Rapport d'Analyse de Projet

Système de Tickets Support avec Chat

Aymane Abounay

Python - Django/React Project

25 mai 2025

Remerciements

Je tiens à exprimer ma profonde gratitude envers toutes les personnes qui ont contribué, de près ou de loin, à la réalisation de ce projet de système de gestion de tickets support avec messagerie intégrée.

Tout d'abord, mes sincères remerciements vont à mon encadrante académique, dont l'accompagnement bienveillant, la rigueur méthodologique et les conseils techniques ont été essentiels tout au long de ce travail. Sa vision claire et son expertise ont largement guidé mes choix de conception et de développement.

Je souhaite également remercier l'ensemble de l'équipe pédagogique pour la qualité de l'enseignement dispensé, notamment dans les domaines du développement web, de l'architecture logicielle et de l'ingénierie des systèmes. Les compétences acquises au fil de la formation ont constitué le socle indispensable à la mise en œuvre de cette application.

Ma reconnaissance s'adresse aussi à toutes les personnes avec qui j'ai pu échanger sur les aspects techniques ou fonctionnels du projet, et dont les retours m'ont permis d'affiner certains choix et d'améliorer la pertinence de la solution.

Enfin, je remercie mes camarades pour leur soutien, leurs encouragements et les discussions enrichissantes qui ont accompagné cette phase d'apprentissage et de création.

Ce projet représente pour moi l'aboutissement d'un travail personnel rigoureux et passionnant, mêlant analyse fonctionnelle, développement logiciel et réflexion sur l'expérience utilisateur. J'espère qu'il pourra servir de base utile pour des projets futurs ou des cas d'usage concrets dans le domaine du support technique.

Aymane Abounay
Étudiant en 3ème année - EMSI

Table des matières

1 Présentation Générale **4**
- 1.1 Introduction du chapitre . . . 4
- 1.2 Contexte et Objectifs . . . 4
- 1.3 Parcours Utilisateur type . . . 4
- 1.4 Stack Technique . . . 5
- 1.5 Conclusion du chapitre . . . 5

2 Gestion des Tickets **6**
- 2.1 Introduction du chapitre . . . 6
- 2.2 Flux fonctionnels . . . 6
  - 2.2.1 Création d'un ticket (Flux Frontend/Backend) . . . 6
  - 2.2.2 Consultation de la liste (Flux Frontend/Backend) . . . 6
  - 2.2.3 Affichage des détails (Flux Frontend/Backend) . . . 7
  - 2.2.4 Modification et Suppression (Flux Frontend/Backend) . . . 7
- 2.3 Gestion des statuts . . . 8
- 2.4 Conclusion du chapitre . . . 8

3 Système de Chat **9**
- 3.1 Introduction du chapitre . . . 9
- 3.2 Flux fonctionnels . . . 9
  - 3.2.1 Accès au chat (Flux Frontend/Backend) . . . 9
  - 3.2.2 Échange de messages (Flux Frontend/Backend) . . . 9
- 3.3 Gestion des utilisateurs (Anonymes/Connectés) . . . 10
- 3.4 Conclusion du chapitre . . . 10

4 Authentification et Autorisation **11**
- 4.1 Introduction du chapitre . . . 11
- 4.2 Flux fonctionnels . . . 11
  - 4.2.1 Inscription et Connexion . . . 11
- 4.3 Gestion du token et routes protégées . . . 12
- 4.4 Rôles et Permissions . . . 12
- 4.5 Conclusion du chapitre . . . 13

5 Architecture Technique Détaillée **14**
- 5.1 Introduction du chapitre . . . 14
- 5.2 Composants Frontend majeurs (React) . . . 14
- 5.3 Endpoints API et logique métier (Django REST Framework) . . . 15
- 5.4 Schéma de la base de données . . . 15
- 5.5 Conclusion du chapitre . . . 15

6 Conclusion Générale et Perspectives **17**
6.1 Introduction du chapitre 17
6.2 Bilan fonctionnel 17
6.3 Bilan technique 17
6.4 Évolutions futures envisagées 18
6.5 Conclusion du chapitre 18

Références **19**

A Annexes **20**

Annexes **20**
A.1 Extraits de Code Significatifs 20
A.1.1 Permission Personnalisée IsAdminOrAuthor 20
A.1.2 Gestion de la création de message (Anonyme/Connecté) 20
A.2 Liste des Dépendances 20
A.2.1 Dépendances Backend (Python - Fichier ‘requirements.txt‘ probable) . . . 21
A.2.2 Dépendances Frontend (Node.js - Fichier ‘package.json‘) 21

Chapitre 1

Présentation Générale

1.1 Introduction du chapitre

Ce premier chapitre vise à introduire le projet de système de tickets support. Nous aborderons le contexte dans lequel ce projet s'inscrit, les objectifs visés par sa réalisation, le parcours typique d'un utilisateur au sein de l'application, ainsi que la pile technologique employée pour son développement. Cette présentation initiale permettra de poser les bases nécessaires à la compréhension des chapitres suivants, qui détailleront les aspects fonctionnels et techniques de l'application.

1.2 Contexte et Objectifs

Le projet consiste en une application web conçue pour faciliter la gestion des demandes d'assistance (tickets) au sein d'une organisation ou pour un service client. L'objectif principal est de fournir une plateforme centralisée où les utilisateurs peuvent soumettre leurs problèmes ou questions, suivre leur résolution, et communiquer avec l'équipe de support via un système de chat intégré à chaque ticket.

Les objectifs spécifiques incluent :

— Permettre la création de tickets par des utilisateurs authentifiés ou anonymes.
— Offrir une interface claire pour visualiser la liste des tickets et leur statut (ouvert, fermé, etc.).
— Fournir une vue détaillée pour chaque ticket, incluant sa description et l'historique des échanges.
— Intégrer un système de messagerie instantanée (chat) spécifique à chaque ticket pour une communication fluide entre l'utilisateur et le support.
— Mettre en place un système d'authentification pour les utilisateurs et des rôles distincts (utilisateur standard, administrateur/support) avec des permissions associées.
— Assurer une architecture découplée entre le frontend (React) et le backend (API Django REST Framework) pour une meilleure maintenabilité et évolutivité.

