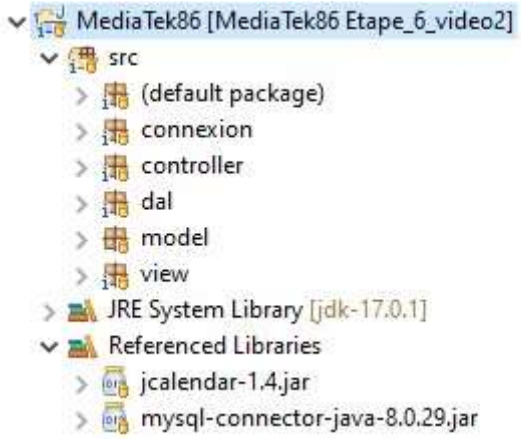
Le squelette du programme étant terminé, un dépôt GitHub est créé à l'adresse https://github.com/DevStaCla/MediaTek86, puis un dépôt Git local via Eclipse pour permettre par la suite la gestion des versions du projet.

2.3. Étape 3 : programmation des outils de connexion et du modèle

À l'étape précédente, nous avons préparé l'aspect visuel des fenêtres Gestion et Login du paquetage *view*. Nous passons maintenant à trois autres paquetages : *connexion*, *dal* et *model*.

2.3.1. Paquetage *connexion* : connexion à la base de données

Avant toute chose, il faut s'assurer de disposer des outils nécessaires à l'établissement d'une connexion à la base de données. En l'occurrence, nous avons besoin de la bibliothèque *mysql-connector-java*, qui vient s'ajouter aux bibliothèques du projet :



Le paquetage *connexion* contient une seule classe, ConnexionBDD, qui est un singleton, c'est-à-dire que la classe ne peut être instanciée qu'une seule fois. En effet, elle dispose d'une instance unique privée :

```
/**
 * instance unique de la classe
 */
private static ConnexionBDD instance = null;
```

Pour accéder à cette instance, on doit utiliser la méthode publique getInstance() :

```
/**
 * renvoie l'instance unique de la classe après l'avoir créée si besoin
 * @param connectionString chaîne de connexion
 * @param login login
 * @param pwd mot de passe
 * @return instance unique de la classe
 */
public static ConnexionBDD getInstance(String connectionString, String login, String pwd) {
    if (instance == null) {
        instance = new ConnexionBDD(connectionString, login, pwd);
    }
    return instance;
}
```

Le constructeur, qui est privé et ne peut être appelé qu'une seule fois, lorsque l'instance vaut *null*, est chargé d'établir la connexion :

```
/**
 * constructeur
 * @param connectionString chaîne de connexion
 * @param login login
 * @param pwd mot de passe
 */
private ConnexionBDD(String connectionString, String login, String pwd) {
    if(connexion == null) {
        try {
            connexion = DriverManager.getConnection(connectionString, login, pwd);
        }
        catch(SQLException e) {
            JOptionPane.showConfirmDialog(null,
                    "Impossible d'accéder à la base de données.\nL'application va être fermée.",
                    "Erreur",
                    JOptionPane.DEFAULT_OPTION,
                    JOptionPane.ERROR_MESSAGE,
                    null);
            System.exit(0);
        }
    }
}
```

Notons que cette classe ne « connaît » pas les informations d'accès à la base de données. Celles-ci (*connectionString*, *login* et *pwd*) lui seront transmises par la classe qui fera appel à ConnexionBDD.

Une fois la connexion établie, la classe ConnexionBDD est également chargée d'envoyer les requêtes SQL à la base de données, requêtes qui peuvent être de type « SELECT » (simple consultation de la base de données) ou autre (modification, suppression, insertion dans une table de la base de données). Deux méthodes différentes se chargent de ces tâches, requeteSelect() et requeteUpdate(). Dans les deux cas, on leur passe les mêmes arguments, la chaîne contenant la requête en SQL, et une liste de paramètres utilisée pour la préparation de requête :

```
/**
 * exécution d'une requête de type SELECT
 * @param requete requête MySQL
 * @param parametres paramètres pour la préparation de la requête
 */
public void requeteSelect(String requete, List<Object> parametres) {
```

Pour les requêtes de type SELECT, qui retournent une liste d'enregistrements, nous avons besoin en plus d'un objet qui servira de curseur pour parcourir la liste de résultats :

```
/**
 * curseur utilisé dans les requêtes de type SELECT
 */
private ResultSet curseur = null;
```

Nous verrons plus loin plus en détails le fonctionnement des méthodes de requêtes.

