Application de gestion de projets et de tâches

Développement Fullstack – Django & Flutter

Momar Talla Salla
Teste technique
25 Juin 2025

---

1. Introduction

Ce document a pour objectif de détailler la conception et l'implémentation de l'application **TaskManager**, une solution complète de gestion de projets et de tâches. Le projet met en œuvre une architecture fullstack, avec un backend robuste développé en Django REST Framework et une application mobile réactive construite avec Flutter.

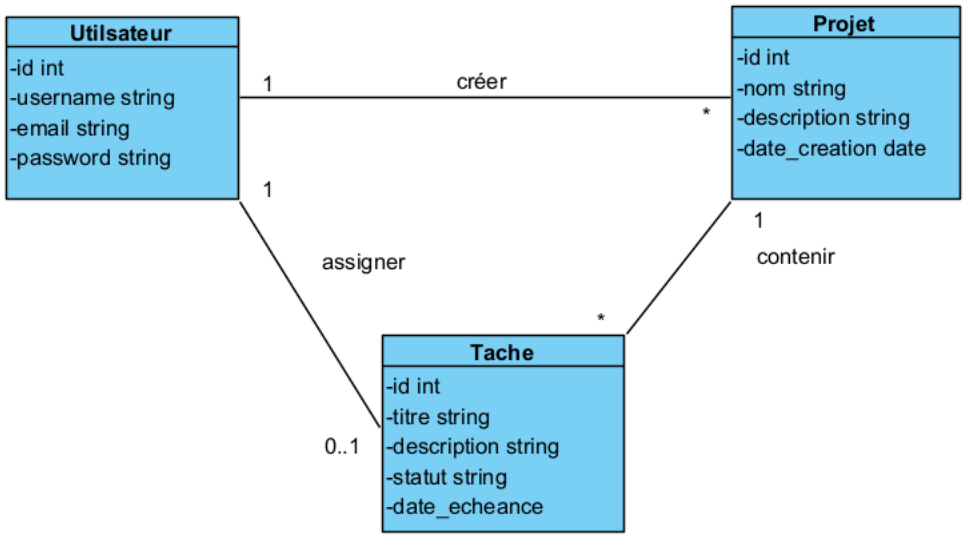
Objectif du projet
L'objectif principal est de fournir aux utilisateurs une plateforme intuitive pour organiser, suivre et gérer leurs projets et les tâches associées. L'application permet la création, la modification et la suppression de projets et de tâches, avec une gestion d'authentification sécurisée.
Utilisateurs ciblés
Cette application cible les professionnels, les équipes ou toute personne souhaitant organiser efficacement ses activités et projets personnels ou collaboratifs.
Technologies utilisées

- **Backend** : Django REST Framework (Python)
- **Base de données** : PostgreSQL (ou SQLite pour le développement local)
- **Authentification** : JSON Web Tokens (JWT) via djangorestframework-simplejwt
- **Frontend** : Flutter (Dart)
- **Stockage sécurisé** : flutter_secure_storage

2. Modélisation

2.1. Modèle Conceptuel de Données (MCD)

Le Modèle Conceptuel de Données (MCD) ci-dessous illustre l'organisation logique des informations de l'application et les relations entre les entités.



Entités :

- **User** : Représente un utilisateur de l'application, capable de s'authentifier et de créer des projets.
- **Projet** : Représente un conteneur pour des tâches. Chaque projet est associé à un utilisateur spécifique.
- **Tâche** : Représente une unité de travail au sein d'un projet. Chaque tâche est associée à un projet et potentiellement à un statut.

Relations et cardinalités :

- **User --- 1 --- crée --- 1, N --- Projet** : Un utilisateur peut avoir plusieurs projets, et un projet appartient à un seul utilisateur.
- **Projet --- 1--- Contient --- 1, N --- Tâche** : Un projet peut contenir plusieurs tâches, et une tâche appartient à un seul projet.

2.2. Modèle UML (Classe)

Le diagramme de classes UML fournit une vue plus détaillée des attributs et méthodes de chaque entité, reflétant leur implémentation technique.

Description des attributs et méthodes par entité :

- **User**
  - Id (PK)
  - username
  - password (haché)
  - email

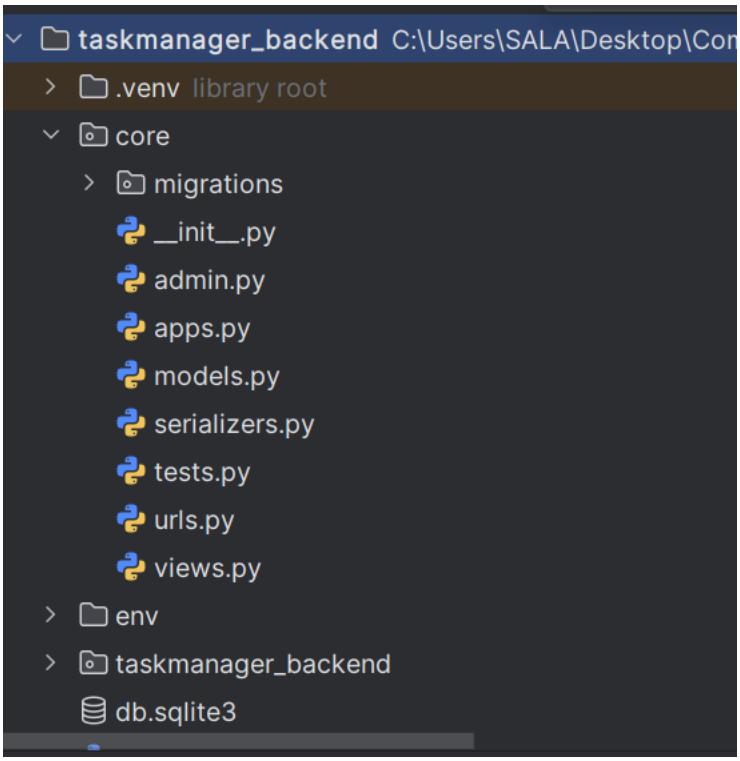
    - is_active
    - is_staff
    - ... (autres champs Django par défaut)
    - *Méthodes pertinentes pour l'authentification et la gestion des utilisateurs.*
- **Projet**
    - id (PK)
    - nom (Chaîne de caractères)
    - description (Texte long, optionnel)
    - date_creation (Date et heure, auto_now_add)
    - utilisateur (FK vers User)
    - *Méthodes : __str__ pour représentation textuelle.*
- **Tâche**
    - id (PK)
    - titre (Chaîne de caractères)
    - description (Texte long, optionnel)
    - statut (Choix : À faire, En cours, Terminé)
    - date_creation (Date et heure, auto_now_add)
    - date_echeance (Date, optionnel)
    - projet (FK vers Projet)
    - assignee_a (FK vers User, optionnel, si plusieurs utilisateurs sur un projet)
    - *Méthodes : __str__ pour représentation textuelle.*

3. Développement Backend – Django

3.1. Architecture générale

Le backend Django est organisé selon les conventions du Django REST Framework.