1.3 Parcours Utilisateur type

Un utilisateur typique interagit avec l'application de la manière suivante :

1. **Accueil :** L'utilisateur arrive sur la page d'accueil qui présente l'application et propose des actions (connexion, inscription, voir les tickets).
2. **Inscription/Connexion :** S'il n'a pas de compte, l'utilisateur peut s'inscrire. Sinon, il se connecte avec ses identifiants. Un utilisateur peut aussi interagir de manière limitée sans compte (création de ticket, chat anonyme).

3. **Consultation des tickets :** L'utilisateur accède à la liste des tickets existants. S'il est administrateur, il voit tous les tickets ; sinon, potentiellement seulement les siens (à vérifier selon les permissions finales implémentées).
4. **Création d'un ticket :** L'utilisateur remplit un formulaire avec un titre et une description pour soumettre une nouvelle demande d'assistance. S'il n'est pas connecté, il peut devoir fournir un pseudo.
5. **Consultation d'un ticket :** L'utilisateur clique sur un ticket dans la liste pour en voir les détails (titre, description, statut, auteur, dates).
6. **Interaction via le Chat :** Depuis la page de détail, l'utilisateur peut accéder au chat associé au ticket pour envoyer et recevoir des messages avec l'équipe de support.
7. **Modification/Suppression (si autorisé) :** Selon ses droits (auteur ou admin), l'utilisateur peut modifier les informations d'un ticket ou le supprimer.
8. **Déconnexion :** L'utilisateur termine sa session.

1.4 Stack Technique

Le projet repose sur une architecture moderne et découplée :
— **Frontend :**
  — **Framework :** React (utilisant Vite pour le build et le développement)
  — **Langage :** JavaScript (JSX)
  — **Routage :** React Router DOM
  — **UI Components :** Shadcn/ui et React Bootstrap (utilisation mixte constatée)
  — **Gestion d'état :** Principalement via les états locaux des composants et 'localStorage' pour le token d'authentification.
  — **Appels API :** Fetch API native de JavaScript.
— **Backend :**
  — **Framework :** Django
  — **API :** Django REST Framework (DRF)
  — **Langage :** Python
  — **Base de données :** MySQL (configurée dans 'settings.py')
— **Authentification :** Basée sur les tokens DRF ('$rest_framework.authtoken$')$**GestionCORS$: $**$'$django-cors-headers$'$pourautoriserlesrequtesdepuislefrontend.$
— **Environnement :**
  — **Serveur web (développement) :** Serveur de développement Django, serveur Vite pour React.
  — **Gestion des dépendances :** 'pip' (Python), 'npm' ou 'yarn' (Node.js).

1.5 Conclusion du chapitre

Ce chapitre a posé les fondations du projet en définissant son périmètre, ses objectifs, le flux d'interaction utilisateur attendu et les technologies mises en œuvre. L'architecture choisie, séparant clairement le frontend React du backend Django via une API REST, est une approche robuste pour ce type d'application web. Les chapitres suivants exploreront plus en détail les fonctionnalités spécifiques et l'implémentation technique de chaque partie du système.

Chapitre 2

Gestion des Tickets

2.1 Introduction du chapitre

Ce chapitre se concentre sur la fonctionnalité centrale de l'application : la gestion des tickets d'assistance. Nous allons détailler les différents flux fonctionnels liés aux tickets, depuis leur création jusqu'à leur consultation, modification et suppression. Nous examinerons également comment le statut des tickets est géré et représenté, en analysant les interactions entre l'interface utilisateur React et l'API Django sous-jacente.

2.2 Flux fonctionnels

La gestion des tickets implique plusieurs opérations CRUD (Create, Read, Update, Delete) orchestrées entre le frontend et le backend.

2.2.1 Création d'un ticket (Flux Frontend/Backend)

Le processus de création d'un nouveau ticket est initié depuis le frontend, typiquement via un bouton "Nouveau ticket" (présent sur la page d'accueil ou la liste des tickets).

Frontend (‘TicketCreate.jsx‘) :

— Un formulaire permet à l'utilisateur de saisir un titre et une description.
— Si l'utilisateur n'est pas authentifié (pas de token dans ‘localStorage‘), un champ supplémentaire lui demande un pseudo.
— À la soumission, une requête POST est envoyée à l'endpoint ‘/api/tickets/‘.
— Le payload contient ‘title‘, ‘description‘, et ‘pseudo‘ (si l'utilisateur est anonyme).
— Si l'utilisateur est authentifié, le token est inclus dans l'en-tête ‘Authorization‘.

Backend (‘api/views.py :TicketViewSet‘, ‘api/serializers.py :TicketSerializer‘) :

— L'endpoint ‘/api/tickets/‘ (méthode POST) est géré par ‘TicketViewSet‘.
— La permission ‘AllowAny‘ est appliquée pour la création, permettant aux utilisateurs anonymes et authentifiés de créer des tickets.
— La méthode ‘perform$_c$reate‘du‘$TicketViewSet$‘$estsurcharge$: $sil'utilisateurestauthentifi$(‘$request.use$ $self.request.user$)‘).$Sinon, leticketestsauvegardsansauteurassoci$(‘$serializer.save$()‘), $etlepseudofour$
— Une réponse HTTP 201 Created est retournée en cas de succès.

2.2.2 Consultation de la liste (Flux Frontend/Backend)

L'affichage de la liste des tickets est géré par le composant ‘TicketsList.jsx‘.

Frontend (‘TicketsList.jsx‘) :

— Au chargement du composant, une requête GET est envoyée à l'endpoint ‘/api/tickets/‘.
— Les données reçues (une liste de tickets) sont stockées dans l'état local (‘useState‘).

— Les tickets sont affichés dans un tableau (‘@/components/ui/table‘), montrant l’ID, le titre, le statut (avec un badge de couleur variable) et la date de création.
— Des boutons d’action permettent de voir les détails ou de supprimer (si l’utilisateur est admin).
— Le composant récupère également les informations de l’utilisateur connecté (‘/api/profile/‘) pour conditionner l’affichage du bouton de suppression.

