



Rapport

Application de gestion et suivi des étudiants

Semestre 3 - 2° Année



Sommaire

<u>Sommaire</u>	2
<u>Introduction</u>	3
<u>Initialisation</u>	3
<u>Page de login</u>	5
<u>Page de navigation</u>	7
<u>Page d'ajout d'étudiant</u>	9
<u>Page liste des étudiants</u>	13
<u>Page vision spécifique étudiant</u>	18
<u>Conclusion</u>	24

Introduction

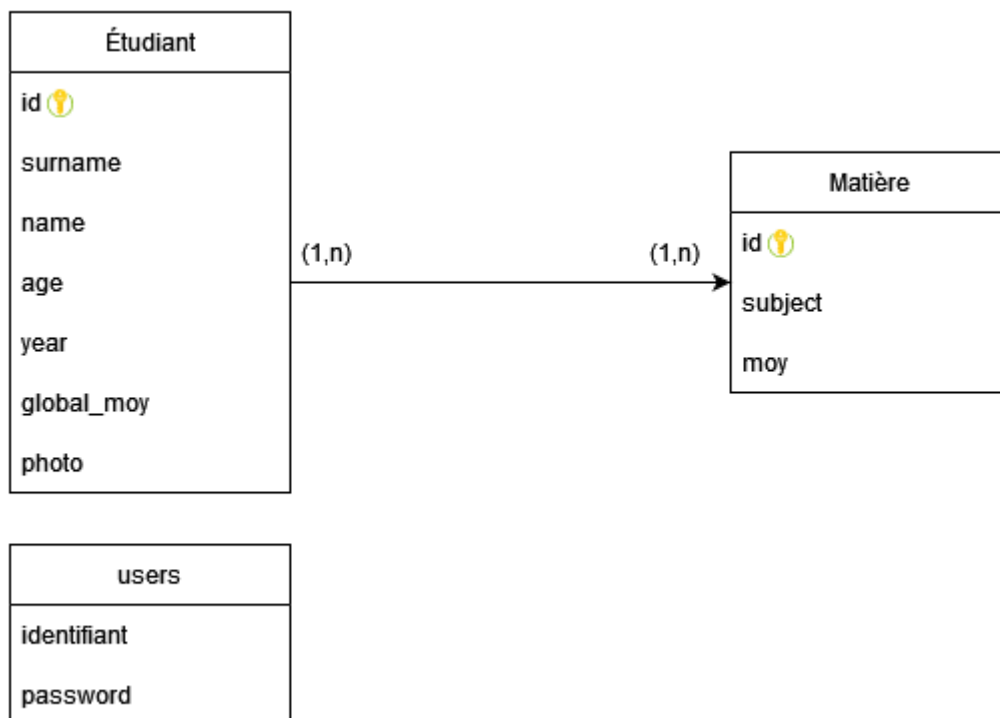
L'idée de ce projet est de développer une application permettant le suivi des moyennes des étudiants d'une structure éducative (ici l'IUT R&T) par les professeurs de cette même structure. L'application est codée en Python, utilise la bibliothèque graphique Kivy et devra permettre l'ajout d'étudiants, de moyennes ainsi que la visualisation des données stockées dans une base de données sur diverses interfaces.

Nous avons travaillé avec Github afin de centraliser nos codes, le répertoire est disponible en suivant ce lien : [SAE302-Gestion_dossier_etudiants](#)

Initialisation

Ce projet requiert une base de données comportant 3 tables : une table users qui contient les différents identifiants et mots de passe des professeurs et qui sera utilisée lors de la connexion via la page de login ; une table contenant les étudiants avec leur identifiant unique, leur nom, leur prénom, leur âge, leur année d'étude, leur moyenne générale et leur photo et enfin, une table contenant les moyennes et les matières correspondantes pour chaque étudiant qui sera lié à cette table par son identifiant.

Nous avons dressé la description des tables suivante :



L'application requiert plusieurs bibliothèques/paquets afin de fonctionner correctement et permettre l'implémentation de fonctionnalités pratiques et optimisées.

Voici les éléments à installer :

- Pip : **sudo apt install python3-pip**
- Kivy : **python3 -m pip install kivy**
- WebSockets et WebSocket-Client :
 - **python3 -m pip install websockets**
 - **python3 -m pip install websocket-client**
- Plyer :
 - **python3 -m pip install plyer**
- Mysql_Connector :
 - **pip3 install mysql_connector**
 - **pip3 install mysql-connector-python**

Afin de permettre à Mysql_Connector de se connecter à la base de données et d'envoyer des requêtes, on initialise la configuration nécessaire à cette connexion en début de programme, afin que celle-ci puisse être utilisée dans n'importe quelle classe.

La configuration (en mode local pour les tests) est initialisée comme suit :

```
config = {'host': '127.0.0.1',  
          'database': 'bdetu',  
          'user': 'etuadmin',  
          'password': 'etuadmin'}
```

Nous avons aussi initialisé :

- La taille de la fenêtre de sorte à : `Window.size = (500,700)`
Soit : 500px*700px
- La couleur de la page : `Window.clearcolor = (237/255, 234/255, 225/255, 1)`
- Une variable "config", qui est un dictionnaire avec les informations pour se connecter à la BDD, donc l'hôte, le nom de la BDD, l'utilisateur et le mot de passe.