- models.py : Définit les structures de données (entités).
- serializers.py : Transforme les données des modèles en formats échangeables (JSON) et vice-versa.
- views.py : Gère la logique métier des requêtes HTTP (CRUD).
- urls.py : Définit les routes API pour diriger les requêtes vers les vues appropriées.



3.2. Modèles

Les modèles sont définis dans core/models.py.

User (modèle étendu ou par défaut de Django si suffisant)

Python

```
# Exemple si extension, sinon utiliser le modèle User par défaut
from django.contrib.auth.models import AbstractUser

class CustomUser(AbstractUser):
    # Ajoutez des champs personnalisés ici si nécessaire
    pass
```

Projet

Python

```python
from django.db import models
from django.conf import settings # Importer settings pour AUTH_USER_MODEL

class Projet(models.Model):
    nom = models.CharField(max_length=255)
    description = models.TextField(blank=True, null=True)
    date_creation = models.DateTimeField(auto_now_add=True)
    # Un projet appartient à un utilisateur
    utilisateur = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name='projets')

    def __str__(self):
        return self.nom
```

Tache

Python

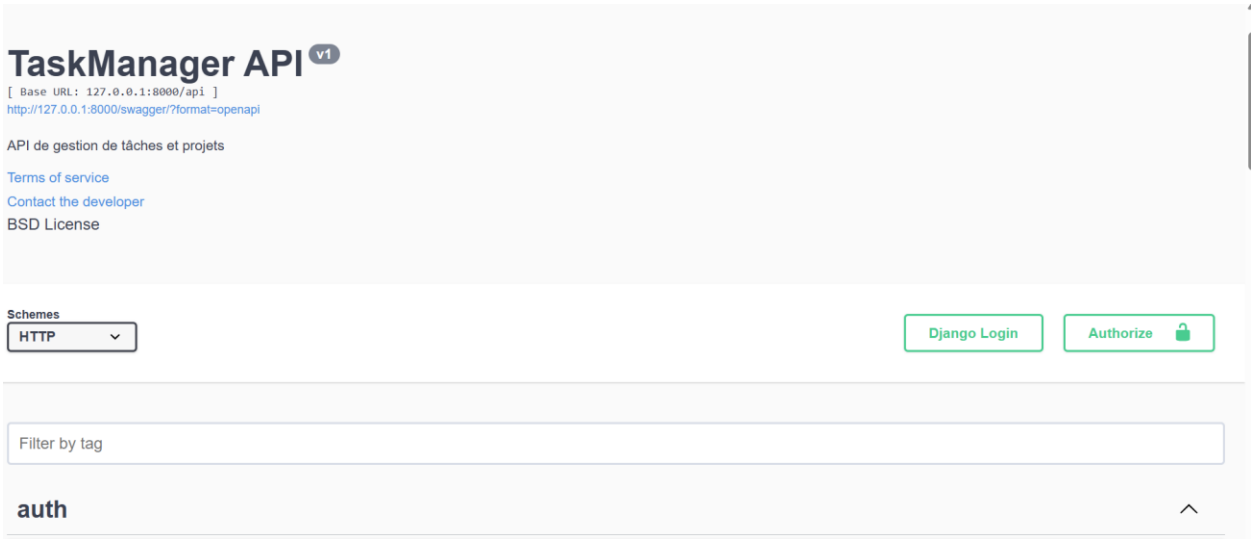
```python
from django.db import models
from django.conf import settings # Importer settings pour AUTH_USER_MODEL

class Tache(models.Model):
    STATUT_CHOICES = [
        ('A_FAIRE', 'À faire'),
        ('EN_COURS', 'En cours'),
        (('TERMINE', 'Terminé'),
    ]

    titre = models.CharField(max_length=255)
    description = models.TextField(blank=True, null=True)
    statut = models.CharField(max_length=10, choices=STATUT_CHOICES, default='A_FAIRE')
    date_creation = models.DateTimeField(auto_now_add=True)
    date_echeance = models.DateField(blank=True, null=True)
    # Une tâche appartient à un projet
    projet = models.ForeignKey('Projet', on_delete=models.CASCADE, related_name='taches')
    # Une tâche peut être assignée à un utilisateur (optionnel ici)
    assignee_a = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.SET_NULL, null=True, blank=True, related_name='taches_assignees')

    def __str__(self):
        return self.titre
```

