

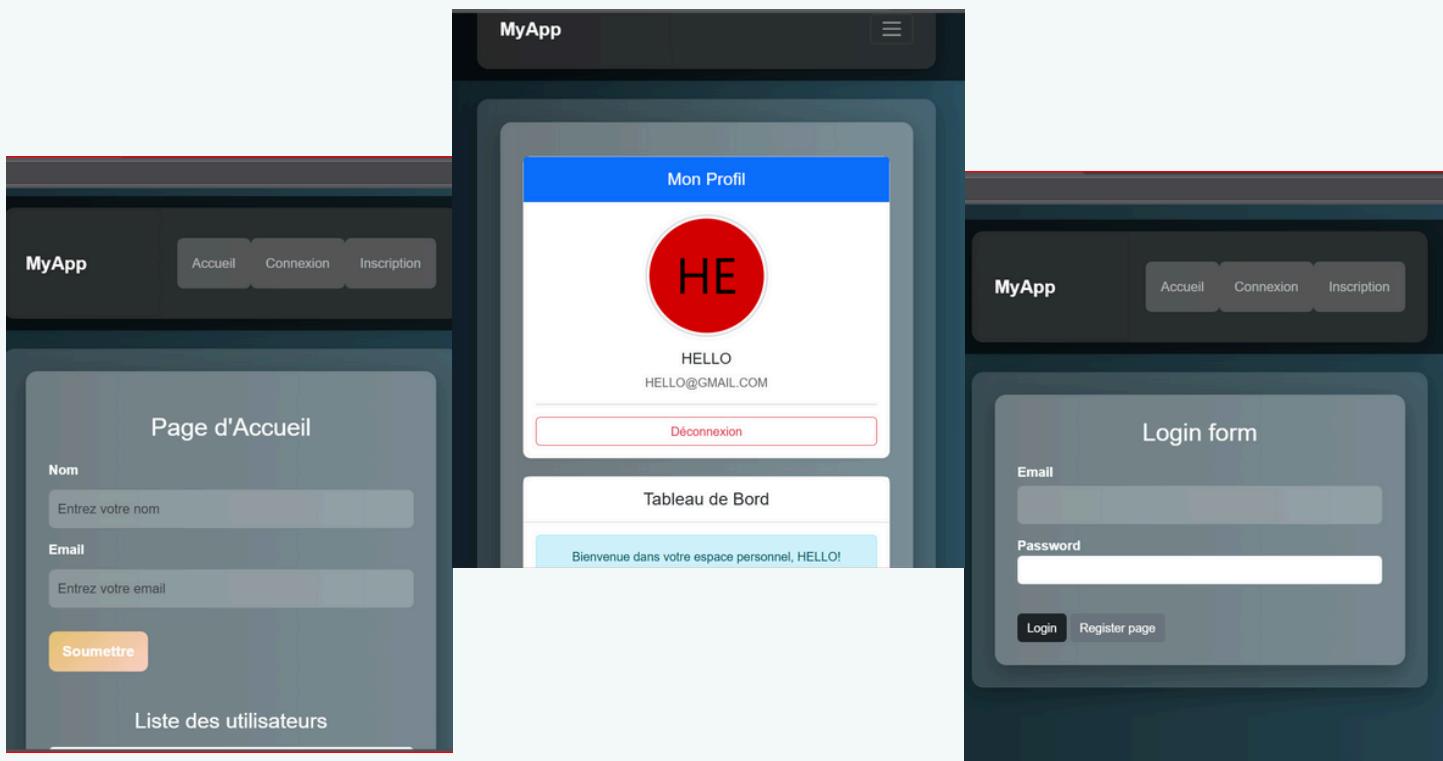
DÉPT. G. INFORMATIQUE / IDAI
S5_ 2024/2025

RAPPORT DE MINI_PROJECT: PYTHON FLASK (ARCHITECTURE MVC)

RÉALISÉE PAR:
GHIZLANE AFAILAL TREBAK



ENCADRE PAR:
Pr. Othman Bakkali



The image displays three screenshots of a mobile application named "MyApp".

- Home Screen:** Shows a form for entering a name and email, and a button to submit. Below the form is a link to "Liste des utilisateurs".
- User Profile Screen:** Displays a profile picture placeholder with "HE", the name "HELLO", and the email "HELLO@GMAIL.COM". It includes a "Déconnexion" button and a "Tableau de Bord" section with a welcome message: "Bienvenue dans votre espace personnel, HELLO!".
- Login Screen:** Features fields for "Email" and "Password", and buttons for "Login" and "Register page".

Introduction

Ce rapport détaille la structure d'un projet Flask implémenté selon l'architecture MVC (Modèle-Vue-Contrôleur). Il décrit les fichiers principaux, leur rôle, et les fonctionnalités associées, en mettant l'accent sur l'utilisation des sessions et des caches pour optimiser l'application.