Du côté du code Kivy, nous avons renseigné la version **1.11** de la bibliothèque, nous avons initialisé le parent "**WindowManager**", qui permet de contenir le nom de toutes les fenêtres que l'on utilise.

Page de login

Quentin DELCHIAPPO

Premièrement, on crée une classe login telle que : `class Login(Screen):`

Tout ce qui concerne l'écran de login est centralisé dans cette classe. On commence par créer une fonction appelée `on_leave`, qui permettra de vider les champs de texte de la page.

```
def on_leave(self):  
    self.ids.connect_login.text = ''  
    self.ids.connect_mdp.text = ''
```

Ici, lorsque la page login est quittée, on remet à vide le texte des TextInput ayants pour ids `"connect_login"` et `"connect_mdp"` qui sont contenus sur la page login.

Ensuite, on ajoute la fonctionnalité même de la connexion, conjointement avec la base de données et les informations qu'elle renvoie afin de s'assurer que les identifiants rentrés permettent bien d'accéder à l'application.

Pour se faire, on crée une fonction appelée `connect` : `def connect(self):`

On initialise une variable `"db"`, lors de la connexion à la base de donnée via `sql.connect()` qui prend pour paramètre les valeurs de la variable `"config"` initialisée plus haut via la syntaxe : `**config`.

On crée ensuite une variable `"cursor"`, qui va placer un curseur au début de notre base de données, au moyen de `db.cursor()`.

Une fois que le curseur est bien placé au début de notre BDD, nous lui demandons d'exécuter la requête SQL suivante : `"SELECT identifiant, password FROM users"`, qui va sélectionner les identifiants et les mots de passe de la table `"USERS"`.

Nous récupérons ensuite toutes les informations retournées par la requête via l'instruction `cursor.fetchall()` que nous assignons à la variable `"data"`. Cela permet d'ajouter toutes les informations retournées par la BDD, dans la variable `"data"` sous forme de liste.

On crée une liste vide, appelée `"users"` telle que `users = []`

On crée une variable appelée `"log"`, qu'on initialise de façon booléen à `False`.

On peut dès à présent commencer à tester les informations entrées dans les champs de login, afin de vérifier l'identité de l'utilisateur.

On crée une boucle qui va, pour chaque **"user"** contenu dans la liste **"data"**, initialiser un dictionnaire appelé **"users_dict"**, qui sera dans notre cas :

```
users_dict = {'ident': user[0], 'pass': user[1]}
```

La clé **"ident"** prendra le premier élément de la liste **"user"**, et la clé **"pass"** prendra le deuxième élément de la liste **"user"**. Par la suite, on ajoute à la liste **"users"**, notre dictionnaire nouvellement rempli

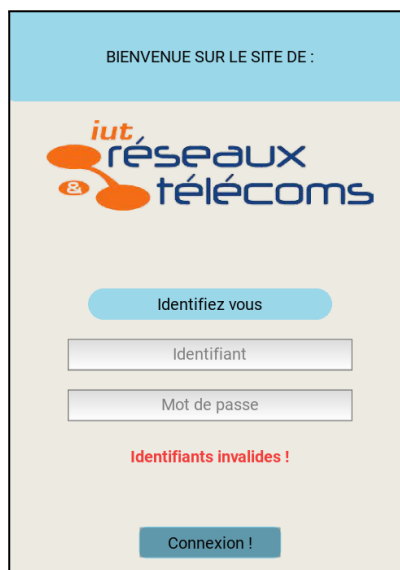
On va créer une deuxième boucle, telle que : pour chaque **"user"** dans la liste **"users"**, si notre variable `self.ids.connect_login.text` correspond à la valeur de l'identifiant, présent dans le dictionnaire et que `self.ids.connect_mdp.text` correspond à la valeur du mot de passe, on passe alors la variable **"log"** au niveau booléen **True**, et on peut faire passer l'application à la fenêtre **"NavigProf"** au moyen de la ligne :

```
App.get_running_app().root.current = "NavigProf"
```

En revanche, si la valeur de **"log"** est égale à **"False"** (ce qui signifie que la boucle vérifiant les champs identifiant et mot de passe s'est terminée sans que la condition qu'elle contient ne se soit déclenchée), alors on affecte à la variable `self.ids.login_error.text` le texte **"Identifiants invalides !"**

Pour finir, on crée une nouvelle fonction appelée **reset_err_msg**, qui nous permet de remettre à vide la valeur de la variable `self.ids.login_error.text`.

Cette fonction sera appelée dans le code Kivy à chaque fois que le curseur rentrera dans un des deux TextInput. Cet appel est réalisé grâce à l'événement Kivy : `on_cursor`.



