

Rapport de l'application ESALAF

Réalisé par : Adam ZBITI

Encadré par : Pr.ELAACHAK,

Pr.NAIMI,

Pr.NDAMA,

Introduction:

Le mini-projet est une application de gestion de magasin qui s'appelle ESALAF développée en Java avec JavaFX et MySQL. L'application est conçue pour permettre aux utilisateurs de gérer les informations de leurs produits, clients, commandes et crédits. Les principales fonctionnalités incluent l'ajout, la suppression, la modification et l'affichage de produits, de clients, de commandes et de crédits.

Le projet est constitué de plusieurs classes qui gèrent différentes fonctionnalités, notamment la classe de connexion à la base de données, la classe de manipulation de données de produits, la classe de manipulation de données de clients, la classe de manipulation de données de commandes et la classe de manipulation de données de crédits.

La classe Main est responsable de l'affichage de l'interface utilisateur et de la gestion des événements. Les différentes fenêtres sont créées à partir de fichiers FXML et sont gérées par des contrôleurs distincts. Les contrôleurs utilisent les classes de manipulation de données pour récupérer et afficher les données dans l'interface utilisateur.

En résumé, le mini-projet est une application de gestion de magasin développée en Java qui permet aux utilisateurs de gérer les informations de produits, de clients, de commandes et de crédits. Les différentes fonctionnalités sont gérées par des classes distinctes et les données sont stockées dans une base de données MySQL.

La classe HelloApplication:

```
public class HelloApplication extends Application {  
    @Override  
    public void start(Stage stage) throws IOException {  
        FXMLLoader fxmlLoader = new  
FXMLLoader(HelloApplication.class.getResource("hello-view.fxml"));  
        Scene scene = new Scene(fxmlLoader.load(), 600, 400);  
        stage.setTitle("Hello!");  
        stage.setScene(scene);  
        stage.show();  
    }  
}
```

```

public static void main(String[] args) {
    launch();
}
}

```

Ce code Java est une classe principale pour une application JavaFX. Elle étend la classe **javafx.application.Application** et implémente la méthode **start()**, qui est appelée lors du lancement de l'application.

La méthode **start()** charge un fichier FXML à partir de la ressource "hello-view.fxml" et crée une scène JavaFX avec cette vue, en la redimensionnant à une taille de 600 x 400 pixels. Ensuite, elle crée une instance de **Stage** (la fenêtre principale de l'application), définit son titre, sa scène et l'affiche en appelant **stage.show()**.

La méthode **main()** est la méthode principale de l'application. Elle appelle simplement la méthode **launch()** de la classe **Application**, qui initialise l'application JavaFX et appelle la méthode **start()** pour lancer l'application.

En résumé, ce code crée une application JavaFX qui charge une vue à partir d'un fichier FXML et l'affiche dans une fenêtre principale de 600 x 400 pixels.

La classe HelloController:

La fonction login():

```

public void login() throws SQLException {
    String sql = "SELECT * FROM admin WHERE username = ? and password = ?";
    try{
        connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
        prepare= connect.prepareStatement(sql);
        prepare.setString(1,si_username.getText());
        prepare.setString(2,si_password.getText());
        result=prepare.executeQuery();
        Alert alert;

        if(si_username.getText().isEmpty() || si_password.getText().isEmpty()){
            alert = new Alert(Alert.AlertType.ERROR);
            alert.setTitle("Error Message");
            alert.setHeaderText(null);
            alert.setContentText("Please fill all terms");
            alert.showAndWait();
        }else {
            if(result.next()){
                alert = new Alert(Alert.AlertType.INFORMATION);
            }
        }
    }
}

```

```
alert.setTitle("Information message");
alert.setHeaderText(null);
alert.setContentText("Successfully login!!");
alert.showAndWait();
try{
    FXMLLoader fxmLoader = new
FXMLLoader(HelloApplication.class.getResource("/com/example/demo/dashboard.fxml"));
    Parent root = (Parent) fxmLoader.load();
    Stage stage = new Stage();
    stage.setTitle("Dashboard");
    stage.setScene(new Scene(root));
    stage.show();
    login_button.getScene().getWindow().hide();
}catch (Exception e){e.printStackTrace();}

}else {
    alert = new Alert(Alert.AlertType.ERROR);
    alert.setTitle("Error Message");
    alert.setHeaderText(null);
    alert.setContentText("Wrong username/password !!");
    alert.showAndWait();
}
}
}catch (SQLException e){
    e.printStackTrace();
}
}
```



Password



Ce code correspond à la méthode **login()**, qui est appelée lorsque l'utilisateur tente de se connecter à l'application. Elle est annotée avec **throws SQLException** pour indiquer qu'elle peut lever une exception de type **SQLException** (problème avec la base de données).

La méthode commence par définir une requête SQL pour récupérer les données d'identification d'un administrateur stockées dans une table **admin**. Elle crée ensuite une connexion à la base de données en utilisant le pilote JDBC pour MySQL et exécute la requête SQL en passant les valeurs des champs de saisie **si_username** et **si_password** pour les paramètres correspondants.

Si l'un des champs est vide, un message d'erreur est affiché à l'utilisateur en utilisant une boîte de dialogue **Alert** de type **ERROR**. Sinon, si la requête SQL renvoie un résultat, cela signifie que les informations d'identification sont correctes, et un message d'information est affiché avec une boîte de dialogue **Alert** de type **INFORMATION**. En outre, un fichier FXML **dashboard.fxml** est chargé et affiché dans une nouvelle fenêtre **Stage** pour accéder au tableau de bord de l'application. La fenêtre de connexion actuelle est masquée en utilisant la méthode **hide()**.

Si la requête SQL ne renvoie aucun résultat, cela signifie que les informations d'identification sont incorrectes, et un message d'erreur est affiché à l'utilisateur en utilisant une boîte de dialogue **Alert** de type **ERROR**.

En cas d'erreur de connexion à la base de données, l'exception **SQLException** est capturée et sa trace d'empilement est affichée en utilisant la méthode **printStackTrace()**.

La fonction signUP:

```
public void signup(){
    String sql = "INSERT INTO admin (email, username, password) VALUES (?, ?, ?)";
    try{
        connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf", "root", "");

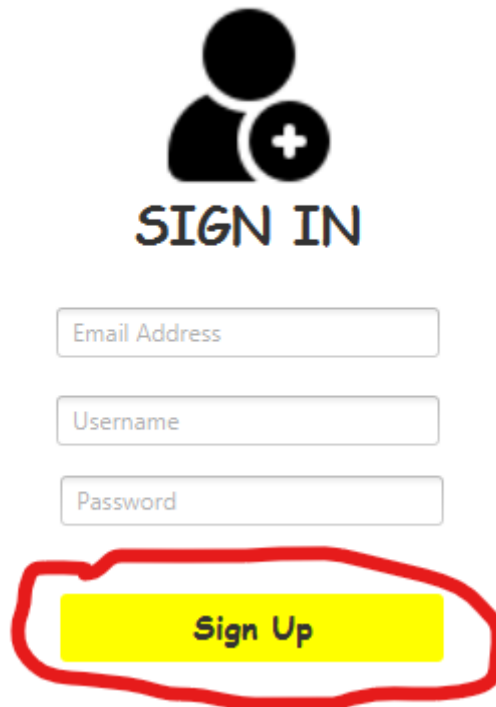
        Alert alert;

        if (su_username.getText().isEmpty() || su_address.getText().isEmpty() ||
            su_password.getText().isEmpty()){
            alert = new Alert(Alert.AlertType.ERROR);
            alert.setTitle("Error Message");
            alert.setHeaderText(null);
            alert.setContentText("Please fill all terms");
            alert.showAndWait();
        }else{
            if (su_password.getText().length() < 8){
                alert = new Alert(Alert.AlertType.ERROR);
                alert.setTitle("Error Message");
                alert.setHeaderText(null);
                alert.setContentText("password invalid");
                alert.showAndWait();
            }else {
                prepare = connect.prepareStatement(sql);
                prepare.setString(1, su_address.getText());
                prepare.setString(2, su_username.getText());
                prepare.setString(3, su_password.getText());
                alert = new Alert(Alert.AlertType.INFORMATION);
                alert.setTitle("Information Message");
                alert.setHeaderText(null);
                alert.setContentText("Successfully Signup");
                alert.showAndWait();
                prepare.executeUpdate();

                su_address.setText("");
                su_username.setText("");
                su_password.setText("");

            }
        }
    }
```

```
}catch (Exception e){e.printStackTrace();}  
}
```



The image shows a 'SIGN IN' form. At the top is a black icon of two people with a plus sign. Below it, the text 'SIGN IN' is displayed. The form consists of three input fields: 'Email Address', 'Username', and 'Password'. Below these fields is a yellow button labeled 'Sign Up', which is highlighted with a red hand-drawn outline.

Cette méthode Java est chargée de gérer le processus d'inscription. Elle ne prend aucun paramètre d'entrée et ne renvoie rien.

La première étape consiste à créer une instruction SQL INSERT qui insère un nouveau enregistrement admin dans la table **admin** de la base de données, avec les valeurs d'email, de nom d'utilisateur et de mot de passe fournies par l'utilisateur. L'instruction est enregistrée en tant que chaîne dans la variable **sql**.

Ensuite, la méthode tente de se connecter à la base de données MySQL à l'aide de la méthode **DriverManager.getConnection()**, et les détails de la base de données (hôte, port, nom, nom d'utilisateur et mot de passe) sont passés en tant que paramètres.

Ensuite, la méthode vérifie si les champs de texte **su_username**, **su_address** et **su_password** sont vides ou non à l'aide de la méthode **isEmpty()**. Si l'un de ces champs est vide, un message d'erreur est affiché à l'aide d'une boîte de dialogue **Alert** et la méthode retourne sans rien faire d'autre.

