JETC_Immo – Guide d'instructions pour IA (mode DEMO / PRO, exécution stricte A→Z)

1. RÔLE ET COMPORTEMENT DE L'IA

Tu es une IA de type développeur (Copilot / Codex) chargée de construire ou refondre l'application **JETC_Immo**.

Tu n'es pas là pour "improviser", mais pour exécuter ce document **strictement**, étape par étape.

1.1 Règles de comportement obligatoires

1. Tu dois suivre les étapes de ce document **dans l'ordre**, de l'Étape 0 à l'étape finale.
2. Tu ne passes **JAMAIS** à l'étape suivante tant que :
    - le code de l'étape en cours n'est pas cohérent,
    - les vérifications indiquées pour cette étape ne sont pas satisfaites.
3. À chaque étape :
    - tu expliques ce que tu vas faire,
    - tu proposes le code nécessaire (backend / SQL / frontend, selon le cas),
    - tu indiques comment tester le résultat (environnement local + mode DEMO si possible).
4. Si une erreur, une incohérence ou une ambiguïté est détectée :
    - tu t'arrêtes immédiatement,
    - tu expliques le problème,
    - tu proposes des pistes de correction,
    - tu attends une confirmation de l'utilisateur avant de continuer.
5. Tu écris un code propre, commenté, structuré, en privilégiant :
    - lisibilité,
    - robustesse,
    - séparation claire des responsabilités.
6. Tu assumes que la base de données est sous **Supabase** (PostgreSQL + Auth + RLS + Storage).

7. Tu indiques clairement les **variables d'environnement nécessaires** et leur rôle, dès qu'elles apparaissent.
8. Tu maintiens la cohérence avec la structure métier décrite dans ce document.
9. Tu dois respecter la distinction **MODE DEMO** / **MODE PRO** pour :
    - les données,
    - les routes API,
    - les paramètres,
    - et les comportements sensibles (paiements, notifications réelles, etc.).

1.2 Objectif global

Construire une application JETC_Immo permettant de gérer les relations entre :

- Régies
- Propriétaires
- Immeubles et logements
- Locataires
- Entreprises
- Techniciens
- Tickets d'intervention
- Missions
- Facturation
- Messagerie
- Notifications
- **Abonnements / modules payants**
- **Vue administrateur JTEC (dashboard)**

avec un niveau de sécurité compatible avec **RLS Supabase** et un **mode DEMO** totalement sécurisé.

---

2. CONTEXTE MÉTIER

2.1 Résumé du domaine

1. La régie gère un portefeuille d'immeubles et de logements.

2. Chaque logement peut avoir un ou plusieurs locataires au cours du temps.
3. Les locataires déclarent des problèmes via des **tickets** (plomberie, électricité, serrurerie, etc.).
4. La régie valide et diffuse les tickets à des **entreprises partenaires**.
    - Mode général : toutes les entreprises éligibles peuvent voir le ticket.
    - Mode restreint : seules les entreprises sélectionnées voient le ticket.
5. Une entreprise accepte une intervention, ce qui crée une **mission**.
6. Un technicien est assigné à la mission.
7. Le technicien intervient, peut signaler du retard, remplir un rapport, et faire signer le locataire.
8. La mission génère une facture, avec une **commission pour JETC_Immo / JTEC**.
9. Le système conserve un historique des interventions, échanges (chat) et notifications.

2.2 Rôles principaux

- **Locataire**
- **Régie**
- **Entreprise**
- **Technicien**
- **Propriétaire** (consultation / reporting)
- **Administrateur JTEC** (vue globale, suivi business, abonnements)

2.3 Modes DEMO et PRO (métier)

- Le **mode DEMO** doit permettre des démonstrations complètes (locataire, régie, entreprise, technicien, JTEC)
  sans utiliser de vraies données ni déclencher de vrais paiements.
- Le **mode PRO** est le mode réel, avec :
    - vrais comptes,
    - vraies régies / entreprises,
    - vrais tickets,
    - vraies factures,
    - et, plus tard, intégration paiement en production.