Backend (‘api/views.py :TicketViewSet‘) :
— L’endpoint ‘/api/tickets/‘ (méthode GET) est géré par ‘TicketViewSet‘.
— La permission par défaut est ‘AllowAny‘, donc tout le monde peut lister les tickets.
— Le ‘queryset = Ticket.objects.all()‘ récupère tous les tickets.
— Le ‘TicketSerializer‘ est utilisé pour sérialiser la liste des tickets avant de la renvoyer au frontend.

2.2.3 Affichage des détails (Flux Frontend/Backend)

La vue détaillée d’un ticket est gérée par ‘TicketDetails.jsx‘.

Frontend (‘TicketDetails.jsx‘) :
— Le composant récupère l’ID du ticket depuis l’URL (‘useParams‘).
— Une requête GET est envoyée à l’endpoint ‘/api/tickets/id/‘.
— Les détails du ticket reçu sont stockés dans l’état et affichés dans des cartes (‘@/components/ui/card‘) : titre, description, statut, auteur, dates.
— Des composants ‘lucide-react‘ sont utilisés pour les icônes.
— Des boutons d’action sont présents : "Rejoindre le chat", "Modifier le ticket" (si auteur ou admin), "Supprimer le ticket" (si admin).
— Les informations de l’utilisateur connecté (‘/api/profile/‘) sont récupérées pour conditionner l’affichage des boutons d’action.

Backend (‘api/views.py :TicketViewSet‘) :
— L’endpoint ‘/api/tickets/id/‘ (méthode GET) est géré par ‘TicketViewSet‘.
— La permission ‘AllowAny‘ s’applique.
— Le ‘TicketViewSet‘ récupère l’instance spécifique du ‘Ticket‘ via son ID.
— Le ‘TicketSerializer‘ sérialise l’objet ticket pour la réponse.

2.2.4 Modification et Suppression (Flux Frontend/Backend)

La modification est gérée par ‘TicketEdit.jsx‘ et la suppression peut être initiée depuis ‘TicketsList.jsx‘ ou ‘TicketDetails.jsx‘.

Modification (Frontend : ‘TicketEdit.jsx‘, Backend : ‘TicketViewSet‘) :
— Le formulaire de ‘TicketEdit.jsx‘ est pré-rempli avec les données du ticket récupérées via GET ‘/api/tickets/id/‘.
— L’utilisateur modifie le titre, la description ou le statut.
— À la soumission, une requête PUT est envoyée à ‘/api/tickets/id/‘ avec les nouvelles données.
— Le backend (‘TicketViewSet‘, action ‘update‘) vérifie les permissions : l’utilisateur doit être authentifié ET être l’auteur du ticket OU être administrateur (‘IsAdminOrAuthor‘).
— Si autorisé, le ‘TicketSerializer‘ met à jour l’instance ‘Ticket‘.

Suppression (Frontend : ‘TicketsList.jsx‘/‘TicketDetails.jsx‘, Backend : ‘TicketViewSet‘) :
— Un bouton "Supprimer" est affiché (conditionnellement pour les admins).
— Une confirmation est demandée à l’utilisateur (‘window.confirm‘).
— Une requête DELETE est envoyée à ‘/api/tickets/id/‘.
— Le backend (‘TicketViewSet‘, action ‘destroy‘) vérifie la permission : l’utilisateur doit être administrateur (‘IsAdminUser‘).

— Si autorisé, l'instance ‘Ticket‘ est supprimée de la base de données.
— Le frontend met à jour l'interface (supprime le ticket de la liste ou redirige l'utilisateur).

2.3 Gestion des statuts

Le statut d'un ticket est un élément clé pour suivre sa progression.
Modèle (‘Tickets/models.py‘) :
— Le modèle ‘Ticket‘ possède un champ ‘status‘ :

```
status = models.CharField(max_length=20, choices=[('open', 'Open'), ('
    closed', 'Closed')], default='open')

```

— Actuellement, seuls les statuts 'open' et 'closed' sont définis, avec 'open' par défaut.
Frontend (‘TicketsList.jsx‘, ‘TicketDetails.jsx‘, ‘TicketEdit.jsx‘) :
— Le statut est affiché en utilisant des badges (‘@/components/ui/badge‘).
— La couleur ou le style du badge varie en fonction de la valeur du statut (par exemple, vert pour 'open', rouge pour 'closed',
— Le composant ‘TicketEdit.jsx‘ permet de modifier le statut via un ‘Select‘ (‘@/components/ui/select‘), proposant les options 'open' et 'closed'.
Backend (‘api/serializers.py‘, ‘api/views.py‘) :
— Le ‘TicketSerializer‘ inclut le champ ‘status‘.
— Lors de la modification (PUT sur ‘/api/tickets/id/‘), la nouvelle valeur du statut fournie par le frontend est enregistrée.
— Il y a une divergence potentielle entre les statuts gérés/affichés par le frontend ('open', 'closed') et ceux définis dans les ‘choices‘ du modèle Django ('open', 'closed').

2.4 Conclusion du chapitre

La gestion des tickets est implémentée de manière fonctionnelle, couvrant les opérations CRUD essentielles et la gestion des statuts. L'interaction entre le frontend React et l'API Django est claire pour ces fonctionnalités. Cependant, une incohérence a été relevée concernant les statuts possibles, ce qui mériterait d'être harmonisé. Le système de permissions semble également bien défini pour contrôler les actions de modification et de suppression. Le chapitre suivant abordera le système de chat intégré aux tickets.

Chapitre 3

Système de Chat

3.1 Introduction du chapitre

Ce chapitre examine le système de messagerie instantanée (chat) intégré à chaque ticket. Cette fonctionnalité permet une communication directe et contextuelle entre l'utilisateur ayant soumis le ticket et l'équipe de support. Nous analyserons le flux d'accès au chat, le processus d'échange de messages, et la manière dont les utilisateurs, qu'ils soient anonymes ou authentifiés, sont gérés dans cet environnement de communication.

3.2 Flux fonctionnels

Le système de chat est accessible depuis la page de détails d'un ticket spécifique.

