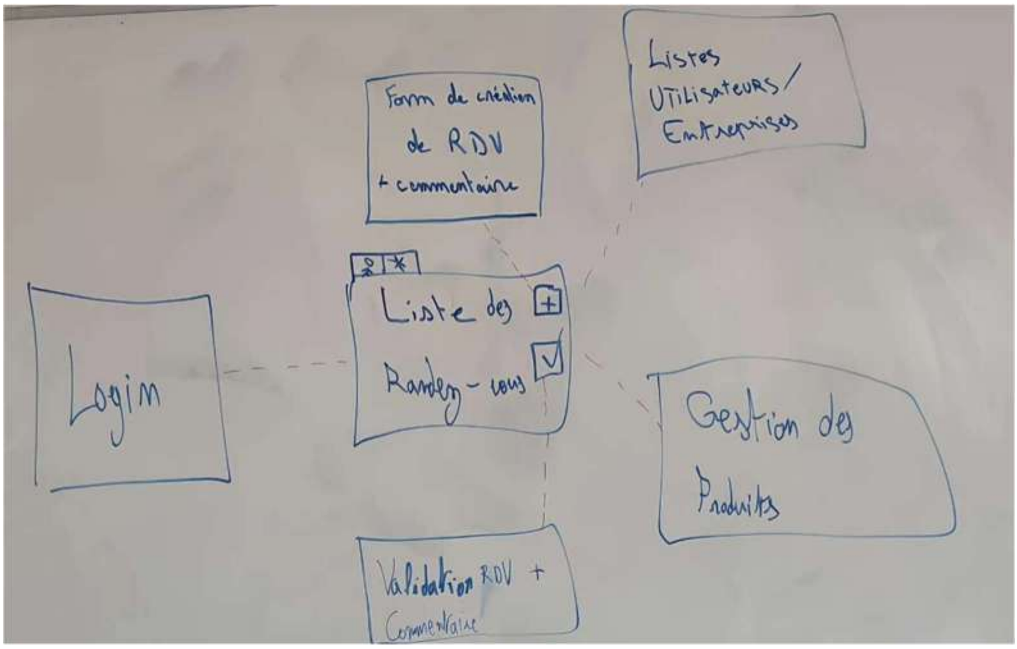
Contre rendu projet application Objet ERP

Sommaire :

1/ Organisation de l’application :

Pour développer une application de type ERP pour notre entreprise qui délivrant des services dans l’évènementiel pour les entreprises et les particuliers, on commence par définir les différentes tâches nécessaires à l’application web.

On fait un schéma de l’organisation des pages web de notre application qui nous servirons à contrôler notre application qui communique avec une base de données :



Comme convenu la première page de notre site sera une page de **connexion** pour les collaborateurs qui permettra, si les identifiants sont corrects, de se connecter et d'accéder à la page qui affiche les **rendez-vous planifiés.**

Cette page permet d'avoir connaissance de l'entreprise/client avec qui un collaborateur à rendez-vous, le lieu, la date, et le/les service(s) à présenter.

Elle permet également de gérer les rendez-vous avec la possibilité de planifier un nouveau rendez-vous, avec la possibilité de mettre un commentaire d'avant rendez-vous via la page de **Planification de rendez-vous.**

Mais aussi de valider des rendez-vous dans la liste lorsque ceux-ci sont effectués, et une possibilité de mettre un commentaire de fin de rendez-vous via la page de **Validation de rendez-vous.**

2/ Technologies utilisées :

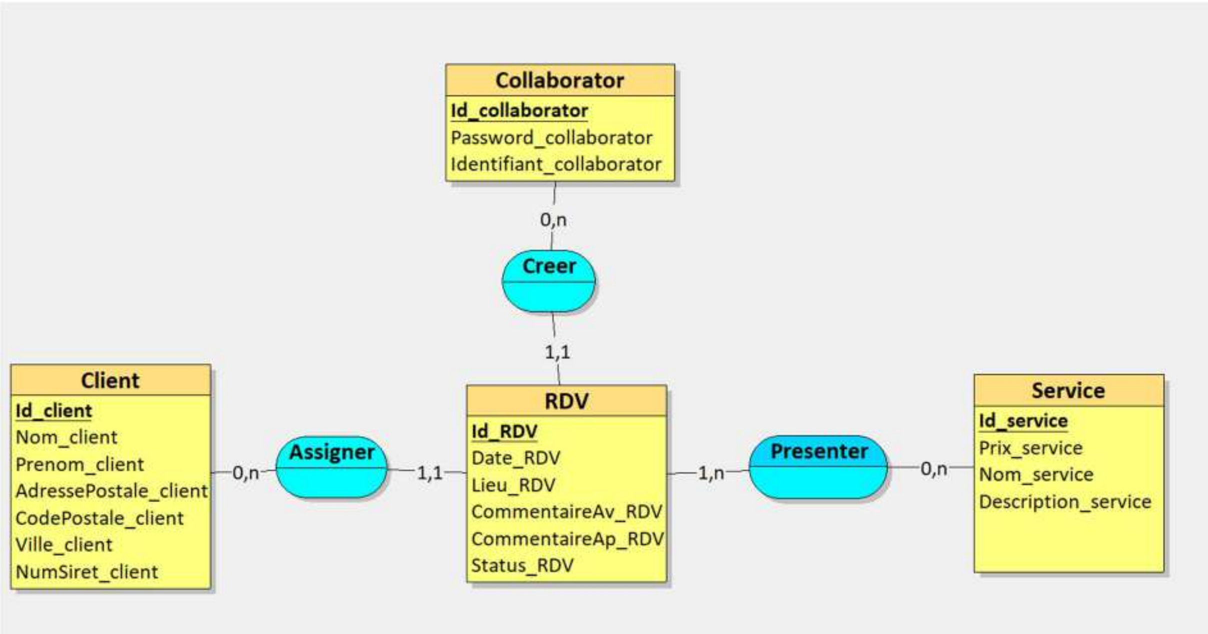
Pour réaliser cette application nous avons décidé d'utiliser le micro-framework Flask pour développer la logique de notre application :