---

3. ARCHITECTURE GLOBALE

3.1 Stack recommandée

- **Backend**
  - Supabase (PostgreSQL + Auth + RLS + Storage)
  - API routes (Vercel, avec Router /api/…)
- **Frontend**
  - Application web (HTML/JS ou future migration vers Next.js)
  - Design moderne, responsive, avec :
    - page d'accueil (landing)
    - choix Mode DEMO / Mode PRO
    - formulaires d'inscription & connexion
- **Auth**
  - Supabase Auth (email/password)
  - Rôles gérés via table profiles
- **Stockage fichiers**
  - Supabase Storage (photos de dégâts, signatures, documents)

3.2 Principes clés

- Le frontend ne contacte jamais la base avec des droits **service_role**.
- Les opérations sensibles passent par des **endpoints backend** sécurisés.
- Les règles RLS empêchent l'accès illégitime aux données même en cas de fuite de clé publique.
- La distinction **DEMO / PRO** doit être **claire** dans l'architecture (projet séparé, schéma, ou flag is_demo + RLS adaptées).

---

4. SCHÉMA DE DONNÉES (RÉSUMÉ FONCTIONNEL)

Le schéma détaillé peut être créé dans Supabase, mais l'IA doit respecter la logique suivante.

4.1 Entités principales existantes

On reprend la base existante, avec :

1. **Régies** (regies)
2. **Propriétaires** (proprietaires)
3. **Immeubles** (immeubles)
4. **Logements** (logements)
5. **Locataires** (locataires)
6. **Profils** (profiles)
    - rôle (locataire, regie, entreprise, technicien, admin_jtec)
    - regie_id, entreprise_id si applicable
    - paramètres de notification
7. **Entreprises** (entreprises)
8. **Techniciens** (techniciens)
9. **Tickets** (tickets)
10. **Missions** (missions)
11. **Matériel** (materiel_categories, materiel_items, mission_materiel)
12. **Factures** (factures)
13. **Messages** (messages)
14. **Notifications** (notifications)

(Tout ça reste cohérent avec le document d'origine, avec la même logique métier.)

JETC_Immo_Guide_IA_AZ

4.2 Nouvelles entités / adaptations

15. **Plans / Modules payants** (plans)

- id
- nom (ex : "Pack Régie", "Pack Entreprise", "Module Messagerie", etc.)
- description
- prix_mensuel, prix_annuel
- flags : inclut_tickets, inclut_messagerie, inclut_facturation, etc.

16. **Abonnements** (subscriptions)

- id
- profile_id ou regie_id / entreprise_id selon le modèle choisi
- plan_id
- statut (actif, en_attente, expiré, annulé)
- date_debut, date_fin
- provider (Stripe, autre)
- payment_reference

17. **Mode DEMO / PRO**
    Au choix (à fixer dans le projet) :

- soit colonne is_demo sur les tables concernées (profiles, tickets, missions, etc.),
- soit schémas séparés (demo. / public.),
- soit projet Supabase spécifique pour DEMO.

Dans tous les cas :

- **Obligation** : les données DEMO ne doivent pas polluer les données PRO.

18. **Vue JTEC / Admin**

- rôle admin_jtec dans profiles
- vues / fonctions SQL (ex : stats_regies, stats_entreprises, stats_tickets, etc.) qui retournent des **données agrégées**, pas les détails nominaux.

---

5. RÈGLES DE SÉCURITÉ (RLS) – PRINCIPES

Les règles RLS doivent être compatibles avec les principes suivants :

1. Un **locataire** :
    - ne voit que ses propres tickets et ses propres données.
2. Une **régie** :
    - ne voit que les données (logements, locataires, tickets, missions, factures) de sa régie.
3. Une **entreprise** :
    - voit les tickets qui lui sont accessibles :
        - en mode général, selon les réglages de la régie,

        - en mode restreint, si explicitement autorisée.
    - voit ses missions et ses techniciens.
4. Un **technicien** :
    - voit uniquement ses missions.
5. Le **public** :
    - n'a pas accès aux données sensibles.
6. Les accès **Storage** (photos, signatures) :
    - sont limités aux acteurs concernés par la mission.
7. **Admin JTEC** :
    - accède à des vues agrégées (statistiques globales).
    - n'accède pas aux données nominatives d'un locataire sans besoin métier explicite (optionnel).
8. **Mode DEMO** :
    - les comptes DEMO, données DEMO et actions DEMO sont isolés des données PRO, via RLS et/ou projet séparé.