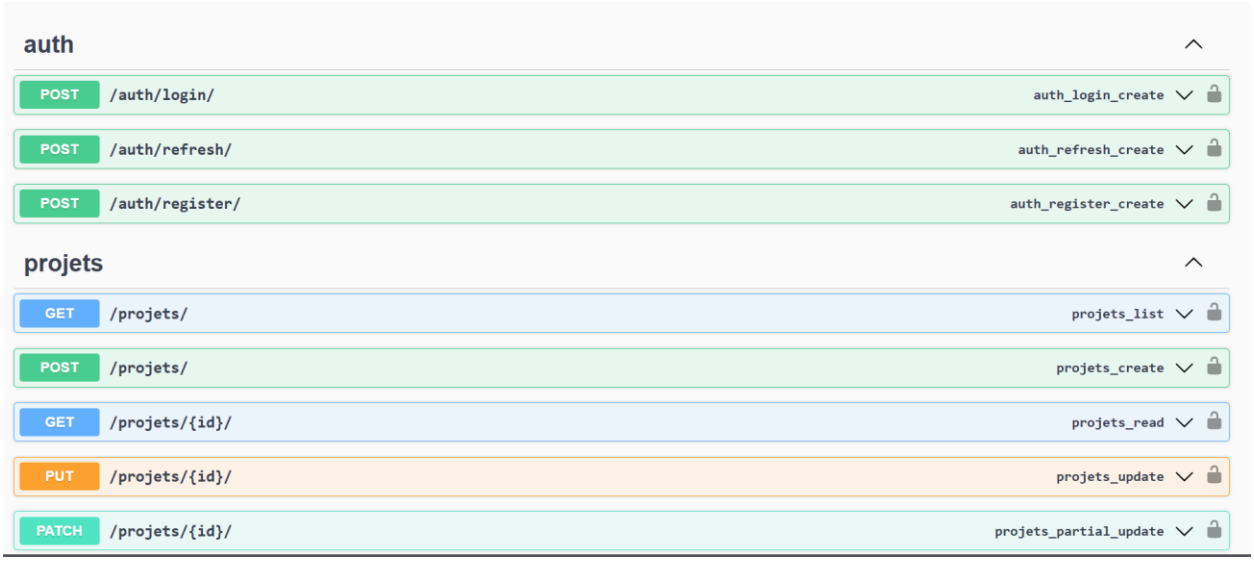
3.3. API REST

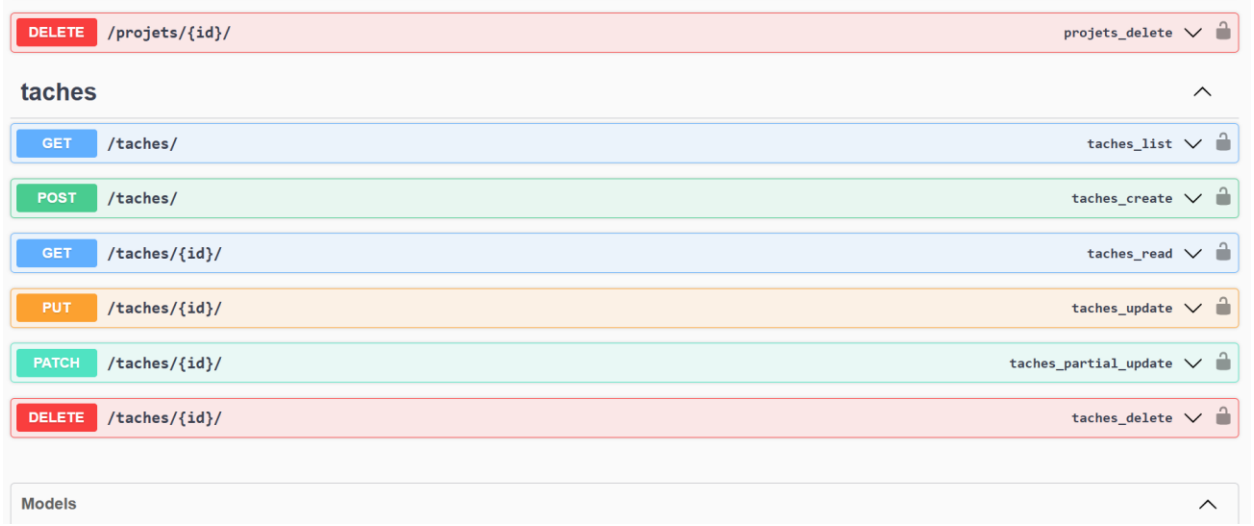
Les API sont gérées via Django REST Framework, fournissant des endpoints RESTful pour les opérations CRUD.

Exemple de route : GET /api/projets/
Cette route permet de récupérer la liste des projets de l'utilisateur authentifié.
Liste des endpoints (CRUD Projet/Tâche, Auth) :

- **Authentification**
  - POST /api/auth/register/ : Enregistrement d'un nouvel utilisateur.
  - POST /api/auth/login/ : Connexion et obtention des tokens JWT (access et refresh).
  - POST /api/auth/token/refresh/ : Rafraîchissement du token d'accès.
- **Projets**
  - GET /api/projets/ : Liste de tous les projets de l'utilisateur authentifié.
  - GET /api/projets/<int:pk>/ : Détails d'un projet spécifique.
  - POST /api/projets/ : Création d'un nouveau projet.
  - PUT /api/projets/<int:pk>/ : Mise à jour complète d'un projet.
  - PATCH /api/projets/<int:pk>/ : Mise à jour partielle d'un projet.
  - DELETE /api/projets/<int:pk>/ : Suppression d'un projet.
- **Tâches**
  - GET /api/projets/<int:projet_pk>/taches/ : Liste des tâches pour un projet donné.
  - GET /api/taches/<int:pk>/ : Détails d'une tâche spécifique.
  - POST /api/projets/<int:projet_pk>/taches/ : Création d'une nouvelle tâche pour un projet.
  - PUT /api/taches/<int:pk>/ : Mise à jour complète d'une tâche.
  - PATCH /api/taches/<int:pk>/ : Mise à jour partielle d'une tâche.
  - DELETE /api/taches/<int:pk>/ : Suppression d'une tâche.







3.4. Authentification JWT

L'authentification est gérée par JSON Web Tokens (JWT) via la librairie djangorestframework-simplejwt.

Fonctionnement :
Lors de la connexion, l'utilisateur reçoit un token d'accès (de courte durée) et un token de rafraîchissement (de longue durée). Le token d'accès est utilisé pour toutes les requêtes API subséquentes nécessitant une authentification. Lorsqu'il expire, le token de rafraîchissement peut être utilisé pour obtenir un nouveau

token d'accès sans que l'utilisateur n'ait à se reconnecter. 1
Sécurité, accès protégé :
Toutes les vues nécessitant une authentification utilisent des classes de permissions (IsAuthenticated) pour s'assurer que seules les requêtes avec un token JWT valide et actif sont autorisées. De plus, des permissions au niveau de l'objet sont implémentées pour garantir qu'un utilisateur ne peut accéder qu'à ses propres projets et tâches. 2
Extrait de code de LoginView, RegisterView

Python

```
# Dans your_app_name/views.py (Exemple simplifié)
from rest_framework import generics, permissions
from rest_framework_simplejwt.views import TokenObtainPairView
from .serializers import RegisterSerializer, MyTokenObtainPairSerializer

# Vue de connexion personnalisée pour JWT
class MyTokenObtainPairView(TokenObtainPairView):
    serializer_class = MyTokenObtainPairSerializer

# Vue d'enregistrement
class RegisterView(generics.CreateAPIView):
    queryset = User.objects.all()
    permission_classes = (permissions.AllowAny,)
    serializer_class = RegisterSerializer
```

3.5. Tests manuels

Des tests manuels des endpoints de l'API ont été réalisés à l'aide d'outils comme Postman ou curl pour vérifier leur bon fonctionnement, l'authentification et les permissions.

Scénarios de test API (Postman ou curl) :

- **Enregistrement d'un nouvel utilisateur** : POST /api/auth/register/ avec username, password. Attendre un succès 201.
- **Connexion d'un utilisateur** : POST /api/auth/login/ avec username, password. Attendre un succès 200 et la réception des tokens access et refresh.
- **Accès aux projets sans authentification** : GET /api/projets/. Attendre un code 401 (Unauthorized).
- **Création d'un projet authentifié** : POST /api/projets/ avec le token access dans l'en-tête Authorization. Attendre un succès 201.
- **Récupération des projets de l'utilisateur authentifié** : GET /api/projets/ avec le token access.

Vérifier que seuls les projets de cet utilisateur sont retournés.