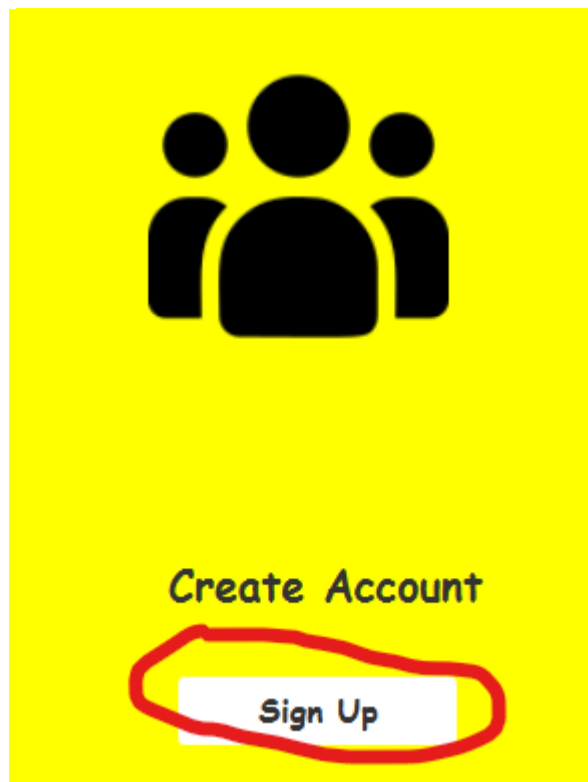
Si tous les champs requis sont remplis, la méthode vérifie si la longueur du mot de passe saisi est d'au moins 8 caractères. Si ce n'est pas le cas, un autre message d'erreur est affiché.

Si toutes les vérifications sont valides, la méthode prépare l'instruction SQL pour son exécution à l'aide de **prepareStatement()**, définit les valeurs d'e-mail, de nom d'utilisateur et de mot de passe à l'aide de **setString()**, puis exécute l'instruction à l'aide de **executeUpdate()**. Un message d'information est affiché à l'utilisateur à l'aide d'une

boîte de dialogue **Alert** pour indiquer que le processus d'inscription a réussi. Enfin, les champs de texte pour l'e-mail, le nom d'utilisateur et le mot de passe sont effacés.

La fonction signUpSlider():

```
public void signUpSlider(){  
    TranslateTransition slider1 = new TranslateTransition();  
    slider1.setNode(sub_form);  
    slider1.setToX(300);  
    slider1.setDuration(Duration.seconds(.5));  
    slider1.play();  
  
    slider1.setOnFinished((ActionEvent event) ->{  
        edit_label.setText("Login Account");  
  
        sub_Sign_button.setVisible(false);  
        sub_login_button.setVisible(true);  
    });  
}
```



Qui donne

Hello!



SIGN IN

Sign Up



Login Account

Sign In

Cette méthode "signupSlider()" est utilisée pour faire glisser un formulaire d'inscription vers la droite, et afficher un formulaire de connexion à la place. Elle utilise l'API JavaFX pour créer une transition d'animation, où le formulaire d'inscription est déplacé de sa position actuelle vers la droite de 300 pixels en 0,5 seconde.

Une fois que la transition est terminée, l'étiquette "edit_label" est mise à jour pour afficher "Login Account". Le bouton de connexion "sub_login_button" devient visible et le bouton d'inscription "sub_Sign_button" est masqué.

Cela permet de fournir une expérience utilisateur fluide et agréable, où les utilisateurs peuvent passer facilement du formulaire d'inscription au formulaire de connexion sans avoir à recharger la page ou ouvrir une nouvelle fenêtre.

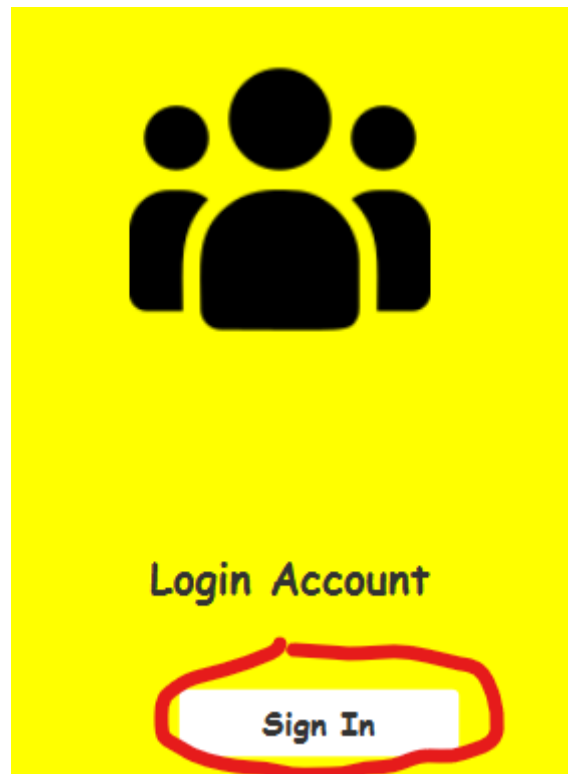
La fonction loginSlider():

```
public void loginSlider(){
    TranslateTransition slider1 = new TranslateTransition();
    slider1.setNode(sub_form);
    slider1.setToX(0);
    slider1.setDuration(Duration.seconds(.5));
    slider1.play();

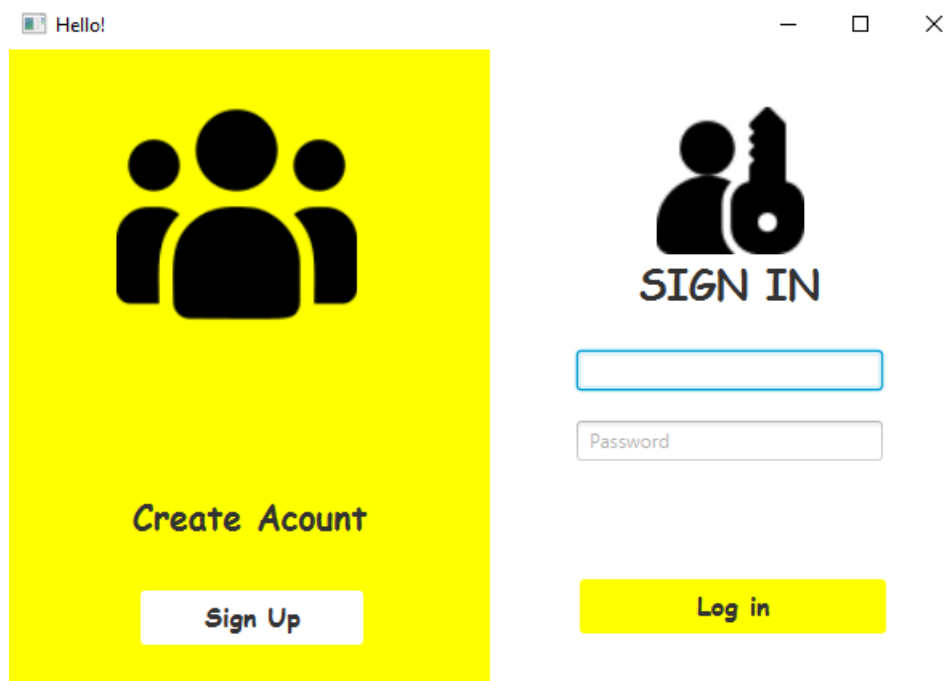
    slider1.setOnFinished((ActionEvent event) ->{
        edit_label.setText("Create Account");
        sub_Sign_button.setVisible(true);
```



```
sub_login_button.setVisible(false);  
});
```



Qui donne



Cette méthode est utilisée pour faire glisser le formulaire de connexion/inscription vers la gauche lorsque l'utilisateur clique sur le bouton "Create Account". Cela se fait à l'aide de la classe `TranslateTransition` de JavaFX qui permet de déplacer un nœud d'une position à une autre avec une animation de transition. Dans ce cas, la position initiale est 0 et la position finale est -300, ce qui signifie que le formulaire sera déplacé de 300 pixels vers la gauche. La durée de l'animation est de 0,5 seconde.

Après avoir terminé l'animation, le texte du label est mis à jour pour afficher "Create Account" et les boutons "Sign Up" et "Login" sont affichés ou cachés en fonction de ce qui est nécessaire pour l'opération en cours.

La classe `DashboardController`:

Interface de Client

La fonction `emptyFields()`:

```
public void emptyFields(){
    Alert alert;
    alert = new Alert(Alert.AlertType.ERROR);
    alert.setTitle("Error Message");
    alert.setHeaderText(null);
    alert.setContentText("Please fill all terms");
    alert.showAndWait();
}
```

La méthode "`emptyFields()`" crée une boîte de dialogue "Alert" de type "ERROR" qui affiche un message demandant à l'utilisateur de remplir tous les champs requis. Cette méthode est utilisée pour alerter l'utilisateur lorsqu'il a laissé des champs vides dans un formulaire ou une entrée de données.