Flask est un micro-framework web écrit en Python, conçu pour être simple, flexible et extensible. Avec son API intuitive et sa structure légère, Flask permet aux développeurs de créer rapidement des applications web de toutes tailles, des petites applications aux sites web à *grande échelle. Il offre une intégration transparente avec Jinja2 pour la gestion des templates et une large gamme d'extensions pour ajouter des fonctionnalités supplémentaires.*

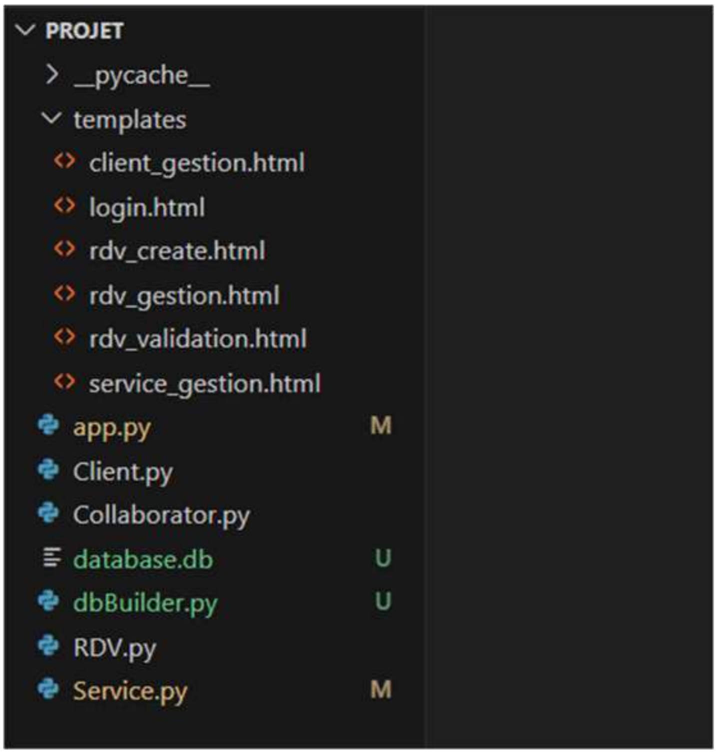
Pour stocker les informations entrées par les collaborateurs dans une base de données nous utiliserons SQLite, avec la bibliothèque Python sqlite3 qui permet la gestion de celle-ci.

Enfin tout le reste de l'application sera développer simplement en HTML, CSS, et python en programmation orienté objet.

3/ Modèle Conceptuel de données (MCD)



4/ Création et intégration de notre base de données



On crée les différents fichiers .py de notre application dans lesquels nous ajouterons plus tard les classes permettant de communiquer avec la base de données.

On ajoute les différentes pages HTML du site.

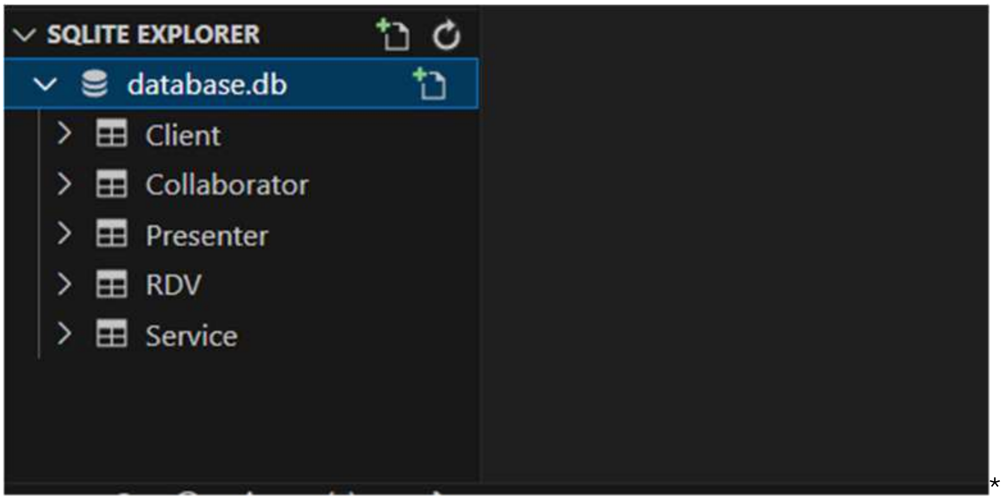
On crée un fichier database.db et un fichier databaseBuilder.py qui servira à construire les tables de notre base de données.