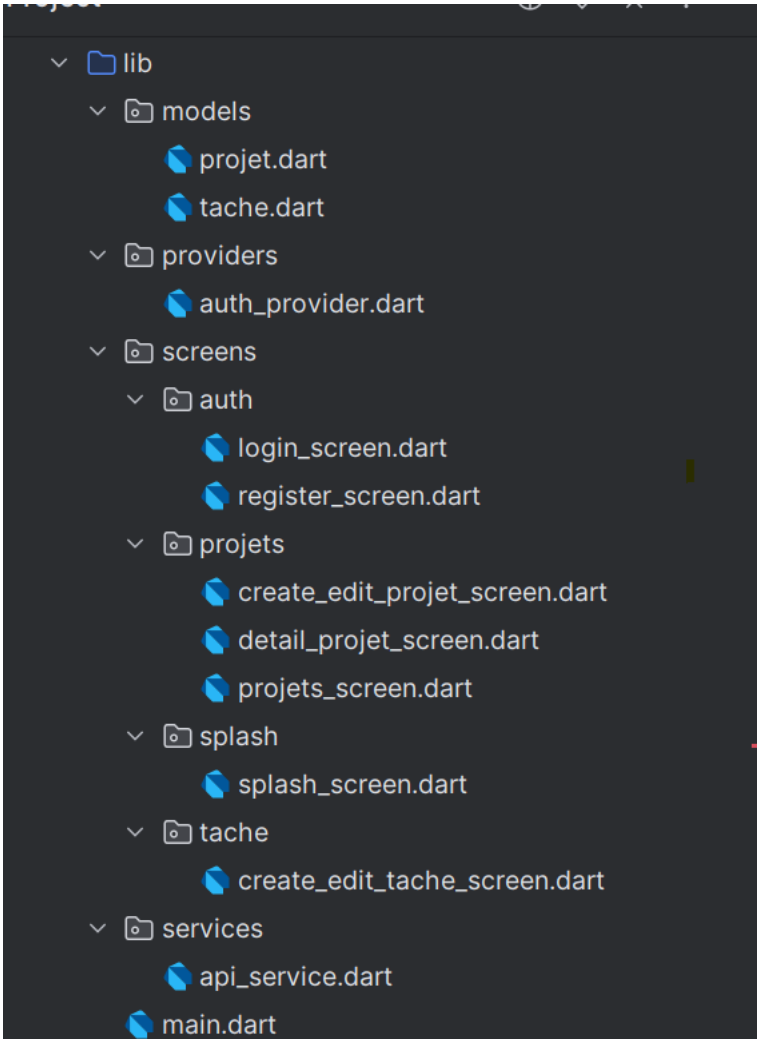
- **Tentative d'accès au projet d'un autre utilisateur** : GET /api/projets/<id_autre_projet>/ avec le token de l'utilisateur actuel. Attendre un code 403 (Forbidden) ou 404 (Not Found).
- **Création d'une tâche pour un projet** : POST /api/projets/<id_projet>/taches/.
- **Modification d'une tâche** : PUT /api/taches/<id_tache>/.
- **Suppression d'un projet/tâche** : DELETE /api/projets/<id_projet>/ ou DELETE /api/taches/<id_tache>/.

4. Développement Frontend – Flutter

4.1. Structure du projet

Le projet Flutter suit une structure modulaire pour une meilleure maintenabilité :

- lib/models/ : Définit les classes Dart pour les modèles de données (Projet, Tâche, User, etc.).
- lib/screens/ : Contient les widgets d'interface utilisateur pour chaque écran de l'application.
- lib/services/ : Encapsule la logique d'appel aux API (e.g., api_service.dart).
- lib/providers/ : Gère l'état global de l'application (e.g., état d'authentification, données partagées).
- lib/utils/ : Fichiers utilitaires (constantes, helpers, etc.).



4.2. Authentification

L'authentification JWT est gérée côté client pour offrir une expérience utilisateur fluide et sécurisée.

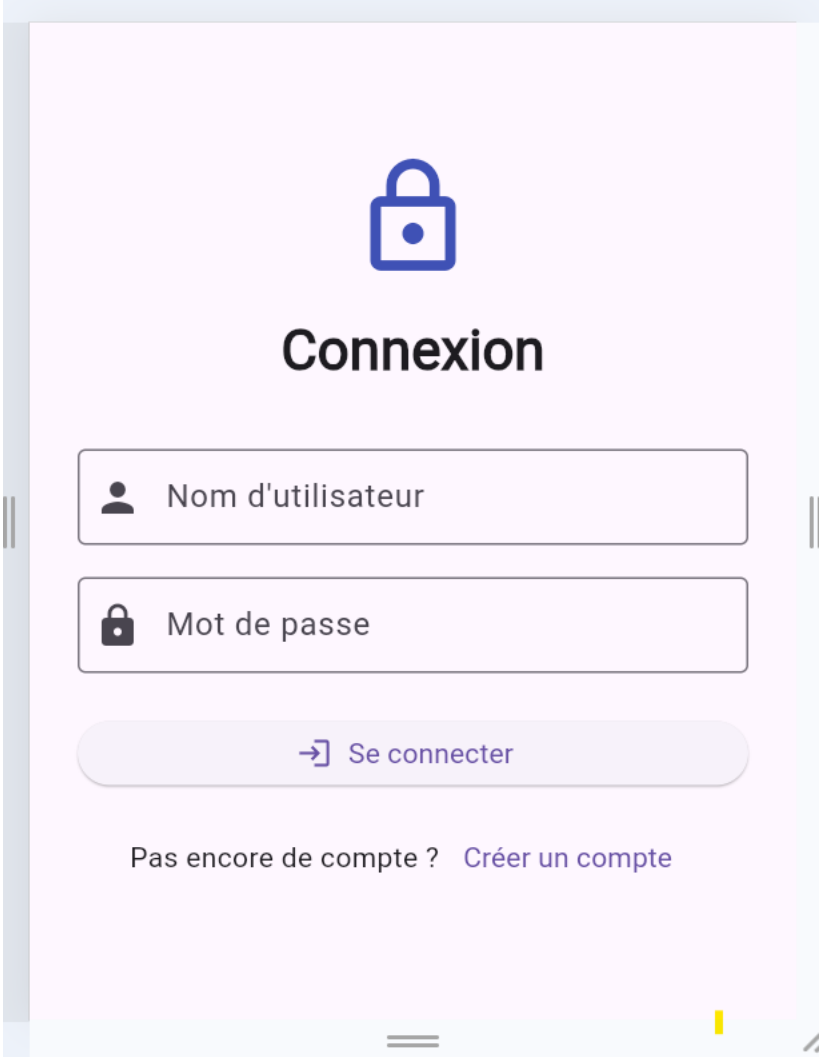
Stockage du token avec flutter_secure_storage
Les tokens JWT (access et refresh) sont stockés de manière sécurisée sur l'appareil de l'utilisateur à l'aide de la librairie flutter_secure_storage. Cette librairie utilise des mécanismes de stockage sécurisés spécifiques à la plateforme (Keychain pour iOS, Keystore pour Android) pour protéger les informations sensibles.
Provider : gestion de session globale
Un Provider (par exemple, avec le package provider ou bloc/riverpod) est utilisé pour gérer l'état de la session utilisateur. Il stocke les tokens, gère leur rafraîchissement automatique et met à jour l'état d'authentification de l'application, permettant une navigation protégée.