Architecture MVC

1. Modèles (Models):

- user.py et info.py
- Gèrent les interactions avec la base de données
- Contiennent la logique métier

2. Vues (Templates):

- Fichiers HTML dans templates/
- Affichent les données aux utilisateurs
- Utilisent l'héritage de templates

3. Contrôleurs:

- auth_controller.py
- Gèrent les routes et la logique d'application
- Fait le lien entre modèles et vues

Fonctionnalités clés

1. Authentification:

- Inscription avec validation d'email unique
- Connexion avec session
- Protection des routes (dashboard)
- Déconnexion

2. Gestion des données:

- CRUD simple pour les informations utilisateur
- Jointures entre tables

3. Sécurité:

- Hash des mots de passe avec bcrypt
- Gestion des sessions sécurisées
- Validation des formulaires côté serveur

4. Interface utilisateur:

- Design responsive avec Bootstrap
- Feedback visuel (messages flash)
- Expérience utilisateur cohérente

1. Structure des Dossiers et Fichiers

Le projet est organisé de la manière suivante :

```
myapp/
|   — app/
|   |   — __pycache__/
|   |   — controllers/
|   |   |   — auth_controller.py
|   |   — models/
|   |   |   — info.py
|   |   |   — user.py
|   |   — static/
|   |   |   — css/
|   |   — templates/
|   |   |   — app.html
|   |   |   — dashboard.html
|   |   |   — index.html
|   |   |   — login.html
|   |   |   — register.html
|   |   — __init__.py
|   |   — config.py
|   |   — forms.py
|   — flask_session/
|       — app.py
```

2. Analyse détaillée des fichiers

1. Fichiers de configuration: (app/config.py)

```
app > 🐍 config.py
1   class Config:
2       SECRET_KEY = 'your_secret_key_here'
3       MYSQL_HOST = 'localhost'
4       MYSQL_PORT = 3307
5       MYSQL_USER = 'root'
6       MYSQL_PASSWORD = ''
7       MYSQL_DB = 'mydatabase'
8       SESSION_TYPE = 'filesystem'
9       CACHE_TYPE = 'simple'
10      |
```

Ce fichier définit une classe Config qui regroupe plusieurs paramètres utilisés dans l'application Flask.

🔑 1. SECRET_KEY:

- Utilisé par Flask pour signer les cookies et sécuriser les sessions.
- Empêche la falsification de sessions et protège les données sensibles.
- Doit être unique et secret en production (peut être stocké dans une variable d'environnement).

2. Paramètres MySQL

Ces paramètres définissent la connexion à la base de données MySQL.

- `MYSQL_HOST = 'localhost'` → L'application se connecte à MySQL en local.
- `MYSQL_PORT = 3307` → Port utilisé pour MySQL (peut être 3306 ou un autre selon la config).
- `MYSQL_USER = 'root'` → Nom d'utilisateur de la base de données.
- `MYSQL_PASSWORD = ""` → Mot de passe de l'utilisateur (à ne jamais laisser vide en production !).
- `MYSQL_DB = 'mydatabase'` → Nom de la base de données utilisée par l'application.

3. Gestion des Sessions

- Définit le stockage des sessions dans le système de fichiers local.
- Flask utilise ces sessions pour suivre les utilisateurs connectés.
- D'autres options existent : redis, memcached, sqlalchemy, etc.

4. Gestion du Cache

- Active un cache de type "simple", utile pour stocker des données temporairement.
- D'autres types de caches disponibles : redis, memcached, filesystem, null, etc.
- Améliore la performance en réduisant les accès répétitifs à la base de données.

Conclusion

Le fichier `config.py` centralise les paramètres essentiels de l'application. Il permet :

- ✓ Une gestion flexible des configurations
- ✓ Une meilleure sécurité avec `SECRET_KEY`
- ✓ Une connexion simplifiée à MySQL
- ✓ Une gestion efficace des sessions et du cache

, 1.2/ fichier app/__init__.py

Le fichier `__init__.py` dans un projet Flask est utilisé pour initialiser l'application et charger les extensions nécessaires. Il permet également d'organiser le projet en suivant l'architecture MVC.

1. Analyse du Code

```
python __init__.py
from flask import Flask
from flask_mysqldb import MySQL
from flask_session import Session
from flask_caching import Cache
from app.config import Config
```

📌 Importation des modules Flask et des extensions

- Flask → Initialise l'application.
- MySQL (flask_mysqldb) → Permet d'interagir avec une base de données MySQL.
- Session (flask_session) → Gère les sessions utilisateurs.
- Cache (flask_caching) → Permet la mise en cache pour améliorer les performances.
- Config → Charge la configuration définie dans config.py.