Système sqlite3 pour ajouter des éléments à la base de données :

Code SQL pour créer nos tables :

CREATE TABLE Collaborator(
  Id_collaborator INTEGER,

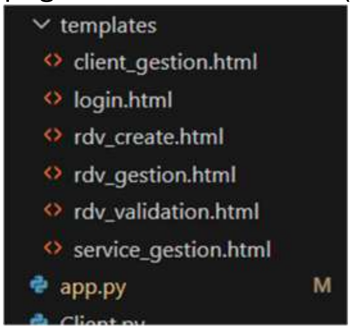
```sql
  Password_collaborator TEXT,
  Identifiant_collaborator TEXT,
  PRIMARY KEY(Id_collaborator)
);

CREATE TABLE Service(
  Id_service INTEGER,
  Prix_service NUMERIC(15,2),
  Nom_service TEXT,
  Description_service TEXT,
  PRIMARY KEY(Id_service)
);

CREATE TABLE Client(
  Id_client INTEGER,
  Nom_client TEXT,
  Prenom_client TEXT,
  AdressePostale_client TEXT,
  CodePostale_client INTEGER,
  Ville_client TEXT,
  NumSiret_client TEXT,
  PRIMARY KEY(Id_client)
);

CREATE TABLE RDV(
  Id_RDV INTEGER,
  Date_RDV NUMERIC,
  Lieu_RDV TEXT,
  CommentaireAv_RDV TEXT,
  CommentaireAp_RDV TEXT,
  Status_RDV NUMERIC,
  Id_collaborator INTEGER NOT NULL,
  Id_client INTEGER NOT NULL,
  PRIMARY KEY(Id_RDV),
  FOREIGN KEY(Id_collaborator) REFERENCES Collaborator(Id_collaborator),
  FOREIGN KEY(Id_client) REFERENCES Client(Id_client)
);

CREATE TABLE Presenter(
  Id_service INTEGER,
  Id_RDV INTEGER,
  PRIMARY KEY(Id_service, Id_RDV),
  FOREIGN KEY(Id_service) REFERENCES Service(Id_service),
  FOREIGN KEY(Id_RDV) REFERENCES RDV(Id_RDV));
```



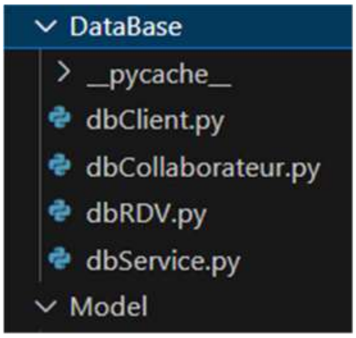
*

5/ Mise en place d’une organisation MVC (Modèle/vue/controlleur)

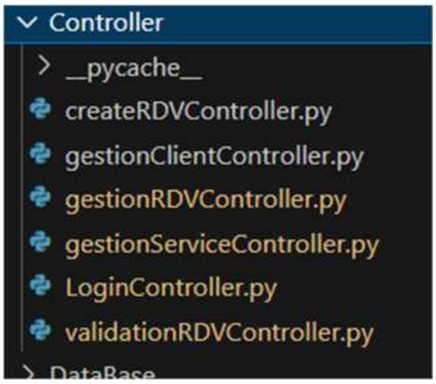
Dans un premier temps, maintenant que l’on a créé dans notre projet la base de donnée et que celle-ci fonctionne, on peut créer un répertoire **Template** pour stocker toutes les pages html de nôtre site (vues) :



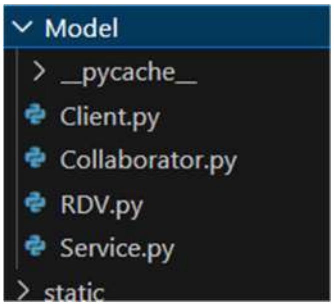
On crée un répertoire **DataBase** pour stocker les fichiers correspondant à chacune de nos tables de données, dans lesquels on va écrire toutes les méthodes pour interagir avec la table : CRUD : create, read, update, delete.



On crée un répertoire **Controller** pour chacune de nos pages, on crée une classe pour la page, pour y coder la logique, récupérer les données des formulaires etc.



On crée un répertoire **<u>Model</u>** pour stocker les classes de chaque table de données.
On créera un objet (une instance de la classe correspondante) pour CHAQUE enregistrement.