2.3.2. Paquetage *dal* : intermédiaire entre le contrôleur et la connexion

Le paquetage *dal* contient une classe unique, AccesDonnees, qui fera le lien entre le contrôleur et la connexion à la base de données. C'est cette classe qui dispose des informations de connexion :

```
/**
 * chaîne de connexion à la base de données
 */
private static String connectionURL = "jdbc:mysql://localhost:3306/mediatek86";
/**
 * login pour accéder à la base de données
 */
private static String login = "mtmanager";
/**
 * mot de passe pour accéder à la base de données
 */
private static String pwd = "Aga,Ajtp86";
```

À ce stade, les méthodes permettant de transmettre des requêtes à la base de données ne sont pas encore créées, elles s'écriront à l'étape suivante au fur et à mesure des besoins.

2.3.3. Paquetage *model* : classes métiers

Les objets qui seront manipulés par l'application sont déjà connus, puisqu'ils correspondent précisément aux tables de la base de données. Il y a donc dans ce paquetage autant de classes qu'il y a de tables : Absence, Motif, Personnel, Responsable et Service.

Ces cinq classes sont créées de la même manière. À titre d'exemple, prenons la classe Absence, qui correspond à la table la plus complexe : sa clé primaire est composée de deux attributs dont une clé étrangère (*idpersonnel*), et elle a une autre clé étrangère (*idmotif*). De plus, la table SQL contient des

champs de type DATE. Au final, la classe Absence, qui utilise java.util.Date, a cinq propriétés : quatre correspondent aux attributs de la table dans la base de données, et la dernière est issue d'une autre table, motif, et permet de stocker le libellé du motif d'absence en plus de son identifiant. Pour chaque propriété, on prépare également un *getter* pour en récupérer la valeur.

```
package model;

import java.util.Date;

/**
 * classe permettant la manipulation d'enregistrements de la table absence
 * @author Claire Stalter
 *
 */
public class Absence {
    /**
     * stocke la valeur du champ idpersonnel de la table absence
     */
    private int idpersonnel;
    /**
     * stocke la valeur du champ datedebut de la table absence
     */
    private Date datedebut;
    /**
     * stocke la valeur du champ datefin de la table absence
     */
    private Date datefin;
    /**
     * stocke la valeur du champ idmotif de la table absence
     */
    private int idmotif;
    /**
     * stocke la valeur du champ libelle de la table motif
     */
    private String motif;

    /**
     * getter sur idpersonnel
     * @return idpersonnel
     */
    public int getIdpersonnel() {
        return idpersonnel;
    }
```

Enfin, chacune des classes du modèle dispose d'un constructeur de ce type :

```
/**
 * constructeur
 * @param idpersonnel champ idpersonnel de la table absence
 * @param datedebut champ datedebut de la table absence
 * @param datefin champ datefin de la table absence
 * @param idmotif champ idmotif de la table absence
 * @param motif champ libelle de la table motif
 */
public Absence(int idpersonnel, Date datedebut, Date datefin, int idmotif, String motif) {
    this.idpersonnel = idpersonnel;
    this.datedebut = datedebut;
    this.datefin = datefin;
    this.idmotif = idmotif;
    this.motif = motif;
}
```

Avant de poursuivre avec la programmation des fonctionnalités, nous générons la documentation technique avec *javadoc*, et nous faisons un *commit and push* vers le dépôt GitHub. La documentation technique est disponible sur le dépôt ; pour l'ouvrir, il suffit d'ouvrir le fichier *index.html* présent dans le dossier *doc*, après avoir récupéré le dépôt sur son poste.