---

6. PLAN DE DÉVELOPPEMENT POUR L'IA (ÉTAPES A→Z)

Les étapes doivent être suivies strictement dans l'ordre.

Étape 0 – Initialisation DEMO/PRO, Supabase, Vercel, Router API

Objectif :

- Poser les fondations techniques **avant** tout le reste.
- Permettre à l'utilisateur de suivre le travail en **mode DEMO** dès le début.

Attendu :

1. Arborescence de base du projet, par exemple :
    - /src (frontend)
    - /api (routes Vercel)
    - /supabase (SQL, policies, seed)
    - /public
2. Fichiers de configuration :

- .env.example contenant au minimum :
  - SUPABASE_URL
  - SUPABASE_ANON_KEY
  - SUPABASE_SERVICE_ROLE_KEY
  - MODE=demo (par défaut au début)
- documentation rapide pour créer .env.local.

3. Clients Supabase :
   - supabaseClient.js (frontend) avec anon_key
   - supabaseServer.js (backend) avec service_role
4. Connexion Vercel prête :
   - projet structuré pour être déployable sur Vercel,
   - logs consultables,
   - distinction demo / prod via variables d'environnement.
5. **Router API** créé avec arborescence minimale :

/api/

healthcheck.js

auth/

demo/

...

Vérifications :

- Une route /api/healthcheck répond avec un JSON simple (ok: true).
- Les clients Supabase se connectent sans erreur.
- L'application peut être lancée en local et déployée en DEMO.

Étape 1 – Landing page, Mode DEMO / Mode PRO

Objectif :

- Avoir une page d'accueil claire, pour choisir DEMO ou PRO, avec un design déjà propre.

Attendu :

- Une **landing page** avec :
    - logo / nom JETC_Immo,
    - bouton “Découvrir en mode DEMO”,
    - bouton “Se connecter / S’inscrire (mode PRO)”,
    - présentation courte du service.
- Changement de mode :
    - Mode DEMO : utilisation de données fictives, ou base DEMO.
    - Mode PRO : redirection vers login/inscription réelle.

Vérifications :

- Depuis la landing, on peut :
    - ouvrir l’interface DEMO,
    - accéder à la page d’authentification PRO (même si encore vide).

Étape 2 – Connexion Supabase & Auth

(Basé sur le doc d’origine, adapté aux nouveaux besoins)

JETC_Immo_Guide_IA_AZ

Objectif :

- Connecter le backend à Supabase Auth (email/password).
- Implémenter une route de login qui renvoie l’utilisateur connecté et son profil.

Attendu :

- Client Supabase côté serveur (déjà créé à l’étape 0).
- Route /api/auth/login :
    - prend email + password
    - appelle Supabase Auth
    - récupère le profil associé (profiles)
    - renvoie : userId, role, regie_id, entreprise_id, etc.
- Route /api/auth/register :

- crée un utilisateur
- crée un profil associé (role par défaut = locataire, ou selon choix).

Vérifications :

- Un utilisateur test peut se connecter et récupérer son profil.
- En cas d'erreur (mauvais mot de passe), message clair.
- Les clés Supabase sont bien dans les variables d'environnement.

Étape 3 – Profils et rôles

Objectif :

- Garantir qu'à la création d'un utilisateur, un profil est créé dans profiles.

Attendu :

- Trigger SQL ou logique backend :
  - création automatique d'un profil lors de la création d'un utilisateur Supabase.
- Rôle par défaut = locataire, sauf si on passe par un flux "régie / entreprise" spécifique.
- Route /api/auth/me pour récupérer le profil courant.

Vérifications :

- Création d'un nouvel utilisateur → profil automatiquement créé.
- Récupération du rôle, regie_id, entreprise_id, etc.

Étape 4 – Gestion Régies / Immeubles / Logements / Locataires

(Comme le document initial, mais intégré dans le nouveau plan)

Objectif :

- Implémenter les opérations minimales pour gérer la structure immobilière.

