

| **Rapport**  **Application de gestion et suivi des étudiants**  Semestre 3 - 2° Année |
| --- |

# Sommaire

[Sommaire](#_aik187jmn7lr)

[Introduction](#_etkprhk8ohzh)

[Initialisation](#_muqvkohqjkdv)

[Page de login](#_f85o8lbqah9a)

[Page de navigation](#_w95yh7sg0sr0)

[Page d’ajout d’étudiant](#_ro4i6xew7oca)

[Page liste des étudiants](#_9mu0k8v4xsft)

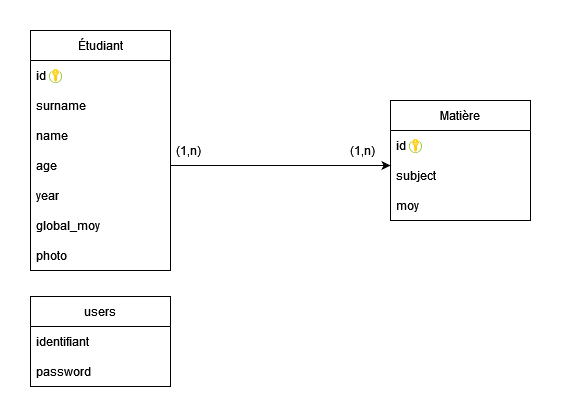
# Introduction

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.

# 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

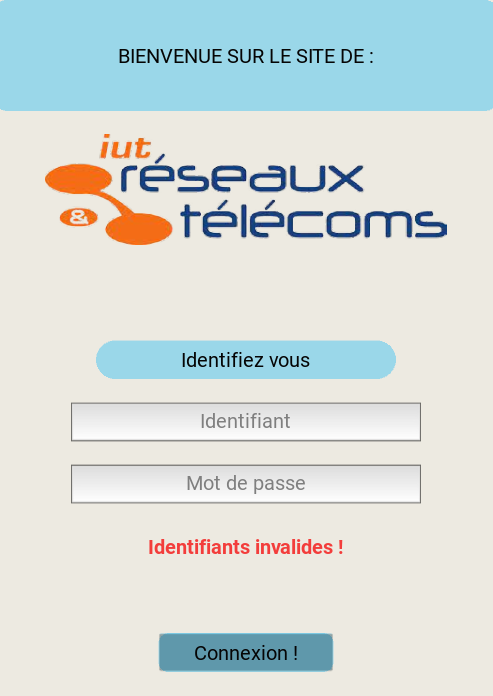
On va créer une deuxième boucle, telle que : pour chaque **“user”** dans la liste **“users”**, si notre variable self.ids.connect\_login.text correspond à la valeur de l’identifiant, présent dans le dictionnaire et que self.ids.connect\_mdp.text correspond à la valeur du mot de passe, on passe alors 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 BBD, 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 information nécessaire :

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 récupérés 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 créée, 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 ScollView comme suit :

ScrollView:

do\_scroll\_x: False

do\_scroll\_y: True

StackLayout:

orientation: "lr-tb"

cols: 1

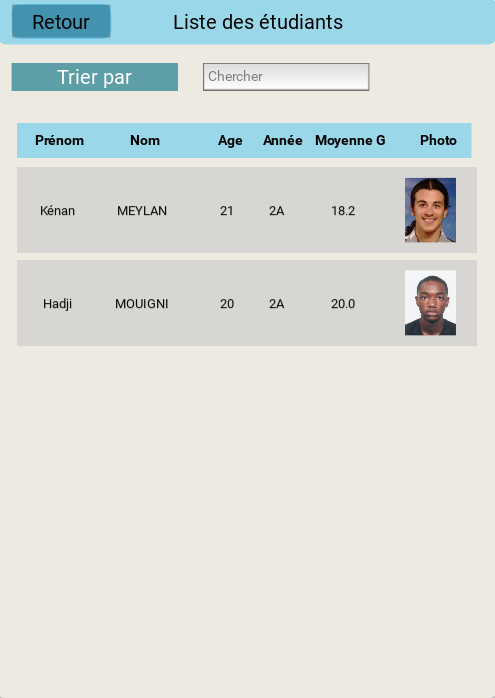
id: stud\_lists

size\_hint: (1, None)

padding: [root.width/25, 0, root.width/25,

root.height/50]

spacing: [root.width/20, root.height/100]

Voici le résulat :

# Page vision spécifique étudiant

Nolan BEN YAHYA