2. Initialisation de l'application Flask

```
# Initialisation de l'application
app = Flask(__name__)
app.config.from_object(Config)
```

- Crée une instance Flask (app).
- Charge la configuration définie dans config.py (connexion MySQL, gestion des sessions, cache, etc.).

3. Initialisation des Extensions

```
# Initialisation des extensions
mysql = MySQL(app)
session = Session(app)
cache = Cache(app)
```

- mysql = MySQL(app) → Initialise l'extension MySQL pour se connecter à la base de données.
- session = Session(app) → Active le stockage des sessions (défini dans config.py).
- cache = Cache(app) → Configure la mise en cache pour optimiser les performances.

4. Importation des routes

```
# Importation des routes
from app.controllers import auth_controller
```

- Pour enregistrer les routes définies dans auth_controller.py et les rendre accessibles dans l'application.
- Évite les imports circulaires en plaçant cette ligne après la création de app.

Conclusion

- Le fichier `__init__.py` permet de configurer et initialiser l'application Flask en suivant une approche modulaire.
- ✓ Séparation des responsabilités (configuration, extensions, routes)
- ✓ Organisation MVC
- ✓ Préparation à la scalabilité et à l'optimisation (sessions, cache, MySQL)

2. Contrôleurs (routes)

(app/controllers/auth_controller.py)

1. Importation des modules

```
# /app/controllers/auth_controller.py
from flask import render_template, redirect, url_for, session, flash
from app import app, mysql
from app.forms import RegisterForm, LoginForm, InfoForm, UserInfoForm
from app.models.user import User
from app.models.info import Info
import bcrypt
```

Flask modules :

- `render_template` : Génère des pages HTML à partir des templates.
- `redirect, url_for` : Redirigent l'utilisateur vers d'autres routes.
- `session` : Gère l'authentification des utilisateurs.
- `flash` : Affiche des messages temporaires (ex : erreurs, confirmations).

Extensions & Modules :

- `app` : Instance Flask définie dans `__init__.py`.
- `mysql` : Connexion à la base de données MySQL.
- `RegisterForm, LoginForm, InfoForm, UserInfoForm` : Formulaires gérés avec Flask-WTF.
- `User, Info` : Modèles représentant les tables `user` et `info` dans la base de données.
- `bcrypt` : Bibliothèque utilisée pour le hachage des mots de passe.

2. Définition des routes

Route / (Page d'accueil)

```
@app.route('/', methods=['GET', 'POST'])
def index():
    form = UserInfoForm()

    if form.validate_on_submit():
        name = form.name.data
        email = form.email.data

        # Stocker les données dans la base de données
        cursor = mysql.connection.cursor()
        cursor.execute("INSERT INTO user_data (name, email) VALUES (%s, %s)", (name, email))
        mysql.connection.commit()
        cursor.close()

        flash("Information ajoutée avec succès !", "success")
        return redirect(url_for('index'))

    # Récupérer les informations stockées
    cursor = mysql.connection.cursor()
    cursor.execute("SELECT name, email FROM user_data ORDER BY id DESC")
    infos = cursor.fetchall()
    cursor.close()

    return render_template('index.html', form=form, infos=infos)
```

- Affiche la page d'accueil avec des informations générales.
- Peut inclure un formulaire pour rechercher des données.
- Méthodes GET et POST : Permet de récupérer et d'envoyer des données.

1. Définition de la Route

```
@app.route('/', methods=['GET', 'POST'])
```

- @app.route('/') : Définit la route pour la page d'accueil.
- methods=['GET', 'POST'] :
 - GET : Affiche la page avec le formulaire et les informations stockées.
 - POST : Traite le formulaire lorsqu'un utilisateur soumet ses informations.

2. Création et Validation du Formulaire

form = UserInfoForm()

- **UserInfoForm()** : Crée une instance du formulaire défini dans app/forms.py.
- Ce formulaire contient probablement des champs name et email.

if form.validate_on_submit():

Vérifie si le formulaire a été soumis et est valide (ex : pas de champs vides, format email correct).

3. Stockage des Données dans la Base MySQL

name = form.name.data

email = form.email.data

Récupère les données saisies par l'utilisateur dans le formulaire.

cursor = mysql.connection.cursor()

**cursor.execute("INSERT INTO user_data (name, email) VALUES (%s, %s)",
(name, email))**

mysql.connection.commit()

cursor.close()

- **cursor = mysql.connection.cursor()** : Crée un curseur pour exécuter des requêtes SQL.
- **INSERT INTO user_data (name, email) VALUES (%s, %s)** : Insère les données dans la table user_data.
- **mysql.connection.commit()** : Sauvegarde la transaction en base de données.
- **cursor.close()** : Ferme la connexion.

4. Message de Confirmation et Redirection

```
flash("Information ajoutée avec succès !", "success")
return redirect(url_for('index'))
```

- `flash()` : Affiche un message temporaire à l'utilisateur.
- `redirect(url_for('index'))` : Redirige vers la page d'accueil pour afficher la mise à jour.