Page de navigation

Quentin DELCHIAPPO

Pour la partie python de la fenêtre "NavigProf", rien de plus simple. Étant donnée que cette page n'a pas besoin de récupérer des informations ou d'avoir des contraintes, nous pouvons simplement lui dire de passer directement :

```
class NavigProf(Screen):  
    pass
```

Comme vu précédemment, lorsque nous devons configurer sous Kivy une nouvelle fenêtre, il nous faut la renseigner comme étant une balise telle que : `<NavigProf>:`.

On n'oublie pas d'assigner le nom qu'aura la fenêtre avec : `name: "NavigProf"`

Une fois que la fenêtre est créée, occupons nous de la mise en page. Pour ce faire, nous utilisons principalement des boutons, et des `BoxLayout` qui nous permettent d'ajouter des éléments de manière organisée sur la page.

Dans le cas de notre fenêtre, le `BoxLayout` général est construit de cette façon :

```
BoxLayout:  
    cols: 1  
    orientation: "vertical"  
    spacing: root.height/20
```

Nous avons notre `BoxLayout` pour toute la page, qui ne contient qu'une seule colonne. Donc, chaque élément se place sous l'élément précédent avec un espacement correspondant à la hauteur de la fenêtre divisé par 20.

Durant tout le fichier Kivy, nous utilisons des tailles, espacements et marges relatives soit à la hauteur ou la largeur de la fenêtre soit à la hauteur ou la largeur des parents. Tout ceci afin de réaliser une application qui puisse être redimensionnée sans risquer de voir les widgets se rentrer les uns dans les autres.

Par la suite, nous avons mis en place un en-tête pour la page.

Sa construction est la même pour toutes les pages. Nous avons un `BoxLayout` qui est orienté verticalement, avec une seule colonne.

Nous utilisons la fonction : `size_hint: (1, None)` de telle sorte à ce que le widget occupe la totalité de la largeur de son parent, mais la hauteur est définie manuellement par `height:` qui prend la taille de la hauteur de la page divisée par 15.

L'en-tête est toujours composé d'un bouton retour (ici un bouton pour se déconnecter puisqu'il renvoie à la page de login) et le titre de la page.

Cette page est simplement composée de deux boutons qui amènent sur les deux pages principales de l'application (ajouter un étudiant et la liste des étudiants).



Page d'ajout d'étudiant

Quentin DELCHIAPPO

Au sein de la fonction `__init__` de la classe de cette page, nous connectons l'événement Kivy `on_drop_file` à la fonction `self.drop_file` via la ligne : `Window.bind(on_drop_file=self.drop_file)`. Ensuite, on initialise à vide la variable du chemin de la photo `"self.photo_path"`.

On crée la fonction `drop_file` telle que :

```
def drop_file(self, window, file_path, x, y):
```

Au moyen des fonctions `split("\\")[-1]`, `strip(" ' ")` et `replace(" ", "_")`, on formate le texte contenu dans le TextInput `self.ids.picture_add_etu` afin que le chemin de la photo déposée dans la fenêtre et passé par la variable `file_path` soit affiché dans le bouton de dépôt de la photo et qu'il soit simple et lisible pour les utilisateurs.

La fonction `file_chooser` telle que : `def file_chooser(self):`, permet d'ouvrir l'explorateur de fichiers du système d'exploitation et ainsi de nous laisser sélectionner une photo à téléverser. Au moyen de la ligne `filechooser.open_file(on_selection=self.picture_selected)`, une fois qu'une photo est choisie le même processus de formatage et d'affichage que lorsque la photo est glissée puis déposée dans la fenêtre de l'application, est exécuté.

La fonction `convert_pic` telle que : `def convert_pic(self, file_path):`, est appelée lors de la requête SQL afin d'envoyer les données à la BDD et permet de convertir la photo sélectionnée ou déposée dans la fenêtre de l'application, en données binaires qui pourront ensuite être envoyés dans un champ BLOB de la BDD.

Pour cela, on ouvre, avec la fonction `with open`, le fichier grâce à son chemin, avec comme paramètre `'rb'`. On initialise la variable `"photo"` à laquelle on ajoute le résultat de la lecture du fichier, donc la photo en données binaires, puis on renvoie la photo.

La fonction `reset`, telle que `def reset(self, type):` se charge de remettre à zéro tous les champs de la page. Que l'étudiant soit rentré dans la BDD ou qu'il y ait eu une erreur, il faut remettre à vide chaque variable.

En revanche, il faut pouvoir distinguer si l'étudiant a été ajouté, ou s'il y a eu une erreur. Pour cela, on utilise une condition sur le type, de telle sorte à ce que :

Si le type est égal à **"ALL"**, alors toutes les variables sont remises à zéro, sinon, si le type est égal à **"ERR"**, de même, tous les champs, sauf le Label affichant l'erreur, sont remis à zéro.

```
def reset(self, type):
    if type == "ALL":
        self.ids.surname_etu.text = ''
        self.ids.name_etu.text = ''
        self.ids.age_etu.text = ''
        self.ids.subject_choice_etu.text = 'Matière ?'
        self.ids.moyenne_etu.text = ''
        self.ids.year_choice_etu.text = 'Année ?'
        self.ids.picture_add_etu.text = 'Cliquez ou glissez
la photo ici'
        self.ids.add_etu_error.text = ''
    elif type == "ERR":
        self.ids.surname_etu.text = ''
        self.ids.name_etu.text = ''
        self.ids.age_etu.text = ''
        self.ids.subject_choice_etu.text = 'Matière ?'
        self.ids.moyenne_etu.text = ''
        self.ids.year_choice_etu.text = 'Année ?'
        self.ids.picture_add_etu.text = 'Cliquez ou glissez
la photo ici'
```

La fonction **on_leave** telle que : `def on_leave(self):` remet tous les champs à zéro lorsque la fenêtre est quittée en appelant la fonction **reset**.