La fonction `produitAddBtn()`:

```
public void produitAddBtn(){
    String Sql = "INSERT INTO produit (produit_id, name, price, category, status) "+"VALUES
    (?, ?, ?, ?, ?)";

    try{
        connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf", "root", "");

        if (produit_produitID.getText().isEmpty()
            || produit_name.getText().isEmpty()
            || produit_price.getText().isEmpty()
            || produit_category.getSelectionModel().getSelectedItem() == null
            || produit_status.getSelectionModel().getSelectedItem() == null)
```

```

{
emptyFields();
}else {

String checkData = "SELECT produit_id FROM produit WHERE produit_id =
"+produit_produitID.getText()+"";

statement = connect.createStatement();
result = statement.executeQuery(checkData);

if (result.next()){
Alert alert;
alert = new Alert(Alert.AlertType.ERROR);
alert.setTitle("Error Message");
alert.setHeaderText(null);
alert.setContentText("produit_id : "+produit_produitID.getText()+"was already taken!");
alert.showAndWait();
}else{
prepare= connect.prepareStatement(Sql);
prepare.setString(1,produit_produitID.getText());
prepare.setString(2,produit_name.getText());
prepare.setString(3,produit_price.getText());
prepare.setString(4, (String) produit_category.getSelectionModel().getSelectedItem());
prepare.setString(5, (String) produit_status.getSelectionModel().getSelectedItem());

Alert alert;
alert = new Alert(Alert.AlertType.INFORMATION);
alert.setTitle("Information Message");
alert.setHeaderText(null);
alert.setContentText("Successfully added");
alert.showAndWait();

//pour entrer data
prepare.executeUpdate();
//to update tableview
produitShowData();
//to clear data
produitClearButtton();
}
}

}catch (Exception e){e.printStackTrace();}

}

```

Produit ID

NOM

Prix

Catégorie

Status

ADD

RESET

UPDATE

DELETE

Cette méthode est utilisée pour ajouter des données dans une table "produit" dans une base de données MySQL.

La méthode commence par récupérer les données entrées par l'utilisateur dans les champs de texte, les menus déroulants, etc. Si l'un des champs est vide, la méthode "emptyFields" est appelée pour afficher un message d'erreur demandant à l'utilisateur de remplir tous les champs.

Ensuite, la méthode vérifie si l'identifiant du produit entré par l'utilisateur n'a pas déjà été utilisé en exécutant une requête SELECT sur la table "produit". Si l'identifiant du produit existe déjà, un message d'erreur est affiché.

Sinon, la méthode prépare une instruction SQL INSERT pour ajouter les données dans la table "produit". Elle utilise une PreparedStatement pour éviter les attaques par injection SQL. Une fois que les données ont été insérées dans la table, un message de confirmation est affiché et les champs de texte sont effacés.

Enfin, la méthode appelle la méthode "produitShowData" pour mettre à jour la table des produits affichée dans l'interface utilisateur.

La fonction produitUpdateBtn():

```

public void produitUpdateBtn(){
    String sql = "UPDATE produit SET name = "
    +produit_name.getText()+", Price = "
    +produit_price.getText()+", category = "
    +produit_category.getSelectionModel().getSelectedItem()+", status = "
    +produit_status.getSelectionModel().getSelectedItem()+"" WHERE produit_id = "
    +produit_produitID.getText()+"";
    try{
        connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
        if (produit_produitID.getText().isEmpty() || produit_name.getText().isEmpty()
        || produit_price.getText().isEmpty()
        || produit_category.getSelectionModel().getSelectedItem()==null
        || produit_status.getSelectionModel().getSelectedItem()==null)
        {
            emptyFields();
        }else{

            Alert alert;
            alert = new Alert(Alert.AlertType.CONFIRMATION);
            alert.setTitle("Confirmation Message");
            alert.setHeaderText(null);
            alert.setContentText("Are you sure you want to update"+produit_produitID.getText()+"?");
            Optional<ButtonType> option = alert.showAndWait();

            if(option.get().equals(ButtonType.OK)){
                prepare = connect.prepareStatement(sql);
                prepare.executeUpdate();

                alert = new Alert(Alert.AlertType.INFORMATION);
                alert.setTitle("Information Message");
                alert.setHeaderText(null);
                alert.setContentText("Successfully Updated");
                alert.showAndWait();

                //to update tableview
                produitShowData();
                //to clear data
                produitClearButtton();

            }else{
                alert = new Alert(Alert.AlertType.ERROR);
                alert.setTitle("Error Message");
                alert.setHeaderText(null);
                alert.setContentText("Update Canceled!!!");
            }
        }
    }
}

```

```

alert.showAndWait();
}
}

}catch(Exception e){e.printStackTrace();}

}

```

The image shows a web form for managing products. It has a light blue background. The form contains the following elements:

- Produit ID**: A text input field.
- NOM**: A text input field.
- Prix**: A text input field.
- Catégorie**: A dropdown menu with the text "Choisir..." and a downward arrow.
- Status**: A dropdown menu with the text "Choisir..." and a downward arrow.
- Buttons**: Four buttons are arranged vertically: "ADD", "RESET", "UPDATE", and "DELETE". The "UPDATE" button is circled in red.

Ce bloc de code est une méthode pour gérer l'événement lorsque l'utilisateur clique sur le bouton "UPDATE" pour mettre à jour un produit existant dans la base de données du système. La méthode commence par construire une instruction de mise à jour SQL en récupérant les valeurs des champs d'entrée sur l'interface utilisateur et en les concaténant dans une chaîne SQL.

Ensuite, elle vérifie si l'un des champs requis est vide et appelle la méthode `emptyFields()` pour afficher un message d'erreur si c'est le cas. Si tous les champs sont remplis, la méthode affiche une boîte de dialogue de confirmation demandant à l'utilisateur s'il est sûr de vouloir mettre à jour le produit. Si l'utilisateur clique sur "OK", la méthode met à jour la base de données avec les nouvelles données en utilisant une instruction préparée et affiche un message de réussite.

Enfin, la méthode produitShowData() est appelée pour rafraîchir la vue de la table avec les données mises à jour, et la méthode produitClearButton() est appelée pour effacer les champs d'entrée.

La fonction produitDeleteBtn():

```
public void produitDeleteBtn(){
    String sql = "DELETE FROM produit WHERE produit_id = '"+produit_produitID.getText()+"'";
    try{
        connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
        if (produit_produitID.getText().isEmpty())
        {
            emptyFields();
        }else{

            Alert alert;
            alert = new Alert(Alert.AlertType.CONFIRMATION);
            alert.setTitle("Confirmation Message");
            alert.setHeaderText(null);
            alert.setContentText("Are you sure you want to Delete '"+produit_produitID.getText()+"'?");
            Optional<ButtonType> option = alert.showAndWait();

            if(option.get().equals(ButtonType.OK)){
                prepare = connect.prepareStatement(sql);
                //to execute query
                prepare.executeUpdate();

                alert = new Alert(Alert.AlertType.INFORMATION);
                alert.setTitle("Information Message");
                alert.setHeaderText(null);
                alert.setContentText("Successfully Deleted");
                alert.showAndWait();

                //to update tableview
                produitShowData();
                //to clear data
                produitClearButton();

            }else{
                alert = new Alert(Alert.AlertType.ERROR);
                alert.setTitle("Error Message");
                alert.setHeaderText(null);
                alert.setContentText("Delete Cancelled!!");
                alert.showAndWait();
            }
        }
    }
```

```

}

}catch(Exception e){e.printStackTrace();}
}

```

The image shows a web form for managing products. It includes the following elements:

- Produit ID**: A text input field.
- NOM**: A text input field.
- Prix**: A text input field.
- Catégorie**: A dropdown menu with 'Choisir...' as the selected option.
- Status**: A dropdown menu with 'Choisir...' as the selected option.
- Buttons**: Four buttons labeled 'ADD', 'RESET', 'UPDATE', and 'DELETE' are arranged vertically. The 'DELETE' button is circled in red.

Ce bloc de code est une méthode pour gérer l'événement lorsque l'utilisateur clique sur le bouton "DELETE" pour supprimer un produit existant dans la base de données du système.

La méthode commence par construire une instruction SQL de suppression en récupérant la valeur de l'ID client à partir du champ texte sur l'interface utilisateur et en la concaténant dans une chaîne SQL.

Ensuite, la méthode vérifie si le champ d'ID client est vide et affiche un message d'erreur en appelant la méthode `emptyFields()` si c'est le cas. Si le champ est rempli, la méthode affiche une boîte de dialogue de confirmation demandant à l'utilisateur s'il est sûr de vouloir supprimer le client. Si l'utilisateur clique sur "OK", la méthode supprime le client de la base de données en utilisant une instruction SQL préparée et affiche un message de succès.

Enfin, la méthode `clientShowData()` est appelée pour rafraîchir la vue de la table avec les données mises à jour et la méthode `clientClearButton()` est appelée pour effacer les champs de saisie.

La fonction produitClearButton():

```
public void produitClearButtton(){
    produit_produitID.setText("");
    produit_name.setText("");
    produit_price.setText("");
    produit_category.getSelectionModel().clearSelection();
    produit_status.getSelectionModel().clearSelection();
}
```

Cette méthode "produitClearButton()" permet de vider les champs de saisie de données concernant un produit sur l'interface utilisateur (UI) du système. Plus précisément, elle réinitialise les champs suivants à une valeur vide :

- produit_produitID
- produit_name
- produit_price
- produit_category
- produit_status

Cela permet à l'utilisateur de saisir de nouvelles données pour un nouveau produit sans avoir à supprimer manuellement les anciennes données.

La fonction ObservableList<ProduitData> produitDataList():

```
public ObservableList<ProduitData> produitDataList(){
    ObservableList<ProduitData> ListData = FXCollections.observableArrayList();
    String sql = "SELECT * FROM produit";

    try{
        connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
        prepare= connect.prepareStatement(sql);
        result=prepare.executeQuery();
        ProduitData pd;
        while (result.next()){
            pd = new ProduitData(result.getInt("id"),result.getString("produit_id"),
            result.getString("name")
            ,result.getDouble("price"),result.getString("category"),
            result.getString("status"));
            ListData.add(pd);
        }
    }
```

```
}catch (Exception e){e.printStackTrace();}
```

```
return ListData;  
}
```

Ce bloc de code contient une méthode qui renvoie une liste observable d'objets `ProduitData`. La méthode récupère les données des produits à partir de la base de données locale en exécutant une requête `SELECT` sur la table "produit". Elle crée ensuite des objets `ProduitData` à partir des résultats de la requête et les ajoute à la liste observable.