5. Récupération des Données Stockées

```
cursor = mysql.connection.cursor()
cursor.execute("SELECT name, email FROM user_data ORDER BY id DESC")
infos = cursor.fetchall()
cursor.close()
```

- Exécute une requête SQL pour récupérer toutes les entrées de la table `user_data`.
- Trie les résultats par `id DESC` (du plus récent au plus ancien).
- `fetchall()` : Récupère toutes les lignes sous forme de liste de tuples

6. Affichage des Données dans le Template

```
return render_template('index.html', form=form, infos=infos)
```

`render_template('index.html', form=form, infos=infos)` :

- Passe le formulaire (`form`) au template pour être affiché.
- Passe la liste des informations (`infos`) pour affichage dans `index.html`.

🔗 Résumé du Fonctionnement

1. Affiche un formulaire (GET).
2. Lorsqu'un utilisateur remplit le formulaire (POST) :
 - Vérifie les données.
 - Insère dans la base MySQL.
 - Affiche un message de confirmation.
 - Redirige vers /.
3. Récupère et affiche les données stockées dans la page index.html.

Route /register

```
@app.route('/register', methods=['GET', 'POST'])
def register():
    form = RegisterForm()
    if form.validate_on_submit():
        name = form.name.data
        email = form.email.data
        password = form.password.data
        hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())
        User.create_user(name, email, hashed_password)
        return redirect(url_for('login'))
    return render_template('register.html', form=form)
```

Ce code définit la route /register, permettant aux utilisateurs de s'inscrire en remplissant un formulaire. Une fois soumis, les données sont enregistrées dans la base de données après hachage du mot de passe.

1. Définition de la Route

```
@app.route('/register', methods=['GET', 'POST'])
```

- @app.route('/register') : Définit la route /register pour la page d'inscription.
- methods=['GET', 'POST'] :
 1. GET : Affiche le formulaire d'inscription.
 2. POST : Traite le formulaire lorsqu'un utilisateur soumet ses informations.

2. Création et Validation du Formulaire

form = RegisterForm()

- RegisterForm() : Crée une instance du formulaire d'inscription défini dans app/forms.py.
- Ce formulaire contient probablement des champs name, email et password.

if form.validate_on_submit():

- Vérifie si le formulaire a été soumis et si les données sont valides (ex : format email correct, mot de passe non vide).

3. Récupération et Hachage du Mot de Passe

name = form.name.data

email = form.email.data

password = form.password.data

- Récupère les données saisies par l'utilisateur.

hashed_password = bcrypt.hashpw(password.encode('utf-8'),

bcrypt.gensalt())

- Hachage du mot de passe avec la bibliothèque bcrypt :
 - bcrypt.gensalt() génère un sel unique pour renforcer le hachage.
 - bcrypt.hashpw() hache le mot de passe saisi avec le sel.

Pourquoi hacher le mot de passe ?

- Protéger les mots de passe en cas de fuite de données.
- Éviter de stocker des mots de passe en texte clair.

4. Stockage des Données

User.create_user(name, email, hashed_password)

- User.create_user() est une méthode du modèle User (dans app/models/user.py).
- Elle enregistre les informations de l'utilisateur dans la base de données.
- Le mot de passe enregistré est haché, donc impossible à récupérer en texte clair.

5. Redirection après Inscription

return redirect(url_for('login'))

- Après une inscription réussie, l'utilisateur est redirigé vers la page de connexion (/login).

6. Affichage du Formulaire

return render_template('register.html', form=form)

- Affiche la page register.html avec le formulaire.

🔗 Résumé du Fonctionnement

1. Affiche un formulaire d'inscription (GET).
2. Lorsqu'un utilisateur remplit le formulaire (POST) :
 - Vérifie les données.
 - Hache le mot de passe.
 - Enregistre l'utilisateur en base de données.
 - Redirige vers /login.
3. Si GET, affiche simplement le formulaire pour l'inscription.

