- Rapport de Stage -

Stage de 2e année du BUT GEII – Développement d'une API sécurisée

Hamari Anir étudiant BUT GEII parcours AII
Tuteur pédagogique : Arnaud Sivert
Tuteur en entreprise : M. Youness Chrifi – Altair Groupe

Effectué du 14/04/2025 au 19/06/2025
Dans le cadre de la validation du BUT GEII (2e année)

Développement d'une API sécurisée Django/PostgreSQL pour la gestion parent-enfant – Projet Altair Parents

---

Lieu du stage :
Altair Soutien Scolaire & Cours Particuliers - Soissons

Adresse :

14 Rue Saint-Léger, 02200 Soissons

Remerciements

Je tiens à exprimer ma profonde gratitude à l'ensemble de l'équipe du projet Altair Parents pour leur accueil chaleureux, leur disponibilité et la confiance qu'ils m'ont accordée tout au long de cette période de stage. L'ambiance de travail, à la fois calme et détendue, m'a permis de progresser dans un cadre serein, propice à l'autonomie et à la concentration. Bien que j'aie travaillé de manière largement autonome, j'ai eu l'opportunité de participer à des points d'avancement réguliers, ce qui a favorisé une organisation claire et un bon suivi du projet.

Je remercie tout particulièrement ***Monsieur Youness Chrifi****, Directeur général et tuteur de stage, pour son encadrement basé sur une* ***autonomie guidée****. Il m'a laissé la liberté d'explorer et de progresser par moi-même, tout en restant disponible lorsque j'avais besoin d'éclaircissements ou de conseils techniques. Cette approche m'a permis de développer ma* ***capacité d'auto-apprentissage****, une méthode que je continuerai à utiliser dans mes projets futurs.*

Je tiens également à exprimer ma reconnaissance à ***Madame Sihem****, responsable des ressources humaines, pour son accueil chaleureux et son accompagnement lors de mon arrivée. Elle a veillé à ce que je sois bien intégré dans l'équipe et que je dispose de* ***toutes les conditions matérielles et humaines nécessaires*** *pour effectuer mon stage dans un environnement serein et motivant.*

Enfin, je remercie l'ensemble de l'équipe pédagogique du BUT GEII de l'IUT de l'Aisne pour la qualité de la formation reçue durant ces deux années. En particulier, les enseignements liés au développement et à la programmation m'ont été directement utiles pour comprendre les fondements du projet. Ce stage m'a permis de mesurer la cohérence entre les savoirs acquis à l'IUT et leur mise en œuvre dans un contexte professionnel réel, tout en identifiant les compétences complémentaires que j'ai pu développer par moi-même.

Sommaire

Introduction

Dans le cadre de ma deuxième année de BUT GEII (Génie Électrique et Informatique Industrielle) à l'IUT de l'Aisne, j'ai effectué un stage de deux mois, du 14 avril au 19 juin 2025, au sein du projet Altair Parents. Ce stage avait pour objectif de consolider mes acquis théoriques par une immersion concrète dans un environnement de développement informatique orienté web.

Altair Parents est un projet éducatif numérique visant à améliorer la communication entre parents et établissements scolaires. Mon rôle a été de participer activement au développement d'une API sécurisée en Django, connectée à une base de données PostgreSQL, et destinée à gérer les profils utilisateurs (parents, enfants), la messagerie et les fiches pédagogiques.

Ce rapport présente les différentes étapes de cette mission : la mise en place de l'environnement technique, la structuration de la base de données, la création des routes de l'API, ainsi que les évolutions vers une architecture plus robuste en passant de Firebase à PostgreSQL.

Il rend compte également des compétences techniques mobilisées, des difficultés rencontrées, des solutions apportées, et enfin d'un bilan personnel sur les apports de cette expérience pour la suite de mon parcours.

Presentation de l’entreprise

1. Historique et mission

Altair Group est un projet numérique innovant développé dans le but de renforcer le lien entre les familles et le cadre éducatif. Pensé comme une plateforme pédagogique, il permet aux parents de suivre plus facilement la scolarité de leurs enfants grâce à une interface claire et intuitive.

Le projet repose sur une volonté simple : centraliser les informations essentielles pour les parents, tout en fluidifiant la communication avec les équipes éducatives (fiches, messages, informations de suivi, etc.). Altair Groupe est une structure de soutien scolaire créée en 2024 à Soissons, spécialisée dans l’accompagnement pédagogique personnalisé des élèves du primaire au lycée. Fondée et dirigée par M. Youness Chrifi, elle propose des cours en petits groupes, des stages de révision, ainsi qu’un suivi individualisé, en présentiel et en ligne.

Dans ce cadre, le projet Altair Parents a vu le jour comme une initiative numérique interne visant à améliorer la communication entre les familles et les encadrants pédagogiques. Il s’agit d’un projet en développement, pensé pour offrir une interface simple et intuitive aux parents afin de consulter les fiches de travail de leurs enfants et de contacter l’équipe pédagogique facilement.

2 Objectifs du projet Altair Parents

Le projet Altair Parents vise à créer une application web éducative permettant aux parents de :

- **Consulter les fiches pédagogiques de leurs enfants selon leur niveau ;**
- **Envoyer des messages au support pédagogique ou aux enseignants ;**
- **Suivre les informations scolaires via un tableau de bord simple et clair.**

Ce projet s'inscrit dans une démarche de numérisation de l'activité de soutien scolaire, en complément des cours en présentiel. L'idée est de proposer un outil accessible, sans authentification complexe, compatible avec les besoins d'un public non technicien, tout en assurant la sécurité des données et la clarté des échanges.

3 Organisation et technologies utilisées

Le développement de l'application repose sur une pile technologique moderne :

- **Back-end** : Django (Python) + Django REST Framework ;
- **Base de données** : PostgreSQL (remplaçant Firebase, pour des raisons de coût et de stabilité) ;
- **Front-end** : React, avec une interface responsive ;
- **Outils utilisés** : Visual Studio Code, GitHub, Postman, pgAdmin.

L'organisation du travail s'est faite en autonomie encadrée, avec des points réguliers avec le tuteur. Le projet a été conduit de manière agile, en ajustant les objectifs au fil des avancées et des tests. Il m'a permis de prendre en main l'ensemble du cycle de développement d'une application web, du modèle de données à l'interface utilisateur.

Objectifs du stage

Ce stage de deuxième année de BUT GEII avait pour objectif principal de permettre une mise en pratique des compétences acquises en développement informatique, architecture logicielle et gestion de projet, dans un cadre réel.

Le projet confié – Altair Parents – s'inscrivait dans une logique de développement interne au sein d'Altair Groupe, avec une forte dimension technique. Il m'a permis de participer à la conception et à la réalisation d'une API REST sous Django, en lien avec une base de données PostgreSQL, et destinée à gérer des données sensibles : parents, enfants, messages et fiches pédagogiques.

Les objectifs spécifiques du stage étaient les suivants :

- Mettre en place un environnement de développement Django/PostgreSQL fonctionnel, sécurisé et bien structuré ;
- Modéliser et structurer les données dans une base relationnelle, avec des relations claires et des contraintes d'intégrité (clés étrangères, unicité) ;
- Créer des routes API REST fiables pour gérer les opérations CRUD sur les entités principales ;
- Migrer les données et la logique métier d'un environnement Firebase vers PostgreSQL, pour une meilleure maîtrise, sécurité et économie ;
- Ajouter des fonctionnalités avancées : système de messagerie parents-support, affichage de fiches pédagogiques par niveau, gestion de rôles et d'authentification ;

- Collaborer avec un tuteur technique, suivre des consignes de travail hebdomadaires, et apprendre à documenter un projet complet.

Ce stage avait aussi pour objectif de renforcer mon autonomie, ma capacité à apprendre par moi-même, et à développer des outils utiles, robustes et réutilisables, dans un contexte semi-professionnel.