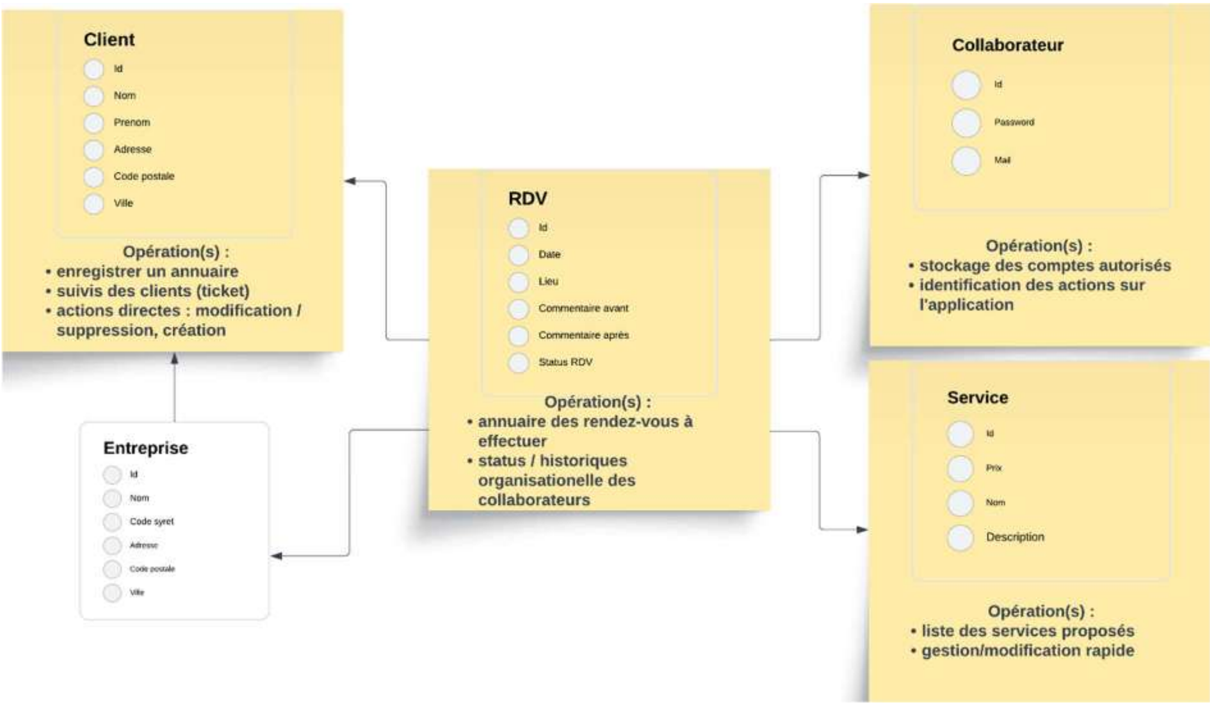


6/ Diagramme UML de nos objets

Un diagramme UML (Unified Modeling Language) est un outil de modélisation visuelle utilisé pour représenter graphiquement différents aspects d'un système logiciel. Il est largement utilisé dans le domaine du génie logiciel pour concevoir, spécifier, visualiser et documenter les différentes parties d'un système logiciel.

Les diagrammes UML offrent un langage standardisé pour représenter les structures. Il en existe plusieurs types, dans le cas de notre projet il s 'agira d'un diagramme de classes afin de présenter la structure statique d'un système en montrant les classes du système, leurs attributs, leurs méthodes et les relations entre elles.

Pour cela nous avons utilisé une application web permettant de construire ses diagrammes de classe en ligne : https://lucid.app :



On peut y apercevoir :

- Les différentes classes répertoriées
- La classe Entreprise qui hérite de la classe parente Client
- Les opérations / actions des différentes classes dans notre application
- Les attribues des constructeurs de chaque classe.
- Le transfert des informations des classes vers la classe RDV

7/ Système de connexion des collaborateurs

```python
from flask import render_template, request, redirect, url_for
from DataBase.dbCollaborateur import CollaborateurDatabase

class LoginController():
    def __init__(self) -> None:
        self.dbCollaborateur = CollaborateurDatabase()

    def page_login(self):
        email = request.args.get('email')
        password = request.args.get('password')
        if email is not None and password is not None:
            if self.dbCollaborateur.getColaboCertif(email, password):
                return redirect(url_for('gestion_rdv_view'))
            else:
                return redirect(url_for('index'))
        return render_template('login.html')
```

Le code présenté ci-dessus concerne le contrôleur de la page de connexion d'une application Flask.

Nous commençons par importer les fonctionnalités nécessaires de Flask pour gérer les requêtes HTTP et les redirections, ainsi que la classe CollaborateurDatabase depuis le module dbCollaborateur pour interagir avec la base de données des collaborateurs.

Ensuite, nous définissons une classe appelée LoginController qui agira comme le contrôleur pour la page de connexion.Dans le constructeur init, nous initialisons un objet CollaborateurDatabase pour accéder à la base de données des collaborateurs.La méthode page_login est appelée lorsqu'un utilisateur tente de se connecter. Elle récupère les données saisies par l'utilisateur pour l'email et le mot de passe à partir de la requête HTTP.

Ensuite, elle vérifie si l'email et le mot de passe sont non vides. Si c'est le cas, elle appelle la méthode getColaboCertif de l'objet dbCollaborateur pour vérifier si les informations fournies correspondent à un collaborateur certifié dans la base de données. Si les informations sont correctes, l'utilisateur est redirigé vers la page de gestion des rendez-vous, sinon il est redirigé vers la page d'accueil.

Si aucun email ou mot de passe n'a été fourni, la méthode renvoie simplement le modèle de la page de connexion pour permettre à l'utilisateur de saisir ses informations.

```python
import dbBuilder
from Model.Collaborator import Collaborator

class CollaborateurDatabase:
    def __init__(self) -> None:
        pass

    def getColaboCertif(self, mail:str, password:str):
        request = 'SELECT * FROM Collaborator WHERE identifiant_collaborator = "{}" AND Password_collaborator = "{}"'.format(mail, password)
        registers = dbBuilder.getRegisterInDataBase(request)
        result = []
        for row in registers:
            collaborator = Collaborator(row[0], row[1], row[2])  # Renamed variable to avoid shadowing class name
            result.append(collaborator)
        return result
```

Ce fragment de code illustre la classe CollaborateurDatabase. Permettez-moi de vous en donner un aperçu.

Le code commence par importer le module dbBuilder et la classe Collaborator du module Model. dbBuilder semble être un module qui gère les requêtes à la base de données, tandis que Collaborator est probablement une classe qui représente un collaborateur dans le système.

Ensuite, nous avons la définition de la classe CollaborateurDatabase. Pour l'instant, dans son constructeur init, il n'y a rien de défini, mais cela pourrait être utilisé pour initialiser des éléments liés à la base de données.

La méthode getColaboCertif est utilisée pour vérifier les informations d'identification d'un collaborateur. Elle prend comme arguments l'adresse e-mail et le mot de passe du collaborateur. Cette méthode construit une requête SQL pour sélectionner les enregistrements correspondants dans la table Collaborator de la base de données. Ensuite, elle exécute cette requête à l'aide de la fonction getRegisterInDataBase du module dbBuilder, qui semble renvoyer les enregistrements correspondants.