Route /login:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        email = form.email.data
        password = form.password.data
        user = User.get_user_by_email(email)
        if user and bcrypt.checkpw(password.encode('utf-8'), user[3].encode('utf-8')):
            session['user_id'] = user[0]
            return redirect(url_for('dashboard'))
        else:
            flash("Login failed. Please check your email and password")
            return redirect(url_for('login'))
    return render_template('login.html', form=form)
```

Cette route permet aux utilisateurs de se connecter en vérifiant leurs identifiants (email et mot de passe). Si les informations sont correctes, l'utilisateur est redirigé vers le tableau de bord.

1. Définition de la Route

`@app.route('/login', methods=['GET', 'POST'])`

- `@app.route('/login')` : Définit la route /login pour la connexion.
- `methods=['GET', 'POST']` :
- GET : Affiche le formulaire de connexion.
- POST : Traite les données saisies par l'utilisateur.

2. Crédit et Validation du Formulaire

`form = LoginForm()`

- `LoginForm()` : Crée une instance du formulaire de connexion défini dans `app/forms.py`.
- Ce formulaire contient probablement des champs `email` et `password`.

`if form.validate_on_submit():`

Vérifie si le formulaire a été soumis et si les données sont valides.

3. Vérification des Identifiants

email = form.email.data

password = form.password.data

- Récupère l'email et le mot de passe saisis par l'utilisateur.

user = User.get_user_by_email(email)

- User.get_user_by_email(email) cherche un utilisateur dans la base de données.
- Cette fonction est probablement définie dans app/models/user.py et retourne les informations de l'utilisateur si trouvé.

Si l'utilisateur existe : **if user and**

bcrypt.checkpw(password.encode('utf-8'), user[3].encode('utf-8')):

- user : Contient les informations de l'utilisateur (ex: id, name, email, hashed_password).
- user[3] : Correspond au mot de passe haché stocké en base de données.
- bcrypt.checkpw() : Compare le mot de passe saisi (haché) avec celui stocké.
- Si les mots de passe correspondent, l'utilisateur est authentifié.

4. Crédit d'une Session

session['user_id'] = user[0]

- Stocke l'ID de l'utilisateur dans la session pour maintenir sa connexion.
- user[0] : Correspond probablement à l'ID de l'utilisateur.

5. Redirection vers le Tableau de Bord

```
return redirect(url_for('dashboard'))
```

- Si l'authentification est réussie, l'utilisateur est redirigé vers la page /dashboard.

6. Gestion des Erreurs

```
session['user_id'] = user[0]flash("Login failed. Please check your email  
and password")  
return redirect(url_for('login'))
```

Si l'utilisateur n'existe pas ou que le mot de passe est incorrect :

- Un message d'erreur est affiché avec flash().
- L'utilisateur est redirigé vers /login.

7. Affichage du Formulaire

```
return render_template('login.html', form=form)
```

- Si la méthode est GET ou si la connexion échoue, affiche la page login.html avec le formulaire.

Résumé du Fonctionnement

1. Affiche un formulaire de connexion (GET).
2. Lorsqu'un utilisateur remplit le formulaire (POST) :
 - Vérifie si l'email existe.
 - Compare le mot de passe saisi avec celui stocké en base de données.
 - Si les informations sont correctes :
 - Crée une session.
 - Redirige vers /dashboard.
 - Sinon, affiche un message d'erreur et recharge la page /login.

Route /dashboard

```
@app.route('/dashboard')
def dashboard():
    if 'user_id' in session:
        user = User.get_user_by_id(session['user_id']) # Utilisez la nouvelle méthode
        if user:
            return render_template('dashboard.html', user=user)
    return redirect(url_for('login'))
```

Cette route gère l'affichage du tableau de bord de l'utilisateur après connexion. Elle vérifie si un utilisateur est bien authentifié avant d'afficher la page.

1. Définition de la Route

```
@app.route('/dashboard')
```

- Associe l'URL /dashboard à la fonction dashboard().
- Seule la méthode GET est utilisée (par défaut).

2. Vérification de la Connexion

```
if 'user_id' in session:
```

- Vérifie si la clé 'user_id' est stockée dans session.
- Cela signifie que l'utilisateur est connecté..

3. Récupération des Informations Utilisateur

```
user = User.get_user_by_id(session['user_id'])
```

- session['user_id'] : Contient l'ID de l'utilisateur connecté.
- User.get_user_by_id() : Fonction définie dans app/models/user.py qui récupère les informations de l'utilisateur en base de données à partir de son ID.

4. Redirection vers /login si l'utilisateur n'est pas connecté

`return redirect(url_for('login'))`

- Si l'utilisateur n'est pas connecté ('user_id' absent de session) ou si l'ID est invalide :
- Redirection vers la page de connexion /login.

Résumé du Fonctionnement

1. Vérifie si l'utilisateur est connecté (session['user_id'] existe).
2. Récupère les informations de l'utilisateur en base de données.
3. Si l'utilisateur est trouvé, affiche la page /dashboard avec ses données.
4. Sinon, redirige vers /login.

Route /logout

```
@app.route('/logout')
def logout():
    session.pop('user_id', None)
    flash("You have been logged out successfully.")
    return redirect(url_for('login'))
```

Cette route gère la déconnexion d'un utilisateur en supprimant sa session active et en le redirigeant vers la page de connexion.

1. Définition de la Route

`@app.route('/logout')`

- Associe l'URL /logout à la fonction logout().
- Seule la méthode GET est utilisée (par défaut).

2. Suppression de la Session