Environnement technique

1 Stack technique: Django, PostgreSQL, React

Le développement du projet **Altair Parents** s'est appuyé sur une stack moderne, cohérente avec les standards actuels du web :

- **Django (Python)** : framework principal côté back-end, utilisé pour la gestion de l'API REST, les modèles de données, les vues et les permissions.
- **Django REST Framework** : utilisé pour exposer les ressources (Parents, Enfants, Messages, Fiches) en API sécurisée et testable.
- **PostgreSQL** : base de données relationnelle choisie pour remplacer Firebase. Elle a permis une structuration plus rigoureuse des données, des relations claires (via clés étrangères) et une meilleure fiabilité.
- **React** : utilisé pour créer l'interface web du côté utilisateur. L'application propose une interface simple pour consulter les fiches, envoyer des messages, et se connecter.
- **Firebase** : initialement utilisé en back-end (Firestore + Auth), il a été progressivement abandonné au profit de PostgreSQL, pour des raisons de coût, de complexité de gestion des relations, et de performance.
- **Token FCM (Firebase Cloud Messaging)** : intégré temporairement pour l'envoi futur de notifications.

2 Outils de travail: Postman, GitHub, VS Code

- **Visual Studio Code (VS Code)** : éditeur principal utilisé, avec les extensions Django, Python, REST Client, GitLens…
- **GitHub** : plateforme de versionnement pour sauvegarder le projet, gérer les commits, et suivre les évolutions.
- **Postman** : utilisé pour tester chaque route de l'API (GET, POST, PUT, DELETE), vérifier les statuts de réponse et le respect des règles de validation.
- **pgAdmin** : outil utilisé pour gérer graphiquement la base PostgreSQL (structure, requêtes, tables, données).

L'ensemble de ces outils m'a permis de travailler avec efficacité, de tester régulièrement le projet, et d'assurer une cohérence entre les différentes couches du développement (modèle, vue, API, interface).

Déroulement du stage

Semaine 1 – Mise en place de l'environnement (Django + PostgreSQL)

Objectif de la semaine

Mettre en place un environnement de travail local pour développer une API Django connectée à PostgreSQL.

J'ai commencé par installer les outils nécessaires au développement : Python, Django, PostgreSQL et pgAdmin sur ma machine.
Ensuite, j'ai créé le projet Django, puis j'ai fait les premières configurations pour le connecter à la base PostgreSQL.
J'ai ensuite défini les modèles Parent et Enfant dans models.py, en utilisant une relation ForeignKey pour associer chaque enfant à son parent.

```
61
62  DATABASES = {
63      'default': {
64          'ENGINE': 'django.db.backends.postgresql',
65          'NAME': 'altair_parents',
66          'USER': 'postgres',
67          'PASSWORD': 'admin123',
68          'HOST': 'localhost',
69          'PORT': '5432',
70      }
71  }
72
```

Figure 1 – Fichier settings.py : configuration de la base PostgreSQL

Dans Django (et dans les bases de données relationnelles), une **ForeignKey** sert à **lier une table à une autre**.

C'est comme dire :

"cet objet appartient à un autre"
ou
"cet enfant est lié à un parent"

Une fois les modèles prêts, j'ai configuré la base PostgreSQL dans settings.py, puis lancé les premières migrations (makemigrations, migrate).
Enfin, j'ai créé un superuser pour accéder à l'interface d'administration Django et tester les premiers enregistrements.

Par exemple, la table Enfant contient une colonne spéciale appelée parent_id qui permet de relier chaque enfant à un parent précis.
Ensuite, j'ai créé un **superutilisateur Django** grâce à la commande createsuperuser, ce qui m'a permis de tester l'enregistrement des premiers parents et enfants depuis l'interface d'administration (/admin), et de vérifier que la relation entre les deux fonctionnait

correctement.

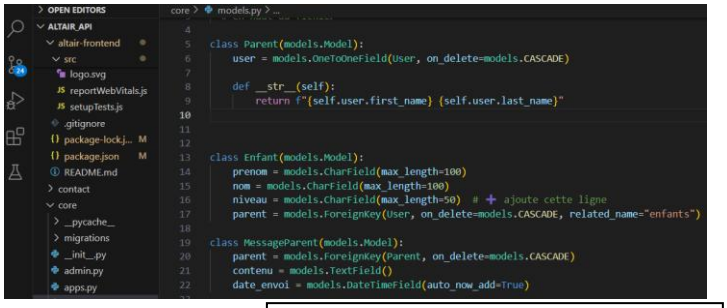


Figure 2 – Fichier models.py avec les modèles Parent et Enfant

Interprétés rencontrées
J'ai rencontré un problème avec le mot de passe PostgreSQL, à cause de caractères spéciaux mal interprétés.
J'ai aussi eu du mal au départ à bien comprendre comment fonctionnaient les clés étrangères dans Django.

Solutions apportées
Pour le mot de passe, j'ai corrigé l'encodage en échappant les caractères spéciaux dans le fichier de configuration.
Pour les clés étrangères, j'ai pris le temps de lire la documentation officielle de Django et de faire des tests locaux pour mieux comprendre leur fonctionnement.

Résultat obtenu
À la fin de cette phase, l'environnement Django/PostgreSQL était prêt et fonctionnel.
Les modèles étaient opérationnels et les premières routes API pouvaient commencer à être développées.

Semaine 2 – Création des modèles et des routes API

Objectif de la semaine
L'objectif de cette semaine était de finaliser les modèles de données, de créer les premières routes API pour les entités Parent et Enfant, et de tester leur bon fonctionnement à l'aide d'outils comme Postman.

Travail réalisé
J'ai commencé par ajouter les fichiers serializers.py dans chaque application. Ces fichiers permettent de convertir les objets Python en JSON, ce qui est nécessaire pour le dialogue entre l'API et le front-end.

J'ai ajouté un fichier appelé serializers.py dans mon application.
Ce fichier sert à **traduire les données** : il transforme les informations de Django (comme un parent ou un enfant) en un format plus simple appelé **JSON**, que l'interface ou un outil de test peut lire facilement.
Par exemple, si on enregistre un parent, le fichier serializers.py va convertir ses informations en texte organisé comme ça :

json

CopierModifier

{

"id": 1,

"nom": "Dupont",

"email": "dupont@example.com"

}

Sans ce fichier, **le site ne pourrait pas afficher les données correctement**, et l'échange entre le serveur et la partie visible ne fonctionnerait pas.
C'est un peu comme un traducteur entre Django et le reste de l'application.

Ensuite, j'ai créé les vues (views.py) en définissant les méthodes GET, POST, PUT et DELETE, pour pouvoir lire, ajouter, modifier et supprimer des données.
J'ai aussi configuré les routes dans urls.py, à la fois dans le projet principal et dans chaque application, afin que l'API soit bien structurée.

Pour permettre les premiers tests, j'ai temporairement laissé les routes en accès libre sans authentification.
J'ai ensuite réalisé des tests complets avec Postman :

- ajout de parents
- ajout d'enfants associés à un parent
- lecture et modification des données via les routes REST

Enfin, j'ai mis en place une contrainte unique_together pour éviter que deux enfants ayant exactement les mêmes informations soient ajoutés plusieurs fois.
J'ai également redirigé l'URL racine / vers /api/ pour simplifier l'accès à l'API pendant les tests.

Difficultés rencontrées
J'ai rencontré plusieurs erreurs 404, dues à des fautes dans les noms de chemins ou de vues.
En modifiant les modèles après avoir déjà fait des migrations, j'ai aussi eu des conflits entre les fichiers de migration.

Solutions apportées
J'ai relu attentivement tous les urlpatterns et corrigé les noms de route.
J'ai aussi supprimé les anciennes migrations erronées, puis recréé des migrations propres pour repartir sur une base stable.
Pour mieux organiser l'API, j'ai utilisé les namespaces dans les fichiers urls.py.