3.2.1 Accès au chat (Flux Frontend/Backend)

L'accès à l'interface de chat se fait via un bouton sur la page de détails du ticket.
Frontend (‘TicketDetails.jsx‘, ‘TicketChat.jsx‘) :
— Sur la page ‘TicketDetails.jsx‘, un bouton "Rejoindre le chat" redirige l'utilisateur vers l'URL ‘/tickets/id/chat‘.
— Le composant ‘TicketChat.jsx‘ est responsable de l'affichage de l'interface de chat.
— Au chargement, ‘TicketChat.jsx‘ récupère l'ID du ticket depuis l'URL (‘useParams‘).
— Il tente de récupérer les informations de l'utilisateur connecté via GET ‘/api/profile/‘ si un token est présent.
— Il effectue une requête GET initiale vers ‘/api/tickets/id/messages/‘ pour charger l'historique des messages du ticket.
— Un mécanisme de rafraîchissement périodique (‘setInterval‘) est mis en place pour récupérer les nouveaux messages toutes les 2 secondes en rappelant GET ‘/api/tickets/id/-messages/‘.
Backend (‘api/views.py :TicketMessageListCreateAPIView‘, ‘api/urls.py‘) :
— L'endpoint GET ‘/api/tickets/ticket$_id$/$messages$/‘$estgrparlavue$‘$TicketMessageListCreateAPIView$‘
— La vue récupère le ticket correspondant à ‘ticket$_id$‘(‘$get_object_or_404(Ticket, id = ticket_id)$‘).$Ellefiltreens$
— Le ‘MessageSerializer‘ est utilisé pour sérialiser la liste des messages avant de la renvoyer au frontend.

3.2.2 Échange de messages (Flux Frontend/Backend)

L'envoi de nouveaux messages est géré par un formulaire dans ‘TicketChat.jsx‘.
Frontend (‘TicketChat.jsx‘) :
— Un champ de saisie (‘Input‘ de shadcn/ui) permet à l'utilisateur d'écrire son message.

— Si l'utilisateur n'est pas authentifié, un champ supplémentaire lui demande de saisir un pseudo.
— À la soumission du formulaire (bouton "Envoyer"), une requête POST est envoyée à l'endpoint '/api/tickets/id/messages/'.
— Le payload contient le 'message' et, si l'utilisateur est anonyme, le 'pseudo'.
— Si l'utilisateur est authentifié, le token est inclus dans l'en-tête 'Authorization'.
— Après un envoi réussi, le champ de message est vidé, et la liste des messages est rafraîchie (implicitement par le 'setInterval' ou par un appel explicite à 'fetchMessages').
— Les messages sont affichés dans une zone scrollable ('ScrollArea' de shadcn/ui), différenciant visuellement les messages de l'utilisateur courant (si connecté) des autres messages.
— Le nom de l'expéditeur ('sender') est affiché pour chaque message.

Backend ('api/views.py :TicketMessageListCreateAPIView', 'api/serializers.py :MessageSerializer', 'Tickets/models.py :Message') :

— L'endpoint POST '/api/tickets/ticket$_id$/$messages$/'$estgalementgrpar$'$TicketMessageListCreateAPIV
— La vue récupère le ticket associé.
— Elle extrait le 'contenu' du message et le 'pseudo' (si fourni) depuis 'request.data'.
— Elle crée une nouvelle instance du modèle 'Message' :
  — 'ticket' : L'instance du ticket courant.
  — 'contenu' : Le texte du message.
  — 'auteur' : 'request.user' si l'utilisateur est authentifié, sinon 'None'.
  — 'pseudo' : La valeur du pseudo fournie si l'utilisateur n'est pas authentifié, sinon 'None'.
— Le 'MessageSerializer' est utilisé pour sérialiser le message nouvellement créé.
— Une réponse HTTP 201 Created est retournée avec les données du message créé.
— Le modèle 'Message' possède une méthode 'get$_d$$isplay_n$$ame$()'$quiretourne$'$auteur.username$'$sil'auteurexiste$($ut

3.3 Gestion des utilisateurs (Anonymes/Connectés)

Le système de chat gère différemment les utilisateurs selon leur statut d'authentification :
— **Utilisateurs Authentifiés :**
  — Leur nom d'utilisateur ('user.username') est automatiquement associé au message envoyé (champ 'auteur' du modèle 'Message').
  — Ils n'ont pas besoin de fournir de pseudo dans l'interface de chat.
  — Leurs messages peuvent être affichés différemment dans le frontend (par exemple, alignés à droite).
— **Utilisateurs Anonymes :**
  — Ils doivent fournir un pseudo lors de l'envoi d'un message.
  — Ce pseudo est stocké dans le champ 'pseudo' du modèle 'Message' (le champ 'auteur' reste 'None').
  — Le frontend affiche une alerte indiquant qu'ils sont en mode invité.

Cette distinction permet à la fois aux utilisateurs enregistrés et aux visiteurs non identifiés de participer à la conversation d'un ticket, tout en conservant une traçabilité de l'expéditeur via le nom d'utilisateur ou le pseudo fourni.

3.4 Conclusion du chapitre

Le système de chat intégré aux tickets offre une fonctionnalité de communication essentielle et semble correctement implémenté. Il gère à la fois la récupération de l'historique et l'envoi de nouveaux messages, en distinguant les utilisateurs authentifiés des utilisateurs anonymes. L'utilisation d'un rafraîchissement périodique ('setInterval') pour récupérer les nouveaux messages. Pour un système de support ticket standard, cette approche est fonctionnelle. Le chapitre suivant abordera l'authentification et la gestion des autorisations.

Chapitre 4

Authentification et Autorisation

4.1 Introduction du chapitre

Ce chapitre aborde les mécanismes d'authentification et d'autorisation mis en place dans l'application. L'authentification vérifie l'identité des utilisateurs, tandis que l'autorisation détermine les actions qu'ils sont autorisés à effectuer. Nous examinerons les processus d'inscription et de connexion, la gestion des sessions via des tokens, la protection des routes nécessitant une authentification, ainsi que la distinction des rôles (utilisateurs standards vs administrateurs) et les permissions associées.

4.2 Flux fonctionnels

4.2.1 Inscription et Connexion

L'application permet aux utilisateurs de créer un compte et de se connecter.