Enfin, les résultats de la requête sont parcourus pour créer des objets Collaborator à partir des données de chaque enregistrement. Ces objets sont ajoutés à une liste result, qui est ensuite renvoyée.

```python

class Collaborator():
    def __init__(self, id, password, email) -> None:
        self.id = id
        self.password = password
        self.email = email

    def getId(self):
        return self.id

    def setId(self, id):
        self.id = id

    def getPassword(self):
        return self.password

    def setPassword(self, password):
        self.password = password

    def getEmail(self):
        return self.email

    def setEmail(self, email):
        self.email = email
```

Cette classe, nommée Collaborator, semble représenter un collaborateur dans un système. Permettez-moi de vous expliquer son fonctionnement :

- Le constructeur __**init**__ est utilisé pour initialiser les attributs de l'objet Collaborator avec les valeurs passées en arguments : id, password et email. Ces attributs sont ensuite accessibles à partir de n'importe quelle instance de la classe Collaborator.
- Les méthodes **getId**, **setId**, **getPassword**, **setPassword**, **getEmail** et **setEmail** fournissent des accesseurs et des mutateurs pour les attributs id, password et email de la classe Collaborator. Les accesseurs permettent de récupérer les valeurs des attributs, tandis que les mutateurs permettent de modifier ces valeurs. Par exemple, **getId()** renvoie l'ID du collaborateur, et **setId()** permet de définir un nouvel ID pour le collaborateur.

Cela rend la classe Collaborator modulaire et permet de manipuler ses attributs de manière sécurisée en utilisant les méthodes définies.

```html
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="{{ url_for('static', filename='login.css') }}">
    <link href='https://unpkg.com/boxicons@2.1.4/css/boxicons.min.css' rel='stylesheet'>
    <title>Login</title>
</head>
<body>
    <div class="form-container">
        <form action="#" method="get">
            <h1>Login</h1>
            <label id="label-email">Email</label>
            <input type="email" class="input-email" placeholder="Email" name="email">
            <label id="label-password">Password</label>
            <input type="password" class="input-password" placeholder="Password" name="password"><i class='bx bx-low-vision'></i>
            <button>Login</button>
        </form>
    </div>
    <script>
        const LowVision = document.querySelector(".bx-low-vision");
        const Password = document.querySelector(".input-password");

        LowVision.addEventListener("click", () => {
            if (Password.type === "password") {
                Password.type = "text";
            } else {
                Password.type = "password";
            }
        });
    </script>
</body>
</html>
```

Ce code HTML représente une page de connexion. Le formulaire contient des champs pour l'email et le mot de passe. L'icône à côté du champ de mot de passe permet de basculer entre l'affichage du mot de passe et son masquage. Le formulaire est soumis via la méthode GET vers une URL non spécifiée. Le CSS est lié via un fichier externe login.css pour styliser la page.

8/ Exemple d’interaction BDD et de manipulation d’objets

Gestion des services proposés :

I/ Création d’une nouvelle Template avec la liste des services :

Dans dbService :

On crée des méthodes permettant d’interagir avec la table Service :

```python
from Model.Service import Service

class ServiceDatabase:
    def __init__(self) -> None:
        pass

    def AddService(self, nom_service:str, prix:float, description:str):
        request = 'INSERT INTO Service (Prix_service, Nom_service, Description_service) VALUES (:prix, :nom, :description)'
        dbBuilder.sendRequestInDataBase(request, {'prix': prix, 'nom': nom_service, 'description': description}) # on envoie 

    def getAllServices(self):
        # Récupérer tous les enregistrements
        registers = dbBuilder.getRegisterInDataBase('SELECT * FROM Service')
        # et les stocker dans une liste d'objets Service
        result = []
        for row in registers:
            service = Service(row[0], row[1], row[2], row[3])
            result.append(service)
        return result

    def deleteService(self, Id_service):
        # récupère un enregistrement avec l'id
        request = 'DELETE FROM Service WHERE Id_service=' + str(Id_service)
        dbBuilder.sendRequestInDataBase(request)
```

On a préalablement créé deux fonctions dans un fichier dbBuilder.py qui permettent d’envoyer des requêtes sql à la base de données ou de récupérer des enregistrements :

```python
# fonction pour envoyer des informations à la BDD
def sendRequestInDataBase(request:str, values:dict={}):
    connect = sqlite3.connect('database.db')
    # on prépare la requête
    cur = connect.cursor()
    cur.execute(request, values)
    connect.commit()
    connect.close()


# fonction pour récupéré un ou plusieurs enregistrement d'une table de la BDD
def getRegisterInDataBase(request:str):
    connect = sqlite3.connect('database.db')
    cur = connect.cursor()
    # Exécuter la requête SQL pour récupérer le(s) enregistrement(s) de la table
    cur.execute(request)
    # on retourne la liste des enregistrements
    list = cur.fetchall()
    # ferme la connexion à la BDD
    connect.close()
    return list
```

Dans gestionServiceController :

On écrit la logique de notre page qui liste les services pour récupérer les données des formulaires, et permettre également de passé dans un tableau les variables à afficher sur la page.

```python
class GestionServiceController():
    def __init__(self) -> None:
        self.dbService = ServiceDatabase()

    def page_gestion_service(self):
        # ajout
        name = request.args.get('name')
        price = request.args.get('price')
        description = request.args.get('description')
        if name != None and price != None and description != None:
            self.dbService.AddService(name, float(price), description) # on crée une enregistrement dans la bdd
            redirect('/')

        # suppression
        if request.args.get('deleteId') != None:
            self.dbService.deleteService(request.args.get('deleteId')) # on récupère l'id de l'enregistrement à supprimer

        # on récupère la liste des services dans la BDD pour les afficher
        allServices = self.dbService.getAllServices()

        return render_template('service_gestion.html', services=allServices, addForm=str(request.args.get('addService')))

MS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS
```

On voit ci-dessus les variables services, et addForm que l'on envoi sur la vue.