Résultat obtenu

À la fin de la semaine, l'API Django REST était fonctionnelle pour les entités Parent et Enfant. Tous les tests CRUD (Create, Read, Update, Delete) ont été validés avec Postman.

core > serializers.py > ...

```python
# serializers.py
from rest_framework import serializers
from .models import Parent, Enfant, MessageParent, Fiche, ContactMessage

class ParentSerializer(serializers.ModelSerializer):
    id = serializers.IntegerField(source='user.id', read_only=True)
    nom = serializers.CharField(source='user.last_name', read_only=True)
    prenom = serializers.CharField(source='user.first_name', read_only=True)
    email = serializers.EmailField(source='user.email', read_only=True)

    class Meta:
        model = Parent
        fields = ['id', 'nom', 'prenom', 'email']


class EnfantSerializer(serializers.ModelSerializer):
    parent = ParentSerializer(read_only=True)
    parent_id = serializers.IntegerField(write_only=True)

    class Meta:
        model = Enfant
        fields = ['id', 'nom', 'prenom', 'niveau', 'parent', 'parent_id']

class MessageParentSerializer(serializers.ModelSerializer):
    class Meta:
        model = MessageParent
        fields = ['id', 'parent', 'email', 'contenu', 'date_envoi']

class FicheSerializer(serializers.ModelSerializer):
    class Meta:
        model = Fiche
        fields = ['id', 'titre', 'contenu', 'niveau']

class ContactMessageSerializer(serializers.ModelSerializer):
    class Meta:
        model = ContactMessage
```

Figure 3– Fichier serializers.py : sérialisation des modèles Parent et Enfant

Semaine 3 – Migration Firebase vers PostgreSQL

Objectif de la semaine

Cette semaine avait pour objectif de remplacer Firebase par PostgreSQL comme base de données principale.

Ce choix a été fait pour avoir une base plus stable, mieux structurée (avec des relations claires entre les tables), et pour éviter les limitations et les coûts variables associés à Firebase.

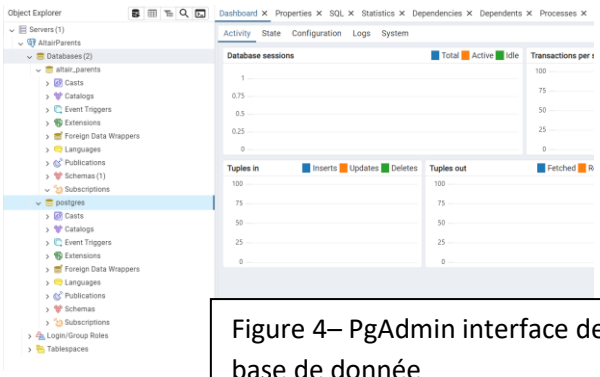


Figure 4– PgAdmin interface de la base de donnée

Travail réalisé

J'ai d'abord fait une analyse des limites de Firebase que j'avais rencontrées :

- *la difficulté à gérer des relations complexes comme celles entre un parent et plusieurs enfants,*
- *la structure de Firestore, qui rend les requêtes croisées assez compliquées à mettre en place,*
- *le coût d'utilisation qui dépend du volume de lecture et d'écriture, ce qui peut vite devenir un problème pour un projet en production.*

Après cette analyse, j'ai décidé de migrer entièrement vers PostgreSQL.

Ce choix me permettait d'avoir :

- *un contrôle total sur les données stockées,*
- *des relations explicites entre les tables (via les clés étrangères),*
- *et aucune dépendance à une plateforme externe payante.*

J'ai donc réécrit les modèles et certaines parties de l'API pour que tout soit compatible avec PostgreSQL uniquement.

J'ai aussi supprimé tous les anciens appels à Firebase qui restaient dans le projet, puis j'ai nettoyé et réorganisé le dépôt GitHub pour repartir sur une base propre.

Difficultés rencontrées
Les anciennes données enregistrées sur Firebase n'étaient pas réutilisables directement, car le format n'était pas compatible avec la nouvelle structure SQL.
J'ai aussi dû restructurer certains modèles Django pour bien profiter des avantages d'une base relationnelle.

Solutions apportées
J'ai exporté manuellement les données utiles depuis Firebase.
Ensuite, j'ai refactoré les vues Django pour qu'elles n'utilisent plus Firebase du tout.
Enfin, j'ai effectué des tests complets avec pgAdmin et Postman pour vérifier que tout fonctionnait correctement avec PostgreSQL.

Résultat obtenu
À la fin de cette étape, l'application utilisait 100 % PostgreSQL, avec une base de données propre et bien structurée.
Les modèles Django ont été adaptés, et l'API fonctionne parfaitement sans Firebase.

Figure 5 – Comparatif Firebase / PostgreSQL : structure, relations, coût :

Critère	Firebase (Firestore)	PostgreSQL
Type de base	NoSQL (documents non relationnels)	SQL relationnelle
Gestion des relations	Faible (relations manuelles, pas de clé étrangère)	Native (clés étrangères, jointures, contraintes)
Structure des données	Souple, sans schéma fixe	Structurée, schéma défini et cohérent
Requêtes complexes	Limité (pas de jointures)	Très performant pour requêtes imbriquées
Coût	Gratuit limité, facturation rapide	Gratuit en local, prévisible en production
Sécurité des accès	Règles Firebase (complexes à maintenir)	Permissions Django + authentification fine
Évolutivité dans le projet	Limité (maintenance difficile à grande échelle)	Robuste et scalable sur le long terme

Semaine 4 – Ajout de la messagerie parents-support

Objectif de la semaine
L'objectif de cette semaine était d'intégrer une fonctionnalité de messagerie, pour permettre aux parents de contacter le support directement depuis l'application.

Travail réalisé
Pour cela, j'ai d'abord créé un nouveau modèle nommé MessageParent, contenant les champs suivants :

- *email_parent : pour enregistrer l'adresse du parent qui envoie le message,*
- *contenu : le texte du message,*
- *date_envoi : automatiquement remplie à l'envoi du message (auto_now_add=True).*

J'ai intégré ce modèle à la base PostgreSQL en effectuant une migration classique.
Ensuite, j'ai développé une vue API (views.py) permettant de recevoir les messages via une requête POST.
J'ai également créé une route dédiée /api/message/ dans urls.py pour rendre cette fonctionnalité accessible.

J'ai réalisé plusieurs tests via Postman pour vérifier le bon fonctionnement de l'envoi :

- *envoi d'un message normal,*
- *test avec un champ vide,*
- *simulation d'envoi massif pour voir la réaction de l'API.*

Enfin, dans l'interface Django Admin, j'ai ajouté un filtrage par date pour faciliter la lecture des messages reçus, et j'ai mis en place l'enregistrement automatique de la date d'envoi.

Difficultés rencontrées
J'ai eu quelques soucis au niveau de la validation des champs dans le sérialiseur (serializers.py). Il fallait aussi penser à se protéger contre les messages trop longs, les injections, ou les abus en cas d'utilisation réelle (spam, envois multiples, etc.).

Solutions apportées

J'ai utilisé le champ EmailField de Django pour valider automatiquement les adresses email.
J'ai ajouté une limite de longueur au champ contenu, et prévu la possibilité de mettre en place une pagination ou une limitation par adresse IP dans une version future.

Résultat obtenu

Le système de messagerie est maintenant 100 % fonctionnel via l'API.
Les messages sont bien enregistrés dans PostgreSQL avec l'heure exacte, et ils sont consultables dans l'interface admin ou en console de développement.

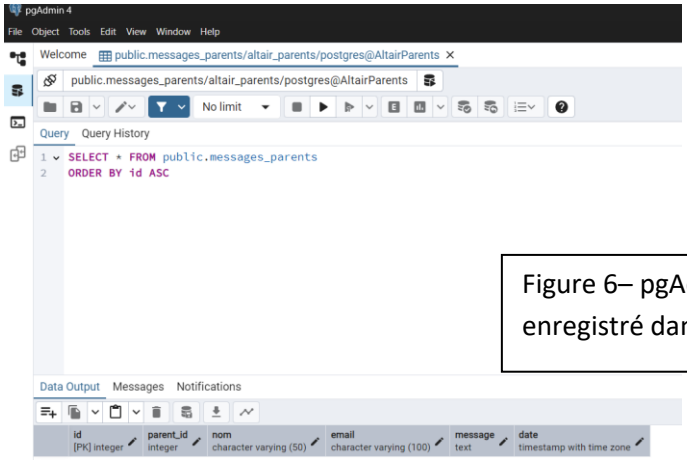


Figure 6– pgAdmin : message enregistré dans la base PostgreSQL

Semaine 5 – Mise en place de l'authentification et des rôles

Objectif de la semaine

Sécuriser l'API en limitant l'accès à certaines routes, et commencer à différencier les utilisateurs selon leur rôle (parent, administrateur…).

