

Nicolas Strekowski
Eric Durand
Julie André
Marine Prunel

Compte rendu projet : **« Grassman »**

Analyse

Dès l'annonce du thème du projet, nous avons réfléchi en groupe à comment répondre aux attentes de l'utilisateur, dans le cadre de la gestion d'un stade à pelouse. Premièrement, nous avons compris qu'il fallait développer une application permettant de récupérer les données de température du stade. L'objectif a donc été de créer une application qui gère les stades appartenant à l'utilisateur comprenant des capteurs et leur courbes associés, du chauffage et de l'arrosage et une gestion du compte client. Cela permettrait aux utilisateurs de pouvoir gérer leur stade plus efficacement, et éviter le dessèchement de la pelouse.

D'abord nous nous sommes répartis les tâches en utilisant Trello:

- L'un a travaillé sur la génération de températures aléatoires mais cohérentes, de précipitation puis sur l'affichage graphique.
- Un autre s'est concentré sur l'affichage du Menu Principal et du design global, ainsi que sur le lien avec la BDD.
- Les deux derniers ont travaillé en coopération sur le menu de création de stade, sur des fonctionnalités diverses comme la création de stade, la personnalisation du nombre de capteurs, la création de compte, et le changement de mot de passe.

Il faut ensuite noter l'utilisation des applications suivantes :

- Utilisation de tkinter comme librairie principale
- Utilisation de POO pour rendre le projet modulaire
- Projet host sur GitHub, le logiciel Git étant très pratique pour synchroniser les changements faits sur différents ordinateurs, et résoudre les problèmes de merge.

Conception

Lien du projet : <https://github.com/JESAIsaps/GrassMan>

Choix :

- Nous avons décidé que les températures ne devraient pas varier trop sur un même terrain, l'ombre n'étant pas vraiment un facteur de température très fort, mis à part pour le ressenti. Donc les températures sont à peu près homogènes sur l'ensemble du terrain, avec quand même quelques variations, de l'ordre du demi-degré.
- Les capteurs ne sont pas affichés sous forme de courbe car cela ne rendrait pas bien dans les variations (trop proches). On affiche donc leur température directement à l'écran.

Généralités :

- Chaque page est un objet
- Créer une base de données qui répertorie les utilisateurs avec leur compte et les stades avec leurs capteurs
- Créer une interface de connexion ou de création de compte
- Possibilité de créer un stade avec un nombre de capteur choisis par l'utilisateur ainsi que leur agencement sur le stade
- Possibilité de changer le mot de passe, de revenir au menu contenant l'accès à tous les stades créés
- Créer des températures et des précipitations aléatoires en fonction des mois selon une moyenne en France
- Afficher la température enregistrée par chaque capteur
- Générer un bouton qui permet d'arroser ou de chauffer lorsque c'est nécessaire en fonction de l'humidité ou des températures
- Générer des prévisions de la température de la journée en fonction de l'allumage du chauffage et/ou de l'arrosage

Organisation de la BDD :

Stade : (Nom :str, Taille :str, nombreCapteurs:int, #clientID:str)

Température : (stade:str, jour:int, temperature:float)

Client : (nomCLient :str, prenomClient:str, identifiant:str, motDePasse:str)

Nous avons décidé de ne stocker dans la base de données qu'une température unique par jour par stade, afin de limiter les appels a la BDD : sinon, on aurait rapidement un problème de place. Pour palier a cette décision, nous avons créé un système de pseudo aléatoire, qui nous permet de créer des températures par heure a partir de ces moyennes.

Le reste n'a rien de très spécifique a notre projet, nous avons essayé de limiter un maximum le stockage d'infos redondantes.

Organisation des fichiers :

L'ensemble des fichiers utilisés dans le code sont stocker sous le dossier « src ». Les données utiles aux programme sont ensuite sous « data ».

Chaque script python représente un objet, que nous développerons dans la partie code.

Structure interne de l'interface :

Nous utilisons Tkinter pour l'interface, ce qui nous amène a devoir décider comment gérer l'interface de Tkinter, qui permet beaucoup de liberté dans la façon d'afficher son code.

Nous avons choisit la voie de la modularité, qui donne a l'utilisateur une liberté maximale.

Structure de l'interface :

L'utilisateur aura accès a plusieurs pages différentes, avec un menu déroulant disponible a tout moment.

Simulation des températures :

Pour simuler nos températures, nous avons fait le choix des simuler tous les jours depuis le 1-1-2000 pour chaque stade à sa création.

Nous avons décidé de ne pas faire appel à une api, car nous ne trouvons pas utile de devoir créer des connexions simplement pour générer des températures qui en plus seraient liées a une zone géographique.

Nous utilisons un algorithme d'aléatoire normalisé que nous développerons dans la partie code.