Addform va permettre s'il est égal à 'yes' d'afficher un formulaire supplémentaire pour ajouter un nouveau service.

```html
<body>
    <h1>Page Gestion des Services</h1>

    <h2>Liste des Services <a id=serviceList href="?addService=yes">Ajouter</a></h2>
    <ul id="serviceList">
        <!-- formulaire d'ajout d'un service -->
        {% if addForm == 'yes' %}
            <form action="" method="get">
                <h2>Ajouter un service</h2>
                <label for="name">Nom du service</label>
                <input type="text" id="name" name="name">
                <label for="price">Prix</label>
                <input type="number" id="price" name="price">
                <label for="description">Description</label>
                <input type="text" id="description" name="description">
                <button type="submit">Créer</button>
            </form>
        {% endif %}
        <!-- Liste des services de la BDD -->
        {% for service in services %}
```

```html
        <!-- Liste des services de la BDD -->
        {% for service in services %}
        <li>
            <span>{{ service.nom }}</span>
            <div>
                {{ service.prix }}€
                <p>{{ service.description }}</p>
            </div>
            <div>
                <a id="serviceList" href="?modifyId={{ service.id }}">Modifier</a>
                <a id="serviceList" href="?deleteId={{ service.id }}">Supprimer</a>
            </div>
        </li>
        {% endfor %}
    </ul>

</body>
```

Gestion des rendez-vous :

Classe Gestion RDV Controller :

```python
def __init__(self) -> None:
    self.dbRDV = RDVDatabase()
```

Cette méthode initialise une instance de Gestion RDV Controller et crée une variable d'instance db RDV, qui est une instance de la classe RDV Database.

Méthode page gestion rdv :

```
def page_gestion_rdv(self):
    allRDVS = self.dbRDV.getAllRDVS()
    if request.args.get('deleteId') != None:
        self.dbRDV.deleteRDV(request.args.get('deleteId'))
    return
render_template('/ProjetAppObjet/templates/rdv_gestion.html',
rdvs=allRDVS, addForm=str(request.args.get('addRDV')))
```

Cette méthode est responsable de la rendu du modèle rdv gestion.html avec tous les rendez-vous (RDV) récupérés depuis la base de données.

Si la requête contient un paramètre 'delete Id', cela indique une demande de suppression d'un rendez-vous avec l'ID correspondant, qui est alors supprimé de la base de données.

La méthode renvoie ensuite le modèle rendu avec les rendez-vous et un formulaire d'ajout si une demande d'ajout de rendez-vous est reçue.

Méthode getAllRDVS :

```
def getAllRDVS(self):
    registers = dbBuilder.getRegisterInDataBase('SELECT * FROM RDV')
    result = []
    for row in registers:
        rdv = RDV(row[0], row[1], row[2], row[3], row[4], row[5],
row[6], row[7])
        result.append(rdv)
    return result
```

Cette méthode récupère tous les rendez-vous (RDV) à partir de la base de données et les retourne sous forme d'une liste d'objets RDV.

Méthode modify RDV :

```
def modifyRDV(self, rdv: RDV):
    request = 'UPDATE RDV SET commentaire_apres=:commentaire_apres, date=:date, lieu=:lieu, commentaire_avant=:commentaire_avant, status=:status WHERE id_rdv=:id_rdv'
    dbBuilder.sendRequestInDataBase(request, {
        'commentaire_apres': rdv.getCommentaireApres(),
        'date': rdv.getDate(),
        'lieu': rdv.getLieu(),
        'commentaire_avant': rdv.getCommentaireAvant(),
        'status': rdv.getStatus(),
        'id_rdv': rdv.getID()
    })
```

Cette méthode modifie un rendez-vous existant dans la base de données avec les nouvelles informations fournies.

Méthode delete RDV :

```
def deleteRDV(self, id_rdv):
```

Cette méthode supprime un rendez-vous spécifié de la base de données en fonction de son identifiant.

Modèle HTML rdv gestion.html :

Ce modèle HTML est utilisé pour afficher la page de gestion des rendez-vous.

Méthode validate RDV :

```
def validateRDV(self, id_rdv):
    rdv = self.getRDVByID(id_rdv)
    if rdv:
```

```
        rdv.setStatus("Validé")
        self.modifyRDV(rdv)
    else:
        pass
```

Cette méthode est utilisée pour valider un rendez-vous spécifique en fonction de son identifiant id rdv.

Elle commence par appeler la méthode get RDV By ID(id rdv) pour obtenir le rendez-vous correspondant à l'ID fourni.

Si le rendez-vous est trouvé (c'est-à-dire s'il n'est pas None), la méthode modifie le statut du rendez-vous en "Validé" en utilisant la méthode setStatus("Validé") sur l'objet RDV.

Ensuite, elle appelle la méthode modify RDV(rdv) pour mettre à jour les informations du rendez-vous dans la base de données.

Si aucun rendez-vous correspondant à l'ID fourni n'est trouvé, la méthode ne fait rien.

Méthode get RDVByID :

```
def getRDVByID(self, id_rdv):
    registers = dbBuilder.getRegisterInDataBase('SELECT * FROM RDV WHERE id_rdv=:id_rdv', {'id_rdv': id_rdv})
    if registers:
        row = registers[0]
        rdv = RDV(row[0], row[1], row[2], row[3], row[4], row[5], row[6], row[7])
        return rdv
    else:
        return None
```

Cette méthode est utilisée pour récupérer un rendez-vous à partir de la base de données en fonction de son identifiant id rdv.

Elle exécute une requête SQL pour sélectionner le rendez-vous avec l'ID spécifié.

Si un enregistrement est retourné (c'est-à-dire si registers n'est pas vide), elle extrait les données de la première ligne de l'enregistrement, crée un objet RDV avec ces données et le retourne.

Si aucun enregistrement correspondant n'est trouvé, la méthode retourne None.

