
DOCUMENTATION TECHNIQUE



Projet « GSB Médecin »

TABLE DES MATIÈRES

1. Présentation du projet
 - 1.1. Contexte du projet
 - 1.2. L'entreprise GSB
 - 1.3. L'activité à gérer
 - 1.4. Mode restreint/complet
2. Fonctionnement de l'application
 - 2.1. Fonctionnement technique
 - 2.1.1. Structure de l'application
 - 2.1.1. L'architecture d'application « MVC »
 - 2.1.1. Le pattern des classes et leurs fonctionnements
3. Composants principaux
4. Diagrammes & UML

PRÉSENTATION DU PROJET

1.1. Contexte du projet

Le laboratoire Galaxy Swiss Bourdin (GSB) est amené dans ses différentes activités à contacter les médecins installés, par exemple les visiteurs médicaux du laboratoire se déplacent afin de présenter les nouveaux produits pharmaceutiques. Une base de données de médecins est utilisée dans ces différents processus. Cette base de données évolue fréquemment, ceci à cause de départ à la retraite de médecins ou d'installation de nouveaux praticiens. Les applications doivent intégrer ces évolutions des données.

Il a été décidé de confier à une société de services informatiques la responsabilité de développer deux applications donnant accès aux informations de la base de données.

1.2. L'entreprise GSB

Le laboratoire Galaxy Swiss Bourdin (GSB) est issu de la fusion entre le géant américain Galaxy (spécialisé dans le secteur des maladies virales dont le SIDA et les hépatites) et le conglomérat européen Swiss Bourdin (travaillant sur des médicaments plus conventionnels), lui même déjà union de trois petits laboratoires. En 2009, les deux géants pharmaceutiques ont uni leurs forces pour créer un leader de ce secteur industriel. L'entité Galaxy Swiss Bourdin Europe a établi son siège administratif à Paris. Le laboratoire Galaxy Swiss Bourdin (GSB) est issu de la fusion entre le géant américain Galaxy (spécialisé dans le secteur des maladies virales dont le SIDA et les hépatites) et le conglomérat européen Swiss Bourdin (travaillant sur des médicaments plus conventionnels), lui même déjà union de trois petits laboratoires. En 2009, les deux géants pharmaceutiques ont uni leurs forces pour créer un leader de ce secteur industriel. L'entité Galaxy Swiss Bourdin Europe a établi son siège administratif à Paris.

1.3. L'activité à réaliser

Un utilisateur (*connecté ou non*) doit pouvoir voir tous les médecins GSB (*avec leurs informations type : nom, prénom, adresse, numéro de téléphone, etc.*) ainsi que les pays et départements qui sont disponibles et créés dans la base de données.

L'activité à réaliser doit permettre de réaliser des actions **CRUD** (Create Read Update Delete) à l'utilisateur qui aurait les accès administrateurs, ce qui inclue donc également un système de connexion. C'est-à-dire que depuis l'application, ils auront comme possibilité d'ajouter, modifier ou bien supprimer un médecin, un pays ou bien un département.

Le développement complet de l'application doit être réalisé en utilisant une architecture **MVC** (Modèle-Vue-Contrôleur) ainsi qu'un langage de programmation orienté objet (**POO**), dans le cas présent, le langage utilisé sera du **Java**.

Suite à la prochaine page

PRÉSENTATION DU PROJET

1.4. Mode restreint/complet

L'application possède un **mode restreint**, mais également un **mode complet**. Le mode complet n'est accessible seulement, et seulement si l'utilisateur du logiciel a pu s'authentifier en tant que privilège administrateur. Pour cela certaines sécurités ont été mise en place, avec un chiffrement notamment du nom d'utilisateur et du mot de passe en format **SHA-512**. Une clé de déchiffrement de 512 caractères doit être utilisée pour pouvoir déchiffrer le mot de passe, à noter qu'une clé est pour le nom d'utilisateur et une autre a été créée pour le mot de passe afin de renforcer au maximum la sécurité. Une fois connecté et le **mode complet** activé, l'utilisateur pourra alors modifier, ajouter ou supprimer ce qu'il souhaite, que cela soit un Pays, un Département ou bien un Médecin. C'est ce qui permet donc de réaliser les actions **CRUD** demandé dans le projet.

Suite à la prochaine page

FONCTIONNEMENT DE L'APPLICATION

2.1. Fonctionnement technique

2.1.1. Structure de l'application

La structure de l'application est telle qu'elle répertorie les différents contrôleurs (*dans le dossier **controllers***), les différents modèles (*dans le dossier **models***) ainsi que les différentes vues (*dans le dossier **views** situé en bas de la structure, dans le dossier **resources***).

Chaque fichier travaille sur un ou plusieurs éléments liés à eux.

Par exemple, le **MedecinController** (*situé dans le dossier **controllers***), le **Medecin** ainsi que le **MedecinAccess** (*situés dans le dossier **models***) fonctionnent ensemble pour tout ce qui est lié de près ou de loin aux différents médecins.