Attendu :

- Endpoints backend (réservés aux régies / admin) pour :
  - créer une régie,
  - créer un immeuble rattaché à une régie,

    - créer un logement rattaché à un immeuble,
    - associer un locataire (user) à un logement.

Vérifications :

- Les relations (FK) sont correctes.
- Un locataire test peut être lié à un logement.
- Une régie ne peut pas voir ou modifier les données d'une autre régie (RLS).

Étape 5 – Création de tickets (locataire)

Objectif :

- Permettre à un locataire connecté de créer un ticket.

Attendu :

- Formulaire frontend pour :
    - catégorie, sous_catégorie, description,
    - disponibilités,
    - (photos plus tard).
- Endpoint backend :
    - crée un ticket lié au locataire, logement, régie,
    - enregistre statut initial ("nouveau").

Vérifications :

- Un locataire ne crée un ticket que pour son logement.
- La régie associée est correctement liée.
- Pas de fuite vers d'autres locataires.

Étape 6 – Diffusion des tickets aux entreprises

Objectif :

- Implémenter le mode général / restreint.

Attendu :

- Champs diffusion_mode et entreprises_autorisees gérés.

- Endpoint pour que la régie configure :
    - mode de diffusion,
    - liste des entreprises autorisées (mode restreint).

Vérifications :

- Une entreprise non autorisée ne voit pas le ticket.
- En mode général, seules les entreprises de la régie voient les tickets.

Étape 7 – Acceptation & création de mission

Objectif :

- Permettre à une entreprise d'accepter un ticket et de créer une mission.

Attendu :

- Endpoint :
    - prend ticket_id,
    - vérifie les droits,
    - crée une mission avec statut initial ("en_attente" ou "planifiée"),
    - empêche les doublons (plusieurs entreprises qui acceptent le même ticket).

Vérifications :

- Un ticket accepté n'est plus proposé à d'autres, ou passe dans un statut différent.

Étape 8 – Gestion des techniciens & planning

Objectif :

- Permettre à l'entreprise de gérer ses techniciens et les assigner.

Attendu :

- Endpoints pour :
    - créer / lier des techniciens,
    - assigner une mission à un technicien,
    - définir la date d'intervention.

Vérifications :

- Un technicien voit uniquement ses missions.
- Une entreprise ne manipule pas les techniciens d'une autre entreprise.

Étape 9 – Intervention, retard, rapport, signature

Objectif :

- Gérer le cycle de vie d'une mission côté technicien.

Attendu :

- Endpoints pour :
    - démarrer une mission,
    - signaler un retard (motifs prédéfinis),
    - clôturer une mission,
    - enregistrer un rapport,
    - associer une signature (Storage).

Vérifications :

- Statuts cohérents ("en_attente" → "en_cours" → "terminée").
- Le retard génère une notification à la régie / locataire.

Étape 10 – Facturation

Objectif :

- Générer des factures à partir des missions terminées.

Attendu :

- Table factures alimentée à la clôture de mission.
- Montant calculé (simple pour commencer).
- Commission JETC renseignée.

Vérifications :

- Une facture = une mission.
- Une entreprise voit uniquement ses factures.

- Une régie peut consulter un aperçu (agrégé).

Étape 11 – Messagerie (chat)

Objectif :

- Permettre des échanges entre locataire / régie / entreprise / technicien.

Attendu :

- Table messages avec :
    - mission_id,
    - sender_id,
    - receiver_id ou rôle,
    - contenu, date.
- Endpoints pour :
    - envoyer un message,
    - lister les messages d'une mission.

Vérifications :

- Accès limité aux acteurs impliqués dans la mission.

Étape 12 – Notifications

Objectif :

- Générer des notifications sur les événements clés.

Attendu :

- Triggers SQL ou logique backend pour insérer dans notifications :
    - nouveau ticket,
    - ticket accepté,
    - mission planifiée,
    - retard,
    - mission terminée,
    - facture créée.

- Endpoints pour :
    - lister les notifications non lues,
    - les marquer comme lues.

Étape 13 – Modules payants & abonnements

Objectif :

- Introduire les plans et abonnements pour les régies / entreprises.