Travail réalisé

- Ajout d'un champ role pour les utilisateurs dans le modèle Parent (par exemple : is_admin = False)
- Implémentation de la **connexion par email et mot de passe** via un endpoint /api/login/
- Génération de **tokens JWT** (JSON Web Token) pour les utilisateurs authentifiés
- Configuration du middleware pour sécuriser les routes (ex. fiches, messages)
- Création d'un décorateur personnalisé pour contrôler les accès à certaines vues (ex : seuls les admins peuvent publier une fiche)
- Ajout du champ is_staff pour les administrateurs éventuels
- Vérification du rôle dans les vues :

Python :

CopierModifier

```
if request.user.is_staff:
    # autoriser action
```

Difficultés rencontrées

- Gestion des tokens expirés ou invalides
- Attribution dynamique des rôles lors de l'inscription (parent par défaut)

Solutions apportées

- Mise en place d'un système de **vérification de token** sur chaque requête protégée

- Utilisation des permissions DRF (IsAuthenticated, IsAdminUser) combinées à des vérifications personnalisées
- Tests approfondis via Postman avec token dans l'en-tête Authorization

Résultat obtenu

- Les routes critiques sont maintenant protégées
- Seuls les utilisateurs connectés peuvent accéder à certaines ressources
- Les rôles sont différenciés entre utilisateurs simples et administrateurs

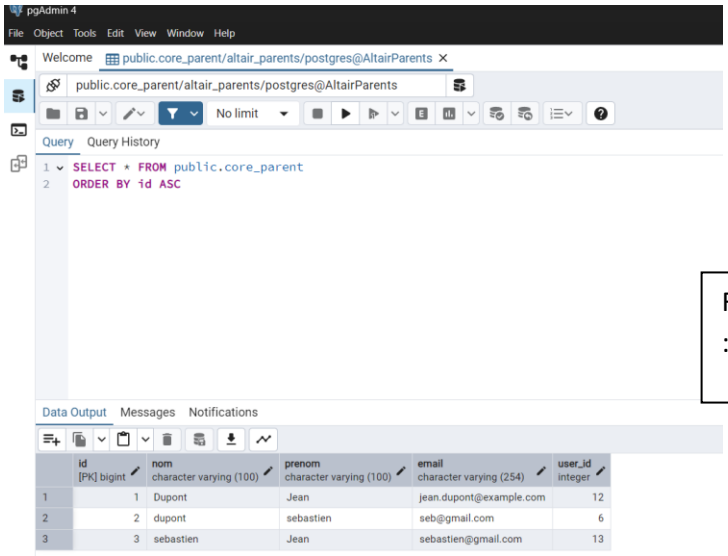


Figure 7 – Interface Django Admin : gestion des utilisateurs avec rôles

Semaine 6 – Intégration des fiches pédagogiques

Objectif de la semaine
Permettre aux parents de consulter des fiches pédagogiques selon le niveau scolaire de leurs enfants, via une interface sécurisée.

Objectif de la semaine
L'objectif était de créer un système de fiches pédagogiques, que chaque parent puisse consulter selon le niveau scolaire de ses enfants. Il fallait que ces fiches soient filtrées automatiquement à la connexion, pour ne pas afficher toutes les fiches à tout le monde.

Travail réalisé
J'ai commencé par créer le modèle Fiche dans Django, avec les champs suivants :

- titre : le titre de la fiche,
- contenu : le texte ou un lien vers un fichier PDF,
- niveau : le niveau scolaire concerné (ex. CP, CE1, CE2...),
- date_publication : enregistrée automatiquement à la création.

J'ai intégré ce modèle dans PostgreSQL grâce aux migrations.
Ensuite, j'ai développé deux routes API :

- /api/fiches/ pour récupérer toutes les fiches,
- /api/fiches?niveau=CE2 pour les filtrer par niveau.

J'ai mis en place une relation indirecte entre les fiches et les parents :
→ un parent est lié à un ou plusieurs enfants,
→ chaque enfant a un niveau,
→ les fiches sont filtrées automatiquement selon ce niveau.

Dans la vue Django, j'ai utilisé une requête du type :

```
python

CopierModifier

fiches = Fiche.objects.filter(niveau=enfant.niveau)
```

Et j'ai étendu ce système pour gérer les cas où un parent a plusieurs enfants avec des niveaux différents.

Difficultés rencontrées
Il a fallu adapter la requête pour qu'elle prenne en compte tous les enfants d'un parent, sans dupliquer les résultats ni générer d'erreurs.
J'ai aussi dû bien réfléchir à la logique entre les fiches, les enfants et les parents, pour que le filtre reste fiable et automatique.

Solutions apportées
J'ai utilisé une boucle sur tous les enfants associés au parent, et j'ai combiné les résultats avec queryset.union() pour éviter les doublons.
J'ai également ajouté un tri par date décroissante, afin que les fiches les plus récentes apparaissent en premier.

Résultat obtenu
L'API retourne désormais uniquement les fiches pédagogiques adaptées au profil du parent connecté.
L'affichage est filtré, sécurisé, et le système est facilement réutilisable pour d'autres niveaux ou matières.

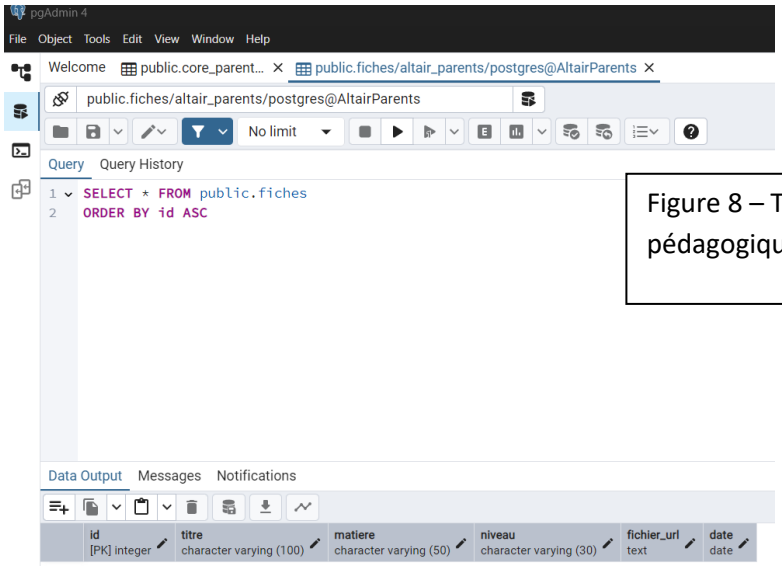


Figure 8 – Table des fiches pédagogiques dans PostgreSQL

Semaine 7 – Interface React et gestion des connexions

Objectif de la semaine

Cette semaine a été consacrée au développement de la partie front-end de l'application Altair Parents, en utilisant le framework React. L'objectif principal était de permettre aux parents de se connecter via une interface simple, puis d'afficher dynamiquement les fiches pédagogiques associées à leurs enfants.

Travail réalisé

J'ai commencé par initialiser le projet avec Vite + React, puis structuré les composants de base de l'application : LoginPage.jsx pour l'authentification, FichesPage.jsx pour l'affichage du contenu, et un Dashboard.jsx pour centraliser les vues principales.

Cette semaine, je me suis concentré sur la **partie visible de l'application**, celle que les parents utilisent. C'est ce qu'on appelle le **front-end**, et j'ai utilisé un outil moderne qui s'appelle **React** (avec Vite, un outil qui aide à créer des projets plus rapidement).

Mon objectif était de permettre aux **parents de se connecter facilement**, puis de leur **afficher les fiches pédagogiques** correspondant au niveau scolaire de leurs enfants.

Pour cela, j'ai organisé l'application en plusieurs **composants** (ce sont comme des blocs qui ont chacun leur rôle) :

- LoginPage.jsx : la page de **connexion** pour que le parent entre son email et mot de passe.
- FichesPage.jsx : la page où **les fiches apparaissent automatiquement** après la connexion.
- Dashboard.jsx : une page qui **regroupe les différentes vues** de l'application (accueil, menu, navigation…).

Grâce à cette structure, l'application est **plus claire**, plus facile à gérer, et les parents peuvent **voir uniquement ce qui les concerne**, sans avoir besoin de tout parcourir manuellement.