Inscription (Frontend : 'Register.jsx', Backend : 'api/views.py :RegisterAPIView') :

- Le composant 'Register.jsx' fournit un formulaire demandant un nom d'utilisateur, une adresse e-mail et un mot de passe (avec confirmation).
- À la soumission, une requête POST est envoyée à l'endpoint '/api/register/' avec ces informations.
- Le backend ('RegisterAPIView') reçoit la requête.
- Il vérifie si les champs requis sont présents et si le nom d'utilisateur n'existe pas déjà.
- Si les vérifications passent, un nouvel utilisateur est créé en utilisant 'User.objects.create$_u ser()$', $quis'occu$
- Le frontend affiche un message de succès et redirige l'utilisateur vers la page de connexion après un court délai.

Connexion (Frontend : 'Login.jsx', Backend : 'rest$framework.authtoken.views.obtain_a uth_t oken$') :

Le composant 'Login.jsx' présente un formulaire demandant le nom d'utilisateur et le mot de passe.

À la soumission, une requête POST est envoyée à l'endpoint '/api/api-token-auth/'.

Cet endpoint est géré par la vue 'obtain$_a uth_t oken' fournie par Django REST Framework. La vue vérif$

Si les identifiants sont valides, elle génère (ou récupère) un token d'authentification associé à l'utilisateur et le retourne dans la réponse JSON (clé 'token').

Si les identifiants sont invalides, une erreur est retournée.

Le frontend ('Login.jsx') reçoit la réponse. Si un token est présent, il le stocke dans 'localStorage' ('localStorage.setItem('token', data.token)'), met à jour l'état global de l'application ('setToken(data.token)'), affiche un message de succès et redirige l'utilisateur vers la page d'accueil.

4.3 Gestion du token et routes protégées

L'authentification des requêtes suivantes et la protection des routes sont basées sur le token.
Stockage et Utilisation du Token (Frontend) :

— Le token obtenu lors de la connexion est stocké dans le ‘localStorage‘ du navigateur.
— Pour chaque requête nécessitant une authentification, le frontend récupère le token depuis ‘localStorage‘ et l'inclut dans l'en-tête ‘Authorization‘ de la requête HTTP, sous la forme.
— La présence du token dans ‘localStorage‘ est également utilisée pour déterminer l'état d'authentification de l'utilisateur dans l'interface.
— La déconnexion (‘handleLogout‘ dans ‘App.jsx‘) consiste simplement à supprimer le token de ‘localStorage‘ et à réinitialiser l'état associé.

Vérification du Token (Backend) :

— La configuration de Django REST Framework (‘settings.py‘) spécifie ‘TokenAuthentication‘ comme méthode d'authentification par défaut :