Attendu :

- Tables plans et subscriptions opérationnelles.
- Endpoint pour :
    - lister les plans,
    - créer un abonnement,
    - vérifier l'accès à un module selon le plan.

Vérifications :

- Régie / entreprise sans abonnement PRO → accès limité.
- Régie / entreprise avec abonnement → modules débloqués.

Étape 14 – Dashboard Administrateur JTEC

Objectif :

- Fournir une vue interne pour JTEC / JETC_Immo.

Attendu :

- Rôle admin_jtec.
- Endpoints + vues SQL pour statistiques :
    - nombre de régies, entreprises, locataires, techniciens,
    - nombre de tickets (ouverts, en cours, terminés),
    - missions par statut,
    - abonnements par plan,
    - revenus estimés.

- Dashboard visuel côté frontend.

Vérifications :

- Accessible uniquement au rôle admin_jtec.
- Distinction visible entre données DEMO et PRO.

Étape 15 – UI/UX avancé (thèmes, animations, mise en page)

Objectif :

- Ajouter les 3 thèmes par vue et les animations de boutons.

Attendu :

- Gestion des thèmes via classes ou variables CSS.
- Au moins :
    - Thème vert clair (boutons animés, layout “cards”),
    - Thème alternatif,
    - Thème sobre.
- Transitions visuelles simples (hover, clic, transitions entre vues).

Étape 16 – Tests, validation, CI & documentation finale

Objectif :

- Vérifier la cohérence globale et finaliser.

Attendu :

- Plan de test clair (manuel ou automatisé).
- Tests couvrant le chemin complet :
    - création utilisateur,
    - profil,
    - logement,
    - ticket,
    - entreprise,
    - mission,

    - technicien,
    - rapport,
    - facture,
    - notifications,
    - dashboard JTEC,
    - mode DEMO vs PRO.
- Documentation synthétique pour déployer en DEMO et PRO.

---

7. COMPORTEMENT EN CAS D'ERREUR

Même logique que le document initial, adaptée :

1. Si une erreur technique est envisagée (conflit RLS, incohérence, doublon, mélange DEMO/PRO, risque de fuite) :
    - l'IA stoppe immédiatement,
    - décrit la cause probable,
    - explique l'impact,
    - propose des solutions possibles.
2. L'IA propose une version corrigée du code ou du schéma **sans avancer d'étape**.
3. L'IA attend la validation explicite de l'utilisateur.

---

8. REPRISE APRÈS CORRECTION

Quand l'utilisateur indique que la correction est faite ("correction faite, continue étape X") :

1. L'IA :
    - réévalue le contexte,
    - rappelle brièvement ce qui était en cours,
    - reprend à l'étape indiquée sans en sauter.
2. L'IA ne recommence pas le projet depuis zéro sauf demande explicite.

9. Système Supabase – SQL, Policies RLS & Organisation des fichiers

Cette section définit **obligatoirement** comment l'IA doit générer les fichiers SQL, où les placer, et comment gérer les clés Supabase dans le projet.

9.1 Création d'un dossier Supabase complet

L'IA doit générer automatiquement un dossier à la racine du projet :

/supabase/

  /schema/

    00_init_schema.sql

    01_tables.sql

    02_relations.sql

    03_views.sql

    04_functions.sql

    05_triggers.sql

  /policies/

    10_policies_profiles.sql

    11_policies_regies.sql

    12_policies_locataires.sql

    13_policies_entreprises.sql

    14_policies_techniciens.sql

    15_policies_tickets.sql

    16_policies_missions.sql

    17_policies_factures.sql

    18_policies_messages.sql

  /demo/

    seed_demo.sql

    demo_reset.sql

RÈGLE OBLIGATOIRE

Copilot doit écrire **chaque fichier SQL** séparément et fournir **le contenu complet** lors de l'étape correspondante.

9.2 Contenu attendu des fichiers SQL

00_init_schema.sql

- Création du schéma (si besoin public / demo).
- Activation RLS sur toutes les tables.

01_tables.sql

Toutes les tables métier :

- profiles
- regies
- entreprises
- techniciens
- immeubles
- logements
- locataires
- tickets
- missions
- factures
- messages
- materiel + tables pivot
- plans
- subscriptions

Copilot doit écrire **les PRIMARY KEY, FOREIGN KEY, CHECK, UNIQUE** nécessaires.