4.3. Écrans de l’application

L'application mobile est composée de plusieurs écrans clés, chacun avec un rôle spécifique. [3333]

1. Écran de Connexion (LoginScreen)

- **Comportement, navigation** : Cet écran permet aux utilisateurs de saisir leurs identifiants. En cas de succès, le token JWT est stocké et l'utilisateur est redirigé vers la liste des projets. Un lien permet également de naviguer vers l'écran d'enregistrement.
- **Extrait de code Flutter (UI, API)** :
  Dart
  ```
  // Voir le code fourni précédemment pour le LoginScreen
  // ... (Widgets pour champs de texte, bouton de connexion, indicateur de chargement, message d'erreur)
  // ...
  // onPressed: _login, // Appel de la méthode de connexion
  ```

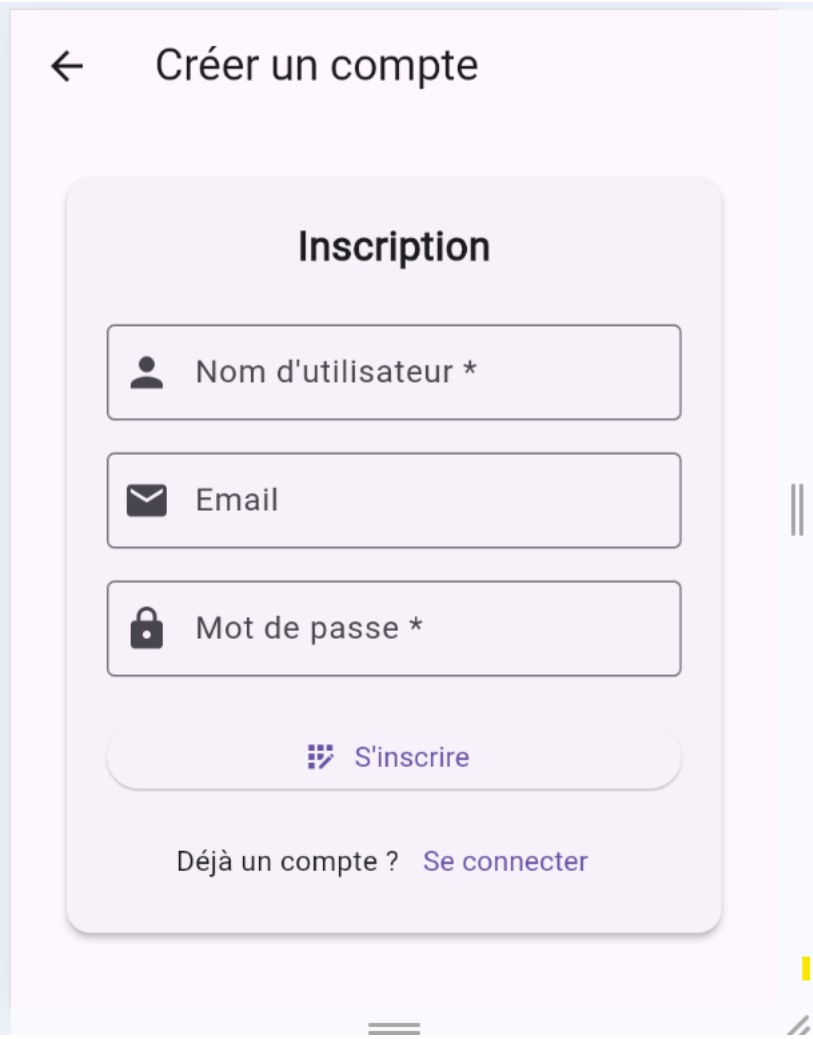


2. Écran d'Enregistrement (RegisterScreen)

- **Comportement, navigation** : Permet aux nouveaux utilisateurs de créer un compte. Après un

enregistrement réussi, l'utilisateur peut être automatiquement connecté ou redirigé vers l'écran de connexion.

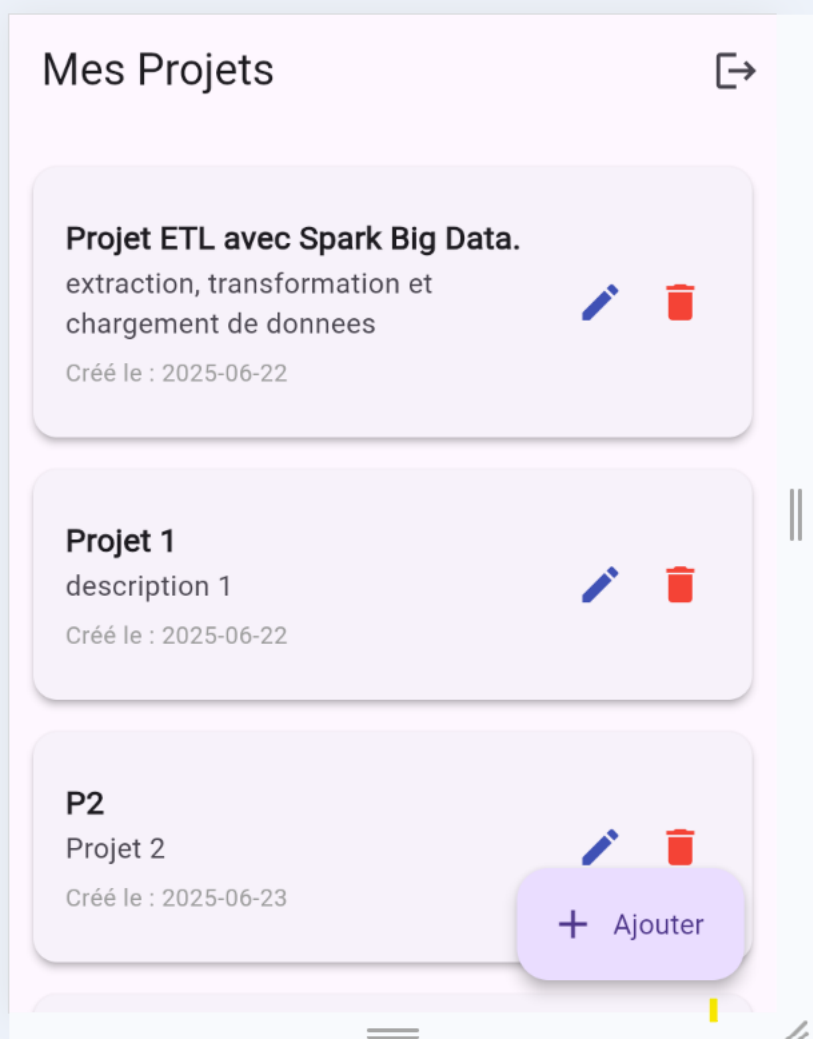
- **Extrait de code Flutter (UI, API)** :
  Dart
  ```
  // Similaire au LoginScreen, avec champs pour nom d'utilisateur, mot de passe, confirmation.
  ```