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.TokenAuthentication',
    ],
}

```

— Pour chaque requête entrante, le middleware d'authentification de DRF tente d'extraire le token de l'en-tête ‘Authorization‘.
— S'il trouve un token valide, il identifie l'utilisateur correspondant et l'attache à l'objet ‘request‘ (‘request.user‘).
— Si aucun token n'est fourni ou si le token est invalide, ‘request.user‘ est défini comme une instance de ‘AnonymousUser‘.

Routes Protégées (Frontend : ‘PrivateRoute.jsx‘, Backend : Permissions DRF) :

— **Frontend :** Le composant ‘PrivateRoute.jsx‘ est utilisé pour envelopper les routes qui ne doivent être accessibles qu'aux utilisateurs connectés (par exemple, ‘/tickets/create‘, ‘/tickets/edit/ :id‘). Il vérifie simplement la présence du token dans ‘localStorage‘. Si le token existe, il rend le composant enfant (la page protégée) ; sinon, il redirige l'utilisateur vers la page de connexion (‘<Navigate to="/login" />‘).
— **Backend :** La protection au niveau du backend est assurée par les classes de permission de DRF. Par exemple :
— La vue ‘UserProfileAPIView‘ utilise ‘permission$_c lasses = [IsAuthenticated]‘, $cequisignifiequeseulunuti
‘$IsAuthenticated‘et‘$IsAdminOrAuthor‘$pourlamodification, ‘$IsAdminUser‘$pourlasuppression.Cespe
— Cette double protection (frontend et backend) est une bonne pratique : le frontend offre une meilleure expérience utilisateur en empêchant l'accès visuel aux pages non autorisées, tandis que le backend garantit la sécurité même si le frontend est contourné.

4.4 Rôles et Permissions

L'application distingue principalement deux rôles :

— **Utilisateur Standard :** Peut créer des tickets, voir ses tickets (potentiellement tous, selon l'implémentation finale), voir les détails, participer au chat, modifier ses propres tickets.
— **Administrateur (Admin/Support) :** Identifié par l'attribut ‘is$_s taff = True‘dumodle‘User‘deDjango.Adesd
$peutvoirtouslestickets, modifiern'importequelticket, supprimern'importequelticket, participerauchat(probab

Les permissions sont implémentées au niveau du backend à l'aide des classes de permissions de DRF :

— ‘permissions.AllowAny‘ : Autorise tout accès (utilisé pour la liste/détail des tickets, la création de tickets, la lecture/écriture des messages de chat).

— ‘permissions.IsAuthenticated‘ : Autorise uniquement les utilisateurs connectés (utilisé pour voir le profil utilisateur).
— ‘permissions.IsAdminUser‘ : Autorise uniquement les administrateurs (‘user.is$_staff$ = $True$‘)($utilispourlasuppressiondetickets$).‘$IsAdminOrAuthor$‘($classepersonnalise$) : $Autorisel'admin$

4.5 Conclusion du chapitre

Le système d’authentification et d’autorisation repose sur des mécanismes standards et éprouvés : authentification par token DRF, stockage du token côté client dans ‘localStorage‘, et utilisation des classes de permissions DRF pour contrôler l’accès aux ressources de l’API. La distinction entre utilisateurs standards et administrateurs est claire et appliquée de manière cohérente dans le backend et le frontend. La protection des routes est assurée à la fois côté client (redirection) et côté serveur (vérification des permissions), ce qui constitue une approche robuste. Les flux d’inscription et de connexion sont fonctionnels.

Chapitre 5

Architecture Technique Détaillée

5.1 Introduction du chapitre

Ce chapitre plonge au cœur de l'implémentation technique du système de tickets support. Nous allons examiner de plus près les composants majeurs de l'interface utilisateur développée avec React, les endpoints clés de l'API RESTful construite avec Django REST Framework, ainsi que la structure de la base de données sous-jacente qui persiste les informations. L'objectif est de fournir une compréhension approfondie de la manière dont les différentes parties du système interagissent et fonctionnent ensemble.

5.2 Composants Frontend majeurs (React)

L'interface utilisateur est construite comme une Single Page Application (SPA) utilisant React et Vite. Voici les rôles des principaux composants identifiés :

- ‘App.jsx‘ : Le composant racine qui configure le routeur (‘BrowserRouter‘ de ‘react-router-dom‘) et définit les routes principales de l'application. Il gère également l'état global lié à l'authentification (token, informations utilisateur) et la fonction de déconnexion.
- ‘Header.jsx‘ : Affiche la barre de navigation supérieure, incluant le nom de l'application, les liens de navigation principaux (Accueil, Tickets, Nouveau), et les options de connexion/inscripti ou le nom de l'utilisateur connecté et le bouton de déconnexion. Utilise ‘react-bootstrap‘ pour la structure de la barre de navigation.
- ‘Footer.jsx‘ : Affiche le pied de page simple en bas de chaque page.
- ‘Home.jsx‘ : La page d'accueil, présentant l'application et offrant des appels à l'action (connexion, inscription, voir/créer tickets). Utilise ‘react-bootstrap‘ pour la mise en page.
- ‘Login.jsx‘ / ‘Register.jsx‘ : Composants responsables des formulaires et de la logique de connexion et d'inscription, interagissant avec les endpoints d'authentification et d'enregistremen de l'API. Utilisent les composants ‘Card‘, ‘Input‘, ‘Label‘, ‘Button‘ de ‘@/components/ui‘ (Shadcn).
- ‘PrivateRoute.jsx‘ : Un composant d'ordre supérieur qui protège certaines routes. Il vérifie la présence du token d'authentification dans ‘localStorage‘ et redirige vers ‘/login‘ si l'utilisateur n'est pas connecté.
- ‘TicketsList.jsx‘ : Affiche la liste des tickets récupérés depuis l'API. Utilise la ‘Table‘ de ‘@/components/ui‘ pour présenter les données et des ‘Badge‘ pour les statuts. Gère également la suppression de tickets (pour les admins).
- ‘TicketDetails.jsx‘ : Affiche les informations détaillées d'un ticket spécifique. Utilise des ‘Card‘, ‘Badge‘, ‘Button‘ de ‘@/components/ui‘ et des icônes ‘lucide-react‘. Permet la navigation vers le chat, l'édition et la suppression (conditionnelle).
- ‘TicketCreate.jsx‘ : Fournit le formulaire pour créer un nouveau ticket. Utilise ‘react-bootstrap‘ pour le formulaire et la carte.
- ‘TicketEdit.jsx‘ : Fournit le formulaire pour modifier un ticket existant. Utilise largement les composants de formulaire de ‘@/components/ui‘ (‘Form‘, ‘Input‘, ‘Textarea‘, ‘Select‘).
- ‘TicketChat.jsx‘ : Gère l'interface de chat pour un ticket donné. Récupère et affiche les messages, permet d'envoyer de nouveaux messages. Utilise ‘Input‘, ‘Button‘, ‘Card‘, ‘ScrollArea‘, ‘Badge‘ de ‘@/components/ui‘. Implémente un polling pour rafraîchir les messages.

L'utilisation mixte de ‘react-bootstrap‘ (dans ‘Home‘, ‘TicketCreate‘, ‘Header‘) et de ‘shadcn/ui‘ (dans ‘Login‘, ‘Register‘, ‘TicketsList‘, ‘TicketDetails‘, ‘TicketEdit‘, ‘TicketChat‘) est notable et pourrait être source d'incohérence visuelle ou de complexité de maintenance à long terme. Une standardisation

sur une seule librairie UI serait préférable.

5.3 Endpoints API et logique métier (Django REST Framework)

Le backend expose une API RESTful pour permettre au frontend d'interagir avec les données.
Configuration (‘settings.py‘, ‘backend/urls.py‘):

- ‘settings.py‘ déclare les applications installées (‘Tickets‘, ‘api‘, ‘rest$_f$ramework‘,‘rest$_f$ramework.authtoken‘,‘
3000ou5174).Le fichier‘backend/urls.py‘(non fourni, maisdduit)inclutlesURLsdel′application‘api‘souslepr fixe‘/api/‘.

Endpoints (‘api/urls.py‘, ‘api/views.py‘):

- ‘/api/tickets/‘ (Router DRF pour ‘TicketViewSet‘):
  - ‘GET‘: Lister tous les tickets (‘AllowAny‘).
  - ‘POST‘: Créer un nouveau ticket (‘AllowAny‘, assigne l'auteur si connecté).
- ‘/api/tickets/id/‘ (Router DRF pour ‘TicketViewSet‘):
  - ‘GET‘: Récupérer les détails d'un ticket (‘AllowAny‘).
  - ‘PUT‘/‘PATCH‘: Mettre à jour un ticket (‘IsAuthenticated‘ ET ‘IsAdminOrAuthor‘).
  - ‘DELETE‘: Supprimer un ticket (‘IsAdminUser‘).
- ‘/api/tickets/ticket$_i$d/messages/‘(‘TicketMessageListCreateAPIView‘) :
  - ‘GET‘: Lister les messages d'un ticket (‘AllowAny‘).
  - ‘POST‘: Ajouter un message à un ticket (‘AllowAny‘, gère utilisateurs anonymes/connectés).
- ‘/api/api-token-auth/‘ (‘obtain$_a$uth$_t$oken‘) :
  - ‘POST‘: Obtenir un token d'authentification en échange d'un nom d'utilisateur et mot de passe.