02_relations.sql

Définition propre des relations, notamment :

- tickets → locataires → logements → régies
- missions → tickets / entreprises / techniciens

03_views.sql

Les vues exposées côté Admin JTEC :

- vue statistiques (globales)
- vue tickets agrégés
- vues anonymisées si nécessaire

04_functions.sql

Fonctions SQL obligatoires :

- get_current_profile()
- fonctions pour récupérer les stats agrégées
- fonctions liées à l'historique / logs DEMO si activé

05_triggers.sql

Triggers essentiels :

- création automatique d'un profil après création d'un user
- triggers notifications (création ticket / mission / facture)

9.3 Fichiers de Policies RLS (RÈGLE STRICTE)

Chaque entité doit avoir son fichier RLS dédié.
Ex :

15_policies_tickets.sql

Contenu obligatoire :

1. **Allow select** selon rôle :
    - locataire : seulement ses tickets
    - régie : tickets de sa régie
    - entreprise : tickets visibles selon diffusion
2. **Insert restrictions** :
    - locataire ne peut créer que pour son logement
3. **Update restrictions** :
    - entreprise ne peut mettre à jour un ticket que si elle a la mission
4. **Prohibition** :
    - aucun accès anonyme

Et Copilot doit inclure :

alter table tickets enable row level security;

create policy "locataire_view_own_tickets"

on tickets for select

using (locataire_id = auth.uid());

et toutes les autres policies pour chaque rôle.

9.4 Gestion du MODE DEMO dans le SQL

Au choix du projet :

OPTION A : schéma demo

create schema if not exists demo;

Toutes les tables sont dupliquées dans /demo/schema.sql.

OPTION B : colonne is_demo

Chaque table doit inclure :

is_demo boolean default false

Toutes les policies RLS doivent filtrer :

(is_demo = false) OR (current_setting('app.mode') = 'demo')

Et l'IA doit proposer cette implémentation proprement.

9.5 Seed DEMO automatique

Le fichier :

/supabase/demo/seed_demo.sql

doit contenir :

- comptes démo
- 1 régie démo
- 2 entreprises démo
- 3 techniciens démo

- logements + locataires fictifs
- tickets et missions d'exemple

Le fichier :

demo_reset.sql

doit **vider** les données DEMO puis relancer seed_demo.sql.

---

10. Gestion des clés Supabase & environnement

Copilot doit expliquer **dans quelle étape** les clés sont nécessaires, et **dans quel fichier les placer**.

10.1 Fichier .env.local (NON commit)

À créer par l'utilisateur après la génération du projet.

Copilot doit y mettre :

SUPABASE_URL="https://xxxxx.supabase.co"

SUPABASE_ANON_KEY="public-anon-key"

SUPABASE_SERVICE_ROLE_KEY="service-role-secret"

MODE="demo"

10.2 Où elles sont utilisées ?

Frontend → seulement ANON_KEY

Dans :

/src/lib/supabaseClient.js

Backend API → SERVICE_ROLE

Dans :

/api/_supabase.js

contenu attendu :

```
import { createClient } from "@supabase/supabase-js";


export const supabaseServer = createClient(
  process.env.SUPABASE_URL,
```

process.env.SUPABASE_SERVICE_ROLE_KEY,

{ auth: { autoRefreshToken: false, persistSession: false } }

);

10.3 Quand Copilot doit te demander les clés ?

Copilot **NE DOIT PAS** demander les clés tant que :

- Étape 0 (structure) n'est pas terminée
- Les fichiers .env.example, supabaseClient.js, supabaseServer.js ne sont pas prêts

Ensuite seulement, Copilot te dira :

"Maintenant, insère tes clés dans .env.local."

🔥 **LA RÈGLE DÉFINITIVE POUR COPILOT**

À CHAQUE FOIS que Copilot génère :

- une table,
- une policy,
- une fonction SQL,
- un trigger,
- un fichier backend,

il doit **automatiquement** produire le fichier complet dans le dossier /supabase/... correspondant.