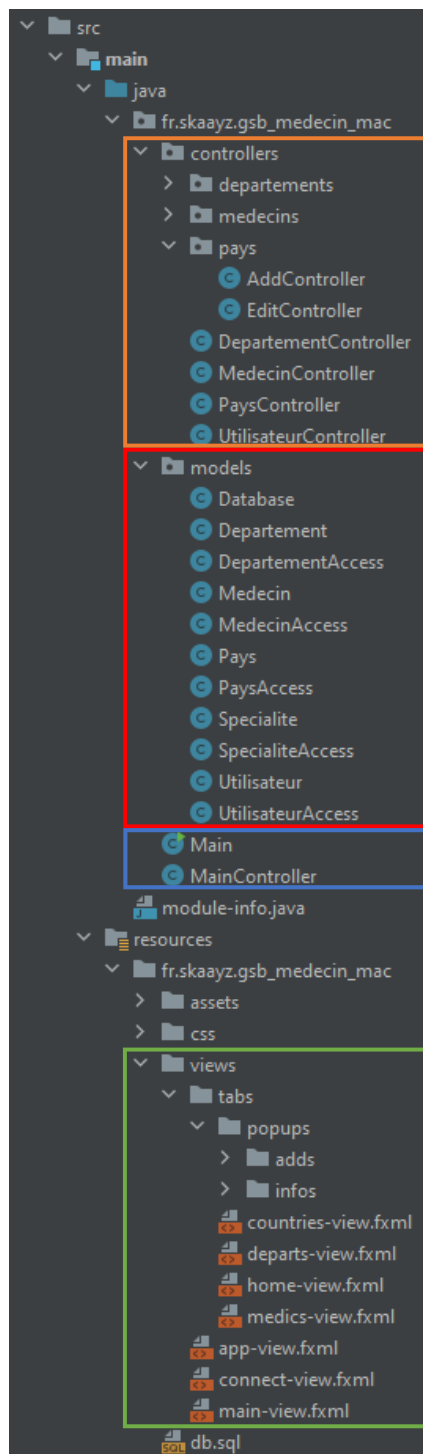
Dans ce même dossier, il est possible voir d'autre dossiers nommés **departements**, **medecins** et **pays**. Ils ont été créés afin de pouvoir stocker les fichiers **AddController** et **EditController** qui seront alors utilisés pour permettre de modifier ou ajouter du contenu dans la table y étant référée. (*Exemple : le **AddController** du dossier **pays** va permettre de travailler sur l'ajout d'autres pays*).

Toutes les différentes méthodes liées au sujet du fichier seront alors inscrites dedans et seulement dedans.

Le fichier **Database** sera la classe principale permettant la connexion à la base de données créée pour le projet, ainsi certaines méthodes à l'intérieur telle que « **execute()** » permettent alors d'effectuer des requêtes directement dans la BDD.

Dans la partie « **views** », quelques dossiers/fichiers sont alors visibles également. Premièrement, le fichier **main-view.fxml** est lié à la page principale qui est affichée lorsque le logiciel se lance. Ainsi, cela est pareil pour le **connect-view.fxml** qui affiche la page de connexion et **app-view.fxml** qui lui se charge d'afficher l'interface principale du logiciel.

Enfin, pour les fichiers **Main** et **MainController** ceux-ci sont nécessaires à l'exécution du projet, car c'est comme cela qu'il est configuré. Il va aller chercher un fichier **Main** et **MainController** qui eux vont permettre de gérer la partie principale de l'application. C'est-à-dire que par exemple, le **Main** et **MainController** vont contrôler le **main-view.fxml** et renverront vers d'autres vues qui utiliseront alors les contrôleurs prévus à cet effet.



FONCTIONNEMENT DE L'APPLICATION

2.1.2. L'architecture d'application « MVC »

Pour la création de ce logiciel, une architecture dites « MVC » (**Modèle-Vue-Contrôleur**) était demandée.

Dans cette architecture, le « **Modèle** » lui, va gérer toute la partie base de données de l'application. Son rôle principal est de récupérer les informations « brutes » venant de la BDD (*base de données*), de les organiser et de les renvoyer directement au **contrôleur** qui les aura au préalable demandé. Donc on y retrouve entre autres les différentes requêtes SQL, qui peuvent permettre de récupérer des informations comme cité plus haut, ou bien alors d'en envoyer via l'exécution de certaines requêtes comme des « **INSERT INTO** » ou bien « **UPDATE** ». L'entièreté des fichiers du dossier **models** sont développés en **Java**.

La « **Vue** » sera l'affichage du logiciel et des différentes pages. Elle ne fera aucun calcul et elle se contentera juste de faire afficher des informations qui pourront être modifiées depuis le contrôleur. (*Exemple, un texte placeholder pour le nom de famille d'un médecin qui pourra alors être modifié directement depuis le contrôleur des médecins*). L'entièreté des fichiers du dossier **views** sont développés en **FXML**. Cependant, un logiciel (*Scene Builder*) existe afin de faire seulement du **drag-n-drop**. Cela simplifie donc la mise en place d'un design pour les pages.

Le « **Contrôleur** » va être la passerelle entre la **vue** et le **modèle**, c'est par ici que la vue pourra alors être modifiée (*comme par exemple, comme cité plus haut, modifier un texte pour qu'il affiche la bonne valeur demandée*), le contrôleur va alors traiter les différentes demandes et les vérifier pour ensuite aller questionner le modèle et récupérer les informations initialement demandées. C'est un peu le « chef d'orchestre » de cette architecture. Il gère la partie logique du code et les décisions. L'entièreté des fichiers du dossier **controllers** sont développés en **Java**.

Suite à la prochaine page

FONCTIONNEMENT DE L'APPLICATION

2.1.3. Le pattern des classes et leurs fonctionnements

Partie contrôleur :

Durant cette partie, je vais décrire le fonctionnement « **basique** » d'une classe contrôleur dans mon logiciel, elle aura pour fonctionnalité principale d'exécuter des méthodes, avec notamment des « **getter** » mais aussi des « **setter** » qui prendront la forme de requêtes **SQL**, envoyée directement à la couche **modèle** de mon architecture **MVC**. Cette classe est nommée « **MedecinController** », j'ai décidé de la prendre en exemple car c'est celle qui a le plus de méthodes concrètes et différentes permettant de mieux comprendre le fonctionnement global de mon code et de ma manière de concevoir l'application. Enfin, nous verrons aussi très rapidement le « **UtilisateurController** » pour que puissiez savoir comment j'ai organisé mon système de connexion à l'application.