3. Écran de Liste des Projets (ProjetListScreen)

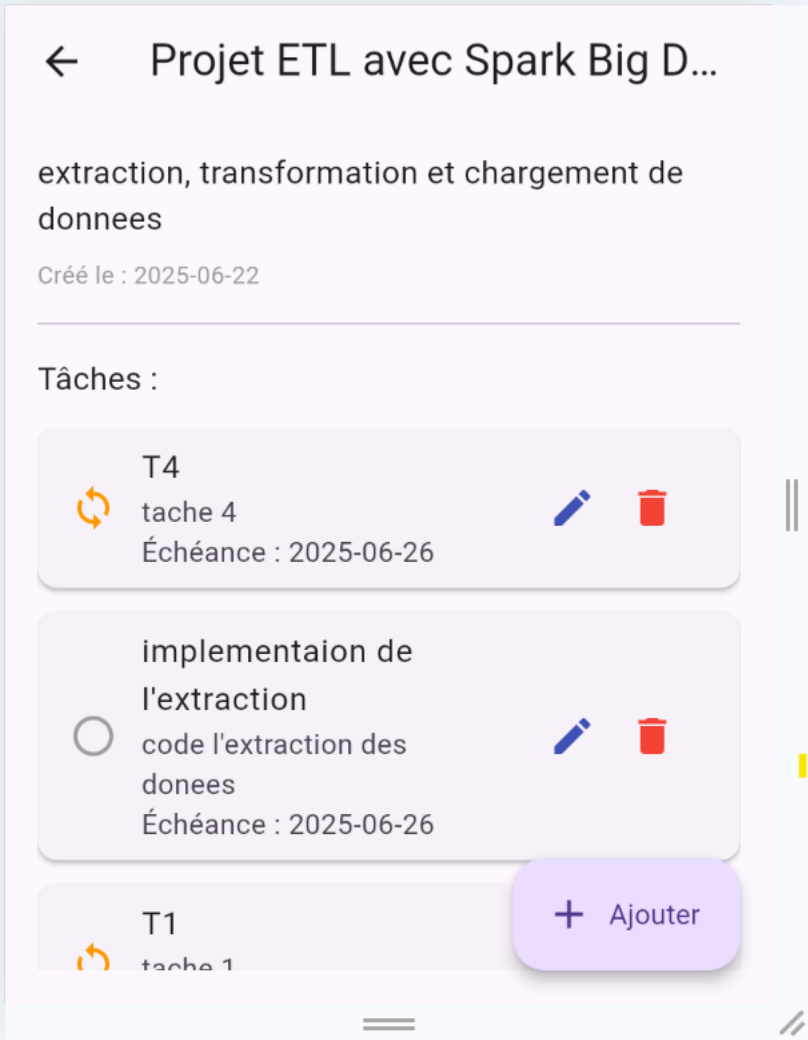
- **Comportement, navigation** : Affiche la liste des projets appartenant à l'utilisateur. Chaque projet est présenté dans une carte (Card) avec son nom et une brève description. Un bouton flottant (FloatingActionButton) permet d'ajouter un nouveau projet. Taper sur un projet navigue vers l'écran de détail du projet.
- **Extrait de code Flutter (UI, API)** :

```
Dart
// Exemple de rendu d'un élément de liste de projet
// Card(
//   elevation: 4,
//   margin: EdgeInsets.symmetric(vertical: 8, horizontal: 16),
//   shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(12)),
//   child: ListTile(
//     title: Text(projet.nom, style: Theme.of(context).textTheme.headline6),
//     subtitle: Text(projet.description ?? ''),
//     onTap: () => Navigator.pushNamed(context, '/detailProjet', arguments: projet),
//     trailing: Icon(Icons.arrow_forward_ios),
//   ),
// )
```



4. Écran de Détail du Projet (ProjetDetailScreen)

- **Comportement, navigation** : Affiche les détails d'un projet spécifique et la liste de ses tâches. Un bouton permet d'ajouter une nouvelle tâche ou de modifier le projet. Chaque tâche peut être modifiée ou marquée comme complétée.
- **Extrait de code Flutter (UI, API)** :
  Dart
  ```
  // ... (Affichage du nom et de la description du projet)
  // ListView.builder pour les tâches
  // FloatingActionButton pour ajouter/modifier
  ```

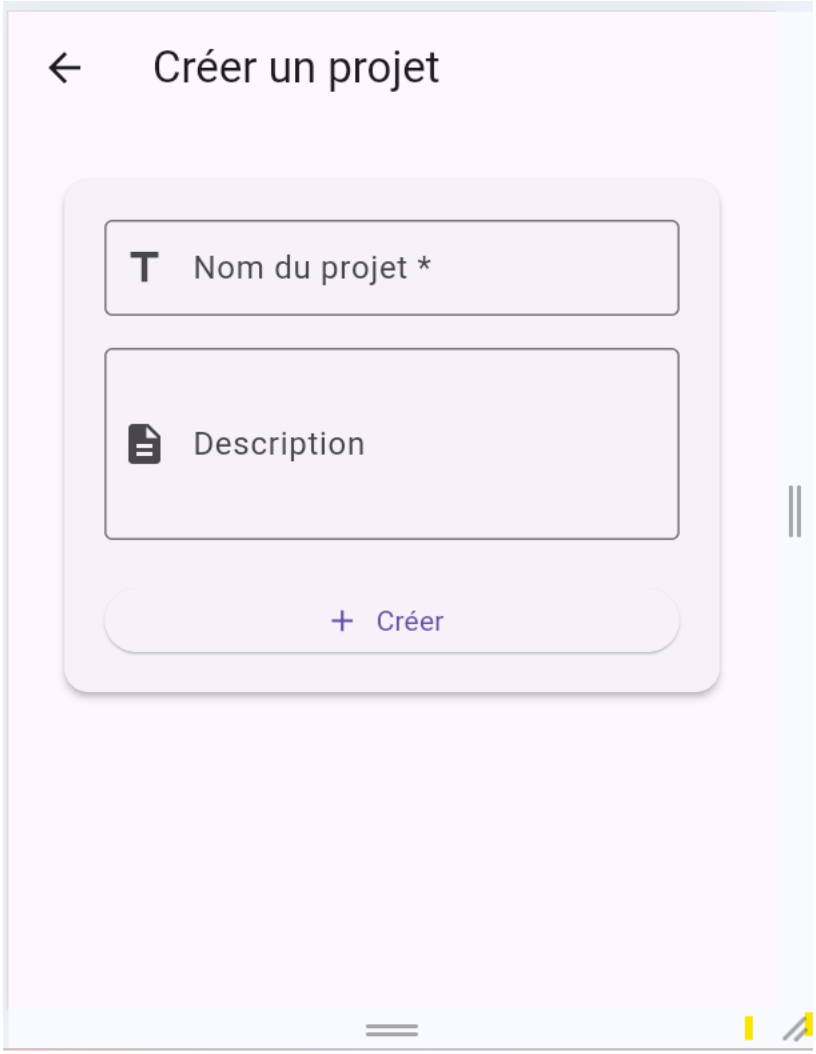


5. Écran de Création/Modification de Projet (CreateEditProjetScreen)

- **Comportement, navigation** : Un formulaire pour saisir le nom et la description d'un projet. Le même écran est utilisé pour la création (champs vides) et la modification (champs pré-remplis). Un bouton "Créer" ou "Mettre à jour" soumet les données à l'API.