11. Gestion des fichiers : photos, documents et signatures

11.1 Création des buckets de stockage

Copilot doit créer automatiquement les buckets suivants :

supabase/storage/buckets/

photos_tickets

photos_missions

signatures

Règles obligatoires :

- Les buckets doivent être **privés** (aucune lecture publique).

- La lecture / écriture se fait uniquement via une **route API sécurisée**, jamais directement depuis le frontend.
- Le nom des fichiers doit être unique :
  format recommandé :
- {ticket_id}/{uuid}.jpg
- {mission_id}/{uuid}.jpg
- {mission_id}/signature_{role}.png

11.2 Routes API obligatoires

Dans le dossier /api/files/, Copilot doit créer :

/api/files/upload-photo.js

/api/files/get-photo.js

/api/files/upload-signature.js

/api/files/get-signature.js

11.2.1 upload-photo.js

Objectif : permettre à

- un locataire (ticket),
- un technicien (mission),
- une entreprise (mission),
  d'envoyer une photo de manière sécurisée.

Contenu minimal :

- Vérification du rôle via le profil Supabase
- Vérification que l'utilisateur est bien lié au ticket/mission
- Utilisation du client **service_role**
- Upload dans le bon bucket
- Retour du chemin stocké en BDD

11.2.2 upload-signature.js

Objectif :

- Le technicien ou le locataire fournit une signature (base64 ou fichier).

- La signature doit être stockée dans le bucket signatures.

Conditions :

- Un technicien ne signe que sa mission
- Un locataire ne signe que son intervention
- Aucun utilisateur tiers n'a le droit d'écrire dans ce bucket

---

11.3 Mise à jour SQL des tables pour stocker les fichiers

Copilot doit mettre à jour les tables concernées :

tickets

Ajouter :

photos text[] default '{}'

missions

Ajouter :

photos text[] default '{}',

signature_locataire text,

signature_technicien text

Chaque entrée correspond à **un chemin Supabase Storage**, jamais le fichier brut.

11.4 Policies RLS pour le Storage

Copilot doit générer les fichiers suivants dans :

/supabase/policies/storage/

policies_photos_tickets.sql

policies_photos_missions.sql

policies_signatures.sql

11.4.1 policies_photos_tickets.sql

Règles :

- **Un locataire** ne peut lire que les photos de **ses propres tickets**
- **Une régie** peut lire les photos des tickets **de sa régie**

- **Une entreprise** peut lire les photos des tickets auxquels elle a accès
- **Lecture / écriture via API uniquement** (DOWNLOAD ou GET signés)

Policy type :

```
-- Allow read only if the user is related to the ticket
create policy "ticket_photos_read"
on storage.objects for select
using (
  bucket_id = 'photos_tickets'
  AND EXISTS (
    select 1 from tickets
    where tickets.id = split_part(object_name, '/', 1)::uuid
    and (
      tickets.locataire_id = auth.uid()
      or tickets.regie_id = (select regie_id from profiles where id = auth.uid())
      or tickets.id in (select ticket_id from missions where entreprise_id = (select entreprise_id from profiles where id = auth.uid()))
    )
  )
);
```

Copilot doit intégrer ce fichier en entier.

11.5 Gestion DEMO / PRO dans les fichiers

Lorsqu'un fichier appartient au mode DEMO, Copilot doit :

- soit ajouter is_demo = true dans le chemin ou les métadonnées,
- soit utiliser des buckets séparés :

photos_tickets_demo

photos_missions_demo

signatures_demo

Les endpoints doivent détecter automatiquement :

```
if (process.env.MODE === "demo") {
   bucket = "photos_tickets_demo";
} else {
   bucket = "photos_tickets";
}
```

11.6 Validation et intégration dans les étapes

Étape où insérer cette partie ?

➡️ **Après l'étape 9 (Intervention / Signature / Rapport) dans ton PDF.**

Ce que Copilot doit faire quand il arrive à cette étape :

1. Générer tous les buckets.
2. Générer les endpoints API.
3. Générer les policies RLS.
4. Mettre à jour les tables SQL.
5. Mettre à jour .env.example si besoin (ex : MAX_FILE_SIZE).
6. Ajouter au frontend :
    - uploader,
    - prévisualisations,
    - téléchargement des signatures.