Fichier « **MedecinController** » :

```
package fr.skaayz.gsb_medecin_mac.controllers;

import fr.skaayz.gsb_medecin_mac.MainController;
import fr.skaayz.gsb_medecin_mac.controllers.medecins.EditController;
import fr.skaayz.gsb_medecin_mac.models.*;
import javafx.beans.property.SimpleStringProperty;
import javafx.collections.ObservableList;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.fxml.Initializable;
import javafx.scene.Cursor;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;
import javafx.util.Callback;

import java.io.IOException;
import java.net.URL;
import java.util.ResourceBundle;
```

Lors de la création d'un contrôleur, il est obligatoire d'importer tous les « **package** » qui seront alors utilisés dans le code. De base, toutes les fonctions ne sont pas implantées dans le code sur lequel nous travaillons, c'est à nous de dire que nous souhaitons importer tel ou tel « **package** » afin d'en utiliser ses fonctions. Par exemple, dans le cas précis le « `import javafx.fxml.FXML;` » va permettre d'importer toutes les méthodes relatives au **FXML**, ce qui me permettra alors par exemple de faire des modifications de texte sur une des pages, etc.

```
public class MedecinController implements Initializable {
    // Regular variables
    private int value = 0;

    // FXML variables
    @FXML
    private TextField search_textbar;

    @FXML
    private Button add_button;

    @FXML
    public TableView<Medecin> tableView;
```



```

@FXML
public TableColumn<Medecin, Integer> id;

@FXML
public TableColumn<Medecin, String> nom, prenom, adresse, tel,
departement, specialite;

@FXML
public TableColumn<Medecin, String> action;

```

Une fois que tous les « **package** » ont été importés, nous pouvons commencer à déclarer alors les différentes variables que va utiliser notre contrôleur, en commençant par ce que j'ai appelé les « **Regular variables** ». Celles-ci correspondent simplement aux variables qui n'utilisent pas de **FXML**, car celles utilisant du **FXML** seront alors dans la partie « **FXML variables** » et seront toujours déclarées avec un « **@FXML** » au-dessus d'elles, qui est obligatoire sinon le code ne l'interprètera pas comme du code **FXML**.

```

// FXML Functions
@FXML
private void searchButtonClicked() {
    setupTableView(MedecinAccess.getByLike(search_textbar.getText()));
}

@FXML
private void addButtonClicked() {
    // Create new stage
    Stage stage = new Stage();
    Pane scene_window = null;
    FXMLLoader loader = new
FXMLLoader(getClass().getClassLoader().getResource("fr/skaayz/gsb_medecin_mac/
views/tabs/popups/adds/medic-view.fxml"));

    try {
        scene_window = loader.load();
    } catch (IOException e) {
        e.printStackTrace();
    }

    // Set Scene
    assert scene_window != null; // scene_window might be null, so assert
    Scene scene = new Scene(scene_window);
    stage.setScene(scene);

    MainController.setupFrame(stage, scene, scene_window, true);

    stage.show();
}

```

Maintenant, nous allons pouvoir créer les fonctions/procédures (*nécessitant d'être déclarée en @FXML*) qui vont permettre pour le moment de faire un système de recherche (*dans la barre de recherche*) et un autre qui nous permettra d'afficher une nouvelle **Stage** (*fenêtre*) dès que l'utilisateur cliquera sur le bouton « + **Ajouter** » de l'onglet **Médecins**. Ces fonctions/procédures là sont rattachées à un bouton dans mon fichier **.fxml** qui a donc défini à l'avance que lorsque celui-ci sera cliqué, il exécutera une fonction.

Extrait du code « **medics-view.fxml** » :

```
<Button fx:id="search_button" layoutX="713.0" layoutY="5.0" minHeight="-Infinity" mnemonicParsing="false"
        onAction="#searchButtonClicked" prefHeight="25.0" prefWidth="133.0"
style="-fx-font-size: 13;"
        styleClass="button-blue" stylesheets="@../css/app.css"
text="Rechercher"/>

<Button fx:id="add_button" layoutX="-1.0" layoutY="5.0" minHeight="-Infinity"
mnemonicParsing="false"
        onAction="#addButtonClicked" prefHeight="25.0" prefWidth="94.0"
style="-fx-font-size: 13;"
        styleClass="button-validate" stylesheets="@../css/app.css" text="+
Ajouter">
```

Nous pouvons remarquer qu'il y a deux balises « **Button** », une pour lancer la recherche, l'autre pour l'ajout d'un nouveau pays. C'est grâce au paramètre « **onAction**='' '' » que je peux alors définir que ce bouton devra exécuter telle fonction quand il détectera une action (*l'action en question est en fait quand le bouton sera cliqué*).

Lors de l'exécution de la première procédure « **searchButtonClicked()** » située plus en haut, nous pouvons remarquer que j'ai utilisé une autre procédure nommée « **setupTableView()** ».

Fichier « **MedecinController** » :

```
// Regular functions
private void setupTableView(ObservableList<Medecin> medecins_list) {
    tableView.getItems().clear();
    tableView.refresh();

    id.setCellValueFactory(new PropertyValueFactory<>("id"));
    nom.setCellValueFactory(new PropertyValueFactory<>("nom"));
    prenom.setCellValueFactory(new PropertyValueFactory<>("prenom"));

    adresse.setCellValueFactory(new PropertyValueFactory<>("adresse"));
    tel.setCellValueFactory(new PropertyValueFactory<>("tel"));

    specialite.setCellValueFactory(c-> new
SimpleStringProperty(SpecialiteAccess.getLibelleByID(c.getValue().getSpecialite_id())));

    departement.setCellValueFactory(c-> new
SimpleStringProperty(DepartementAccess.getLibelleByID(c.getValue().getDepartement_id())));

    // Reload
    reload(medecins_list);
    tableView.setPlaceholder(new Label("Médecin(s) non trouvé(s)."));
}
```

Ici nous pouvons remarquer que la procédure « **setupTableView()** » nécessite un paramètre qui sera obligatoirement une « **ObservableList** » dans laquelle se retrouveront donc une liste d'objet « **Medecin** ». C'est grâce à cela que l'affichage de tous les médecins dans le tableau va pouvoir se faire. Une fois le paramètre créé, j'ai ensuite lancé deux autres procédures, nommée « **clear()** » et « **refresh()** », celles-ci sont à but « préventif » mais ne servent qu'à réinitialiser le tableau avant son utilisation au cas où il y aurait des valeurs qui se seraient intégrées sans l'avoir demandé au préalable.