La fonction `add_student`, telle que `def add_student(self):`, est la plus importante, c'est cette fonction qui fait en sorte d'ajouter ou non un étudiant dans la BDD.

Tout d'abord, on initialise une variable `"etu_exist"` à un état booléen, ici `"False"` qui montre que l'étudiant n'existe pas.

Ensuite, on pose une condition général pour chaque champs :

Si un des champs est vide, alors le texte du widget `add_etu_error` devra renvoyer `'Un ou des champs sont vides'`. Sinon, on se connecte à la BDD, comme vu précédemment, puis on exécute la requête SQL suivante : `"SELECT surname, name FROM etudiants"` qui nous permet de sélectionner tous les étudiants présents dans la base de données (ou en tout cas leurs prénoms et noms) et ainsi de vérifier, selon la même méthode que pour la page de login, si les informations que l'on souhaite ajouter (un nouvel étudiant) n'est pas déjà présent dans la BDD. Le message `"Cet étudiant existe déjà !"` est alors affiché dans le Label d'erreur de la page d'ajout si le nom et le prénom rentrés appartiennent déjà à un étudiant présent dans la base de données.

D'autres messages peuvent être affichés en fonction des erreurs rencontrées :

`"La moyenne n'est pas valide !"`

`"L'âge n'est pas valide !"`

L'état booléen de `"etu_err"` devient `"True"` si l'étudiant existe déjà, si l'âge est inférieur à 17 ans ou supérieur à 100 ans ou si la moyenne rentrée n'est pas strictement comprise entre 0 et 20.

On utilise la fonction `"reset("Err")"`, pour remettre à vide chaque champ lors d'une erreur, et on utilise `"break"` pour sortir de la boucle. La variable `"etu_err"` n'étant pas égale à `False`, l'ajout de l'étudiant ne pourra se faire.

On utilise la fonction `lower()` pour tester les champs de nom et prénom, afin d'ignorer la casse, et d'être sûr de comparer chaque texte.

On crée enfin une variable `"ident"`, qui recevra la première lettre des variables `"surname_etu"` + `"name_etu"` + un identifiant aléatoire entre 2000 et 9000.

La variable **“to_insert_stud”**, représente tous les éléments qui devront être ajoutés dans la table étudiant de la BDD.

On affecte alors à la variable, toutes les informations nécessaires :

```
to_insert_stud = [ident, self.ids.surname_etu.text,
                  self.ids.name_etu.text,
                  int(self.ids.age_etu.text),
                  self.ids.year_choice_etu.text,
                  round(float(self.ids.moyenne_etu.text), 2),
                  self.convert_pic(self.photo_path)]
```

À noter que dans toute l'application nous utilisons **round(float(), 2)**, afin d'arrondir toutes les moyennes avec deux chiffres après la virgule.

De même pour la variable **“to_insert_subject”**, qui représente tous les éléments qui seront ajoutés dans la table matières de la BDD (donc la matière sélectionnée, la moyenne correspondante et l'identifiant unique généré).

```
to_insert_subject = [ident, self.ids.subject_choice_etu.text,
                    float(self.ids.moyenne_etu.text)]
```

On utilise la fonction **“cursor_execute”** pour envoyer les requêtes SQL suivantes :

```
"INSERT INTO etudiants (id, surname, name, age, year,
global_moy, photo) VALUES (%s, %s, %s, %s, %s, %s, %s)",
to_insert_stud)
```

```
"INSERT INTO matières (id, subject, moy) VALUES (%s, %s, %s)",
to_insert_subject)
```

On spécifie quelle table doit être remplie avec le nom des éléments, puis on précise **“%s”** pour spécifier les valeurs à insérer qui sont récupérées via les variables **“to_insert_stud”** et **“to_insert_subject”**.

On fait appel à la fonction **reset()**, afin de remettre à vide tous les champs de la page **“add”**.

Puis on envoie les données dans la BDD avec **db.commit()**

Et on ferme la connexion à la BDD avec **db.close()**

Page liste des étudiants

Nolan BEN YAHYA

Cette fenêtre permet la visualisation sous forme de liste des étudiants présents dans la base de données.

À l'entrée dans la page, la fonction `populate(self)` est appelée et permet d'ajouter les données de chaque étudiant présent dans la base de données.

Après s'être connecté à la base de données, les données renvoyées par la requête `"SELECT surname, name, age, year, global_moy, photo FROM etudiants " + key_search + order_search` sont formatées dans une liste de dictionnaires en suivant exactement la même méthode que pour les vérifications lors de l'ajout d'un étudiant ou lors de la connexion.