L'authentification a été connectée à l'API Django développée en back-end. Lorsqu'un parent entre son email et mot de passe, une requête est envoyée à l'API /api/login/. Si les informations

sont correctes, un token JWT est retourné, stocké dans le local storage pour authentifier les appels futurs.

Une fois connecté, le parent est redirigé automatiquement vers la page FichesPage, où s'affichent les fiches liées au niveau scolaire de ses enfants, récupérées via une requête GET sécurisée. J'ai mis en place un système de gestion d'état (useState, useEffect) pour déclencher cette récupération de données au chargement de la page.

Difficultés rencontrées

L'un des principaux défis a été de bien synchroniser la gestion du token JWT entre le front-end et l'API. Il fallait également éviter que des utilisateurs non connectés puissent accéder à certaines pages sensibles. Enfin, j'ai dû gérer les cas où aucun enfant n'était lié au parent, ou lorsqu'aucune fiche n'était encore disponible.

Solutions apportées

Pour répondre à ces enjeux, j'ai mis en place un contexte global (UserContext) afin de conserver les informations de l'utilisateur tout au long de la session. Une fonction fetchFiches() a été développée pour lancer automatiquement la requête des fiches après connexion. Des redirections conditionnelles ont aussi été ajoutées : un parent non connecté est systématiquement renvoyé vers la page de connexion.

Résultat obtenu

À ce stade du projet, l'interface est fonctionnelle : un parent peut se connecter, et les fiches pédagogiques correspondant au niveau de ses enfants s'affichent dynamiquement. L'interface est sobre, responsive, et reste en cours d'amélioration graphique pour les prochaines itérations.

1

```jsx
import React, { useEffect, useState } from "react";
import axios from "axios";
import { useNavigate, useLocation } from "react-router-dom";

const FichesPage = () => {
  const [fiches, setFiches] = useState([]);
  const [niveaux, setNiveaux] = useState([]);
  const navigate = useNavigate();
  const location = useLocation();

  const token = localStorage.getItem("access");
  const email = location.state?.email || localStorage.getItem("email");

  useEffect(() => {
    const fetchFiches = async () => {
      try {
        const parentRes = await axios.get(`http://localhost:8000/api/parents/?email=${email}`, {
          headers: { Authorization: `Bearer ${token}` },
        });

        const parentId = parentRes.data[0]?.id;
        if (!parentId) return;

        const enfantsRes = await axios.get(`http://localhost:8000/api/enfants/?parent_id=${parentId}`, {
          headers: { Authorization: `Bearer ${token}` },
        });

        const niveauxTrouvés = enfantsRes.data.map((e) => e.niveau);
        const fichesRes = await axios.get(`http://localhost:8000/api/fiches/`, {
          headers: { Authorization: `Bearer ${token}` },
        });

        const fichesFiltrées = fichesRes.data.filter((fiche) =>
          niveauxTrouvés.includes(fiche.niveau)
        );
```

2

```jsx
        setNiveaux([...new Set(niveauxTrouvés)]);
      } catch (error) {
        console.error("Erreur lors du chargement des fiches :", error);
      }
    };

    fetchFiches();
  }, [email, token]);

  return (
    <div className="max-w-3xl mx-auto py-10 px-4">
      <h2 className="text-3xl font-bold text-purple-700 mb-4 text-center">📚 Fiches pédagogiques</h2>

      <p className="text-sm text-gray-600 text-center mb-6">
        Niveaux de vos enfants : <strong>{niveaux.join(", ") || "Aucun"}</strong>
      </p>

      {fiches.length === 0 ? (
        <p className="text-center text-gray-500">Aucune fiche disponible pour vos enfants.</p>
      ) : (
        <div className="grid gap-6">
          {fiches.map((fiche) => (
            <div
              key={fiche.id}
              className="border rounded-lg shadow hover:shadow-lg transition p-5 bg-white"
            >
              <h3 className="text-xl font-semibold text-purple-800 mb-1">{fiche.titre}</h3>
              <p className="text-gray-500 text-sm mb-2">Niveau : {fiche.niveau}</p>
              <p className="text-gray-800 whitespace-pre-wrap">{fiche.contenu}</p>
            </div>
          ))}
        </div>
      )}
```

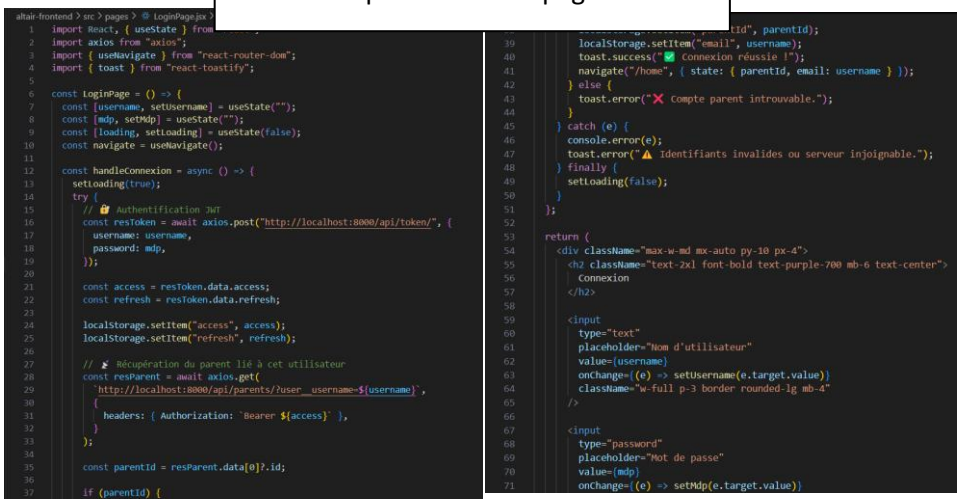
Figure 9 – Interface React : affichage dynamique des fiches pédagogiques

```
        <button
          onClick={() => navigate("/home")}
          className="mt-10 block mx-auto bg-purple-700 hover:bg-purple-800 text-white px-6 py-2 rounded shadow"
        >
          ← Retour à l'accueil
        </button>
      </div>
    );
  };

  export default FichesPage;
```

3

Figure 10 – extrait Formulaire de connexion parent dans la page



Semaine 8 – Finalisation, tests et ameliorations

Objectif de la semaine

Pour cette dernière semaine de stage, j'ai principalement travaillé sur la vérification de tout ce qui avait été fait, la correction des petits problèmes rencontrés, et l'amélioration générale de l'application avant la fin.

Travail réalisé

J'ai commencé par relire le code pour le rendre plus propre et plus clair. J'ai aussi supprimé les parties inutiles ou répétées. Ensuite, j'ai testé toutes les routes de l'API, comme l'ajout d'un

parent, d'un enfant, l'envoi d'un message, la connexion, ou encore la récupération des fiches pédagogiques.

Du côté de la base de données PostgreSQL, j'ai vérifié avec pgAdmin que tout était bien enregistré, que les liens entre les tables fonctionnaient, et qu'il n'y avait pas de doublons ou d'erreurs.

Pour la partie front-end avec React, j'ai amélioré certains messages affichés à l'écran, par exemple quand il n'y a pas de fiche disponible, ou quand la connexion échoue. J'ai aussi vérifié que le tableau des fiches se recharge bien après la connexion.

Enfin, j'ai sécurisé les routes sensibles en vérifiant que seul un utilisateur connecté avec un bon token pouvait y accéder.

Difficultés rencontrées

J'ai eu quelques soucis pour gérer les cas où un parent a plusieurs enfants. Il y a aussi eu des petits bugs quand le token de connexion était expiré, ou quand l'utilisateur actualisait la page manuellement.

Solutions apportées

J'ai modifié les requêtes pour bien prendre en compte tous les enfants d'un parent. J'ai aussi ajouté des vérifications pour savoir si le token est encore valide, et rediriger vers la page de connexion si ce n'est pas le cas.

Résultat obtenu

Le projet est désormais dans un état fonctionnel et stable, prêt à être mis en ligne ou enrichi par d'autres développeurs.

La base de données PostgreSQL est propre et bien structurée, les routes API sont testées et sécurisées, et l'interface React offre une expérience fluide malgré quelques éléments encore en amélioration.