```
session.pop('user_id', None)
```

- `session.pop('user_id', None)` : Supprime la clé 'user_id' de la session si elle existe.
- Pourquoi `None` ?
- Si 'user_id' n'existe pas, `pop()` ne génère pas d'erreur et renvoie `None`.

Cela déconnecte l'utilisateur en supprimant son identifiant de session.

3. Message de Confirmation

```
flash("You have been logged out successfully.")
```

- `flash()` : Ajoute un message temporaire stocké en session.
- Ce message sera affiché sur la page de connexion après la redirection.

4. Redirection vers la Page de Connexion

```
return redirect(url_for('login'))
```

- Redirige l'utilisateur vers `/login` après la déconnexion.

🔗 Résumé du Fonctionnement

1. Supprime l'ID utilisateur de la session (`session.pop('user_id')`).
2. Affiche un message indiquant que l'utilisateur a été déconnecté.
3. Redirige l'utilisateur vers la page de connexion (`/login`).

3. Modèles

Classe Info:

```

1 from app import mysql
2
3 class Info:
4     @staticmethod
5     def add_info(user_id, data):
6         cursor = mysql.connection.cursor()
7         cursor.execute("INSERT INTO user_info (user_id, data) VALUES (%s, %s)", (user_id, data))
8         mysql.connection.commit()
9         cursor.close()
10
11    @staticmethod
12    def get_all_info():
13        cursor = mysql.connection.cursor()
14        cursor.execute("SELECT users.name, user_info.data FROM user_info JOIN users ON user_info.user_id = users.id")
15        infos = cursor.fetchall()
16        cursor.close()
17        return infos
18
19

```

Cette classe représente un modèle pour gérer les informations des utilisateurs dans une base de données MySQL. Elle fournit deux méthodes statiques :

1. `add_info(user_id, data)` → Ajoute une nouvelle information liée à un utilisateur.

2. `get_all_info()` → Récupère toutes les informations des utilisateurs.

1. Importation de MySQL

`from app import mysql`

- `mysql` est une instance de MySQL initialisée dans `app/__init__.py`.
- Permet d'interagir avec la base de données MySQL.

2. Méthode `add_info(user_id, data)`

`@staticmethod`

`def add_info(user_id, data):`

`cursor = mysql.connection.cursor()`

`cursor.execute("INSERT INTO user_info (user_id, data) VALUES (%s, %s)", (user_id, data))`

`mysql.connection.commit()`

`cursor.close()`

◆ Explication :

1. Création d'un curseur : mysql.connection.cursor()
 - Permet d'exécuter des requêtes SQL.
2. Exécution de la requête SQL : **INSERT INTO user_info (user_id, data)**

VALUES (%s, %s)

- %s est un espace réservé pour éviter les injections SQL.
 - Les valeurs (user_id, data) sont passées en paramètres.
3. Validation des modifications : mysql.connection.commit()
 - Sauvegarde les modifications dans la base de données.
 4. Fermeture du curseur : cursor.close()
 - Libère les ressources.

Permet d'ajouter une nouvelle entrée dans la table user_info.

3. Méthode get_all_info()

`@staticmethod`

```
def get_all_info():
    cursor = mysql.connection.cursor()
    cursor.execute("SELECT users.name, user_info.data FROM user_info
JOIN users ON user_info.user_id = users.id")
    infos = cursor.fetchall()
    cursor.close()
    return infos
```

◆ Explication :

- Création d'un curseur pour exécuter des requêtes SQL.
- Exécution de la requête SQL :

```
SELECT users.name, user_info.data
FROM user_info
JOIN users ON user_info.user_id = users.id
```

- jointure (JOIN) entre user_info et users sur user_id pour récupérer les noms des utilisateurs avec leurs informations.

1. Récupération des résultats : cursor.fetchall()

- Retourne une liste de tuples (nom, data).

2. Fermeture du curseur : cursor.close()

- Libère les ressources.

3. Retour des résultats à l'appelant.

Permet d'afficher toutes les informations stockées avec les noms des utilisateurs.

Résumé du Fonctionnement

- add_info(user_id, data) ajoute une information utilisateur dans user_info.
- get_all_info() récupère toutes les informations stockées, en affichant le nom des utilisateurs.

Classe User:

```
from app import mysql
import bcrypt

class User:
    @classmethod
    def create_user(cls, name, email, password):
        cursor = mysql.connection.cursor()
        cursor.execute("INSERT INTO users (name, email, password) VALUES (%s, %s, %s)",
                      (name, email, password))
        mysql.connection.commit()
        cursor.close()

    @classmethod
    def get_user_by_email(cls, email):
        cursor = mysql.connection.cursor()
        cursor.execute("SELECT * FROM users WHERE email = %s", (email,))
        user = cursor.fetchone()
        cursor.close()
        return user