Voici la boucle qui permet le formatage :

```
studs = []
for etu in reversed(data):
    stud_dict = {"surname": etu[0], "name": etu[1],
                 "age": etu[2], "year": etu[3],
                 "global_moy": etu[4], "photo": etu[5]}
    studs.append(stud_dict)
```

Il faut ensuite s'assurer que les éléments ajoutés lors d'une première visite de la page ne persistent pas afin d'éviter que des données qui ne sont plus présentes dans la BDD soient toujours affichées ou que des étudiants soient affichés deux fois.

La boucle suivante se charge de ce problème :

```
studs_lists = [i for i in self.ids.stud_lists.children]
for stud_list in studs_lists:
    self.ids.stud_lists.remove_widget(stud_list)
```

La variable **"studs_lists"** contient tous les widgets présents dans le StackLayout **stud_lists** et permet à la boucle suivante de supprimer chaque enfant de **stud_lists** un par un.

Une fois cela fait, il suffit d'ajouter les données récupérées plus haut dans la base de données et qui attendent dans la liste “**studs**” par une dernière boucle que voici :

```
for etu in studs:
    stud = StudList()
    stud.ids.stud_list_surname.text = etu['surname']
    stud.ids.stud_list_name.text = etu['name'].upper()
    stud.ids.stud_list_age.text = str(etu['age'])
    stud.ids.stud_list_year.text = str(etu['year'])
    stud.ids.stud_list_global_moy.text =
        str(round(etu['global_moy'], 2))
    stud.ids.stud_list_picture.texture =
        CoreImage(io.BytesIO(etu['photo']), ext="png").texture
    self.ids.stud_lists.add_widget(stud)
```

Cette boucle crée, pour chaque étudiant, une instance du template **StudList** défini dans le fichier Kivy, en modifie les informations avec les données présentes dans le dictionnaire de l'étudiant sélectionné par la boucle, dont la photo qui est convertie en texture afin d'être affichée par Kivy via la ligne :

```
CoreImage(io.BytesIO(etu['photo']), ext="png").texture
```

Enfin, l'instance modifiée est ajoutée à la suite des précédentes contenues dans le StackLayout.

La page contient aussi un champ de texte pour la recherche précise d'un étudiant ainsi qu'un élément de type Spinner afin de trier les étudiants de manière efficace.

Ces recherches et tris sont dirigés par deux fonctions distinctes qui sont appelées à chaque repopulation de la liste par la fonction **populate**. Lorsque du texte est entré dans le TextInput de recherche ou lors de la sélection d'une méthode de tri depuis le Spinner, la fonction **populate** est appelée.

La fonction qui s'occupe de trier les données en fonction du texte du Spinner utilise un dictionnaire, afin de mettre en relation les choix disponibles et les bouts de requêtes SQL correspondants.

Voici à quoi ressemble la fonction qui prend en paramètre le texte du Spinner au moment de sa sélection :

```
def sort_list(self, search):
    search_list = {"1A": "AND year = '1A'",
                   "2A": "AND year = '2A'",
                   "Prénoms A-Z": "ORDER BY surname",
                   "Prénoms Z-A": "ORDER BY surname DESC",
                   "Noms A-Z": "ORDER BY name",
                   "Noms Z-A": "ORDER BY name DESC",
                   "Age >": "ORDER BY age",
                   "Age <": "ORDER BY age DESC",
                   "Moyenne G >": "ORDER BY global_moy",
                   "Moyenne G <": "ORDER BY global_moy DESC"}

    if search == "Trier par":
        return ""
    return search_list[search]
```

La fonction qui se charge de chercher un étudiant spécifique en entrant son prénom et son nom dans le TextInput reprend le même principe et prend en paramètre le texte du TextInput. Étant donné que la fonction **populate** est appelée à chaque fois qu'un caractère est entré dans le champ de texte, il suffit alors de séparer le texte en deux et de prendre le premier élément pour chercher le prénom et les éléments restants pour chercher un nom de famille, qui peut donc être composé de deux parties séparées par un espace.

Voici la fonction en question :

```
def sort_specific(self, search):
    if len(search.split(' ')) == 1:
        return "WHERE surname LIKE '" + search + "%' "
    elif len(search.split(' ')) >= 2:
        return "WHERE surname LIKE '" + search.split(' ')[0]
            + "%' AND name LIKE '"
            + " ".join(search.split(' ')[1:]) + "%' "
```


Au tout début de la fonction **populate**, ces deux fonctions sont appelées avec comme paramètres les textes de leurs widgets respectifs puis les résultats de leur exécution sont passés dans la requête vue plus haut et qui permet de récupérer les étudiants dans la base de données.

Le template contenant les données de chaque utilisateur et qui est ajoutée récursivement sur la page est simplement un élément `BoxLayout` horizontal auquel plusieurs labels et une image sont ajoutés. Seulement, ce `BoxLayout` a cela de particulier qu'il hérite des comportements des éléments Boutons (**ButtonBehavior**) ce qui nous permet de faire en sorte de lier l'événement de clic à une action. Autrement dit, il est possible de cliquer sur un élément `BoxLayout` contenant les informations de l'étudiant et d'être redirigé vers une page spécifique qui liste plus d'informations sur l'étudiant sélectionné.