‘/api/register/‘ (‘RegisterAPIView‘) :
— ‘POST‘ : Créer un nouveau compte utilisateur (‘AllowAny‘).

‘/api/profile/‘ (‘UserProfileAPIView‘) :
— ‘GET‘ : Récupérer les informations de l'utilisateur connecté (‘IsAuthenticated‘).
La logique métier est principalement encapsulée dans les vues DRF (‘TicketViewSet‘, ‘TicketMessageListCreateAPIView‘, etc.) et les serializers (‘TicketSerializer‘, ‘MessageSerializer‘) qui gèrent la validation, la création, la mise à jour et la représentation des données des modèles ‘Ticket‘ et ‘Message‘.

5.4 Schéma de la base de données

La persistance des données est assurée par une base de données MySQL, dont la structure est définie par les modèles Django.

Modèles (‘Tickets/models.py‘) :
— **‘User‘ (modèle intégré de Django) :** Utilisé pour l'authentification et pour lier les tickets et messages à un auteur enregistré. Les champs clés sont ‘username‘, ‘email‘, ‘password‘, ‘is$_s$taff‘.**‘Ticket‘ :***Reprsenteunedemanded′assistance.Contient‘title‘,‘description‘,‘status‘(open/closed)*
— **‘Message‘ :** Représente un message dans le chat d'un ticket. Contient ‘contenu‘, ‘date$_e$nvoi‘.*Possdeunecltrangre*
−plusieurs), *unecltrangre‘auteur‘vers‘User‘*(*nullable*), *etunchamp‘pseudo‘*(*nullable*)*pourlesmessagesanonyn*

Diagramme de Classe : Le diagramme suivant illustre les relations entre ces modèles.
Ce diagramme montre que :
— Un ‘User‘ peut être l'auteur de zéro ou plusieurs ‘Ticket‘s.
— Un ‘User‘ peut être l'auteur de zéro ou plusieurs ‘Message‘s.
— Un ‘Ticket‘ contient zéro ou plusieurs ‘Message‘s.
— Les relations ‘auteur‘ dans ‘Ticket‘ et ‘Message‘ sont optionnelles (nullable), permettant la gestion des contributions anonymes (bien que pour ‘Ticket‘, le pseudo ne soit pas stocké directement sur le modèle ‘Ticket‘).

5.5 Conclusion du chapitre

L'architecture technique détaillée révèle une application structurée de manière classique pour un projet Django/React. Le découplage frontend/backend via une API REST est bien implé-

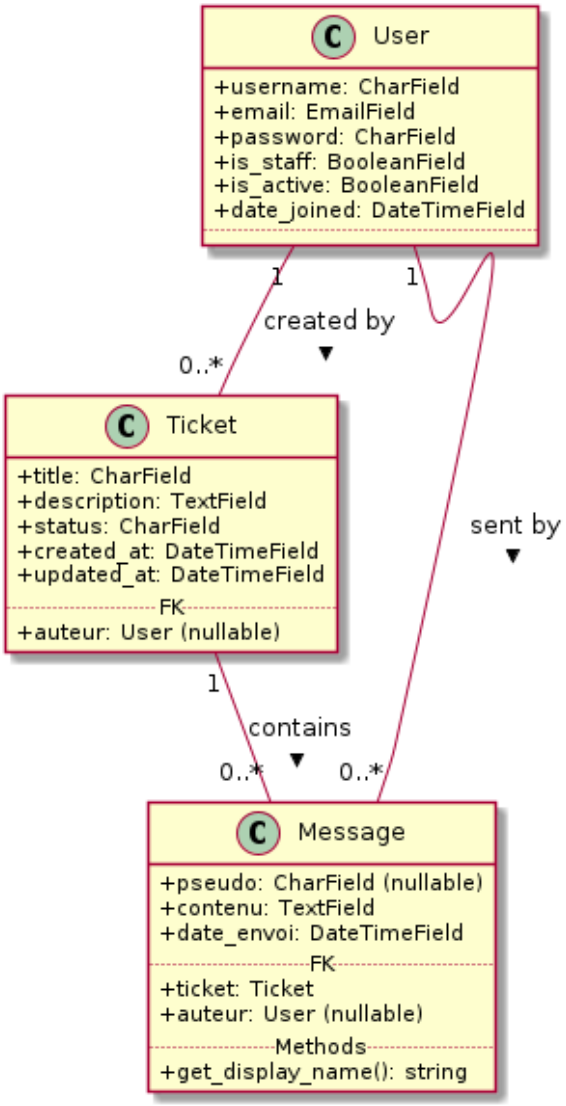

FIGURE 5.1 – Diagramme de classe des modèles principaux.

menté. Les composants React sont organisés par fonctionnalité, bien que l'utilisation mixte de librairies UI puisse être améliorée. L'API Django utilise efficacement DRF pour exposer les données et gérer la logique métier et les permissions. La structure de la base de données est simple et adaptée aux besoins fonctionnels décrits. Les points d'attention techniques incluent l'harmonisation des librairies UI frontend et l'optimisation potentielle du chat (polling vs WebSockets).

Chapitre 6

Conclusion Générale et Perspectives

6.1 Introduction du chapitre

Ce dernier chapitre propose une synthèse du projet de système de tickets support. Nous effectuerons un bilan fonctionnel et technique basé sur l'analyse des composants frontend et backend. Ensuite, nous identifierons quelques pistes d'amélioration et d'évolutions futures possibles pour enrichir l'application et optimiser son fonctionnement. Enfin, nous conclurons ce rapport d'analyse.

6.2 Bilan fonctionnel

L'application remplit les objectifs principaux définis initialement : elle permet la création, la consultation, la modification (conditionnelle) et la suppression (administrateur) de tickets d'assistance. Le système de chat intégré offre un canal de communication direct pour chaque ticket, accessible aux utilisateurs authentifiés comme anonymes (via pseudo). Le système d'authentification et la gestion des rôles (utilisateur standard vs administrateur) sont fonctionnels et assurent un contrôle d'accès adéquat aux différentes fonctionnalités.

Les parcours utilisateurs clés (inscription, connexion, gestion des tickets, chat) sont couverts par les composants React et les endpoints API Django. L'interface utilisateur, bien qu'utilisant deux librairies de composants distinctes (Shadcn/ui et React Bootstrap), fournit les éléments nécessaires à l'interaction.

Quelques points fonctionnels pourraient être précisés ou améliorés :

— **Gestion des statuts :**
— **Visibilité des tickets :** Il n'est pas explicitement défini si un utilisateur standard peut voir uniquement ses propres tickets ou tous les tickets. L'implémentation actuelle (‘TicketViewSet‘ avec ‘queryset = Ticket.objects.all()‘ et ‘AllowAny‘) expose tous les tickets à tout le monde. Une logique de filtrage basée sur l'auteur ou un groupe pourrait être nécessaire selon les besoins.
— **Notifications :** Le système ne semble pas inclure de notifications (par exemple, lorsqu'un nouveau message est posté dans le chat ou que le statut d'un ticket change), ce qui est souvent attendu dans ce type d'application.

6.3 Bilan technique

L'architecture technique est solide, basée sur un découplage clair entre le frontend React et le backend API Django REST Framework. L'utilisation de DRF simplifie la création de l'API et la gestion des permissions. La base de données MySQL est un choix standard pour les applications Django.

Points techniques notables et améliorations possibles :

— **Cohérence UI Frontend :** Standardiser l'utilisation des composants UI sur une seule librairie (probablement Shadcn/ui, qui est plus utilisée dans les composants récents) améliorerait la cohérence visuelle et la maintenabilité.
— **Chat en temps réel :** Remplacer le polling (‘setInterval‘) par une solution basée sur WebSockets (par exemple, avec Django Channels) offrirait une expérience de chat plus réactive et réduirait la charge sur le serveur.
— **Gestion des erreurs :** L'affichage des erreurs dans le frontend pourrait être plus détaillé et convivial (par exemple, messages d'erreur spécifiques au lieu de "Erreur serveur").
— **Déploiement :** La configuration actuelle (‘DEBUG = True‘, ‘ALLOWED$_HOSTS = []$‘, $clsecrteendur, basede$

6.4 Évolutions futures envisagées

Plusieurs évolutions pourraient être envisagées pour enrichir l'application :
— **Assignation de tickets :** Permettre aux administrateurs d'assigner des tickets à des membres spécifiques de l'équipe de support.
— **Priorités et Catégories :** Ajouter des champs pour définir la priorité ou la catégorie d'un ticket.
— **Pièces jointes :** Permettre aux utilisateurs d'ajouter des fichiers (captures d'écran, logs) aux tickets ou aux messages de chat.
— **Notifications par email :** Envoyer des notifications par email lors de la création d'un ticket, d'un nouveau message, ou d'un changement de statut.
— **Recherche et Filtrage :** Implémenter des fonctionnalités de recherche et de filtrage avancées dans la liste des tickets.
— **Tableau de bord Admin :** Créer un tableau de bord pour les administrateurs avec des statistiques (nombre de tickets ouverts/fermés, temps de réponse moyen, etc.).

6.5 Conclusion du chapitre

En conclusion, le projet de système de tickets support constitue une base fonctionnelle solide répondant aux besoins essentiels de gestion des demandes d'assistance et de communication associée. L'architecture technique choisie est pertinente, bien que des améliorations puissent être apportées, notamment concernant la cohérence de l'interface utilisateur, l'efficacité du chat en temps réel, et l'ajout de tests automatisés. Les pistes d'évolution proposées ouvrent la voie à une application plus complète et robuste à l'avenir.

Références

Cette section peut être complétée pour lister les documentations officielles (React, Django, DRF, Shadcn, etc.) ou toute autre ressource externe utilisée comme référence majeure durant le développement ou l'analyse.

— Documentation React : https://react.dev/
— Documentation Django : https://docs.djangoproject.com/
— Documentation Django REST Framework : https://www.django-rest-framework.org/
— Documentation Vite : https://vitejs.dev/
— Documentation Shadcn/ui : https://ui.shadcn.com/
— Documentation React Bootstrap : https://react-bootstrap.github.io/
— Documentation PlantUML : https://plantuml.com/

Annexe A

Annexes

A.1 Extraits de Code Significatifs

Cette section peut inclure des extraits de code particulièrement pertinents ou complexes qui ont été discutés dans le rapport, par exemple, la classe de permission personnalisée, la logique de création de message, etc.

A.1.1 Permission Personnalisée IsAdminOrAuthor