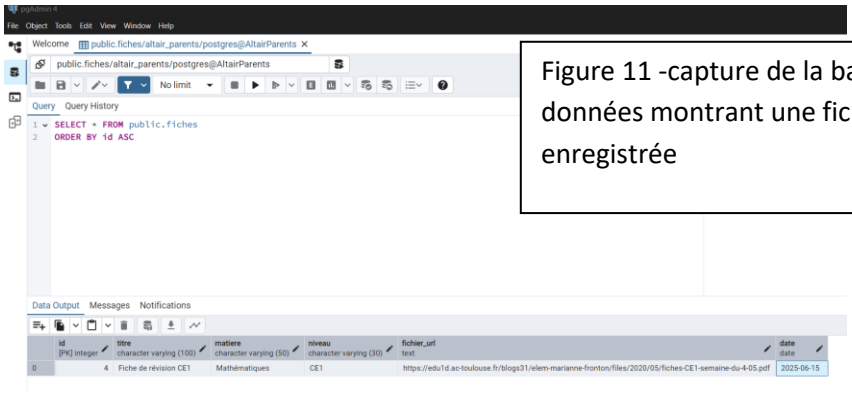


Figure 11 -capture de la base de données montrant une fiche enregistrée

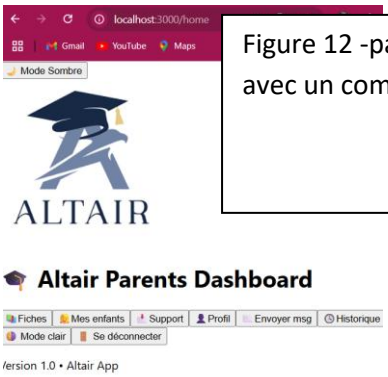


Figure 12 -page après connexion avec un compte parent

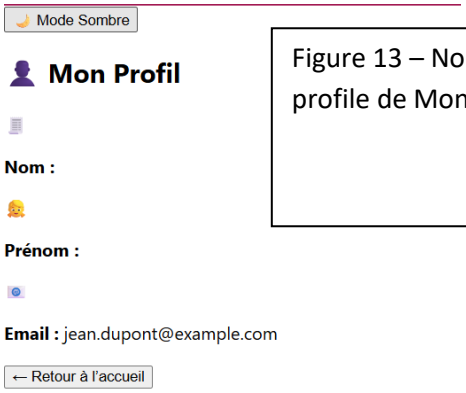


Figure 13 – Nous sommes sur le profile de Monsieur Jean Dupont

Analyse technique du projet

1 Architecture Générale

L'architecture du projet Altair Parents est divisée en deux grandes parties :

- un **back-end** développé avec **Django** (Python) qui gère la logique, les routes API, la base de données, les rôles, la sécurité ;
- un **front-end** développé avec **React**, qui gère l'affichage pour les utilisateurs (parents) et l'interaction avec l'API.

Les deux parties communiquent via des **requêtes HTTP** : le front envoie des demandes (login, récupérer des fiches…) à l'API, qui lui répond avec les données.

Une base de données **PostgreSQL** est utilisée pour stocker les informations : parents, enfants, messages, fiches pédagogiques.

9.2 Modèles de données

Le cœur du projet repose sur plusieurs modèles Django reliés entre eux :

- **Parent** : contient l'email, le nom, le mot de passe, et le rôle (admin ou non)
- **Enfant** : rattaché à un parent, avec son nom, prénom et niveau scolaire
- **Fiche** : contient un titre, une matière, un niveau, et une URL de fichier
- **MessageParent** : message envoyé par un parent vers le support

Chaque relation est définie proprement avec des **clés étrangères**, ce qui assure une bonne cohérence dans la base.

9.3 Sécurité et authentification

Le système utilise des **tokens JWT** (JSON Web Token) pour sécuriser les accès. Quand un parent se connecte, il reçoit un token, qu'il doit renvoyer avec chaque requête pour prouver son identité.

Les routes sont protégées selon le rôle de l'utilisateur. Par exemple :

- seuls les parents connectés peuvent voir les fiches liées à leurs enfants ;
- seul un administrateur peut ajouter une fiche (rôle is_staff activé).

9.4 API REST : L'API est construite avec **Django REST Framework**. Chaque entité du projet (Parent, Enfant, Fiche, Message) a :

- un modèle (models.py)
- un sérialiseur (serializers.py)

- une vue (views.py)
- une ou plusieurs routes (urls.py)

L'API accepte des requêtes GET, POST, PUT, DELETE selon les besoins.
Les réponses sont retournées au format **JSON**, ce qui facilite l'intégration avec le front-end React.

9.5 Intégration front-end

L'interface React utilise **Axios ou fetch()** pour appeler l'API.
Les parents peuvent :

- se connecter ;
- consulter les fiches pédagogiques selon le niveau de leurs enfants.

Les composants principaux sont :

- LoginPage.jsx
- FichesPage.jsx
- Dashboard.jsx (page d'accueil générale après connexion)

Le **token** est stocké localement dans le navigateur pour authentifier les requêtes.

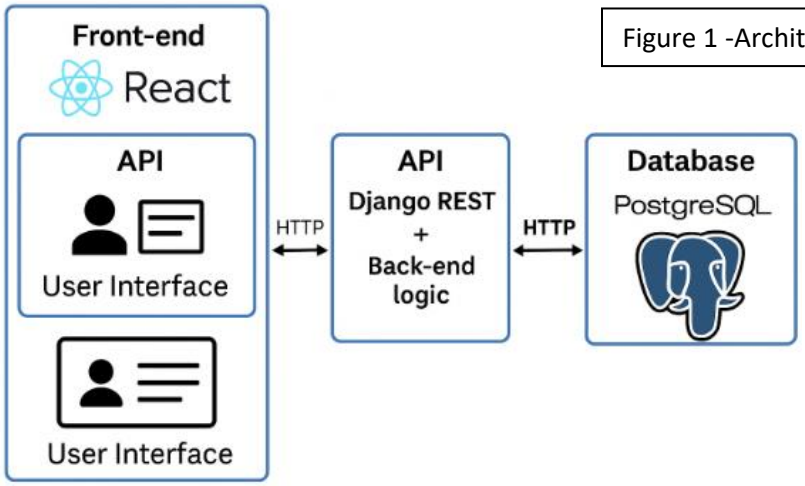


Figure 1 -Architecture

Schéma général de l'architecture du projet Altair Parents

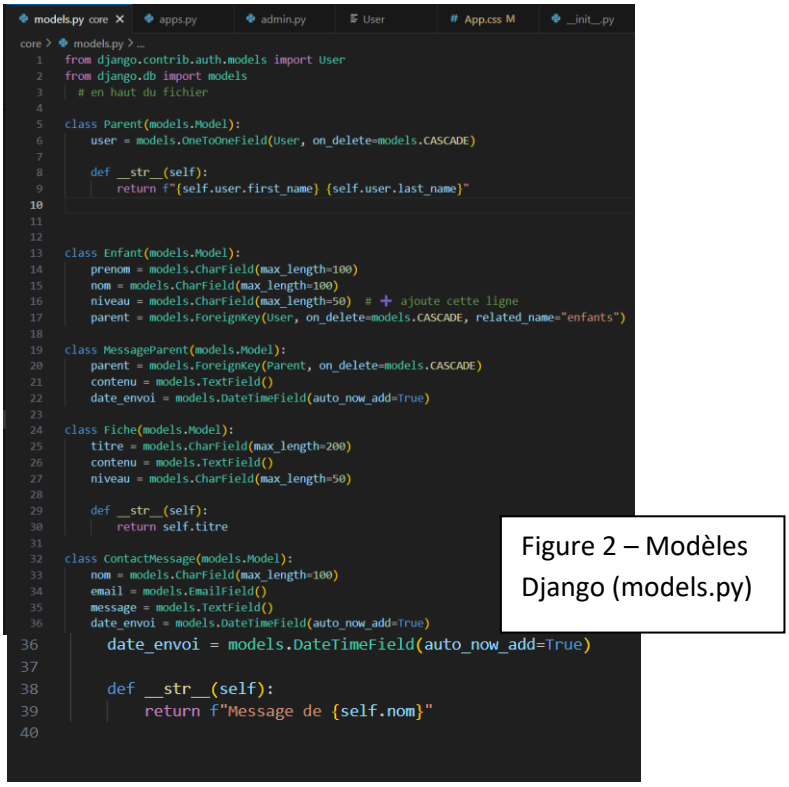


Figure 2 – Modèles Django (models.py)

Bilan personnel

Avant ce stage, j'avais déjà commencé à me former seul au développement web. J'avais réalisé quelques projets personnels, comme un portfolio, ou du machine learning, mais toujours avec des outils comme Firebase, et sans vraiment connaître les bases plus avancées comme PostgreSQL ou Django.