    @classmethod
    def get_user_by_id(cls, user_id):
        cursor = mysql.connection.cursor()
        cursor.execute("SELECT * FROM users WHERE id = %s", (user_id,))
        user = cursor.fetchone()
        cursor.close()
        return user
```

La classe User représente un modèle pour gérer les utilisateurs dans une base de données MySQL. Elle offre trois méthodes de gestion des utilisateurs :

1. `create_user(name, email, password)` → Crée un nouvel utilisateur dans la base de données.
2. `get_user_by_email(email)` → Récupère un utilisateur par son email.
3. `get_user_by_id(user_id)` → Récupère un utilisateur par son ID.

1. Importation des modules

```
from app import mysql
import bcrypt
```

- `mysql` : Instance de MySQL pour interagir avec la base de données.
- `bcrypt` : Module permettant de sécuriser les mots de passe par hachage.

2. Méthode `create_user(cls, name, email, password)`

```
@classmethod
def create_user(cls, name, email, password):
    cursor = mysql.connection.cursor()
    cursor.execute("INSERT INTO users (name, email, password)
VALUES (%s, %s, %s)",
               (name, email, password))
    mysql.connection.commit()
    cursor.close()
```

◆ Explication :

1. Création d'un curseur : `mysql.connection.cursor()`
 - Permet d'exécuter des requêtes SQL.
2. Exécution de la requête SQL : **INSERT INTO users (name, email, password) VALUES (%s, %s, %s)**

3. Validation des modifications : mysql.connection.commit()

- Sauvegarde les changements dans la base de données.

4. Fermeture du curseur : cursor.close()

- Libère les ressources.

Cette méthode crée un nouvel utilisateur dans la table users.

3. Méthode `get_user_by_email(cls, email)`

`@classmethod`

`def get_user_by_email(cls, email):`

`cursor = mysql.connection.cursor()`

`cursor.execute("SELECT * FROM users WHERE email = %s", (email,))`

`user = cursor.fetchone()`

`cursor.close()`

`return user`

◆ Explication :

1. Création du curseur pour exécuter des requêtes SQL.

2. Exécution de la requête SQL : **SELECT * FROM users WHERE email = %s**

- Recherche un utilisateur avec l'email donné.

3. Récupération du premier résultat : cursor.fetchone()

- Retourne un tuple (id, name, email, password).

4. Fermeture du curseur : cursor.close()

5. Retourne l'utilisateur (ou None si non trouvé).

Permet de récupérer un utilisateur par son email, utile pour la connexion.

La classe User représente un modèle pour gérer les utilisateurs dans une base de données MySQL. Elle offre trois méthodes de gestion des utilisateurs :

- 1.create_user(name, email, password) → Crée un nouvel utilisateur dans la base de données.
- 2.get_user_by_email(email) → Récupère un utilisateur par son email.
- 3.get_user_by_id(user_id) → Récupère un utilisateur par son ID.

4. Méthode `get_user_by_id(cls, user_id)`

`@classmethod`

`def get_user_by_id(cls, user_id):`

`cursor = mysql.connection.cursor()`

`cursor.execute("SELECT * FROM users WHERE id = %s", (user_id,))`

`user = cursor.fetchone()`

`cursor.close()`

`return user`

◆ Explication

1. Création d'un curseur pour exécuter des requêtes SQL.

2. Exécution de la requête SQL :

SELECT * FROM users WHERE id = %s

◦ Recherche un utilisateur par son ID.

3. Récupération du premier résultat : `cursor.fetchone()`

◦ Retourne un tuple (id, name, email, password).

4. Fermeture du curseur : `cursor.close()`

5. Retourne l'utilisateur (ou None si non trouvé).

Permet d'identifier un utilisateur par son ID, utile après connexion.

4. Formulaires

```
# /app/forms.py
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, SubmitField
from wtforms.validators import DataRequired, Email, ValidationError
from app.models.user import User
import bcrypt
```

- FlaskForm : Classe de base pour les formulaires Flask-WTF.
- StringField : Champ de texte pour les entrées utilisateur.
- PasswordField : Champ de mot de passe (masque les entrées).
- SubmitField : Bouton de soumission.
- DataRequired : Valide que le champ n'est pas vide.
- Email : Vérifie que l'entrée est une adresse email valide.
- ValidationError : Permet de lever des erreurs de validation personnalisées.
- User : Modèle des utilisateurs, utilisé pour vérifier si un email existe déjà.
- bcrypt : Module de hachage pour sécuriser les mots de passe.

1. Formulaire RegisterForm (Inscription)