Ensuite nous commençons à utiliser les variables type « **TableColumn** » que nous avons créées précédemment et nous leurs attribuons une valeur de cellule. C'est grâce à ce système que nous allons définir que telle cellule, faisant parti de la colonne « **id** » aura telle valeur, et ainsi de suite pour les variables « **nom** », « **prénom** », « **adresse** », « **tel** », « **specialite** » et « **departement** ».

Une fois les variables définies, nous allons « **reload()** » le tableau, donc le rafraîchir afin que toutes les dernières modifications soient prises en compte. Et dans le cas où rien ne s'afficherait (*problème dans l'ObservableList, rien dans la base de données, etc.*) nous allons afficher un texte dit « **placeholder** » dans lequel il est écrit qu'aucun médecin n'a été trouvé.

```
private void reload(ObservableList<Medecin> medecins_list) {
    tableView.getItems().clear();
    tableView.refresh();
    tableView.getItems().addAll(medecins_list);
}

@Override
public void initialize(URL location, ResourceBundle resources) {
    setupTableView(MedecinAccess.getAll());
}
```

Voici le code de la procédure « **reload()** », celle-ci prend également en compte une **ObservableList**, notamment celle qui aura été prédéfinie plus haut, nous utilisons la même. La procédure va simplement s'occuper de renettoyer la **TableView** avec les procédures « **clear()** » et « **refresh()** » déjà utilisée en haut, puis va ajouter la liste des médecins qui aura été donnée dans le tableau.

Pour finir, nous utiliserons la méthode « **initialize()** » qui a été importée grâce au package « **javafx.fxml.Initializable** ». Celle-ci va nous permettre de lancer de manière instantanée, dès la première utilisation du contrôleur, une procédure. Ici, « **setupTableView()** », qui va donc permettre que dès que l'utilisateur clique sur l'onglet « **Médecins** », alors tous les médecins soient chargés et affichés dans la **TableView** définie plus haut.

Voici la structure globale d'un contrôleur, il faut savoir que tous les autres contrôleurs « **PaysController** », « **DepartementController** » sont exactement pareils avec pour seules différences les paramètres des procédures/TableColumn, qui doivent donc retourner les objets prévus.

Suite à la prochaine page

Fichier « **UtilisateurController** » :

```
public class UtilisateurController {
    @FXML
    private TextField connect_username;

    @FXML
    private PasswordField connect_password;
```

```
@FXML
private javafx.scene.control.Button main_close_button;

@FXML
private Label connect_Infos;
```

Voici la déclaration des variables qui seront utilisées dans mon contrôleur de connexion. À noter que pour la troisième variable, il aurait été possible de simplement marquer « **private Button ..** » plutôt que « **private javafx.scene.control.Button** », cela est juste pour vous montrer qu'il est possible d'y arriver de plusieurs manières, en qu'en utilisant celle qui est écrite il n'est du coup plus utile d'importer le package « **javafx.scene.control.Button** ».

```
// FXML Functions
@FXML
private void connectButtonClicked(ActionEvent event) throws IOException {
    if(connect_username.getText().trim().isEmpty() ||
    connect_password.getText().trim().isEmpty()) {
        connect_Infos.setLayoutX(45);
        connect_Infos.setText("Tous les champs doivent être complétés");
    } else {
        String username = encryptToSHA512(connect_username.getText(), "key1");
        String password = encryptToSHA512(connect_password.getText(), "key2");

        Utilisateur user = UtilisateurAccess.getUtilisateur(username,
        password);

        if(user == null) {
            connect_Infos.setLayoutX(29);
            connect_Infos.setText("Nom d'utilisateur ou mot de passe
incorrect");
        } else {
            MainController.changePage("views/app-view.fxml", event);
            MainController.setIsOnSoftware(true);
        }
    }
}
```

Ici, nous voyons la procédure « **connectButtonClicked()** » qui sera lancée lorsque l'utilisateur cliquera sur le bouton « **Connexion** » de la page prévu à cet effet. Donc, quand l'utilisateur cliquera, nous allons d'abord vérifier si les entrées sont vides, c'est-à-dire que nous allons vérifier si le champs « **Nom d'utilisateur** » est vide OU alors si c'est le champs « **Mot de passe** » qui l'est. Si l'une des deux conditions est respectée, l'application retournera alors le message suivant :

Tous les champs doivent être complétés

Sinon, si les champs sont en effets remplis, nous allons pouvoir alors chiffrer le nom d'utilisateur et le mot de passe qui viennent d'être rentrés. J'ai utilisé le principe de chiffrement « **SHA-512** » afin de garantir un maximum de sécurité. Grâce à ce chiffrement, le nom d'utilisateur et mot de passe qui devaient initialement être « **admin** » et « **aaaa** » se transforment en :

« **a48c985784c99de0feb6f327c1ae08ae05c2cfae8550b463c0089001fb61728def6295253d1b4a06411c242a6c8adbe0e396bf9cc65d24ea83ab490c3cf217d** » pour le nom d'utilisateur ;

Et

« **d640a809dd1a77ceafbb50160941d250f8844c5c3517b51dcca7aca08c273c430f166433da954177c92f8f0557a65cdc59256d6b39e782ead57eb64fd2818319** » pour le mot de passe.