2.4. Étape 4 : programmation des fonctionnalités de l'application

La réalisation de cette étape se découpe assez naturellement en deux parties, la première étant la gestion du personnel et la seconde étant celle des absences. Cependant, nous avons rencontré des problématiques similaires dans les deux parties, et il est plus logique ici d'aborder les principaux défis de l'étape dans son ensemble : authentification de l'utilisateur, affichage du résultat d'une requête SELECT, prise en charge des autres requêtes, gestion des dates.

2.4.1. Fenêtre de connexion

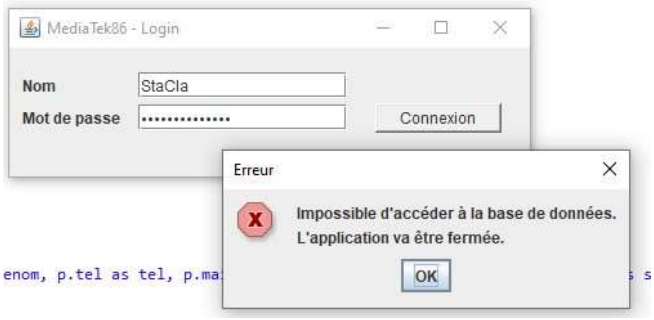
Dans la version finale de l'application, celle-ci s'ouvre sur la fenêtre de connexion déjà vue plus haut. L'utilisateur est invité à entrer son nom et son mot de passe, qui sont comparés au contenu de la table *responsable* de la base de données. Cependant, on se souvient que le mot de passe du responsable n'est pas stocké en clair dans la base de données, mais chiffré. Ici, il nous faut donc une méthode capable de chiffrer le mot de passe entré par l'utilisateur avec l'algorithme SHA-2, avant de le comparer avec la valeur contenue dans la table de la base de données. Cette méthode de chiffrement a été trouvée sur Internet :

```
/**
 * génération d'un mot de passe avec algorithme SHA-2
 * @param motdepasse mot de passe à hasher
 * @return hash SHA-2 du mot de passe
 */
public String getSHA256SecurePassword(String motdepasse) {
    String generatedPassword = null;
    try {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        byte[] bytes = md.digest(motdepasse.getBytes());
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < bytes.length; i++) {
            sb.append(Integer.toString((bytes[i] & 0xff) + 0x100, 16).substring(1));
        }
        generatedPassword = sb.toString();
    }
    catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
    return generatedPassword;
}
```

Au clic sur le bouton Connexion, un message s'affiche si le nom ou le mot de passe est incorrect :

Si le nom et le mot de passe correspondent bien à une entrée de la table *responsable*, la fenêtre de gestion du personnel s'ouvre.

À noter qu'un troisième cas de figure est possible. Si la connexion à la base de données ne peut se faire, l'application affiche un message d'erreur puis se ferme :



2.4.2. Affichage des résultats d'une requête de type SELECT

À l'ouverture de la fenêtre principale, la liste des membres du personnel doit s'afficher dans la partie gauche. Ce qui était au départ un objet JList est devenu par la force des choses un objet JTable, bien plus approprié pour afficher des lignes de plusieurs valeurs. Cependant, nous découvrons que son fonctionnement est assez éloigné de celui des objets DataGridView de .NET. Notamment, il ne semble pas possible de lier directement une liste d'objets (par exemple de type Personnel dans notre cas) à un objet JTable.

Côté vue, nous avons d'abord besoin de créer un « modèle », sorte de source depuis laquelle seront chargées les données de l'objet JTable. Notre méthode reçoit une liste d'objets de type Personnel, et retourne un modèle de tableau, qui comporte obligatoirement des noms de colonnes. Certaines colonnes, en réalité, ne devront pas apparaître à l'écran, mais on a tout de même besoin de conserver ces données :

```
/**
 * construction du modèle pour afficher les données de la table tblPersonnel
 * @param lePersonnel liste d'objets de type Personnel
 * @return modèle de données
 */
private DefaultTableModel modelTblPers(List<Personnel> lePersonnel) {
    String[] colonnes = new String[] { "idpers", "Nom", "Prénom", "Téléphone", "Mail", "idserv", "Service"};
    Object[][] donnees = new Object[lePersonnel.size()][7];
    for (int i = 0; i < lePersonnel.size(); i++) {
        donnees[i][0] = lePersonnel.get(i).getIdpersonnel();
        donnees[i][1] = lePersonnel.get(i).getNom();
        donnees[i][2] = lePersonnel.get(i).getPrenom();
        donnees[i][3] = lePersonnel.get(i).getTel();
        donnees[i][4] = lePersonnel.get(i).getMail();
        donnees[i][5] = lePersonnel.get(i).getIdservice();
        donnees[i][6] = lePersonnel.get(i).getService();
    }
    return new DefaultTableModel(donnees, colonnes);
}
```