Ce stage m'a permis de franchir une étape importante dans mon apprentissage. J'ai appris énormément de choses concrètes que je n'avais jamais vues en cours au BUT GEII. j'ai appris à construire une application complète, depuis la base de données jusqu'à l'interface, en passant par l'API et la sécurisation des accès.

Même si le projet n'est pas encore terminé à 100 %, je suis très satisfait de mon avancée. Ce stage m'a permis de progresser techniquement, de mieux organiser mon

travail, et surtout de travailler comme dans un vrai projet, avec des tests, des erreurs, et des améliorations continues.

J'ai aussi confirmé mon intérêt pour le développement, et je me sens plus motivé pour continuer dans cette voie. À l'avenir, j'aimerais peut-être me spécialiser dans un domaine comme la cybersécurité ou le réseaux.

En résumé, ce stage a été une des expériences les plus utiles et concrètes de ma formation, car il m'a permis de confronter ce que j'avais appris seul et à l'IUT à un vrai projet, et de progresser fortement dans le domaine de l'informatique.

Conclusion

Ce stage a été une étape très importante dans ma formation, car il m'a permis de passer de la théorie à la pratique, avec un vrai projet à construire, tester et améliorer. J'ai pu mettre en application ce que j'ai appris à l'IUT, mais surtout découvrir et apprendre par moi-même beaucoup de choses que je n'avais encore jamais vues.

Avant le stage, je connaissais un peu le développement web à travers des projets personnels simples, mais je n'avais encore jamais travaillé avec des outils comme Django ou PostgreSQL, ni même mis en place une architecture complète avec API, base de données, et interface utilisateur. Le fait de devoir tout construire moi-même, de comprendre comment relier chaque partie, et de m'adapter à de nouveaux outils, a vraiment été une grande avancée pour moi.

Même si certaines parties sont encore en cours de finalisation, la structure principale est bien là : les modèles sont en place, l'API fonctionne, les données sont bien gérées, et l'interface utilisateur est en cours d'intégration. Cela m'a appris à organiser mon travail, tester chaque étape, corriger mes erreurs, et surtout à rester motivé même quand ça devient compliqué.

Ce stage m'a aussi conforté dans mon choix de continuer dans l'informatique. Je suis très intéressé par le développement, mais aussi par des domaines comme les réseaux, les bases de données ou la cybersécurité. Grâce à cette expérience, je me sens plus confiant, plus autonome, et j'ai envie de continuer à apprendre et à progresser.

En résumé, ce stage m'a permis de me dépasser, de travailler sur un projet concret, et de mieux comprendre ce que je voulais faire par la suite. C'est sans aucun doute l'une des expériences les plus utiles et formatrices que j'ai vécues depuis le début de mes études.

Ce rapport retrace une expérience de stage complète et formatrice. Il marque une étape importante dans ma progression personnelle et professionnelle.

Ce stage m'a également permis de mieux me situer professionnellement et de renforcer mon envie de poursuivre dans le développement et les technologies web.

Annexes

*Les annexes suivantes regroupent les **captures d'écran** et **éléments techniques** utilisés pour illustrer le fonctionnement du projet Altair Parents.*

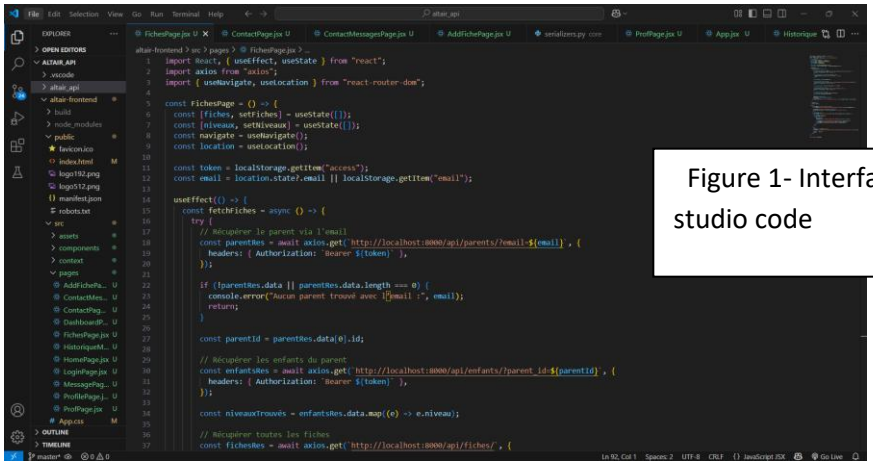


Figure 1- Interface visual studio code

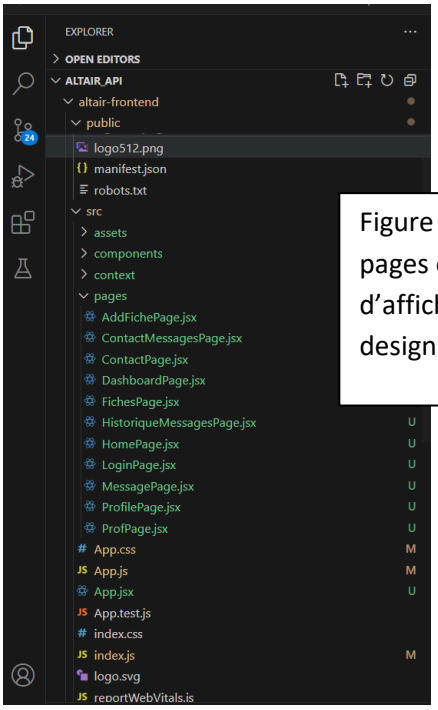
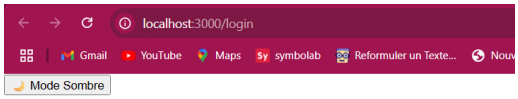


Figure 2- Ce sont les différentes pages contenant les code d'affichage (mise en page, design...)



Connexion

Email Mot de passe Se connecter

Figure 3 - Page de connexion pour le parent d'élève

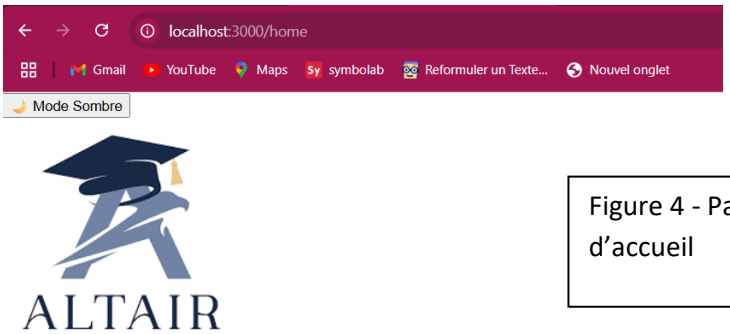


Figure 4 - Page d'accueil

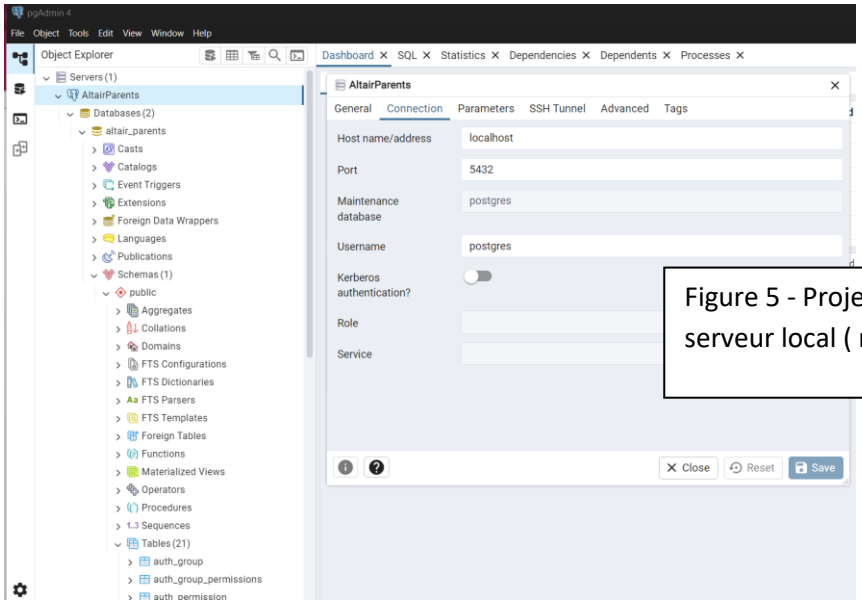


Figure 5 - Projet fonctionnant sur un serveur local (mon PC)