Voici la fonction appelée lors de chaque clic sur un `BoxLayout` contenant les données d'un étudiant et qui est donc renseignée dans la classe spécifique au template Kivy :

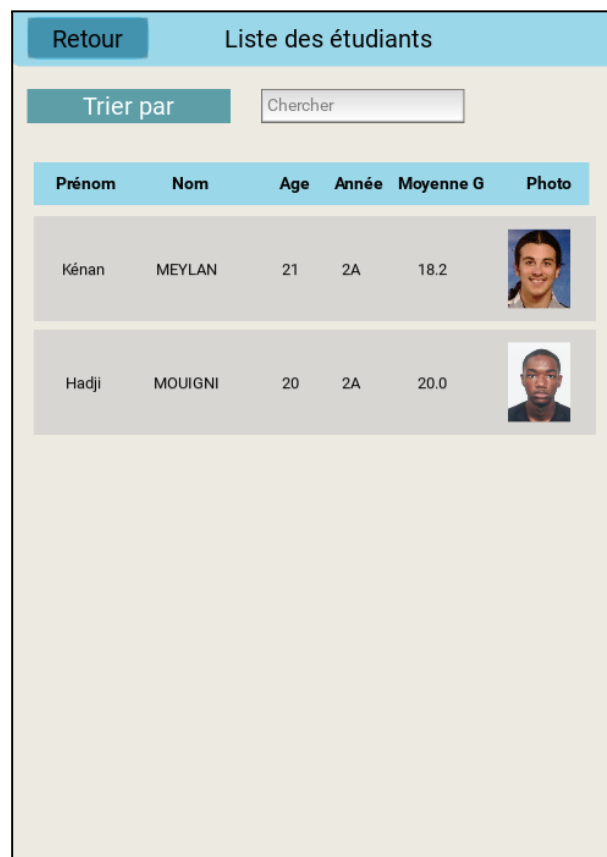
```
def stud_choose(self):  
    global choosen_surname  
    global choosen_name  
    choosen_surname = self.ids.stud_list_surname.text  
    choosen_name = self.ids.stud_list_name.text  
    App.get_running_app().root.current = "StudentView"
```



Cette fonction initialise deux variables globales, une pour le prénom et une autre pour le nom de l'étudiant sélectionné, car ces informations doivent être utilisées dans plusieurs autres classes (notamment dans la classe de la fenêtre qui affiche la vision spécifique d'un étudiant), puis change la fenêtre actuelle de l'application pour la fenêtre de vision spécifique de l'étudiant.

Afin que les différents étudiants soient placés les uns à la suite des autres de manière empilée et que l'on puisse faire défiler tous les étudiants, on utilise un `StackLayout` placé sous un `ScrollView` comme suit :

```
ScrollView:
    do_scroll_x: False
    do_scroll_y: True
    StackLayout:
        orientation: "lr-tb"
        cols: 1
        id: stud_lists
        size_hint: (1, None)
        padding: [root.width/25, 0, root.width/25,
                  root.height/50]
        spacing: [root.width/20, root.height/100]
```

Voici le résultat :



Prénom	Nom	Age	Année	Moyenne G	Photo
Kénan	MEYLAN	21	2A	18.2	
Hadji	MOUIGNI	20	2A	20.0	

Page vision spécifique étudiant

Nolan BEN YAHYA

Lorsqu'on clique sur un étudiant présent dans la liste, on est redirigés vers une page contenant des informations spécifiques à l'étudiant telles que le nom et prénom, la photo, l'âge, l'année d'étude, la moyenne générale et la liste des moyennes de l'étudiant avec les matières correspondantes.

Cette page permet de modifier la photo de l'étudiant dans le cas où celle-ci est invalide ou changée, ajouter une matière avec une moyenne ou supprimer une moyenne existante.

Juste avant l'entrée dans la fenêtre, on remet à zéro les champs de saisie d'une nouvelle moyenne et on appelle la fonction **populate**.

```
def on_pre_enter(self):  
    self.ids.stud_view_moy_error.text = ""  
    self.ids.stud_view_subject_choice_etu.text = "Matière ?"  
    self.ids.stud_view_moyenne_etu.text = ""  
    self.populate(choosen_surname, choosen_name)
```

La fonction populate récupère d'abord les données générales de l'étudiant depuis la base de données au moyen des variables globales initialisées plus tôt comme suit :

```
"SELECT id, age, year, global_moy, photo FROM etudiants WHERE  
surname = '" + stud_surname + "' AND name = '" + stud_name +  
"'"
```

Une fois cela fait, on range les informations obtenues dans un dictionnaire et on ajoute la photo de l'étudiant à la page via la même technique que pour la liste des étudiants :

```
stud_dict = {"ident": data[0], "age": data[1],  
            "year": data[2], "global_moy": data[3],  
            "photo": data[4]}  
self.ids.stud_view_picture.texture =  
CoreImage(io.BytesIO(stud_dict['photo']), ext="png").texture
```