Cette méthode est appelée par une autre, chargée d'afficher les données dans l'objet JTable :

```
/**
 * chargement de la liste du personnel dans la JTable tblPersonnel
 */
private void chargePersonnel() {
    List<Personnel> lePersonnel = controle.getPersonnel();
    tblPersonnel.setModel(modelTblPers(lePersonnel));
    tblPersonnel.getColumnModel().getColumn(0).setPreferredWidth(30);
    tblPersonnel.getColumnModel().getColumn(1).setPreferredWidth(30);
    tblPersonnel.getColumnModel().getColumn(2).setPreferredWidth(30);
    tblPersonnel.getColumnModel().getColumn(3).setPreferredWidth(60);
    tblPersonnel.removeColumn(tblPersonnel.getColumnModel().getColumn(0));
    tblPersonnel.removeColumn(tblPersonnel.getColumnModel().getColumn(4));
}
```

On voit notamment que des colonnes sont supprimées au moment de l'affichage : en effet, on ne veut afficher ni le champ *idpersonnel*, ni le champ *idservice*. On remarque d'ailleurs une subtilité : pour supprimer la première colonne, *idpersonnel*, on supprime bien la colonne d'id 0. Mais pour supprimer la colonne *idservice*, c'est l'id 4 qu'il faut indiquer, et non 5, puisque la colonne d'id 1 est devenue la colonne d'id 0…

Bien sûr, la vue doit obtenir la liste du personnel. Pour cela, elle fait appel au contrôleur, via la ligne :

```
List<Personnel> lePersonnel = controle.getPersonnel();
```

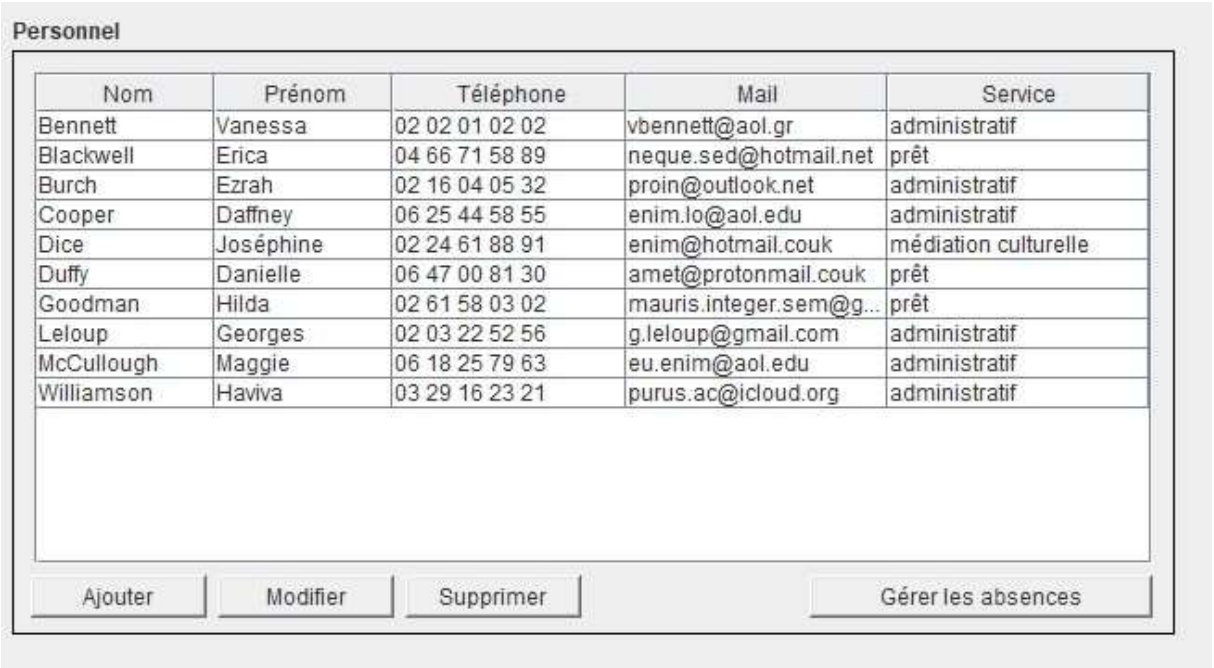
Quant au contrôleur, on a vu qu'il transmettait les requêtes à AccesDonnees. Côté contrôleur, dans la classe Controle, on a donc cette méthode :

```
/**
 * transfère au DAL la demande getPersonnel() de la vue
 * @return liste du personnel sous forme de ArrayList
 */
public List<Personnel> getPersonnel() {
    return AccesDonnees.getPersonnel();
}
```