Ces deux méthodes travaillent ensemble pour valider un rendez-vous en récupérant d'abord le rendez-vous par son ID, puis en mettant à jour son statut et enfin en enregistrant les modifications dans la base de données.

Méthode page validation rdv :

```
def page_validation_rdv(self):
    if request.args.get('validate_id') is not None:
        rdv_id = request.args.get('validate_id')
        self.dbRdv.validateRDV(rdv_id)
        return redirect('/validation-rdv')

    rdvs_to_validate = self.dbRdv.getRDVToValidate()

    return render_template('validation_rdv.html',
rdvs=rdvs_to_validate)
```

Cette méthode est responsable de l'affichage de la page de validation des rendez-vous.

Si la requête contient un paramètre 'validate_id', cela signifie qu'un rendez-vous doit être validé. Dans ce cas, l'ID du rendez-vous à valider est extrait de la requête, puis la méthode validate RDV de RDV Database est appelée pour valider ce rendez-vous. Ensuite, la page est redirigée vers elle-même pour rafraîchir la liste des rendez-vous à valider.

En l'absence d'une demande de validation de rendez-vous, la méthode récupère la liste des rendez-vous à valider en appelant la méthode get RDV Validation de RDV Database.

Enfin, elle rend le modèle HTML validation rdv.html avec la liste des rendez-vous à valider.

Gestion des Clients et Entreprises :

1) Dans dbClients :

Ce code crée une classe ***ClientDatabase*** pour interagir avec une base de données contenant des informations sur les clients. Il fournit des méthodes pour ajouter, récupérer, modifier et supprimer des clients dans cette base de données.

```python
import dbBuilder
from Model.Client import Client

class ClientDatabase:
    def __init__(self) -> None:
        pass

    def addClient(self, client: Client):
        request = 'INSERT INTO Client (Nom_client, Prenom_client, AdressePostale_client, CodePostale_client, Ville_client, NumSiret_client) VALUES (:nom, :prenom, :adresse_postale, :code_postal, :ville, :numsiret)'
        dbBuilder.sendRequestInDataBase(request, {'nom': client.getNom(), 'prenom': client.getPrenom(), 'adresse_postale': client.getAdresse_Postale(), 'code_postal': client.getCode_Postal(), 'ville': client.getVille(), 'numsiret': client.getNumSiret()})

    def getAllClients(self):
        registers = dbBuilder.getRegisterInDataBase('SELECT * FROM Client')
        result = []
        for row in registers:
            client = Client(row[0], row[1], row[2], row[3], row[4], row[5], row[6])
            result.append(client)
        return result

    def modifyClient(self, client: Client):
        request = 'UPDATE Client SET Nom_client=:nom, Prenom_client=:prenom, AdressePostale_client=:adresse_postale, CodePostale_client=:code_postal, Ville_client=:ville, NumSiret_client=:numsiret WHERE Id_client=:id'
        dbBuilder.sendRequestInDataBase(request, {'nom': client.getNom(), 'prenom': client.getPrenom(), 'adresse_postale': client.getAdresse_Postale(), 'code_postal': client.getCode_Postal(), 'ville': client.getVille(), 'numsiret': client.getNumSiret(), 'id': client.getID()})

    def deleteClient(self, id_client):
        request = 'DELETE FROM Client WHERE Id_client=' + str(id_client)
        dbBuilder.sendRequestInDataBase(request)
```

2) Dans dbEntreprise :

Ce code définit une classe ***EntrepriseDatabase*** qui hérite de la classe ***ClientDatabase***. Cette classe est spécifiquement conçue pour interagir avec une base de données contenant des informations sur les entreprises.

```python
from Model.Client import Entreprise
from dbClient import ClientDatabase

class EntrepriseDatabase(ClientDatabase):
    def __init__(self):
        super().__init__()

    def addEntreprise(self, entreprise: Entreprise):
        super().addClient(entreprise)
        request = 'UPDATE Client SET NumSiret_client=:numsiret WHERE Id_client=:id'
        dbBuilder.sendRequestInDataBase(request, {'numsiret': entreprise.getNumSiret(), 'id': entreprise.getID()})
```

3) Dans gestionClientController :

On a un contrôleur Flask appelé ***GestionClientController*** pour gérer les interactions liées aux clients dans une application web. Il utilise un objet ***ClientDatabase*** pour interagir avec la base de données des clients. La méthode ***page_gestion_client*** gère l'affichage de la page de gestion des clients et les actions associées, telles que l'ajout, la modification et la suppression de clients.

Il récupère les données du formulaire soumis, effectue les opérations nécessaires sur la base de données, puis renvoie le rendu de la page avec les données des clients et le formulaire de modification.

```python
from flask import render_template, request, redirect
from DataBase.dbClient import ClientDatabase
from Model.Client import Client

class GestionClientController():
    def __init__(self) -> None:
        self.client_db = ClientDatabase()

    def page_gestion_client(self):
        # Ajout d'un client
        if request.method == 'POST':
            id = request.form['modifyId']
            nom = request.form['nom']
            prenom = request.form['prenom']
            adresse_postale = request.form['adresse_postale']
            code_postal = request.form['code_postal']
            ville = request.form['ville']
            numsiret = request.form['numsiret']
            if id is None and nom is not None and prenom is not None and adresse_postale is not None and code_postal is not None and ville is not None and numsiret is not None:
                new_client = Client(None, nom, prenom, adresse_postale, code_postal, ville, numsiret)
                self.client_db.addClient(new_client)
            elif nom is not None and prenom is not None and adresse_postale is not None and code_postal is not None and ville is not None and numsiret is not None:
                modify_client = Client(int(id), nom, prenom, adresse_postale, code_postal, ville, numsiret)
                # modification d'un client
                self.client_db.modifyClient(modify_client)

        # Suppression d'un client
        delete_id = request.args.get('deleteId')
        if delete_id is not None:
            self.client_db.deleteClient(delete_id)

        # Récupération de la liste des clients pour les afficher
        all_clients = self.client_db.getAllClients()
        # formulaire de modification
        modify_form = request.args.get('modify')

        return render_template('client_gestion.html', clients=all_clients, modify_form=modify_form)
```

4) Dans gestionEntrepriseController :

On a un contrôleur Flask appelé ***GestionEntrepriseController*** pour gérer les interactions liées aux entreprises dans une application web. Il hérite des fonctionnalités de gestion des clients de la classe ***GestionClientController***.