```
class RegisterForm(FlaskForm):
    name = StringField("Name", validators=[DataRequired()])
    email = StringField("Email", validators=[DataRequired(), Email()])
    password = PasswordField("Password", validators=[DataRequired()])
    submit = SubmitField("Register")

    def validate_email(self, field):
        if User.get_user_by_email(field.data):
            raise ValidationError('Email Already Taken')
```

- Champs :
 - name : Nom de l'utilisateur, champ obligatoire.
 - email : Adresse email, champ obligatoire avec validation d'email.
 - password : Mot de passe, champ obligatoire.
 - submit : Bouton d'envoi.
- Validation personnalisée (validate_email) :
 - Vérifie si l'email est déjà enregistré dans la base de données.
 - Si oui, lève une ValidationError("Email Already Taken").

Permet d'inscrire un utilisateur tout en empêchant les doublons d'emails.

2. Formulaire LoginForm (Connexion)

```
class LoginForm(FlaskForm):    You, 19 hours ago • Initial commit
    email = StringField("Email", validators=[DataRequired(), Email()])
    password = PasswordField("Password", validators=[DataRequired()])
    submit = SubmitField("Login")
```

- Champs :
 - email : Email de l'utilisateur, requis et validé.
 - password : Mot de passe, requis.
 - submit : Bouton de connexion.

Utilisé pour authentifier un utilisateur en vérifiant ses identifiants.

3. Formulaire InfoForm (Informations utilisateur)

```
class InfoForm(FlaskForm):
    data = StringField("Enter your information", validators=[DataRequired()])
    submit = SubmitField("Submit")
```

- Champs :
 - data : Informations à stocker, requises.
 - submit : Bouton d'envoi.

Utilisé pour permettre aux utilisateurs d'ajouter des informations.

4. Formulaire UserInfoForm (Informations utilisateur - Version étendue)

```
class UserInfoForm(FlaskForm):
    name = StringField("Nom", validators=[DataRequired()])
    email = StringField("Email", validators=[DataRequired(), Email()])
    submit = SubmitField("Soumettre")
```

- Champs :
 - name : Nom de l'utilisateur, requis.
 - email : Adresse email, requis et validé.
 - submit : Bouton d'envoi.

Peut être utilisé pour modifier les informations d'un utilisateur existant.

5. Templates (vues)

- templates/app.html

```
<!DOCTYPE html>
<html lang="fr">
<!-- Template de base avec:
- Configuration responsive
- Intégration de Bootstrap
- Barre de navigation dynamique (change selon la session)
- Blocs pour le titre et le contenu
-->
```

Fonctionnalités:

- Template de base hérité par toutes les pages
- Navbar dynamique qui change selon l'état de connexion
- Intègre Bootstrap CSS/JS
- Définit la structure HTML de base

- templates/dashboard.html

```
{% extends 'app.html' %}

<!-- Tableau de bord utilisateur avec:
- Profil utilisateur (avatar généré, nom, email)
- Statistiques fictives
- Liste d'activités récentes
- Style CSS personnalisé
-->
```

Fonctionnalités:

- Affiche les informations de l'utilisateur connecté
- Section de profil avec avatar généré
- Statistiques et activités (données factices pour l'exemple)
- Design responsive avec Bootstrap

- **templates/index.html**

```
{% extends 'app.html' %}  
<!-- Page d'accueil avec:  
- Formulaire pour ajouter des données  
- Liste des utilisateurs enregistrés  
- Affichage des messages flash  
-->
```

Fonctionnalités:

- Formulaire principal pour ajouter des données
- Liste des utilisateurs existants
- Gestion des messages flash (succès/erreurs)

- **templates/login.html**

```
{% extends 'app.html' %}  
<!-- Page de connexion avec:  
- Formulaire de login  
- Gestion des erreurs  
- Lien vers l'inscription  
-->
```

Fonctionnalités:

- Formulaire de connexion avec validation
- Affichage des erreurs de validation
- Lien vers la page d'inscription

- **templates/register.html**

Fonctionnalités:

- Formulaire d'inscription avec tous les champs requis
- Validation côté client et serveur
- Messages d'erreur détaillés
- Lien vers la page de connexion

6. Fichiers d'exécution (app.py)

Ce fichier est le point d'entrée de votre application Flask. Il sert à démarrer le serveur Flask et exécuter l'application.

```
from app import app

if __name__ == '__main__':
    app.run(debug=True)
```

1. Importation de l'application (app) : **from app import app**

- Cela importe l'objet app défini dans app/__init__.py.
- app est l'instance principale de l'application Flask.

2. Exécution du serveur Flask :

```
if __name__ == '__main__':
    app.run(debug=True)
```

- if __name__ == '__main__': Vérifie si le script est exécuté directement (et non importé dans un autre module).
- app.run(debug=True) : Démarré le serveur avec :
 - _ debug=True : Mode débogage activé 
 - _ Recharge automatiquement le serveur en cas de modification du code.
 - Affiche les erreurs détaillées en cas de problème.

Fonctionnalités:

- Point d'entrée pour exécuter l'application
- Active le mode debug pour le développement