C'est AccesDonnees qui prépare la requête proprement dite. Ici, il s'agit de retourner sous forme de liste tous les enregistrements de la table *personnel*. La méthode getPersonnel() de AccesDonnees prépare donc la requête SQL, récupére l'instance de connexion à la base de données, transmet à celle-ci sa requête via la méthode requeteSelect(), puis lit le « curseur » obtenu et range chacune des lignes dans un objet de type Personnel. La liste d'objets Personnel est renvoyée au contrôleur, qui peut la transmettre à la vue :

```
/**
 * demande à ConnexionBDD la liste d'enregistrements de la table personnel
 * @return liste du personnel sous forme de ArrayList
 */
public static List<Personnel> getPersonnel() {
    ArrayList<Personnel> lePersonnel = new ArrayList<>();
    String req = "select p.idpersonnel as idpersonnel, p.nom as nom, p.prenom as prenom, p.tel as tel,"
            + " p.mail as mail, s.idservice as idservice, s.nom as service ";
    req += "from personnel p join service s on (p.idservice = s.idservice) ";
    req += "order by nom, prenom;";
    ConnexionBDD conn = ConnexionBDD.getInstance(connectionURL, login, pwd);
    conn.requeteSelect(req, null);
    while (Boolean.TRUE.equals(conn.lireCurseur()))
    {
        Personnel personnel = new Personnel(
                (int)conn.champ("idpersonnel"),
                (String)conn.champ("nom"),
                (String)conn.champ("prenom"),
                (String)conn.champ("tel"),
                (String)conn.champ("mail"),
                (int)conn.champ("idservice"),
                (String)conn.champ("service"));
        lePersonnel.add(personnel);
    }
    conn.fermeCurseur();
    return lePersonnel;
}
```

Les données ainsi obtenues peuvent ensuite être affichées dans la vue :



2.4.3. Requêtes de modification de la base de données

L'application comporte plusieurs cas d'utilisation impliquant la modification de la base de données : ajout, modification, suppression d'enregistrements dans les tables *personnel* et *absence*. Selon la demande de la vue (au clic sur le bouton concerné), différentes méthodes du contrôleur vont être appelées pour transmettre les requêtes à AccesDonnees, qui se charge ensuite de préparer les requêtes proprement dites. Par exemple, cette méthode de AccesDonnees supprime toutes les absences d'un membre du personnel :

```
/**
 * demande à ConnexionBDD de supprimer des enregistrements dans la table absence
 * @param personnel membre du personnel dont on veut supprimer les absences
 */
public static void supprToutesAbsences(Personnel personnel) {
    String req = "delete from absence where idpersonnel = ?;";
    ArrayList<Object> lesParametres = new ArrayList<>();
    lesParametres.add(personnel.getIdpersonnel());
    ConnexionBDD conn = ConnexionBDD.getInstance(connectionURL, login, pwd);
    conn.requeteUpdate(req, lesParametres);
}
```