La méthode renvoie finalement la liste observable de `ProduitData` contenant les informations sur tous les produits stockés dans la base de données locale.

La fonction `produitShowData()`:

```
public void produitShowData(){  
    produitListData = produitDataList();  
    produit_col_ID.setCellValueFactory(new PropertyValueFactory<>("produit_id"));  
    produit_col_name.setCellValueFactory(new PropertyValueFactory<>("name"));  
    produit_col_prix.setCellValueFactory(new PropertyValueFactory<>("price"));  
    produit_col_categorie.setCellValueFactory(new PropertyValueFactory<>("category"));  
    produit_col_status.setCellValueFactory(new PropertyValueFactory<>("status"));  
  
    produit_TableView.setItems(produitListData);  
}
```

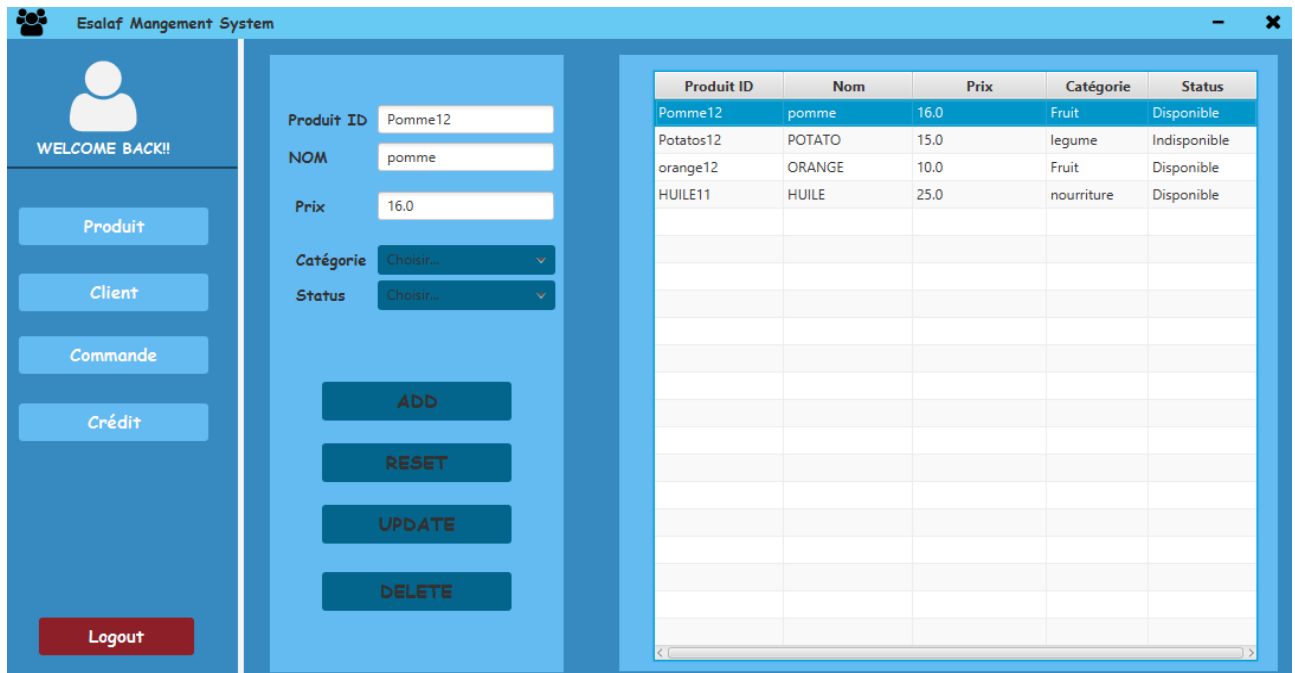
[illegible]

Cette méthode permet d'afficher les données des produits dans un tableau. Tout d'abord, elle récupère les données des produits en appelant la méthode `produitDataList()`. Ensuite, elle définit les valeurs des différentes colonnes du tableau en utilisant `PropertyValueFactory`, qui permet de mapper chaque colonne aux propriétés correspondantes dans la classe `ProduitData`.

Enfin, elle utilise la méthode `setItems()` pour affecter les données de la liste `produitListData` au `TableView`, ce qui permet d'afficher les données dans le tableau.

La fonction produitSelect():

```
public void produitSelect(){
    ProduitData pd = produit_TableView.getSelectionModel().getSelectedItem();
    int num = produit_TableView.getSelectionModel().getSelectedIndex();
    produit_produitID.setText(pd.getProduit_id());
    produit_name.setText(pd.getName());
    produit_price.setText(String.valueOf(pd.getPrice()));
}
```



Ce bloc de code définit une méthode nommée `produitSelect()`, qui est appelée lorsqu'un utilisateur sélectionne une ligne dans la table des produits (`produit TableView`).

La méthode commence par récupérer l'élément sélectionné (`pd`) et son index (`num`) à partir de la table. Ensuite, elle met à jour les champs d'entrée de l'interface utilisateur avec les valeurs correspondantes de l'élément sélectionné. Plus précisément, la méthode utilise les méthodes `setProduit_id()`, `setName()`, et `setPrice()` pour mettre à jour les champs de l'ID du produit, du nom et du prix, respectivement.

Cela permet à l'utilisateur de modifier les valeurs actuelles du produit sélectionné directement dans les champs d'entrée de l'interface utilisateur.

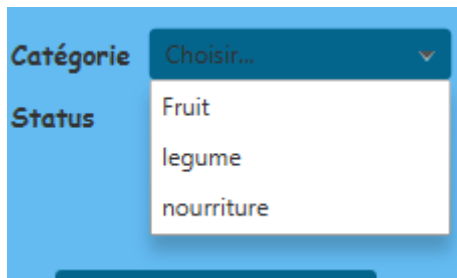
La fonction produitCategoryListe():

```
private String categories [] = {"Fruit", "legume", "nourriture"};

public void produitCategoryListe(){
    List<String> genderlist = new ArrayList<>();

    for (String data : categories){
        genderlist.add(data);
    }

    ObservableList ListData = FXCollections.observableArrayList(genderlist);
    produit_category.setItems(ListData);
}
```



Le code ci-dessus crée une liste de catégories pour les produits et l'affiche dans un menu déroulant sur l'interface utilisateur.

La variable **categories** est un tableau de chaînes de caractères qui contient les différentes catégories possibles pour les produits.

La méthode **produitCategoryListe()** itère sur ce tableau de chaînes de caractères et ajoute chaque élément à une liste d'objets **String** appelée **genderlist**.

Ensuite, la méthode crée une liste observable à partir de la liste **genderlist** à l'aide de la méthode **FXCollections.observableArrayList()**. Cette liste observable peut être utilisée pour fournir des données au menu déroulant dans l'interface utilisateur.

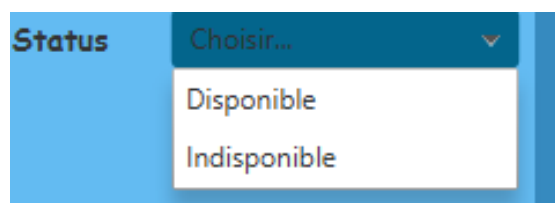
Enfin, la méthode définit la liste observable comme la source de données pour le menu déroulant en utilisant la méthode **setItems()** de l'objet **ComboBox** appelé **produit_category**.

La fonction produitStatusListe():

```
private String status [] = {"Disponible", "Indisponible"};

public void ProduitStatusList(){
    List<String> statuslist = new ArrayList<>();

    for (String data : status){
        statuslist.add(data);
    }
    ObservableList ListData = FXCollections.observableArrayList(statuslist);
    produit_status.setItems(ListData);
}
```



Ce code est une méthode qui permet de remplir la liste déroulante (ComboBox) du statut des produits avec les options "Disponible" ou "Indisponible".

La méthode commence par initialiser un tableau de chaînes de caractères contenant les deux options possibles : "Disponible" et "Indisponible". Ensuite, elle crée une liste vide pour stocker les options et boucle sur le tableau de chaînes pour ajouter chaque option à la liste.

Enfin, la méthode crée un objet ObservableList à partir de la liste remplie et l'ajoute à la liste déroulante du statut des produits à l'aide de la méthode setItems(). Cela permet à l'utilisateur de choisir le statut approprié pour chaque produit dans le formulaire de saisie.

Interface de Client

La fonction clientAddButton():

```
public void clientAddBtn(){
    String Sql = "INSERT INTO client (clientId, name, phone, produit, status, price) "+ "VALUES
    (?, ?, ?, ?, ?, ?)";

    try{
        connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf", "root", "");

        if (client_ID.getText().isEmpty()
            || client_name.getText().isEmpty()
            || client_phone.getText().isEmpty()
            || client_produit.getSelectionModel().getSelectedItem() == null
            || client_status.getSelectionModel().getSelectedItem() == null
            || client_price.getSelectionModel().getSelectedItem() == null)
        {
            emptyFields();
        } else {

            String checkData = "SELECT clientId FROM client WHERE clientId =
            '"+client_ID.getText()+"'";

            statement = connect.createStatement();
            result = statement.executeQuery(checkData);

            if (result.next()){
                Alert alert;
                alert = new Alert(Alert.AlertType.ERROR);
                alert.setTitle("Error Message");
                alert.setHeaderText(null);
                alert.setContentText("client_id : '"+client_ID.getText()+"' was already taken!");
                alert.showAndWait();
            } else {
```

```

prepare= connect.prepareStatement(Sql);
prepare.setString(1,client_ID.getText());
prepare.setString(2,client_name.getText());
prepare.setString(3,client_phone.getText());
prepare.setString(4, (String) client_produit.getSelectionModel().getSelectedItem());
prepare.setString(5, (String) client_status.getSelectionModel().getSelectedItem());
prepare.setString(6, (String) client_price.getSelectionModel().getSelectedItem());

Alert alert;
alert = new Alert(Alert.AlertType.INFORMATION);
alert.setTitle("Information Message");
alert.setHeaderText(null);
alert.setContentText("Successfully added");
alert.showAndWait();

//pour entrer data
prepare.executeUpdate();
//to update tableview
clientShowData();
//to clear data
clientClearButtton();
}
}

}catch (Exception e){e.printStackTrace();}

}

```

Ce code correspond à la méthode qui s'exécute lorsqu'on clique sur le bouton "ADD" pour ajouter un nouveau client dans la base de données.