Ensuite, on sélectionne toutes les moyennes et matières correspondantes attribués à l'étudiant :

```
"SELECT subject, moy FROM matieres WHERE id = '" +  
stud_dict["ident"] + "'"
```

On ajoute ensuite les moyennes et matières récupérées à une liste de dictionnaires comme pour la liste des étudiants :

```
for moy in data:
    moy_dict = {"subject": moy[0], "moy": round(moy[1], 2)}
    stud_moy_list.append(moy_dict)
```

On s'assure que le StackLayout contenant les moyennes est vide afin de pouvoir ajouter les matières et moyennes récupérées sans problème :

```
stud_moy_del = [i for i in self.ids.stud_view_moy.children]
for stud_moy_del in stud_moy_del:
    self.ids.stud_view_moy.remove_widget(stud_moy_del)
```

Enfin, dans la dernière boucle qui doit ajouter les moyennes de l'étudiant présentes dans la BDD sur la fenêtre, on récupère la meilleure moyenne dans la matière traitée par la boucle pour l'année d'étude correspondante à celle de l'étudiant visualisé.

Voici la requête SQL qui fait intervenir un INNER JOIN :

```
"SELECT MAX(moy) FROM matières INNER JOIN etudiants ON
matières.id = etudiants.id WHERE subject = '" +
stud_moy_i['subject'] + "' AND year = '" + stud_dict['year'] +
"' LIMIT 1"
```

Finalement, suivant la même technique que pour la liste des étudiants, on crée une instance du template Kivy **EtuMoy** dans laquelle on modifie les informations avant de l'ajouter à la suite du StackLayout **stud_view_moy** :

```
stud_moy = EtuMoy()
stud_moy.ids.stud_view_moy_subject.text =
    stud_moy_i['subject']
stud_moy.ids.stud_view_moy_moy.text =
    str(round(stud_moy_i['moy'], 2))
stud_moy.ids.stud_view_moy_best_moy.text =
    str(round(best_moy, 2))
self.ids.stud_view_moy.add_widget(stud_moy)
```

On ajoute enfin les informations spécifiques à l'étudiant dans un Label sous la photo. Cet ajout se fait à la fin car il faut pouvoir accéder à la moyenne générale récupérée depuis la base de données :

```
self.ids.stud_view_infos.text = "[b]" + stud_surname + " " +  
                                stud_name.upper() + "[/b]\n" +  
                                str(stud_dict['age']) + " ans\n" +  
                                stud_dict['year'] + "\nMoyenne générale : [b]" +  
                                str(round(stud_dict['global_moy'], 2)) + "[/b]"
```

Si on souhaite modifier l'image de l'étudiant, il suffit de cliquer sur la photo de l'étudiant présente sur la page. Ensuite, on convertit l'image selon la même technique que lors de l'ajout d'un étudiant via les fonctions `picture_selected` et `convert_pic`.

On met à jour le BLOB de l'étudiant dans la base de données :

```
new_image = [self.convert_pic(self.photo_path)]  
cursor.execute("UPDATE etudiants SET photo = %s WHERE id =  
(SELECT id FROM etudiants WHERE surname = '" + choosen_surname  
+ "' AND name = '" + choosen_name + "')", new_image)
```

Enfin, on met à jour les informations de la page en changeant deux fois de fenêtre et donc en appelant une nouvelle fois les fonctions de récupération et d'affichage des informations de la base de données.

```
App.get_running_app().root.current = "Liste"  
App.get_running_app().root.current = "StudentView"
```

Il est possible d'ajouter une moyenne et la matière correspondante à l'étudiant depuis cette page. Un Spinner, un TextInput et un Bouton permettent de réaliser cette opération.

Lorsqu'on appuie sur le bouton d'envoi, la fonction `add_moy(self)` est appelée et permet de réaliser les tests suivants :

- Vérifie si la matière est sélectionnée :

```
if self.ids.stud_view_subject_choice_etu.text == "Matière ?":
```

- Vérifie si la moyenne rentrée est valide :

```
elif self.ids.stud_view_moyenne_etu.text == "" or  
round(float(self.ids.stud_view_moyenne_etu.text), 2) < 0 or  
round(float(self.ids.stud_view_moyenne_etu.text), 2) > 20:
```

Des messages d'erreur sont affichés dans un Label au-dessus du bouton selon ces conditions :

```
"Matière invalide"  
"Moyenne invalide"
```

On sélectionne ensuite l'identifiant de l'utilisateur auquel on ajoute la moyenne avec la requête :

```
"SELECT id FROM etudiants WHERE surname = '" + choosen_surname  
+ "'" AND name = '" + choosen_name + "'" LIMIT 1"
```

On ajoute les nouvelles données à envoyer à une liste **moy_add** :

```
moy_add = [ident, self.ids.stud_view_subject_choice_etu.text,  
round(float(self.ids.stud_view_moyenne_etu.text), 2)]
```