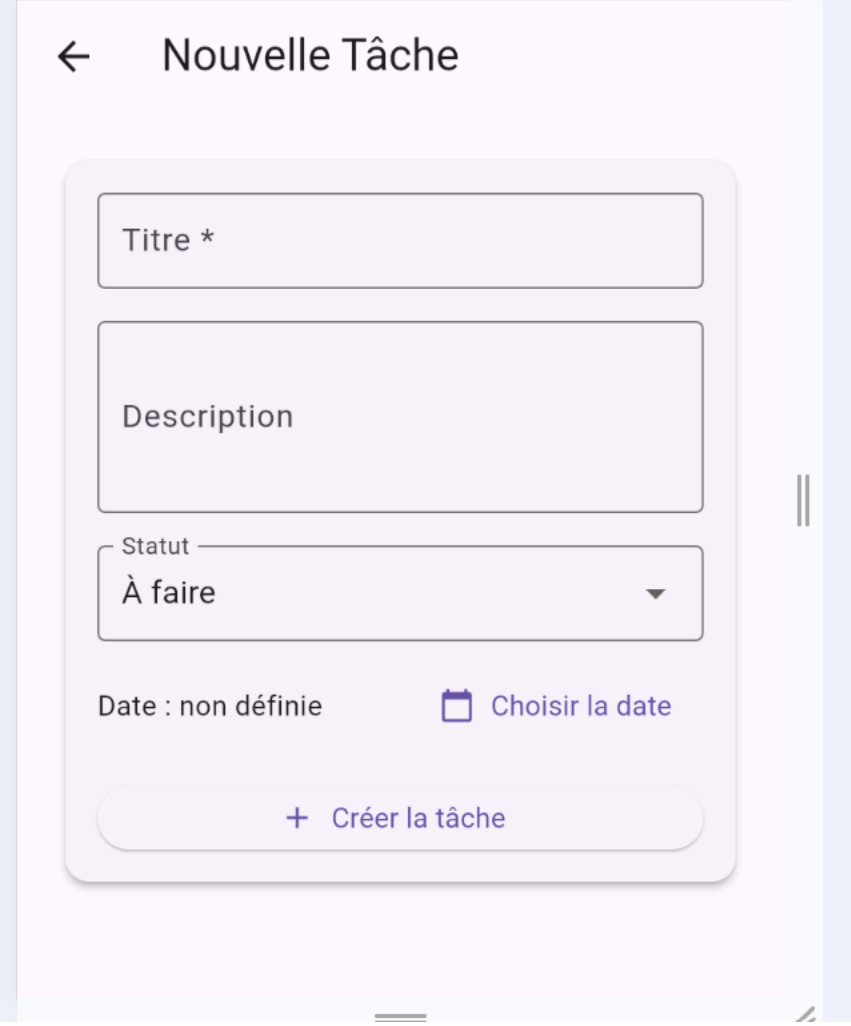
- **Extrait de code Flutter (UI, API)** :
  Dart

```
// Voir le code fourni précédemment pour le CreateEditProjetScreen
// ... (TextFormField pour nom, description)
// ... (Bouton "Créer" / "Mettre à jour")
```



6. Écran de Création/Modification de Tâche (CreateEditTacheScreen)

- **Comportement, navigation** : Formulaire pour le titre, la description, le statut et éventuellement la date d'échéance d'une tâche. Intégration de Dropdown pour le statut et DatePicker pour la date.
- **Extrait de code Flutter (UI, API)** :
  Dart
  ```
  // ... (TextFormField pour titre, description)
  // DropdownButton pour le statut
  // ElevatedButton pour DatePicker
  // Bouton "Créer" / "Mettre à jour"
  ```



4.4. Appels réseau (ApiService)

Le service ApiService gère toutes les interactions avec le backend, encapsulant la logique des requêtes HTTP et la gestion des tokens JWT.

Code simplifié de getProjets, createTache, etc.

Dart

```
// Dans lib/services/api_service.dart
import 'package:http/http.dart' as http;
import 'dart:convert';
import 'package:flutter_secure_storage/flutter_secure_storage.dart';
```

```dart
class ApiService {
  static const String _baseUrl = "http://127.0.0.1:8000/api"; // URL de votre backend
  static const _storage = FlutterSecureStorage();

  static Future<Map<String, String>> _getHeaders() async {
    final token = await _storage.read(key: 'access_token');
    return {
      'Content-Type': 'application/json',
      'Authorization': 'Bearer $token',
    };
  }

  static Future<http.Response> login(String username, String password) async {
    // Appel POST à /api/auth/login/
    final url = Uri.parse('$_baseUrl/auth/login/');
    return await http.post(
      url,
      headers: {'Content-Type': 'application/json'},
      body: json.encode({'username': username, 'password': password}),
    );
  }

  static Future<http.Response> getProjets() async {
    final url = Uri.parse('$_baseUrl/projets/');
    final headers = await _getHeaders();
    return await http.get(url, headers: headers);
  }

  static Future<http.Response> createProjet(Map<String, dynamic> data) async {
    final url = Uri.parse('$_baseUrl/projets/');
    final headers = await _getHeaders();
    return await http.post(url, headers: headers, body: json.encode(data));
  }

  static Future<http.Response> updateProjet(int id, Map<String, dynamic> data) async {
    final url = Uri.parse('$_baseUrl/projets/$id/');
    final headers = await _getHeaders();
    return await http.put(url, headers: headers, body: json.encode(data));
  }

  static Future<http.Response> deleteProjet(int id) async {
    final url = Uri.parse('$_baseUrl/projets/$id/');
    final headers = await _getHeaders();
    return await http.delete(url, headers: headers);
```

```
  }

  // Fonctions similaires pour les tâches (getTaches, createTache, etc.)
  static Future<http.Response> getTaches(int projetId) async {
    final url = Uri.parse('$_baseUrl/projets/$projetId/taches/');
    final headers = await _getHeaders();
    return await http.get(url, headers: headers);
  }
  // ... autres méthodes pour Tache
}
```

4.5. UI moderne

L'interface utilisateur de l'application est conçue pour être moderne, claire et intuitive, en suivant les principes du Material Design de Flutter.