Tout d'abord, la méthode vérifie que tous les champs nécessaires sont remplis en utilisant la méthode isEmpty() pour vérifier si les champs sont vides. Si un champ est vide, la méthode emptyFields() est appelée pour afficher une alerte informant l'utilisateur que certains champs sont vides.

Si tous les champs nécessaires sont remplis, la méthode vérifie si le clientID saisi n'existe pas déjà dans la base de données. Si le clientID existe déjà, une alerte est affichée pour en informer l'utilisateur.

Si le clientID n'existe pas encore, la méthode crée une requête SQL INSERT pour ajouter les données saisies dans la base de données. Les données sont récupérées à partir des champs texte et des menus déroulants correspondants. Ensuite, une alerte de confirmation est affichée pour informer l'utilisateur que les données ont été ajoutées avec succès.

Enfin, la méthode exécute la requête INSERT pour insérer les données dans la base de données, actualise le TableView affichant la liste des clients en appelant la méthode

clientShowData() et efface les données saisies dans les champs texte et les menus déroulants en appelant la méthode clientClearButton().

La fonction clientUpdateButton():

```
public void clientUpdateBtn(){
    String sql = "UPDATE client SET name = '"
    +client_name.getText()+"', phone = '"
    +client_phone.getText()+"', produit = '"
    +client_produit.getSelectionModel().getSelectedItem()+"', status = '"
    +client_status.getSelectionModel().getSelectedItem()+"', price = '"
    +client_price.getSelectionModel().getSelectedItem()+"' WHERE clientId='"
    +client_ID.getText()+"'";
    try{
        connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
        if (client_ID.getText().isEmpty()
        || client_name.getText().isEmpty()
        || client_phone.getText().isEmpty()
        || client_produit.getSelectionModel().getSelectedItem()==null
        || client_status.getSelectionModel().getSelectedItem()==null
        || client_price.getSelectionModel().getSelectedItem()==null)
        {
            emptyFields();
        }else{

            Alert alert;
            alert = new Alert(Alert.AlertType.CONFIRMATION);
            alert.setTitle("Confirmation Message");
            alert.setHeaderText(null);
            alert.setContentText("Are you sure you want to update"+produit_produitID.getText()+"?");
            Optional<ButtonType> option = alert.showAndWait();

            if(option.get().equals(ButtonType.OK)){
                prepare = connect.prepareStatement(sql);
                prepare.executeUpdate();

                alert = new Alert(Alert.AlertType.INFORMATION);
                alert.setTitle("Information Message");
                alert.setHeaderText(null);
                alert.setContentText("Successfully Updated");
                alert.showAndWait();

                //to update tableview
                clientShowData();
            }
        }
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
}
```



```
//to clear data
clientClearButtton();

}else{
alert = new Alert(Alert.AlertType.ERROR);
alert.setTitle("Error Message");
alert.setHeaderText(null);
alert.setContentText("Update Canceled!!!");
alert.showAndWait();
}
}

}catch(Exception e){e.printStackTrace();}

}
```

Ce code est une méthode qui est appelée lorsque le bouton "**UPDATE**" est cliqué. Cette méthode met à jour les informations d'un client dans la base de données en utilisant une requête SQL "UPDATE".

La méthode commence par créer une chaîne SQL en utilisant les valeurs des champs de texte et des menus déroulants. Ensuite, elle vérifie si tous les champs sont remplis. Si au moins un champ est vide, la méthode appelle la méthode "emptyFields" pour afficher un message d'erreur à l'utilisateur. Sinon, elle affiche une boîte de dialogue de confirmation pour que l'utilisateur puisse confirmer la mise à jour.

Si l'utilisateur confirme la mise à jour, la méthode prépare la requête SQL en utilisant la méthode "prepareStatement" et exécute la requête en utilisant la méthode "executeUpdate". Enfin, elle affiche un message d'information pour indiquer que la mise à jour a été effectuée avec succès. La méthode met ensuite à jour la table de données en appelant la méthode "clientShowData" et efface les champs en appelant la méthode "clientClearButton".

Si une exception est levée à n'importe quel moment de la méthode, elle est imprimée dans la console.

La fonction clientDeleteButton():

```
public void clientDeletBtn(){
String sql = "DELETE FROM client WHERE clientId = '"+client_ID.getText()+"'";
try{
connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
if (client_ID.getText().isEmpty())
{
emptyFields();
}
```

```

}else{

Alert alert;
alert = new Alert(Alert.AlertType.CONFIRMATION);
alert.setTitle("Confirmation Message");
alert.setHeaderText(null);
alert.setContentText("Are you sure you want to Delete "+client_ID.getText()+"?");
Optional<ButtonType> option = alert.showAndWait();

if(option.get().equals(ButtonType.OK)){
prepare = connect.prepareStatement(sql);
//to execute query
prepare.executeUpdate();

alert = new Alert(Alert.AlertType.INFORMATION);
alert.setTitle("Information Message");
alert.setHeaderText(null);
alert.setContentText("Successfully Deleted");
alert.showAndWait();

//to update tableview
clientShowData();
//to clear data
clientClearButtton();

}else{
alert = new Alert(Alert.AlertType.ERROR);
alert.setTitle("Error Message");
alert.setHeaderText(null);
alert.setContentText("Delete Cancelled!!");
alert.showAndWait();
}
}

}catch(Exception e){e.printStackTrace();}
}

```

Ce code est une méthode qui permet de supprimer une entrée dans une base de données MySQL.

La méthode commence par récupérer la connexion à la base de données en utilisant la méthode **DriverManager.getConnection**. Ensuite, elle vérifie si l'identifiant du client (**clientId**) est vide. Si c'est le cas, elle appelle une méthode **emptyFields()** qui est censée afficher un message d'erreur pour informer l'utilisateur que certains champs sont vides.

Sinon, elle affiche une boîte de dialogue de confirmation pour demander à l'utilisateur s'il est sûr de vouloir supprimer cette entrée. Si l'utilisateur clique sur "OK", la méthode prépare une instruction SQL de suppression (**DELETE**) avec l'identifiant du client saisi par l'utilisateur. Ensuite, elle exécute cette instruction SQL en utilisant la méthode **executeUpdate()**.

Si la suppression a réussi, un message d'information est affiché pour informer l'utilisateur. Enfin, la méthode met à jour le tableau d'affichage des données avec la méthode **clientShowData()** et efface les champs de saisie avec la méthode **clientClearButtton()**.

Si l'utilisateur clique sur "Annuler" dans la boîte de dialogue de confirmation, un message d'erreur est affiché pour informer l'utilisateur que la suppression a été annulée. Si une exception est levée lors de l'exécution de la méthode, elle est simplement imprimée à la console en utilisant la méthode **printStackTrace()**.

La fonction clientClearButtton():

```
public void clientClearButtton(){
    client_ID.setText("");
    client_name.setText("");
    client_phone.setText("");
    client_produit.getSelectionModel().clearSelection();
    client_status.getSelectionModel().clearSelection();
    client_price.getSelectionModel().clearSelection();
}
```

Cette fonction "clientClearButtton" est appelée pour effacer tous les champs de saisie et les sélections des menus déroulants liés aux informations des clients. Elle utilise les méthodes "setText()" pour effacer le contenu des champs de saisie et "getSelectionModel().clearSelection()" pour effacer les sélections des menus déroulants.

La fonction ObservableList<ClientData> clientDataList() :

```
public ObservableList<ClientData> clientDataList(){
    ObservableList<ClientData> ListData = FXCollections.observableArrayList();
    String sql = "SELECT * FROM client";

    try{
        connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
        prepare= connect.prepareStatement(sql);
        result=prepare.executeQuery();
        ClientData cd;
```

```

while (result.next()){
    cd = new ClientData(result.getInt("id"),result.getString("clientId"),
    result.getString("name")
    ,result.getString("phone"),
    result.getString("produit"),
    result.getString("status"),
    result.getDouble("price"));
    ListData.add(cd);
}

}catch (Exception e){e.printStackTrace();}

return ListData;
}

```

La méthode **clientDataList()** est utilisée pour récupérer les données des clients à partir de la base de données et les stocker dans une liste observable.

Tout d'abord, une nouvelle liste observable **ListData** est créée en utilisant la méthode **FXCollections.observableArrayList()**. Ensuite, la requête SQL "**SELECT * FROM client**" est préparée et exécutée pour récupérer toutes les données des clients à partir de la table **client**.

Les résultats sont ensuite parcourus à l'aide d'une boucle while et chaque ligne de résultat est stockée dans un objet **ClientData** créé à l'intérieur de la boucle. L'objet **ClientData** est créé à partir des valeurs récupérées de la base de données pour chaque colonne de la table **client**.

Enfin, chaque objet **ClientData** créé est ajouté à la liste observable **ListData** à l'aide de la méthode **add()**. La liste observable **ListData** est ensuite retournée.