Il est utile de noter que dans la procédure permettant le chiffrement, appelée « **encryptToSHA512** », une clé est également définie pour permettre de déchiffrer le code si besoin est. Dans le cas présent, la clé de déchiffrement pour le nom d'utilisateur est « **key1** », et celle du mot de passe est « **key2** ». Bien sûr, là aussi j'ai pu générer des clés aléatoires de 512 caractères afin que celles-ci soient introuvables, mais pour les bienfaits de cette documentation, je n'ai pas pu les rentrer sinon le code devenait illisible.

Une fois que le nom d'utilisateur et mot de passe sont chiffrés, je vais alors pouvoir les comparer à ce qui est en base de données (*et les valeurs en base de données, sont également chiffrées, nous allons voir donc si le résultat du chiffrement du nom d'utilisateur « **admin** » et mot de passe « **aaaa** » est bien égal à celui qui est rentré dans la BDD*). Si c'est le cas, alors la fonction « **UtilisateurAccess.getUtilisateur()** » doit normalement nous renvoyer un objet « **Utilisateur** », sinon il renverra « **null** » et c'est grâce à cela que nous allons savoir si un utilisateur existe avec ces informations rentrées. Dans le cas où il existe, le contrôleur nous renverra sur la page principale du logiciel, dans le cas où l'utilisateur n'existe pas et donc que la fonction « **UtilisateurAccess.getUtilisateur()** » renvoie « **null** », le code va faire afficher un message d'erreur signifiant que le nom d'utilisateur ou le mot de passe est incorrect.

Nom d'utilisateur ou mot de passe incorrect

Code du « **UtilisateurController** » permettant le chiffage en **SHA-512** :

```
// Regular Functions
public String encryptToSHA512(String passwordToHash, String salt){
    String generatedPassword = null;
    try {
        MessageDigest md = MessageDigest.getInstance("SHA-512");
        md.update(salt.getBytes(StandardCharsets.UTF_8));
        byte[] bytes =
md.digest(passwordToHash.getBytes(StandardCharsets.UTF_8));
        StringBuilder sb = new StringBuilder();
        for (byte aByte : bytes) {
            sb.append(Integer.toString((aByte & 0xff) + 0x100,
16).substring(1));
        }
        generatedPassword = sb.toString();
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
    return generatedPassword;
}
```

FONCTIONNEMENT DE L'APPLICATION

2.1.3. Le pattern des classes et leurs fonctionnements

Partie modèle :

Durant cette partie, je vais expliquer le fonctionnement « **basique** » de trois classes modèles de mon logiciel, elles auront pour fonctionnalité principale d'exécuter des méthodes.

Fichier « **Departement** » :

```
package fr.skaayz.gsb_medecin_mac.models;

public class Departement {
    // Variables
    private int id;
    private String libelle;
    private Pays pays;

    // Constructor
    public Departement(int id, String libelle, Pays pays) {
        this.id = id;
        this.libelle = libelle;
        this.pays = pays;
    }

    // Functions
    public int getId() {
        return id;
    }

    public String getLibelle() {
        return libelle;
    }

    public Pays getPays() {
        return pays;
    }

    public int getPays_id() {
        return pays.getId();
    }
}
```

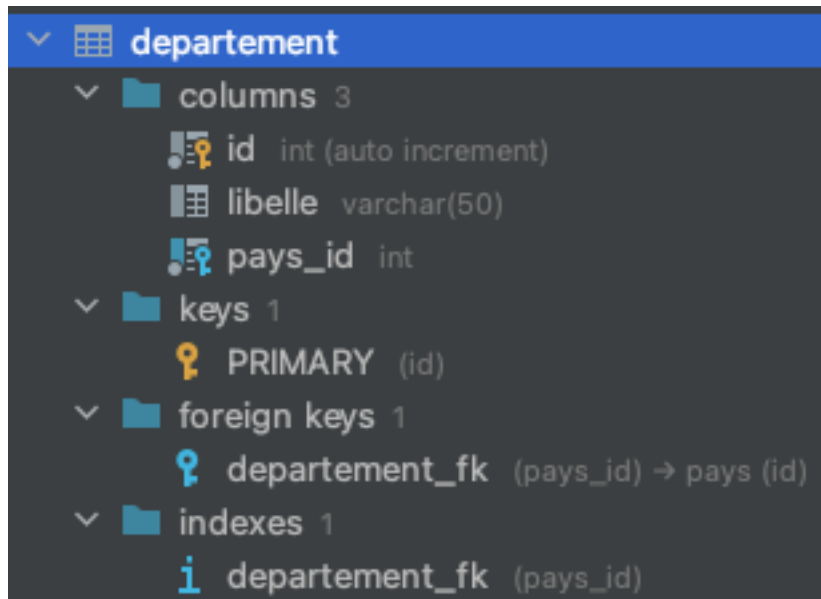
Pour la classe « **Departement** », qui va donc être utilisée dans la classe « **Medecin** » mais également pour l'affichage de tous les départements dans l'onglet « **Départements** » du logiciel, il faut (*et comme pour toutes les classes*) lui attribuer le package qui va en direction du dossier dans lequel il se trouve, ici le dossier « **models** ».

Après l'ajout du package, nous allons pouvoir créer la classe et lui attribuer les variables qui seront utilisées.

```
// Variables
private int id;
private String libelle;
private Pays pays;
```

Ici, nous allons dire que la classe département aura 3 variables, une concernant son id, une autre son libellé et finalement la dernière concernera un objet « **Pays** » qui sera récupéré grâce à au « **pays_id** » qui est une clé étrangère de la table « **département** » car un département est forcément relié à un pays, alors nous devons définir à quel pays appartient ce département.