On vérifie si la matière est déjà enregistrée pour l'étudiant au moyen d'une nouvelle requête qui doit retourner 0 si la matière n'existe pas et 1 ou un autre chiffre supérieur sinon.

```
"SELECT count(*) FROM matières WHERE id = '" + ident + "'" AND  
subject = '" + self.ids.stud_view_subject_choice_etu.text + "'"  
LIMIT 1"
```

Si la moyenne n'est pas déjà enregistrée pour l'étudiant, on l'ajoute à la table **matières**, puis on met à jour la colonne contenant la moyenne générale de l'étudiant pour que celle-ci prenne en compte la nouvelle moyenne.

```
if subject_exist == 0:  
    cursor.execute("INSERT INTO matières (id, subject, moy)  
                    VALUES (%s, %s, %s)", moy_add)  
    cursor.execute("UPDATE etudiants SET global_moy = (SELECT  
AVG (moy) FROM matières WHERE id = (SELECT id FROM  
etudiants WHERE surname = '" + choosen_surname + "'" AND  
name = '" + choosen_name + "')) WHERE id = (SELECT id  
FROM etudiants WHERE surname = '" + choosen_surname + "'"  
AND name = '" + choosen_name + "'")")
```

Enfin, on envoie les modifications sur la base de données et on ferme la connexion avec celle-ci avant de recharger la page en suivant la même technique que pour la mise à jour de la photo :

```
db.commit()
db.close()
App.get_running_app().root.current = "Liste"
App.get_running_app().root.current = "StudentView"
```

Si la matière est déjà liée à l'étudiant dans la base de données, on affiche le message d'erreur "La matière existe déjà" dans le Label d'erreur au-dessus du bouton d'envoi.

Lorsqu'une moyenne est incorrecte ou que l'étudiant n'en a plus besoin dans son fichier (qu'il y a un changement de semestre par exemple), il faut pouvoir facilement supprimer la moyenne.

Sur le template Kivy EtuMoy qui permet l'affichage de différentes moyennes, un bouton peut être cliqué et fera appel à la fonction `moy_suppr(self)` définie dans sa classe afin de gérer la suppression de la moyenne.

Lorsque l'on appuie sur le bouton, une requête SQL se charge de supprimer la moyenne que l'on veut supprimer dans la base de données :

```
"DELETE FROM matières WHERE id = (SELECT id FROM etudiants
WHERE surname = '" + choosen_surname + "' AND name = '" +
choosen_name + "') AND subject = '" +
self.ids.stud_view_moy_subject.text + '"
```

Une autre requête SQL met à jour la moyenne générale de l'étudiant en prenant en compte la suppression de la moyenne.

```
"UPDATE etudiants SET global_moy = (SELECT AVG (moy) FROM
matières WHERE id = (SELECT id FROM etudiants WHERE surname =
'" + choosen_surname + "' AND name = '" + choosen_name + "'))
WHERE id = (SELECT id FROM etudiants WHERE surname = '" +
choosen_surname + "' AND name = '" + choosen_name + "')
```

Si la moyenne qu'on veut supprimer est la dernière dans la liste des moyennes, on supprime en même temps l'étudiant. En effet, si un étudiant n'a pas au moins une moyenne dans une matière, il n'est pas nécessaire de le garder dans l'application (cela signifie que l'année est terminée, que le semestre change ou que l'étudiant est parti de la structure éducative).

On exécute donc cette requête SQL :

```
"DELETE FROM etudiants WHERE surname = '" + choosen_surname +  
" ' AND name = '" + choosen_name + '"
```

On retourne ensuite sur la page Liste via la ligne :

```
App.get_running_app().root.current = "Liste"
```

Dans le cas où la moyenne supprimée n'est pas la dernière, on recharge simplement la page avec la même technique que lors de l'ajout d'une nouvelle matière :

```
App.get_running_app().root.current = "Liste"  
App.get_running_app().root.current = "StudentView"
```

Dans les deux cas, on publie les changements sur la base de données et on ferme la connexion à celle-ci :

```
db.commit()  
db.close()
```

[Retour](#) Voir/Modifier un étudiant



Kénan MEYLAN
21 ans
2A
Moyenne générale : **18.2**

Ajouter une moyenne

Matière ?

Moyenne

Ajouter

Matière	Moyenne étudiant	Meilleure moyenne	Action
Prog	18.2	18.2	Supprimer

Conclusion

Au cours de cette SAÉ, nous avons pu mettre en pratique nos connaissances acquises en Kivy, base de données, mais aussi en développement Python plus poussé. Nous avons ainsi pu développer une application communicante de gestion d'étudiants et de leurs moyennes.

Au cours de ce projet, nous avons été confrontés à divers problèmes qui nous ont forcé à revoir notre stratégie et les fonctionnalités que nous voulions implémenter comme par exemple le Trombinoscope qui a finalement été supprimé après plusieurs tentatives infructueuses afin d'organiser les informations des étudiants.

Enfin, ce projet aura été pour nous un premier pas dans le développement d'applications et fonctionnalités se rapprochant d'une utilisation concrète dans le monde du travail.