La fonction clientShowData() :

```

public void clientShowData(){
    clientListData = clientDataList();
    client_col_ID.setCellValueFactory(new PropertyValueFactory<>("clientId"));
    client_col_name.setCellValueFactory(new PropertyValueFactory<>("name"));
    client_col_phone.setCellValueFactory(new PropertyValueFactory<>("phone"));
    client_col_produit.setCellValueFactory(new PropertyValueFactory<>("produit"));
    client_col_status.setCellValueFactory(new PropertyValueFactory<>("status"));
    client_col_prix.setCellValueFactory(new PropertyValueFactory<>("price" ));
}

```

```
client_tableview.setItems(clientListData);  
  
}
```

La méthode **clientShowData()** permet d'afficher les données clients dans un TableView dans l'interface graphique de l'application.

Dans cette méthode, on appelle la méthode **clientDataList()** qui retourne une liste observable d'objets **ClientData**. Ensuite, on assigne cette liste observable à la variable **clientListData**.

Ensuite, on configure les colonnes du TableView en utilisant la méthode **setCellValueFactory()** pour chaque colonne. On spécifie ainsi quelle propriété de l'objet **ClientData** correspond à chaque colonne.

Enfin, on ajoute les données de la liste observable au TableView en utilisant la méthode **setItems()**. Les données s'affichent alors dans le TableView avec les colonnes configurées précédemment.

La fonction clientSelect() :

```
public void clientSelect(){  
    ClientData cd = client_tableview.getSelectionModel().getSelectedItem();  
    int num = client_tableview.getSelectionModel().getSelectedIndex();  
    //if((num-1) < 1) return;  
    client_ID.setText(cd.getClientId());  
    client_name.setText(cd.getName());  
    client_phone.setText(cd.getPhone());  
  
}
```

Ce code est une méthode qui est appelée lorsque l'utilisateur sélectionne une ligne dans le TableView. Il récupère les données de la ligne sélectionnée en utilisant la méthode **getSelectedItem()** de la TableView et les stocke dans un objet de type **ClientData**. Ensuite, il récupère également l'index de la ligne sélectionnée en utilisant la méthode **getSelectedIndex()** de la TableView.

Ensuite, il met à jour les champs de texte correspondants (**client_ID**, **client_name** et **client_phone**) avec les valeurs de l'objet **ClientData** récupéré. Cela permet à l'utilisateur de voir les détails de la ligne sélectionnée dans les champs de texte correspondants pour une édition éventuelle.

La fonction clientProduitName() :

```
public void clientProduitName(){

String sql = "SELECT name FROM produit WHERE status = 'Disponible'";

try{
ObservableList listData = FXCollections.observableArrayList();
connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
prepare= connect.prepareStatement(sql);
result=prepare.executeQuery();

while (result.next()) {
listData.add(result.getString("name"));
}
client_produit.setItems(listData);
clientProduitPrice();

}catch (Exception e){
e.printStackTrace();
}

}
```

Ce code établit une connexion à la base de données "esalaf" et exécute une requête SELECT pour récupérer tous les noms des produits qui ont le statut "Disponible" de la table "produit". Ensuite, il ajoute chaque nom de produit à une liste observable appelée "listData". La liste observable est ensuite définie comme source de données pour le menu déroulant "client_produit". Enfin, il appelle une méthode "clientProduitPrice()" pour mettre à jour le prix du produit sélectionné. Si une exception est levée, l'exception est imprimée.

La fonction clientProduitPrice() :

```
public void clientProduitPrice(){
String sql = "SELECT price FROM produit WHERE name =
"+client_produit.getSelectionModel().getSelectedItem()+"";

try{
ObservableList listData = FXCollections.observableArrayList();
connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
prepare= connect.prepareStatement(sql);
result=prepare.executeQuery();
```

```

while (result.next()) {
listData.add(result.getString("price"));
}
client_price.setItems(listData);

}catch (Exception e){
e.printStackTrace();
}
}

```

Ce code permet de récupérer le prix du produit sélectionné dans la liste déroulante de produits ("client_produit") et d'afficher ce prix dans une autre liste déroulante ("client_price").

Tout d'abord, une requête SQL est construite pour sélectionner le prix de la base de données en fonction du nom du produit sélectionné. Cette requête est stockée dans une variable "sql".

Ensuite, une liste observable "listData" est initialisée pour stocker les résultats de la requête.

La méthode "getSelectedItem()" est utilisée pour obtenir le nom du produit sélectionné dans la liste déroulante "client_produit". Ce nom est utilisé pour construire la requête SQL et récupérer le prix correspondant dans la base de données.

La connexion à la base de données est établie, la requête est préparée, exécutée et les résultats sont stockés dans la liste "listData".

Enfin, la liste "listData" est utilisée pour remplir la liste déroulante "client_price" avec les prix récupérés de la base de données.

La fonction clientStatusList() :

```

private String statut [] = {"Maintenant", "Credit"};
public void ClienttStatusList(){
List<String> statuslist = new ArrayList<>();

for (String data : statut){
statuslist.add(data);
}
ObservableList ListData = FXCollections.observableArrayList(statuslist);
client_status.setItems(ListData);
}

```

La méthode **ClienttStatusList** remplit la liste déroulante (**ComboBox**) **client_status** avec les valeurs contenues dans le tableau **statut**. Elle crée une liste vide appelée **statuslist**, puis parcourt le tableau **statut** à l'aide d'une boucle **for-**

each. Pendant chaque itération de la boucle, elle ajoute l'élément courant du tableau à la liste **statuslist**. Une fois que tous les éléments du tableau ont été ajoutés à la liste, elle crée une liste observable à partir de **statuslist** et la définit comme source de données de la liste déroulante **client_status**. Ainsi, la liste déroulante affichera les éléments "Maintenant" et "Credit" comme options sélectionnables.

Interface de Commande

La fonction `ObservableList<ClientData> commandeDataList():`

```
public ObservableList<ClientData> commandeDataList(){
    ObservableList<ClientData> ListData = FXCollections.observableArrayList();
    String sql = "SELECT * FROM client ";

    try{
        connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
        prepare= connect.prepareStatement(sql);
        result=prepare.executeQuery();
        ClientData cd;
        while (result.next()){
            cd = new ClientData(result.getInt("id"),result.getString("clientId"),
            result.getString("name"),
            ,result.getString("phone"),
            result.getString("produit"),
            result.getString("status"),
            result.getDouble("price"));
            ListData.add(cd);
        }

    }catch (Exception e){e.printStackTrace();}

    return ListData;
}
```

Cette méthode permet de récupérer toutes les données des clients stockées dans la table "client" de la base de données "esalaf" et de les stocker dans une liste observable de type ClientData.

La méthode commence par initialiser une liste observable appelée "ListData" qui va contenir les données des clients. Ensuite, elle définit une requête SQL pour sélectionner toutes les données de la table "client".

La méthode établit ensuite une connexion à la base de données, prépare la requête SQL et exécute la requête pour récupérer les résultats. Ensuite, elle utilise une boucle "while" pour parcourir les résultats et stocke chaque résultat dans un objet ClientData, qui est ensuite ajouté à la liste observable "ListData".

Enfin, la méthode renvoie la liste observable contenant toutes les données des clients stockées dans la table "client".

La fonction commandeShowData:

```
public void commandeShowData(){
    commandeListData = commandeDataList();
    Commende_col_clientID.setCellValueFactory(new PropertyValueFactory<>("clientId"));
    Commende_col_nom.setCellValueFactory(new PropertyValueFactory<>("name"));
    Commende_col_phone.setCellValueFactory(new PropertyValueFactory<>("phone"));
    Commende_col_produit.setCellValueFactory(new PropertyValueFactory<>("produit"));
    Commende_col_status.setCellValueFactory(new PropertyValueFactory<>("status"));
    Commende_col_prix.setCellValueFactory(new PropertyValueFactory<>("price" ));

    Commende_tableview.setItems(commandeListData);

}
```

Cette méthode est utilisée pour afficher les données de la table "client" dans la vue "commande". Tout d'abord, elle appelle la méthode "commandeDataList()" pour récupérer les données à afficher. Ensuite, elle spécifie quelle colonne de la table "commande" sera associée à quelle propriété de la classe "ClientData" en utilisant les méthodes "setCellValueFactory()" et "PropertyValueFactory()". Enfin, elle associe les données récupérées à la table en utilisant la méthode "setItems()". Cela permet d'afficher les données de la table "client" dans la vue "commande".

La fonction commandeClientId():

```
public void commandeClientId(){

    String sql = "SELECT clientId FROM client WHERE status = 'Maintenant'";

    try{
```

```

ObservableList listData = FXCollections.observableArrayList();
connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
prepare= connect.prepareStatement(sql);
result=prepare.executeQuery();

while (result.next()) {
listData.add(result.getString("clientId"));
}
Commende_id.setItems(listData);
commendeClientName();
commendeProduitName();
commendeProduitPrice();

}catch (Exception e){
e.printStackTrace();
}

}

```

Ce code est une méthode pour afficher les identifiants des clients qui ont un statut **"Maintenant"**. Il exécute une requête SQL pour sélectionner les identifiants des clients dans la table "client" où le statut est égal à "Maintenant". Les résultats sont stockés dans une liste observable et affectés à la liste déroulante **"Commende_id"**. Enfin, il appelle d'autres méthodes pour charger les noms des clients, les noms des produits et les prix des produits correspondants à l'identifiant de client sélectionné dans la liste déroulante.

La fonction commendeClientName():

```

public void commendeClientName(){
String sql = "SELECT name FROM client WHERE clientId =
"+Commende_id.getSelectionModel().getSelectedItem()+"";

try{
ObservableList listData = FXCollections.observableArrayList();
connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
prepare= connect.prepareStatement(sql);
result=prepare.executeQuery();

while (result.next()) {
listData.add(result.getString("name"));
}
Commende_nom.setItems(listData);
}

```

```

}catch (Exception e){
e.printStackTrace();
}
}

```

Le code permet de récupérer le nom du client à partir de l'ID du client sélectionné dans la liste déroulante "Commende_id".