- **Utilisation de Card** : Les éléments de liste (projets, tâches) sont encapsulés dans des widgets Card pour une meilleure délimitation visuelle et un effet d'élévation agréable.
- **Dropdown et DatePicker** : Utilisation de widgets spécifiques pour une saisie de données facilitée et une meilleure expérience utilisateur (ex: choix de statut, sélection de date).
- **Icônes, couleurs par statut** : Les icônes et les couleurs sont utilisées de manière cohérente pour indiquer les actions (ex: icône de crayon pour éditer, poubelle pour supprimer) et les statuts (ex: couleur verte pour "Terminé", orange pour "En cours").
- **Accessibilité mobile** : L'interface est conçue pour être responsive et s'adapter aux différentes tailles d'écran, garantissant une bonne lisibilité et interactivité sur divers appareils mobiles.

5. Déploiement

5.1. Backend en local (ou Railway / Render)

Installation Python, PostgreSQL, Django :

1. Installer Python (version 3.9+) et pip.
2. Installer PostgreSQL et créer une base de données.
3. Cloner le dépôt GitHub du backend.
4. Créer et activer un environnement virtuel :
   ```
   Bash
   python -m venv venv
   source venv/bin/activate  # ou `venv\Scripts\activate` sur Windows
   ```
5. Installer les dépendances :
   ```
   Bash
   pip install -r requirements.txt
   ```
6. Configurer la base de données dans settings.py.
7. Appliquer les migrations :

Bash
python manage.py makemigrations
python manage.py migrate

8. Créer un superutilisateur (pour l'administration) :
   Bash
   python manage.py createsuperuser

Commandes de lancement :

- Lancer le serveur de développement Django :
  Bash
  python manage.py runserver
  L'API sera accessible généralement sur http://127.0.0.1:8000/api/.

5.2. Frontend (émulateur / Android Studio)

Flutter doctor :

1. Installer Flutter SDK et configurer votre environnement de développement (Android Studio, VS Code).
2. Vérifier l'installation avec :
   Bash
   flutter doctor

3. Accepter les licences Android :
   Bash
   flutter doctor --android-licenses

flutter run ou .apk :

1. Cloner le dépôt GitHub du frontend.
2. Accéder au répertoire du projet.
3. Installer les dépendances :
   Bash
   flutter pub get

4. Lancer l'application sur un émulateur ou un appareil connecté :
   Bash
   flutter run

5. Pour générer un fichier .apk de release :
   Bash
   flutter build apk --release

6. Résultats & Tests

Cette section présente des captures d'écran des principales fonctionnalités de l'application et décrit les scénarios de test validés.

Captures des écrans principaux :

Scénarios testés :
Les scénarios suivants ont été testés avec succès :

- **Connexion** : Un utilisateur existant peut se connecter avec succès et accéder à ses projets.
- **Création de projet** : Création d'un nouveau projet qui apparaît ensuite dans la liste.
- **Édition de projet** : Modification du nom et de la description d'un projet existant.
- **Suppression de projet** : Suppression d'un projet, entraînant la suppression de ses tâches associées.
- **Création de tâche** : Ajout de nouvelles tâches à un projet existant.
- **Édition de tâche** : Modification du titre, de la description, du statut ou de la date d'échéance d'une tâche.
- **Gestion de statut** : Changement du statut d'une tâche (À faire, En cours, Terminé) et mise à jour visuelle.
- **Affectation automatique** : Lors de la création d'un projet, il est automatiquement lié à l'utilisateur authentifié. Lors de la création d'une tâche, elle est automatiquement liée au projet sélectionné.
- **Navigation fluide** : Navigation entre les écrans (connexion -> liste projets -> détail projet -> création/modification) sans problème.
- **Persistance de l'authentification** : L'utilisateur reste connecté entre les sessions (grâce à flutter_secure_storage).

7. Améliorations futures

Plusieurs pistes d'amélioration sont envisagées pour enrichir les fonctionnalités et l'expérience utilisateur :

- **Rafraîchissement automatique des tokens** : Implémenter une logique de rafraîchissement des tokens JWT en arrière-plan pour éviter les déconnexions intempestives.
- **Affectation multi-utilisateur** : Permettre d'assigner des tâches à différents utilisateurs (nécessiterait d'étendre la logique de permission et le modèle Tache).
- **Tableau de bord statistiques** : Ajouter un écran de tableau de bord affichant des statistiques sur les projets et les tâches (nombre de tâches terminées, en cours, etc.).
- **Export PDF / Excel** : Offrir la possibilité d'exporter les listes de projets ou de tâches sous différents formats.
- **Notifications push** : Implémenter des notifications pour les rappels de tâches ou les mises à jour de projet.
- **Recherche et filtrage** : Ajouter des fonctionnalités de recherche et de filtrage avancées pour les projets et les tâches.
- **Internationalisation** : Support de plusieurs langues pour l'interface utilisateur.

8. Annexes

Code source complet sur GitHub :

- **Backend (Django)** : [Lien vers votre dépôt GitHub du backend]
- **Frontend (Flutter)** : [Lien vers votre dépôt GitHub du frontend]

Liste des packages Flutter et modules Django utilisés :

Flutter (pubspec.yaml extraits) :

- http: ^x.x.x (pour les appels API)
- flutter_secure_storage: ^x.x.x (pour le stockage sécurisé des tokens)
- provider: ^x.x.x (ou autre package de gestion d'état)

Django :

- Django==x.x.x
- djangorestframework==x.x.x
- djangorestframework-simplejwt==x.x.x
- psycopg2-binary==x.x.x (si PostgreSQL est utilisé)

JSON de réponse type pour un projet :

JSON


```
{
  "id": 1,
  "nom": "Développement de l'Application TaskManager",
  "description": "Création d'une application de gestion de projets et tâches en Django/Flutter.",
  "date_creation": "2025-06-25T14:30:00Z",
  "utilisateur": {
    "id": 1,
    "username": "monutilisateur"
  },
  "taches": [
    {
      "id": 101,
      "titre": "Concevoir le MCD",
      "description": "Élaborer le modèle conceptuel de données pour les entités.",
      "statut": "TERMINE",
      "date_creation": "2025-06-25T14:35:00Z",
      "date_echeance": "2025-06-26",
```

```
            "assignee_a": null
        },
        {
            "id": 102,
            "titre": "Mettre en place l'API d'authentification",
            "description": "Créer les endpoints pour l'enregistrement et la connexion JWT.",
            "statut": "EN_COURS",
            "date_creation": "2025-06-25T15:00:00Z",
            "date_echeance": "2025-06-28",
            "assignee_a": null
        }
    ]
}
```