Extrait de la table « **département** » provenant de la BDD :



departement	
columns 3	
id	int (auto increment)
libelle	varchar(50)
pays_id	int
keys 1	
PRIMARY	(id)
foreign keys 1	
departement_fk	(pays_id) → pays (id)
indexes 1	
departement_fk	(pays_id)

Fichier « **Département** » :

```
// Constructor
public Département(int id, String libelle, Pays pays) {
    this.id = id;
    this.libelle = libelle;
    this.pays = pays;
}
```

Nous créons ensuite le constructeur, ici il n'y en aura qu'un seul et nous lui réattribuons donc les variables qui auront été au préalable créée (*celles en haut*). Il faut donc attribuer l'id, le libellé ainsi que le pays du département.

```
// Functions
public int getId() {
    return id;
}

public String getLibelle() {
    return libelle;
}

public Pays getPays() {
    return pays;
}

public int getPays_id() {
    return pays.getId();
}
```

Et nous développons finalement les méthodes « **getter** » que la classe utilisera. À noter que j'ai créé une méthode supplémentaire permettant de récupérer directement l'ID du pays auquel le département appartient via un « **getPays_id()** » qui permet donc de retourner un entier (*int*) récupéré grâce à la méthode « **getId()** » faisant également parti de la liste des « **getter** » de la classe « **Pays** ».

Fichier « **Database** » :

```
package fr.skaayz.gsb_medecin_mac.models;

import java.sql.*;

public class Database {
    // Variables
    private static Connection conn_db = null;

    private static final String host_db = "127.0.0.1";
    private static final String name_db = "gsb_medecins";
    private static final String user_db = "root";
    private static final String pass_db = "";

    private static final String link_db = "jdbc:mysql://" + host_db + "/" +
name_db;

    // Functions
    private static Connection getConnection() {
        if(conn_db == null) {
            try {
                conn_db = DriverManager.getConnection(link_db, user_db, pass_db);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return conn_db;
    }

    protected static ResultSet query(String request) throws SQLException {
        Connection connect_db = getConnection();
        ResultSet queryOutput;

        Statement statement = connect_db.createStatement();
        queryOutput = statement.executeQuery(request);

        return queryOutput;
    }

    protected static void execute(String request) throws SQLException {
        Connection connect_db = getConnection();
        Statement statement = connect_db.createStatement();

        statement.execute(request);
    }
}
```

Voici une présentation du fichier « **Database** », et cela vous montre quel pattern j'utilise pour la création d'un code me permettant d'utiliser la base de données.

Premièrement, j'attribue le package qui va en direction du dossier dans lequel le fichier se trouve, ici le dossier « **models** » et je prends soin d'importer le package « **java.sql.*** ». Le « ***** » ici va permettre d'ajouter toutes les fonctions/procédures du package **SQL** de **Java**.

Je vais ensuite déclarer les variables qui vont me permettre d'initialiser la connexion à la BDD, avec donc la première variable qui sera de type « **Connection** » qui va donc être un objet de la connexion BDD que je vais pouvoir réutiliser afin de ne pas ré-instancier une connexion à la BDD à chaque fois, je réutilise seulement celle qui a été initialement créée.

Je définis ensuite les variables d'« **host** », du nom de la base de donnée ainsi que l'utilisateur et son mot de passe pour s'y connecter. Ces valeurs seront donc créées avec le typage « **String** ».

```
private static Connection getConnection() {
    if(conn_db == null) {
        try {
            conn_db = DriverManager.getConnection(link_db, user_db, pass_db);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    return conn_db;
}
```

Après cela, je vais créer la fonction de type « **Connection** » (le même type que pour la variable *conn_db*) nommée « **getConnection()** » qui va alors me permettre de récupérer la connexion si elle a déjà été instanciée ou alors de la créer et ensuite la récupérer selon les variables qui auront été au préalable mise en place.

```
protected static ResultSet query(String request) throws SQLException {
    Connection connect_db = getConnection();
    ResultSet queryOutput;

    Statement statement = connect_db.createStatement();
    queryOutput = statement.executeQuery(request);

    return queryOutput;
}

protected static void execute(String request) throws SQLException {
    Connection connect_db = getConnection();
    Statement statement = connect_db.createStatement();

    statement.execute(request);
}
```

Après cela, la connexion maintenant récupérable, nous allons l'utiliser dans les méthodes « **query()** » (qui servira à récupérer des informations depuis la BDD, avec des requêtes de type *SELECT*) et « **execute()** » (qui elle, servira à envoyer des informations, exécuter des requêtes type *INSERT INTO, UPDATE, DELETE, etc.*).

Pour chacune des méthodes, nous allons récupérer la connexion et la stocker dans une variable, ici **connect_db** de type « **Connection** ». Nous allons pouvoir donc par la suite créer des « **statement** » que nous utiliserons pour effectuer soit une requête visant à récupérer des informations, ou alors une requête visant à en envoyer.

Suite à la prochaine page

Fichier « **DepartementAccess** » :

```
package fr.skaayz.gsb_medecin_mac.models;

import javafx.collections.FXCollections;
import javafx.collections.ObservableList;

import java.sql.ResultSet;
import java.sql.SQLException;

public class DepartementAccess extends Database {
```

Nous allons voir maintenant une classe DAO (*Data Access Object*) qui va permettre de lancer des requêtes depuis des méthodes qui seront créées au préalable.

Premièrement, comme dans toutes les autres classes, il faut l'attribuer son package principal à celui de son propre dossier, ici « **models** ». Nous importerons alors tous les autres packages utiles à la création des différentes méthodes, et nous pourrons finalement créer la classe en prenant soin d'obligatoirement l'« **extends** » depuis la classe « **Database** » que j'ai montré en haut. Car les méthodes créées dans la classe « **Database** » sont en « **protected** », ce qui signifie qu'elles sont utilisables seulement par la classe elle-même ou bien ses enfants (*principe d'héritage*), en écrivant donc « **extends Database** » nous déclarons la classe « **DepartementAccess** » comme étant un enfant de « **Database** », ce qui fait qu'elle pourra utiliser les fonctions de son parent.