La requête SQL est construite en utilisant l'ID du client sélectionné pour sélectionner le nom correspondant dans la table "client". Les résultats de la requête sont stockés dans une liste observable, puis affichés dans la liste déroulante "Commende_nom".

La méthode "commendeClientName" est appelée après que l'utilisateur ait sélectionné un ID de client dans la liste déroulante "Commende_id", et elle récupère le nom du client correspondant en utilisant la requête SQL construite à partir de cet ID.

La fonction commendeProduitName():

```

public void commendeProduitName(){
String sql = "SELECT produit FROM client WHERE clientId =
"+Commende_id.getSelectionModel().getSelectedItem()+"";

try{
ObservableList listData = FXCollections.observableArrayList();
connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
prepare= connect.prepareStatement(sql);
result=prepare.executeQuery();

while (result.next()) {
listData.add(result.getString("produit"));
}
Commende_produit.setItems(listData);

}catch (Exception e){
e.printStackTrace();
}
}

```

Ce code est une méthode qui permet de récupérer les noms des produits achetés par un client en fonction de son identifiant (clientId).

Tout d'abord, la méthode crée une requête SQL qui sélectionne le produit de la table "client" où l'identifiant du client correspond à celui sélectionné dans la liste déroulante Commende_id.

Ensuite, la méthode exécute cette requête et récupère le résultat dans un objet ResultSet.

La méthode boucle sur le ResultSet pour extraire les noms des produits et les ajouter à une ObservableList appelée listData.

Finalement, la méthode ajoute la liste de noms de produits à une autre liste déroulante appelée Commende_produit, qui permettra à l'utilisateur de sélectionner le produit correspondant à la commande.

La fonction commendeProduitPrice():

```
public void commendeProduitPrice(){
    String sql = "SELECT price FROM client WHERE clientId =
    '"+Commende_id.getSelectionModel().getSelectedItem()+"'";

    try{
        ObservableList listData = FXCollections.observableArrayList();
        connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
        prepare= connect.prepareStatement(sql);
        result=prepare.executeQuery();

        while (result.next()) {
            listData.add(result.getString("price"));
        }
        Commende_Prix.setItems(listData);

    }catch (Exception e){
        e.printStackTrace();
    }
}
```

Ce code permet de récupérer le prix du produit sélectionné dans la liste déroulante "Commende_id". Il utilise la même approche que la méthode "commendeProduitName" : en fonction de l'ID du client sélectionné, il exécute une requête SQL qui extrait le prix du produit correspondant. La valeur de prix est ensuite ajoutée à une ObservableList, qui est liée à la liste déroulante "Commende_Prix" pour l'afficher à l'utilisateur.

La fonction paymentButton():

```

public void paymentButton(){
    String sql = "UPDATE client SET status = 'Payé' WHERE clientId =
    '"+Commende_id.getSelectionModel().getSelectedItem()+"''";
    try{
        connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
        prepare = connect.prepareStatement(sql);
        prepare.executeUpdate();

        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle("Information message!!");
        alert.setHeaderText(null);
        alert.setContentText("successfully payed!!");
        alert.showAndWait();

        commendeShowData();
        clearPayment();

    }catch(Exception e){
        e.printStackTrace();
    }

}

```

Ce code permet de mettre à jour la colonne "status" de la table "client" pour le client sélectionné dans la liste de commande. Plus précisément, le code exécute une requête SQL qui modifie la valeur de la colonne "status" en "Payé" pour le client dont l'identifiant est égal à celui sélectionné dans la liste de commande. Ensuite, il affiche une boîte de dialogue "Alert" pour informer l'utilisateur que le paiement a été effectué avec succès. Enfin, il appelle la méthode "commendeShowData" pour mettre à jour l'affichage de la liste de commande et la méthode "clearPayment" pour vider les champs de saisie.

La fonction clearPayment():

```

public void clearPayment(){
    Commende_id.getSelectionModel().clearSelection();
    Commende_nom.getSelectionModel().clearSelection();
    Commende_produit.getSelectionModel().clearSelection();
    Commende_Prix.getSelectionModel().clearSelection();
}

```

La méthode **clearPayment** est utilisée pour effacer les sélections faites par l'utilisateur dans les champs de la fenêtre d'interface utilisateur pour effectuer un paiement. Elle ne prend aucun paramètre en entrée et ne renvoie rien.

Dans cette méthode, on appelle les méthodes **clearSelection()** de chaque champ de sélection de la fenêtre d'interface utilisateur pour effacer leur sélection respective. Cela permet à l'utilisateur de sélectionner de nouveaux éléments lorsqu'il effectue un nouveau paiement sans avoir à décocher manuellement les anciennes sélections.

Interface de Crédit:

La fonction `ObservableList<ClientData> creditDataList()`:

```
public ObservableList<ClientData> creditDataList(){
    ObservableList<ClientData> ListData = FXCollections.observableArrayList();
    String sql = "SELECT * FROM client ";

    try{
        connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
        prepare= connect.prepareStatement(sql);
        result=prepare.executeQuery();
        ClientData Md;
        while (result.next()){
            Md = new ClientData(result.getInt("id"),result.getString("clientId"),
            result.getString("name")
            ,result.getString("phone"),
            result.getString("produit"),
            result.getString("status"),
            result.getDouble("price"));
            ListData.add(Md);
        }

    }catch (Exception e){e.printStackTrace();}

    return ListData;
}
```

Ce code définit une méthode nommée **creditDataList()** qui renvoie une liste observable d'objets **ClientData**. Cette méthode interroge une base de données MySQL locale (**jdbc:mysql://localhost:3306/esalaf**) pour récupérer les données de tous

les clients. La requête SQL utilisée pour récupérer les données de la table client est stockée dans une variable de chaîne nommée **sql**.

Ensuite, cette méthode exécute la requête SQL en utilisant un objet **Connection** pour établir une connexion à la base de données, un objet **PreparedStatement** pour préparer la requête SQL et un objet **ResultSet** pour stocker le résultat de la requête. Elle crée également un objet **ClientData** en utilisant les données de chaque ligne dans le résultat et les ajoute à la liste observable.

Enfin, la méthode renvoie la liste observable contenant tous les objets **ClientData** récupérés de la base de données.

La fonction creditShowData():

```
public void creditShowData(){
    creditListData = creditDataList();
    crediti_col_ID.setCellValueFactory(new PropertyValueFactory<>("clientId"));
    crediti_col_nom.setCellValueFactory(new PropertyValueFactory<>("name"));
    crediti_col_phone.setCellValueFactory(new PropertyValueFactory<>("phone"));
    crediti_col_produit.setCellValueFactory(new PropertyValueFactory<>("produit"));
    crediti_col_status.setCellValueFactory(new PropertyValueFactory<>("status"));
    crediti_col_prix.setCellValueFactory(new PropertyValueFactory<>("price" ));

    credit_tableview.setItems(commendeListData);
}
```

Il s'agit d'une méthode qui affiche les données des clients ayant un statut de crédit (status = "Crédit") dans une TableView.

La première ligne de la méthode appelle la méthode creditDataList() pour récupérer la liste des données clients de la base de données.

Ensuite, chaque colonne de la TableView est configurée avec une PropertyValueFactory qui indique quelle propriété de l'objet ClientData doit être associée à cette colonne.

Enfin, la TableView est mise à jour avec les données de la liste des clients en utilisant la méthode setItems().

Il est important de noter que la dernière ligne contient une erreur: elle utilise commendeListData au lieu de creditListData pour définir les données de la TableView.

La fonction creditClientID():

```

public void creditClientID(){

String sql = "SELECT clientId FROM client WHERE status = 'Credit' ";

try{
ObservableList listData = FXCollections.observableArrayList();
connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
prepare= connect.prepareStatement(sql);
result=prepare.executeQuery();

while (result.next()) {
listData.add(result.getString("clientId"));
}
crediti_ID.setItems(listData);
creditClientName();
creditProduitName();
}catch (Exception e){
e.printStackTrace();
}

}

```

Ce code permet de remplir une liste déroulante (ComboBox) avec les identifiants des clients qui ont un statut "Crédit" dans la table "client". La méthode s'appelle "creditClientID".

La requête SQL sélectionne les identifiants ("clientId") dans la table "client" où le statut est "Crédit".

Ensuite, la méthode crée une liste observable "listData" et récupère les résultats de la requête en itérant à travers le résultat avec une boucle "while". Pour chaque ligne, elle ajoute l'identifiant à la liste "listData".

Enfin, la liste déroulante "crediti_ID" est remplie avec la liste "listData". La méthode appelle également deux autres méthodes "creditClientName" et "creditProduitName" pour remplir deux autres listes déroulantes avec le nom du client et le nom du produit.

Si une exception se produit, elle est attrapée et affichée dans la console.

La fonction creditClientName():

```

public void creditClientName(){
String sql = "SELECT name FROM client WHERE clientId =
""+crediti_ID.getSelectionModel().getSelectedItem()+""";

```



```

try{
ObservableList listData = FXCollections.observableArrayList();
connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
prepare= connect.prepareStatement(sql);
result=prepare.executeQuery();

while (result.next()) {
listData.add(result.getString("name"));
}
crediti_nom.setItems(listData);

}catch (Exception e){
e.printStackTrace();
}
}

```

Ce code est une méthode qui récupère le nom du client à partir de son ID de client sélectionné dans une liste déroulante. Il exécute une requête SQL pour sélectionner le nom correspondant à l'ID de client sélectionné, puis ajoute le nom dans une liste observable et l'affiche dans une autre liste déroulante appelée "crediti_nom".