Evolution des température :

Ayant contrôle complet sur les températures, nous avons décidé de créer les températures de manière à ce qu'il y ait un phénomène de réchauffement climatique, que vous pourrez voir principalement en comparant les températures des années 2000 et 2020.

Pour les température de la journée actuelle, nous mettons en place un système de prédiction, qui sera affecté si l'utilisateur arrose ou chauffe le stade.

Code

Généralités :

- Programmation orienté objet
- Langage : Python (pas le choix)
- Chaque scripte contient une classe
- Environ 1000 lignes dans le code complet, sans compter les bibliothèques importés
- Grand travail d'optimisation (a peu près toutes les opérations sont réalisées a la connexion, c'est donc le seul moment avec une demi seconde de chargement)

Gestion de l'aléatoire :

Pour générer nos températures, nous avons rencontré un problème majeur : de l'aléatoire, pas aléatoire. Pour gérer nos températures affichées, nous voulons qu'elles soient aléatoires, mais toujours les mêmes. On pourrait stocker nos données dans de grandes liste et tables, mais a cause de la manière dont sont faites les listes, le parcours des listes à chaque appel de fonction rendrait tout très lent.

La solution : à chaque fois qu'on veut des températures, on les génère, a partir d'une seed, qui sera très facilement accessible.

Les seeds seront les température moyennes assignées à chaque jour, qui elles même sont créées aléatoirement à la création du stade, de manière aléatoire du module Random.

Après, lorsque l'on veut montrer une plage de températures au cours d'une journée, on va prendre notre valeur moyenne, lui appliquer notre fonction aléatoire, pour obtenir un pseudo aléatoire de qualité.

Notre aléatoire, inspiré d'algos connus d'aléatoire, mais en enlevant beaucoup de cet aléatoire :

```
def aleatoire(a):  
    """  
    Retourne un nombre pseudo-aleatoire entre 0 et 9, a l'aide de la seed a  
    """  
    return int(str(abs((np.exp(a**1.2))%997)*5.3)[0])
```

Créations des température au jour :

Nous avons déjà vu que nous ne créons qu'une seule température par jour.

Pour nous permettre une température pendant un jour qui ne sera pas uniforme, nous avons utilisé la moyenne générée pour le jour en question, ce à quoi nous avons appliqué un sinus, avec un déphasage de sorte qu'il fasse plus chaud le jour que la nuit avec en plus une touche de notre pseudo-aleatoire:

```
def CreateDayTemp(heure, dayMedium)->int:  
    """  
    Renvoie une temperature selon une heure et une moyenne d'un jour.  
    """  
    rep = dayMedium-2 + np.sin(heure*np.pi/12 +aleatoire(dayMedium)/12+4.5)*6  
    return rep
```

Création des précipitations :

Semblable aux températures, se base aussi sur les températures moyennes, mais gère différemment son aléatoire.

Organisation du code :

Nous avons décidé de séparer le code en plusieurs classes. Cette partie sera spécifique à Tkinter.

Le fichier main.py contient l'appli principale, la fenêtre que l'on ouvre, ensuite il va chercher dans les autres scripts les classes requises à l'affichage des pages, qui héritent toutes de tk.Frame. Les codes vraiment effectifs à du calcul sont dans Graphs.py, et les appels à la base de données dans BDDapi.py, le reste ne gère que l'affichage et n'est pas très intéressant, mis à part qu'il est super optimisé et donc rapide dans l'exécution.

Sécurité :

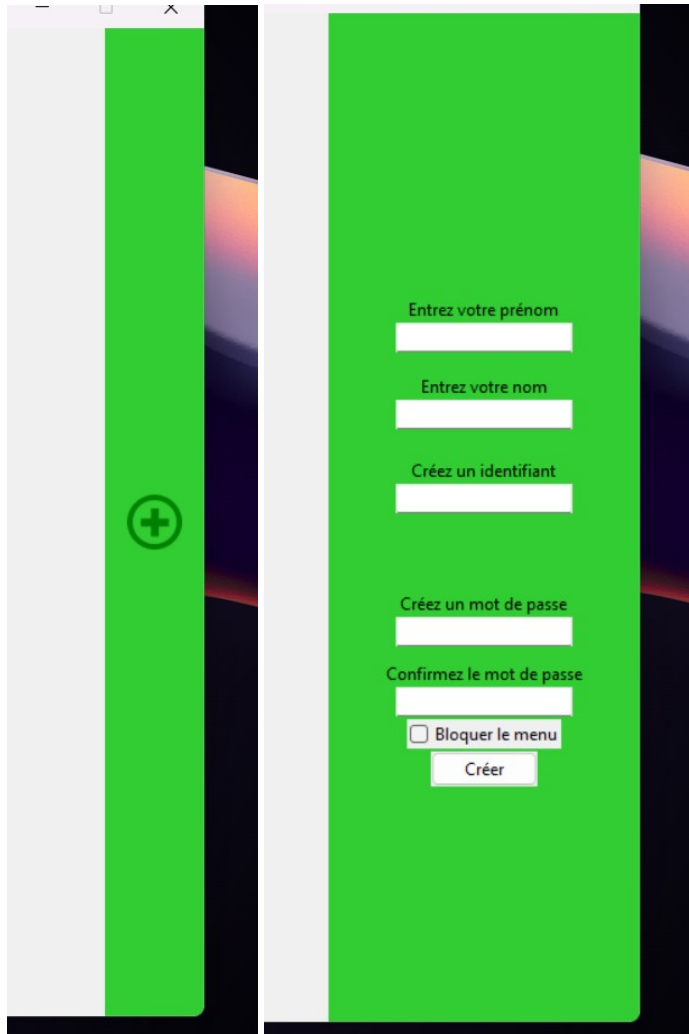
Notre code est entièrement sécurisé : il est protégé contre les injections SQL, et les mots de passe des clients sont stockés sous forme hachée, grâce au module Bcrypt.