Pour les méthodes de cette classe, je vais simplement en montrer 3, la première permettra de récupérer une liste de départements, dans le cas actuel grâce à une requête de type « **SELECT * FROM departement ;** », la seconde permettra de récupérer seulement un département grâce à une requête « **SELECT * FROM departement WHERE id = 1 ;** » et enfin je montrerais également une méthode permettant d'exécuter une requête qui n'attend pas de retour, comme un « **DELETE** ».

Suite à la prochaine page

```
public static ObservableList<Departement> getAll() {
    ObservableList<Departement> departement_List =
    FXCollections.observableArrayList();

    try {
        ResultSet request = Database.query("SELECT * FROM departement;");

        while(request.next()) {
            departement_List.addAll(
                new Departement(
                    request.getInt("id"),
                    request.getString("libelle"),
                    PaysAccess.getCountryByID(request.getInt("pays_id"))
                )
            );
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return departement_List;
}
```

Dans cette méthode « **getAll()** », le principe va être récupérer la liste de tous les départements existants dans la base de données. Pour se faire, nous allons créer la méthode et lui indiquer que le type qu'elle devra retourner sera obligatoirement un « **ObservableList<Departement>** », comme ça ce que la méthode va retourner ne contiendra que des objets « **Departement** ».

Ensuite, nous allons effectuer un « **try/catch** » permettant donc d'effectuer une requête contenue dans la variable de type **ResultSet** nommée « **request** » et nous vérifions que **tant que** (*while*) la requête peut avoir du contenu (*tant que la requête retourne quelque chose*) alors nous allons créer un nouvel objet **Departement** grâce aux informations qui seront retournées et nous allons l'ajouter dans la liste des départements de type **ObservableList**. S'il y a une erreur lors de la requête, le « **catch** » nous informera de cela. Maintenant, il suffit de retourner le tableau qui est rempli (*ou non*) afin qu'on puisse le traiter.

```
public static Departement getDepartementByID(int id) {
    Departement departement = null;

    try {
        ResultSet request = Database.query("SELECT * FROM departement WHERE id
= " + id + ";");

        if (request.next()) {
            departement = new Departement(
                request.getInt("id"),
                request.getString("libelle"),
                PaysAccess.getCountryByID(request.getInt("pays_id"))
            );
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return departement;
}
```

Maintenant, nous allons voir comment retourner un seul objet notamment grâce à une requête SQL qui ne retournera obligatoirement qu'une seule valeur (*car dans le cas précis, nous cherchons un département grâce à son ID, ID qui est forcément unique*).

Pour commencer, nous créons la méthode en indiquant bien que le type du retour sera forcément un objet « **Departement** », ainsi nous rentrons également en paramètre de cette même méthode que l'utilisateur devra rentrer un entier (*int*) qui concernera donc l'id du département.

Nous créons une variable ayant comme type « **Departement** » que nous mettons sur **null**, car lorsque nous ferons la requête SQL, si celle-ci trouve un département selon son ID, alors la variable changera et ne sera plus égale à **null**, ce qui permettra à l'utilisateur lors du retour (*return*) de vérifier si la requête a trouvé ou pas un département, dans le cas où le département n'est pas trouvé elle renverra **null** car la variable ne sera pas touchée.