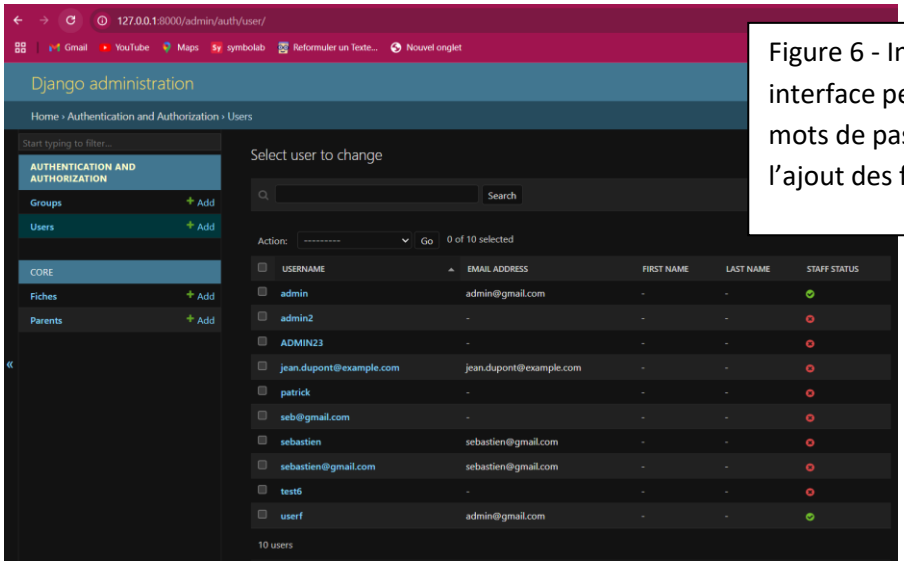


Figure 6 - Interface Django admin, interface permettant de gérer les mots de passes des utilisateurs et l'ajout des fiches et de parents

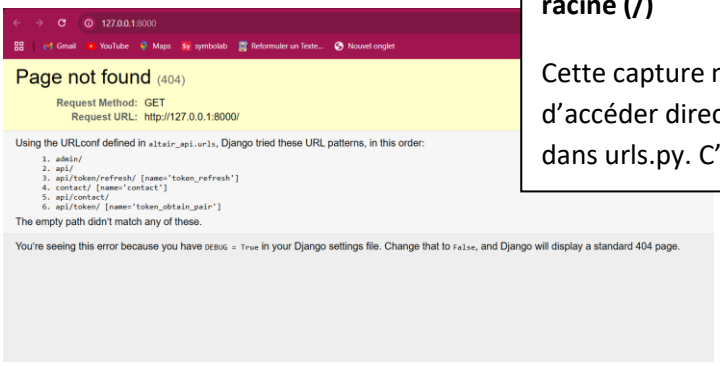


Figure 7 - **Page d'erreur 404 Django indiquant l'absence de route définie à la racine (/)**

Cette capture montre que Django renvoie une erreur 404 lorsque l'on tente d'accéder directement à http://127.0.0.1:8000/, car aucune route ('') n'est définie dans urls.py. C'est une situation normale si la racine n'est pas redirigée ou utilisée

Figure 8 - Résultat d'une requête GET sur /api/parents/ affichant la liste des parents enregistrés via l'API Django REST

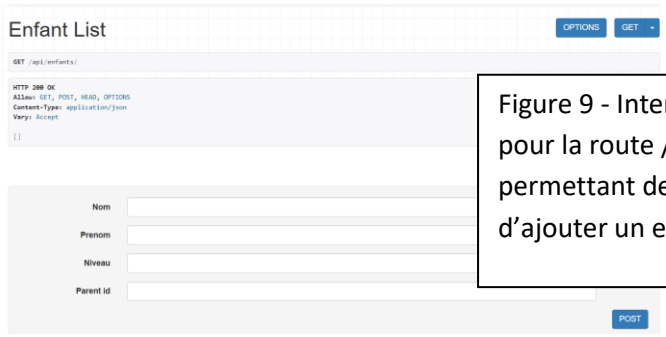


Figure 9 - Interface de l'API Django pour la route /api/enfants/ permettant de consulter ou d'ajouter un enfant

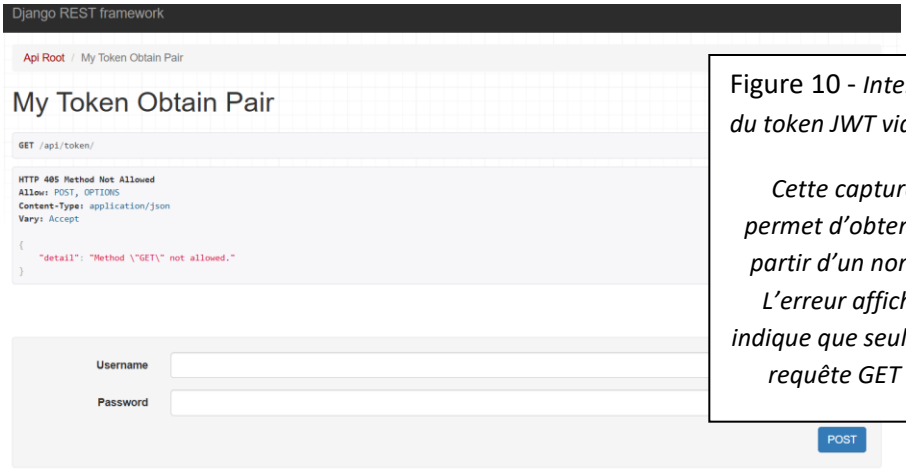


Figure 10 - *Interface de l'API Django pour l'obtention du token JWT via /api/token/*

Cette capture montre la route /api/token/, qui permet d'obtenir un token d'authentification JWT à partir d'un nom d'utilisateur et d'un mot de passe. L'erreur affichée ici (405 – Method Not Allowed) indique que seule la méthode POST est autorisée. Une requête GET n'est pas valable pour cette route.

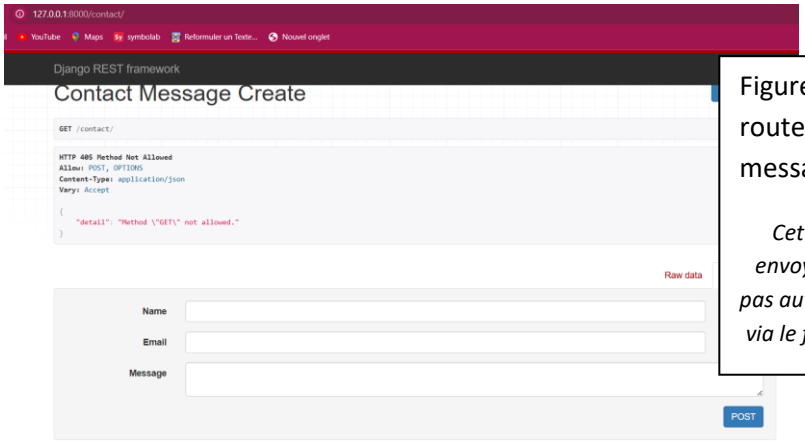


Figure 11 - Interface de l'API Django pour la route /contact/ permettant l'envoi d'un message au support

Cette capture montre la route /contact/, utilisée pour envoyer un message de contact. La méthode GET n'étant pas autorisée, l'erreur 405 s'affiche. Seule la méthode POST, via le formulaire ci-dessous (nom, email, message), permet d'enregistrer une demande.