- La méthode commence par définir une requête SQL pour sélectionner le nom du client à partir de son ID de client sélectionné dans la liste déroulante "crediti_ID".
- La méthode définit une liste observable appelée "listData" qui sera utilisée pour stocker les noms récupérés à partir de la base de données.
- La méthode établit une connexion à la base de données en utilisant `DriverManager.getConnection()`.
- La méthode crée un objet `PreparedStatement` appelé "prepare" en utilisant la connexion établie précédemment, puis exécute la requête SQL en utilisant `prepare.executeQuery()`.
- La méthode parcourt tous les résultats de la requête en utilisant `while(result.next())`, et pour chaque résultat, ajoute le nom correspondant à la liste observable "listData" en utilisant `listData.add(result.getString("name"))`.
- La méthode associe la liste observable "listData" à la liste déroulante "crediti_nom" en utilisant `crediti_nom.setItems(listData)`.

La fonction `creditProduitName()`:

```

public void creditProduitName(){
    String sql = "SELECT produit FROM client WHERE clientId =
    '"+crediti_ID.getSelectionModel().getSelectedItem()+"''";

    try{
        ObservableList listData = FXCollections.observableArrayList();
        connect = DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","root","");
        prepare= connect.prepareStatement(sql);
        result=prepare.executeQuery();

        while (result.next()) {
            listData.add(result.getString("produit"));
        }
        crediti_prix.setItems(listData);

    }catch (Exception e){
        e.printStackTrace();
    }
}

```

Ce code est une méthode qui permet de récupérer le nom du produit que le client doit rembourser dans le système de gestion de crédit.

Tout d'abord, la méthode récupère l'identifiant du client sélectionné à partir de la liste déroulante "crediti_ID". Ensuite, une requête SQL est exécutée pour récupérer le nom du produit associé à cet identifiant. La requête est exécutée en utilisant une connexion à une base de données MySQL.

Le résultat de la requête est stocké dans une liste observable nommée "listData". La liste est ensuite affectée à la liste déroulante "crediti_prix", qui affichera le nom du produit que le client doit rembourser.

Enfin, des blocs try-catch sont utilisés pour gérer les exceptions potentielles qui peuvent survenir lors de l'exécution de la requête SQL ou de la récupération des données de la base de données.

La fonction enregistrerButton():

```

public void enregistrerButton(){
    String sql = "UPDATE client SET status = 'Enregistrer' WHERE

```

```

clientId = ""+crediti_ID.getSelectionModel().getSelectedItem()+"";
try{
    connect =
DriverManager.getConnection("jdbc:mysql://localhost:3306/esalaf","ro
ot","");
    prepare = connect.prepareStatement(sql);
    prepare.executeUpdate();

    Alert alert = new Alert(Alert.AlertType.INFORMATION);
    alert.setTitle("Information message!!");
    alert.setHeaderText(null);
    alert.setContentText("successfully Saved!!");
    alert.showAndWait();

    creditShowData();
    clearEnregistrement();

}catch(Exception e){
    e.printStackTrace();
}
}

```

Ce code correspond à une méthode qui est appelée lorsqu'on clique sur le bouton "Enregistrer". Cette méthode met à jour le statut d'un client dans la base de données en le passant de "Crédit" à "Enregistré". Pour cela, elle utilise la méthode "getSelectionModel()" pour obtenir l'élément sélectionné dans la liste des ID des clients en crédit ("crediti_ID"), puis elle utilise cet ID pour construire une requête SQL qui mettra à jour le statut de ce client dans la base de données. Ensuite, elle affiche une boîte de dialogue d'alerte pour indiquer que l'opération s'est déroulée avec succès, elle rafraîchit la table des données en appelant la méthode "creditShowData()" et elle efface les champs de sélection en appelant la méthode "clearEnregistrement()".

La fonction clearEnregistrement():

```

public void clearEnregistrement(){
    crediti_ID.getSelectionModel().clearSelection();
    crediti_nom.getSelectionModel().clearSelection();
    crediti_prix.getSelectionModel().clearSelection();
}

```

Cette méthode est appelée lorsque l'utilisateur souhaite effacer les données qui ont été sélectionnées pour l'enregistrement. Elle efface les sélections de la liste déroulante pour le numéro de client, le nom du client et le nom du produit. Cela permet à l'utilisateur de réinitialiser les champs de sélection pour l'enregistrement suivant.

La fonction switchForm():

```
public void switchForm(ActionEvent event){
    if (event.getSource()==ProduitButton){
        produit_form.setVisible(true);
        client_form.setVisible(false);
        Commende_form.setVisible(false);
        credit_form.setVisible(false);
        produitCategoryListe();
        ProduitStatusList();
        produitShowData();
    } else if (event.getSource() == ClientButton){
        produit_form.setVisible(false);
        client_form.setVisible(true);
        Commende_form.setVisible(false);
        credit_form.setVisible(false);
        clientProduitName();
        clientProduitPrice();
        clientShowData();
        ClienttStatusList();
    } else if(event.getSource() == CommendeButton){
        produit_form.setVisible(false);
        client_form.setVisible(false);
        Commende_form.setVisible(true);
        credit_form.setVisible(false);
        commendeShowData();
        commendeClientID();
        commendeClientName();
        commendeProduitName();
        commendeProduitPrice();

    } else if (event.getSource() == creditButton){
        produit_form.setVisible(false);
        client_form.setVisible(false);
        Commende_form.setVisible(false);
        credit_form.setVisible(true);
        creditShowData();
        creditClientID();
        creditClientName();
    }
}
```

```

    creditProduitName();
}
}

```

Ce code définit une méthode qui est appelée lorsqu'un bouton est cliqué pour changer de formulaire dans l'interface utilisateur. Selon le bouton cliqué, il affichera le formulaire correspondant et masquera les autres formulaires. Par exemple, si le bouton "Client" est cliqué, le formulaire "client_form" sera affiché et les autres formulaires (produit_form, Commende_form, credit_form) seront masqués. Cette méthode appelle également d'autres méthodes pour afficher les données et les listes déroulantes appropriées pour chaque formulaire.

La fonction logout():

```

public void logout (){
    Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
    alert.setTitle("Confirmation Logout!!");
    alert.setHeaderText(null);
    alert.setContentText("You want to leave us!!");
    Optional<ButtonType> option = alert.showAndWait();
    if(option.get().equals(ButtonType.OK)){
        try{
            FXMLLoader fxmLoader = new
FXMLLoader(HelloApplication.class.getResource("/com/example/demo/hello-view.fxml"));
            Parent root = (Parent) fxmLoader.load();
            Stage stage = new Stage();
            stage.setTitle("Dashboard");
            stage.setScene(new Scene(root));
            stage.show();

            logoutBtn.getScene().getWindow().hide();
        }catch (Exception e){e.printStackTrace();}

    }

}

```

Ce code définit une méthode appelée **logout()** qui gère l'action de déconnexion de l'utilisateur de l'application. Lorsque l'utilisateur clique sur le bouton de déconnexion, une boîte de dialogue de confirmation est affichée, demandant à l'utilisateur s'il souhaite réellement quitter l'application.

Si l'utilisateur clique sur le bouton "OK" de la boîte de dialogue de confirmation, le code récupère le fichier FXML de la vue d'accueil de l'application et charge son contenu dans un objet **Parent**. Ensuite, il crée une nouvelle fenêtre de l'application en utilisant cet objet **Parent** comme racine de la scène.

Enfin, la méthode **hide()** est appelée sur la fenêtre actuelle (celle contenant le bouton de déconnexion), cachant ainsi cette fenêtre et fermant la session actuelle de l'application.

En résumé, cette méthode gère le processus de déconnexion de l'utilisateur de l'application en affichant une boîte de dialogue de confirmation et en redirigeant l'utilisateur vers la page de connexion si celui-ci décide de se déconnecter.

La fonction minimize():

```
public void minimize(){  
    Stage stage = (Stage) mainForm.getScene().getWindow();  
    stage.setIconified(true);  
}
```

Ce code est une méthode qui permet de minimiser la fenêtre de l'application. La méthode récupère la référence à la fenêtre principale de l'application en utilisant la méthode **getScene()** de l'élément **mainForm**. Ensuite, elle convertit la fenêtre en objet **Stage** et utilise la méthode **setIconified(true)** pour minimiser la fenêtre. Cela permet à l'utilisateur de réduire la fenêtre de l'application sans la fermer complètement.

La fonction close():

```
public void close(){  
    javafx.application.Platform.exit();  
}
```

La méthode **close()** ferme l'application JavaFX en appelant la méthode **exit()** de la classe **Platform** du package **javafx.application**. La méthode **exit()** arrête tous les threads JavaFX et détruit tous les stades JavaFX, ce qui entraîne la fermeture de l'application.

ClientData :

Définit une classe **ClientData** qui représente les données d'un client. Elle possède les attributs suivants :

- **id** : un entier représentant l'identifiant de la ligne dans la table de la base de données

- **clientId** : une chaîne de caractères représentant l'identifiant unique du client
- **name** : une chaîne de caractères représentant le nom du client
- **phone** : une chaîne de caractères représentant le numéro de téléphone du client
- **produit** : une chaîne de caractères représentant le produit acheté par le client
- **status** : une chaîne de caractères représentant le statut de la commande (par exemple "en attente", "en cours", "terminée")
- **price** : un nombre décimal représentant le prix du produit acheté par le client.

La classe possède également un constructeur qui permet d'initialiser les attributs, ainsi que des méthodes getters pour chaque attribut.

ProduitData:

Définit une classe **ProduitData** qui stocke les informations relatives à un produit dans une application. Les attributs de la classe incluent :

- **id** : l'identifiant numérique unique du produit.
- **produit_id** : l'identifiant alphanumérique unique du produit.
- **name** : le nom du produit.
- **price** : le prix du produit.
- **category** : la catégorie à laquelle le produit appartient.
- **status** : le statut actuel du produit.

La classe dispose également d'un constructeur qui permet d'initialiser ces attributs, ainsi que de méthodes d'accès (**getters**) pour récupérer les valeurs des attributs.