Après cette variable, nous allons refaire le même principe de « **try/catch** » et nous allons donc écrire la requête qui nous convient. Contrairement à la méthode « **getAll()** », nous allons pas vérifier que « **tant qu'il y a du contenu de retourné** » (*while*) mais simplement « **s'il y a du contenu qui est retourné** » (*if*) car nous avons juste besoin de vérifier s'il y a maximum 1 retour, et si c'est le cas, alors l'objet **Departement** sera créé dans la variable « **departement** » définie juste avant. Enfin, nous retournons la valeur de « **departement** ».

```
public static void deleteDepartmentByID(int id) {
    try {
        Database.execute("DELETE FROM departement WHERE id = " + id + ";");
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Pour l'exécution d'une requête, il suffit simplement de déclarer une nouvelle procédure avec comme paramètre (*dans le cas actuel*) l'ID du département que nous souhaitons supprimer.

Nous aurons alors simplement à effectuer un « **try/catch** » permettant de récupérer les erreurs possibles lors de la requête. Nous utiliserons alors la méthode « **execute()** » vue plus haut pour effectuer la demande SQL de type « **DELETE** ».

COMPOSANTS PRINCIPAUX

Les composants principaux de ce logiciel sont au nombre de 4.

J'ai tout d'abord utilisé « **gradle** » dans mon projet **Java**, celui-ci est ce qui est appelé un « **moteur de production** » fonctionnant sur la plateforme **Java**. Il permet notamment de construire des projets en **Java**, **Scala**, **Groovy**, voir **C++**. J'ai décidé de choisir **gradle** car celui-ci permet d'avoir les mêmes atouts qu'Apache Maven et **Apache Ant**. Il allie donc 3 moteurs de production en un.

J'ai également utilisé une architecture dite « **MVC** » comme cité plus haut, me permettant grâce à des fichiers de bien répertorier les différents travaux que ceux-ci sont censés effectuer. Par exemple, la vue pour l'affichage dans le dossier « **view** », le contrôleur servant de passerelle dans le dossier « **controller** », etc.

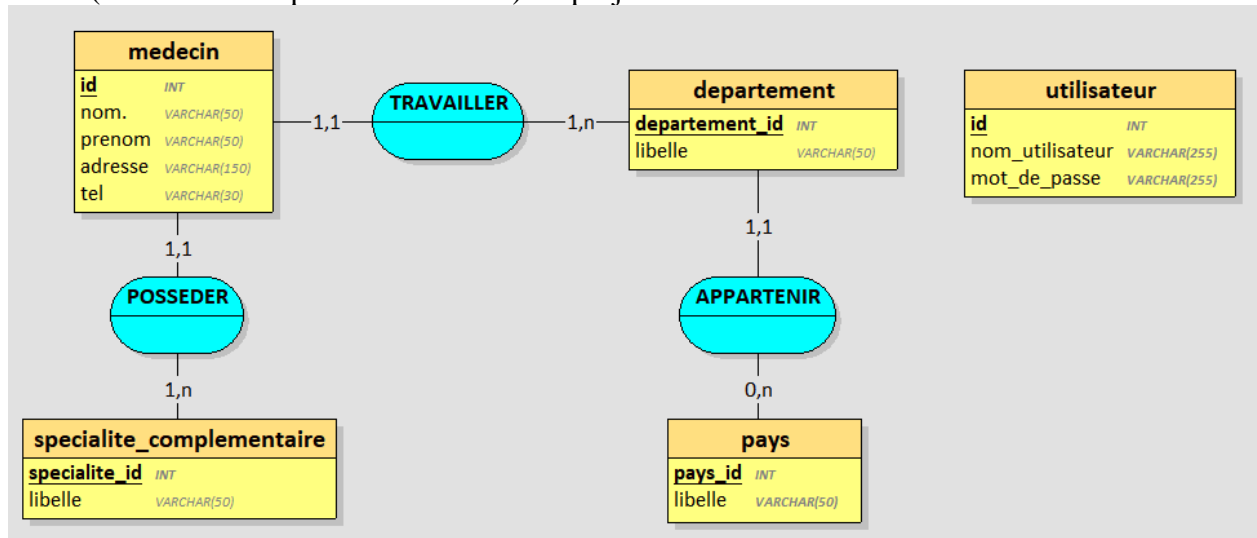
Également, j'ai utilisé la **librairie** (*bibliothèque*) nommée « **JavaFX** » pour réaliser toutes les fenêtres dont j'avais besoin, car cet outil dispose d'un autre nommé « **Scene Builder** » qui permet de créer ses propres vues, fenêtre grâce à du drag'n'drop. Par exemple, nous pouvons créer une fenêtre et décider qu'à tel endroit, il y aurait un bouton avec tel taille, telle couleur, etc. Un outil bien plus rapide pour la création de vue que de faire ça en HTML/CSS.

Et enfin, pour toute la partie **SQL** j'ai utilisé l'**API** (*Application Programming Interface*) **JDBC** (*Java DataBase Connectivity*) qui permet donc de se connecter à des bases de données, c'est-à-dire que **JDBC** constitue un ensemble de classes permettant de développer des applications capables de se connecter à des serveurs de bases de données (**SGBD**).

Suite à la prochaine page

DIAGRAMMES & UML

MCD (Modèle Conceptuel de Données) du projet :



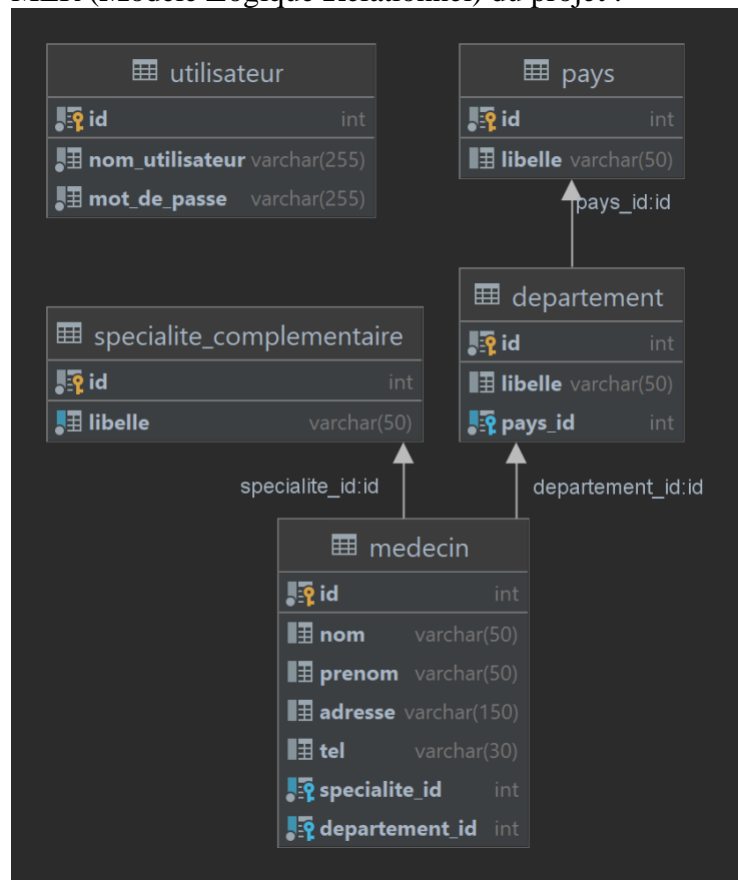
MLD (Modèle Logique de Données) du projet :

```

pays = (pays_id INT, libelle VARCHAR(50));
departement = (departement_id INT, libelle VARCHAR(50), #pays_id);
specialite_complementaire = (specialite_id INT, libelle VARCHAR(50));
medecin = (id INT, nom_ VARCHAR(50), prenom VARCHAR(50), adresse VARCHAR(150), tel VARCHAR(30), #departement_id, #specialite_id);
utilisateur = (id INT, nom_utilisateur VARCHAR(255), mot_de_passe VARCHAR(255));
  
```

Suite à la prochaine page

MLR (Modèle Logique Relationnel) du projet :



UML (Unified Model Language) du projet :