Ici, ce n'est plus la méthode requeteSelect() de ConnexionBDD qui est appelée, mais requeteUpdate(), puisqu'il s'agit d'une modification de la base de données. En plus de la chaîne contenant la requête, elle accepte une liste de paramètres. Voyons ce qui se passe côté ConnexionBDD :

```java
/**
 * exécution d'une requête autre que SELECT
 * @param requete requête MySQL
 * @param parametres paramètres pour la préparation de la requête
 */
public void requeteUpdate(String requete, List<Object> parametres) {
    if (connexion != null) {
        try {
            PreparedStatement commandeprep = connexion.prepareStatement(requete);
            if (parametres != null) {
                int i = 1;
                for (Object param : parametres) {
                    commandeprep.setObject(i, param);
                    i++;
                }
            }
            commandeprep.executeUpdate();
        }
        catch(SQLException e) {
            if (e.getErrorCode() == 1062) {
                JOptionPane.showConfirmDialog(null,
                        "Enregistrement impossible, car une autre absence ayant la même date"
                        + " de début existe déjà pour cette personne",
                        "Erreur",
                        JOptionPane.DEFAULT_OPTION,
                        JOptionPane.ERROR_MESSAGE,
                        null);
            }
            else {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

La méthode requeteUpdate réalise une préparation de la requête avec un objet PreparedStatement, en ajoutant à la requête, via la méthode générale setObject(), les paramètres qu'elle reçoit.

2.4.4. Gestion des dates

On a vu plus haut que nous avions choisi la bibliothèque JCalendar pour insérer des sélecteurs de date. À l'usage, il s'est révélé compatible avec les besoins de la mission et a donc été conservé dans la version finale. La gestion des dates a cependant soulevé quelques questions, notamment de conversion.

Le type java.util.Date semble directement compatible avec le format de date de SQL, et c'est donc celui que nous avons choisi pour stocker les dates des objets de la classe métier Absence. De plus, les entrées des éléments JDateChooser de la bibliothèque JCalendar sont également directement récupérables au format java.util.Date. Il nous a donc paru opportun de l'utiliser, même si certaines de ses méthodes sont considérées comme obsolètes et que java.time.LocalDate semble aujourd'hui davantage utilisé.

Il nous a toutefois posé un problème, car comme le précise la documentation officielle :

« The class `Date` represents a specific instant in time, with millisecond precision. »

Par conséquent, une date entrée dans un élément JDateChooser contient un temps par défaut (le moment précis de la saisie de la date) et non la date seule. Or, nous avons besoin de pouvoir comparer deux dates sans tenir compte d'une heure quelconque, pour obliger l'utilisateur à saisir une date de fin strictement postérieure à la date de début. C'est la raison pour laquelle nous avons recours à cette méthode, trouvée sur internet :

```
/**
 * mise à zéro de la partie "temps" d'un objet Date
 * @param date date dont on veut supprimer les données h, m, s, ms
 * @return date sans les valeurs h, m, s, ms
 */
private Date supprTemps(Date date) {
    Calendar cal = Calendar.getInstance();
    cal.setTime(date);
    cal.set(Calendar.HOUR_OF_DAY, 0);
    cal.set(Calendar.MINUTE, 0);
    cal.set(Calendar.SECOND, 0);
    cal.set(Calendar.MILLISECOND, 0);
    return cal.getTime();
}
```

Elle est utilisée uniquement lors de la récupération des dates des deux éléments JDateChooser, à l'enregistrement d'une absence :

```
/**
 * événement clic sur btnSaveAbs : sauvegarde d'absence après confirmation de l'utilisateur
 */
private void btnSaveAbs_clic() {
    if (dtcDateDeb.getDate() != null && dtcDateFin.getDate() != null) {
        Date datedebut = supprTemps(dtcDateDeb.getDate());
        Date datefin = supprTemps(dtcDateFin.getDate());
        // si la date de début est strictement antérieure à la date de fin
        if (datedebut.before(datefin)) {
```

Par ailleurs, se pose la question du format des dates. En effet, le format des dates simples (sans heure) dans SQL est YYYY-MM-DD. Dans le modèle de la JTable des absences, ainsi que dans les composants JDateChooser, nous les avons formatées pour que toutes les dates de l'application s'affichent au format DD/MM/YYY (FORMATDATE étant une constante de classe contenant cette chaîne) :