```python
class IsAdminOrAuthor(permissions.BasePermission):
    def has_object_permission(self, request, view, obj):
        # obj est ici l'instance du Ticket
        # Autorise si l'utilisateur est staff OU s'il est l'auteur du ticket
        return request.user.is_staff or obj.auteur == request.user
```

Listing A.1 – api/views.py - Classe IsAdminOrAuthor

A.1.2 Gestion de la création de message (Anonyme/Connecté)

```python
def post(self, request, ticket_id):
    ticket = get_object_or_404(Ticket, id=ticket_id)
    contenu = request.data.get('message')
    pseudo = request.data.get('pseudo')

    if not contenu:
        return Response({'error': 'Message vide.'}, status=status.
    HTTP_400_BAD_REQUEST)

    message = Message.objects.create(
        ticket=ticket,
        auteur=request.user if request.user.is_authenticated else None,
        pseudo=pseudo if not request.user.is_authenticated else None,
        contenu=contenu
    )
    serializer = MessageSerializer(message)
    return Response(serializer.data, status=status.HTTP_201_CREATED)
```

Listing A.2 – api/views.py - Méthode POST de TicketMessageListCreateAPIView

A.2 Liste des Dépendances

Cette section pourrait lister les dépendances clés du projet (fichiers requirements.txt et package.json). Pour la concision, seules les dépendances majeures sont mentionnées ici.

A.2.1 Dépendances Backend (Python - Fichier ‘requirements.txt‘ probable)

— Django
— djangorestframework
— mysqlclient (ou équivalent pour MySQL)
— django-cors-headers
— django-extensions (mentionné dans ‘settings.py‘)

A.2.2 Dépendances Frontend (Node.js - Fichier ‘package.json‘)

— react
— react-dom
— react-router-dom
— vite
— @vitejs/plugin-react
— bootstrap
— react-bootstrap
— @radix-ui/* (dépendances de shadcn/ui)
— tailwindcss
— lucide-react
— etc. (autres dépendances de shadcn/ui et potentiellement d’autres librairies)