Il utilise un objet ***EntrepriseDatabase*** pour interagir avec la base de données des entreprises. La méthode ***page_gestion_entreprise*** gère l'affichage de la page de gestion des entreprises et les opérations associées, telles que l'ajout, la modification et la suppression d'entreprises.

Il récupère les données du formulaire soumis, effectue les opérations nécessaires sur la base de données des entreprises, puis renvoie le rendu de la page avec les données des entreprises et le formulaire de modification.

```python
from flask import render_template, request, redirect
from DataBase.dbClient import EntrepriseDatabase
from GestionClientController import GestionClientController
from Model.Client import Entreprise

class GestionEntrepriseController(GestionClientController):
    def __init__(self):
        super().__init__()
        self.entreprise_db = EntrepriseDatabase()

    def page_gestion_entreprise(self):
        if request.method == 'POST':
            id = request.form['modifyId']
            nom = request.form['nom']
            prenom = request.form['prenom']
            adresse_postale = request.form['adresse_postale']
            code_postal = request.form['code_postal']
            ville = request.form['ville']
            numsiret = request.form['numsiret']
            if id is None and nom is not None and prenom is not None and adresse_postale is not None and code_postal is not None and ville is not None and numsiret is not None:
                new_entreprise = Entreprise(None, nom, prenom, adresse_postale, code_postal, ville, numsiret)
                self.entreprise_db.addEntreprise(new_entreprise)
            elif nom is not None and prenom is not None and adresse_postale is not None and code_postal is not None and ville is not None and numsiret is not None:
                modify_entreprise = Entreprise(int(id), nom, prenom, adresse_postale, code_postal, ville, numsiret)
                self.entreprise_db.modifyClient(modify_entreprise)

        delete_id = request.args.get('deleteId')
        if delete_id is not None:
            self.entreprise_db.deleteClient(delete_id)

        all_entreprises = self.entreprise_db.getAllClients()
        modify_form = request.args.get('modify')

        return render_template('entreprise_gestion.html', entreprises=all_entreprises, modify_form=modify_form)
```

5) Voici la classe Client :

La classe Client représente un client dans le système. ***Elle possède les attributs suivants :***

- ID : identifiant du client
- Nom : nom du client
- Prénom : prénom du client
- Adresse_Postale : adresse postale du client
- Code_Postal : code postal du client
- Ville : ville du client

Elle possède également des méthodes pour accéder et modifier ces attributs :

- getID() et setID(id): pour accéder et modifier l'identifiant du client
- getNom() et setNom(nom): pour accéder et modifier le nom du client
- getPrenom() et setPrenom(prenom): pour accéder et modifier le prénom du client
- getAdresse_Postale() et setAdresse_Postale(adresse_postale): pour accéder et modifier l'adresse postale du client
- getCode_Postal() et setCode_Postale(code_postal): pour accéder et modifier le code postal du client
- getVille() et setVille(ville): pour accéder et modifier la ville du client

```python
import dbBuilder

# Classe Clients
class Client :
    def __init__(self, ID, Nom, Prenom, Adresse_Postale, Code_Postal, Ville):
        self.id = ID
        self.nom = Nom
        self.prenom = Prenom
        self.adresse_postale = Adresse_Postale
        self.code_postal = Code_Postal
        self.ville = Ville


    def getID(self):
        return self.id

    def setID(self, id):
        self.id = id


    def getNom(self):
        return self.nom

    def setNom(self, nom):
        self.nom = nom


    def getPrenom(self):
        return self.prenom

    def setPrenom(self, prenom):
        self.nom = prenom


    def getAdresse_Postale(self):
        return self.adresse_postale

    def setAdresse_Postale(self, adresse_postale):
        self.adresse_postale = adresse_postale


    def getCode_Postal(self):
        return self.code_postal

    def setCode_Postale(self, code_postal):
        self.code_postal = code_postal


    def getVille(self):
        return self.ville

    def setVille(self, ville):
        self.ville = ville
```

6) Voici la classe Entreprise :

La classe Entreprise hérite de la classe Client et représente une entreprise dans le système.

Elle possède les mêmes attributs que la classe Client, ainsi qu'un attribut supplémentaire :

- NumSiret : numéro SIRET de l'entreprise

Elle possède également des méthodes pour accéder et modifier cet attribut spécifique :

- getNumSiret() et setNumSiret(NumSiret): pour accéder et modifier le numéro SIRET de l'entreprise.

```python
from dbBuilder import Client

class Entreprise(Client):
    def __init__(self, ID, Nom, Prenom, Adresse_Postale, Code_Postal, Ville, NumSiret):
        super().__init__(ID, Nom, Prenom, Adresse_Postale, Code_Postal, Ville, NumSiret)
        self.NumSiret =  NumSiret

    def getNumSiret(self):
        return self.NumSiret

    def setNumSiret(self, NumSiret):
        self.NumSiret = NumSiret
```