```
/**
 * construction du modèle pour afficher les données de la table tblAbsences
 * @param lesAbsences liste d'objets de type Absence
 * @return modèle de données
 */
private DefaultTableModel modelTblAbs(List<Absence> lesAbsences) {
    String[] colonnes = new String[] { "idpers", "Date de début", "Date de fin", "idmotif", "Motif"};
    Object[][] donnees = new Object[lesAbsences.size()][5];
    for (int i = 0; i < lesAbsences.size(); i++) {
        donnees[i][0] = lesAbsences.get(i).getIdpersonnel();
        donnees[i][1] = lesAbsences.get(i).getDatedebut();
        donnees[i][2] = lesAbsences.get(i).getDatefin();
        donnees[i][3] = lesAbsences.get(i).getIdmotif();
        donnees[i][4] = lesAbsences.get(i).getMotif();
        // formatage des dates
        SimpleDateFormat date = new SimpleDateFormat(FORMATDATE);
        donnees[i][1] = date.format(donnees[i][1]);
        donnees[i][2] = date.format(donnees[i][2]);
    }
    return new DefaultTableModel(donnees, colonnes);
}
```

En dehors de ces quatre points délicats, la programmation des fonctionnalités de l'application n'a pas présenté de problème particulier.

2.5. Étape 5 : création de la documentation utilisateur

Une vidéo d'un peu moins de 6 minutes, disponible sur notre portfolio, présente le fonctionnement de l'application de gestion du personnel de MediaTek86. Elle a été réalisée avec le logiciel gratuit Wink.

2.6. Étape 6 : déploiement de l'application

Le déploiement de l'application sur un poste nécessite deux choses : d'une part l'exécutable de l'application, d'autre part la mise en place de la base de données.

L'exécutable peut se créer de plusieurs manières. Ici, nous avons choisi d'utiliser Eclipse, en faisant un clic droit sur le nom du projet et en choisissant « Export… », puis « Runnable JAR file », c'est-à-dire une archive Java exécutable. L'exécutable obtenu, MediaTek86.jar, est disponible dans le dépôt GitHub du projet. Notons que, la version 17 du Java Development Kit étant celle installée sur la machine ayant servi au développement de l'application, le poste auquel l'application est destinée devra impérativement disposer de la version 17 ou supérieure du JDK.

Le script de restauration de la base de données a quant à lui été généré via phpMyAdmin. Avec un ajout manuel cependant, les quelques lignes destinées à créer un utilisateur ayant accès à la base de données mediatek86 :

```
--
-- Base de données : `mediatek86`
--
CREATE DATABASE IF NOT EXISTS `mediatek86` DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci;
CREATE USER 'mtmanager'@'%' IDENTIFIED BY 'Aga,Ajtp86';
GRANT USAGE ON *.* TO 'mtmanager'@'%';
GRANT ALL PRIVILEGES ON `mediatek86`.* TO 'mtmanager'@'%';
USE `mediatek86`;
```

Ce script de restauration, nommé DBRestore_MediaTek86.sql, est également disponible dans le dépôt GitHub du projet. Pour installer la base de données sur un poste, il est nécessaire d'installer Wamp puis d'exécuter le script complet dans la console SQL de phpMyAdmin.

3. Bilan final

La version finale de l'application paraît fonctionnelle et conforme aux attentes du client.

Le temps prévu pour l'activité était de 29 heures, mais nous pensons y avoir consacré 40 heures, du fait de certains problèmes rencontrés et déjà mentionnés. La découverte du fonctionnement des objets JTable a notamment exigé des recherches et des tâtonnements, de même pour le sélecteur de dates. Si les solutions choisies nous semblent acceptables dans le contexte de cet atelier, nous avons découvert les limites de la bibliothèque graphique Swing et savons que d'autres pistes sont à explorer à l'avenir.