Le menu déroulant :

Afin d'avoir un menu accessible a tout moment, pour se déconnecter, retourner au menu... Nous avons décidé de faire un menu déroulant, qui sera toujours instancé.

Pour y accéder, il suffit de déplacer la souris dessus, et il se rétracte quand on enlève la souris.

Si on veut garder ce menu ouvert, on peut sélectionner « bloquer le menu », et alors le menu de se rétractera plus.



The image displays two side-by-side screenshots of a web application interface. The left screenshot shows a green sidebar with a white plus icon. The right screenshot shows the sidebar expanded into a form with the following fields and elements:

- Entrez votre prénom
- Entrez votre nom
- Créez un identifiant
- Créez un mot de passe
- Confirmez le mot de passe
- ☐ Bloquer le menu
- Créer

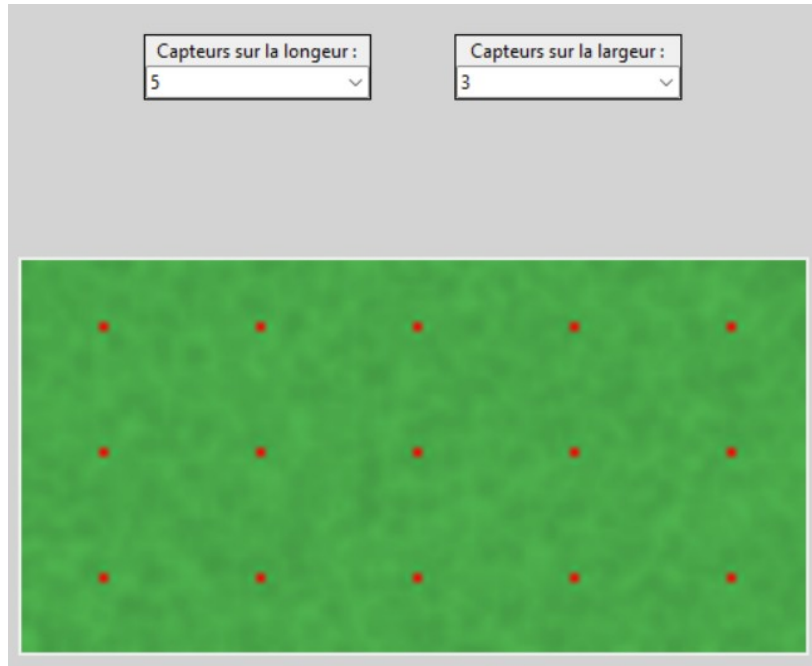
Le menu possède plusieurs versions différentes, selon le contexte, pour créer un utilisateur, ou bien gérer son compte.

La création de stade :

Pour créer le stade, nous voulions au depart laisser libre choix a l'utilisateur de placer autant de capteurs qu'il le veut. Mais pour l'affichage

des courbes, les temps de chargements on commencé a se faire sentir, quand nous avons commencé a avoir 5000 capteurs par stade (100x50).

Nous avons donc décidé de bloquer le choix des capteurs a du 10x5, avec une image entièrement générée algorithmiquement, qui donne un visuel de la répartition des capteurs sur le stade :



```
def DrawStadiumExample(nbX, nbY):  
    """  
    Renvoie le visuel de la repartition des capteurs du stade selon  
    leur nombre nbX et nbY qui correspondent au nombre de capteurs sur  
    x et y  
    """  
  
    imageData = gaussian_filter(  
        [[(0,(40 + randint(1, 10))*7,0) for _ in range(100)] for _ in range(50)],  
        sigma=0.75)  
  
    for i in range(int(25//nbY), 50, int(50//nbY)):  
        for j in range(int(50//nbX), 100, int(100//nbX)):  
            imageData[i][j] = (255, 0, 0)  
  
    finalImage = np.array(imageData).astype(np.uint8)
```

Tests

-Fait au fil du développement, un test final par des tiers une fois la phase principale de développement terminée.