# Marieteam Web - Documentation



# I. Vue d’ensemble du projet

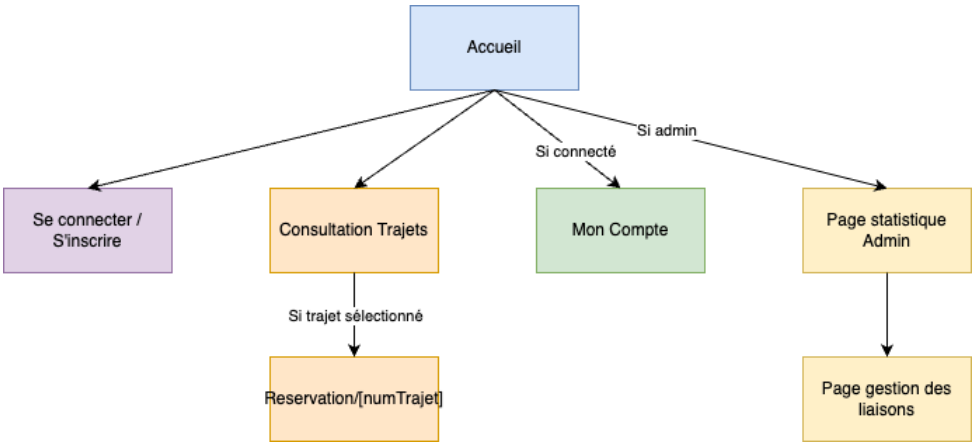
## A. Les stacks

Marieteam Web est développé en JavaScript avec le framework Next.js. Le style est réalisé avec la bibliothèque Tailwind CSS.

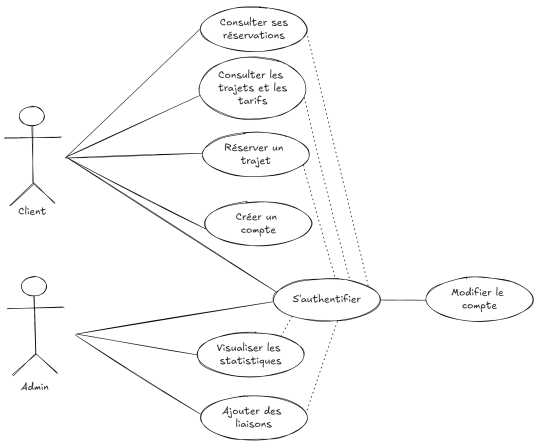
Pour la base de données, le choix a été fait de la réaliser en PostgreSQL avec la solution Supabase, qui permet de gérer plusieurs méthodes plus facilement comme l’authentification, etc.

Pour connecter le projet à la base de données, il faut récupérer les variables d’environnement sur Supabase et les mettre dans un fichier .env à la racine du projet.

## B. Architecture du site

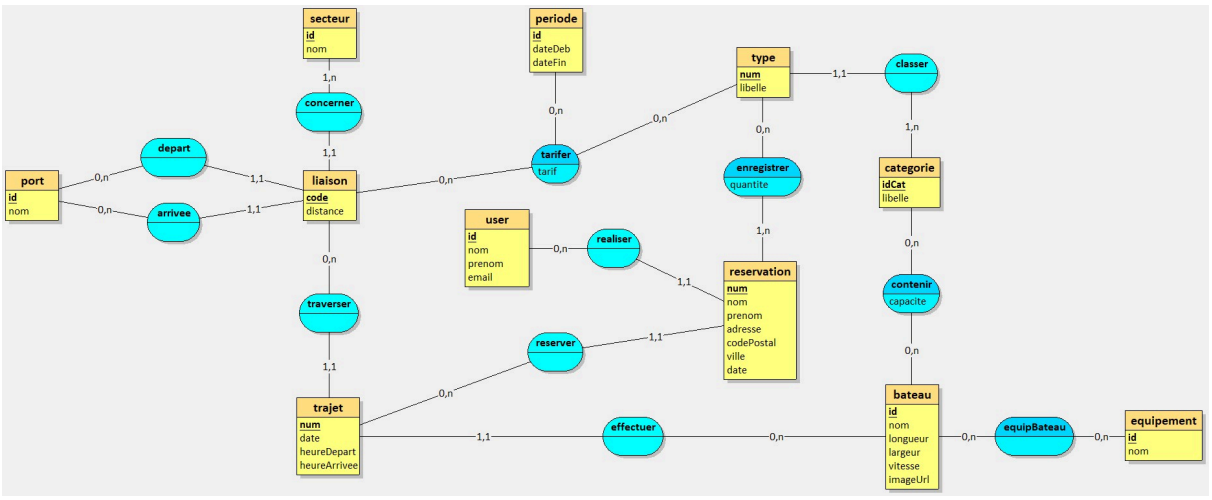


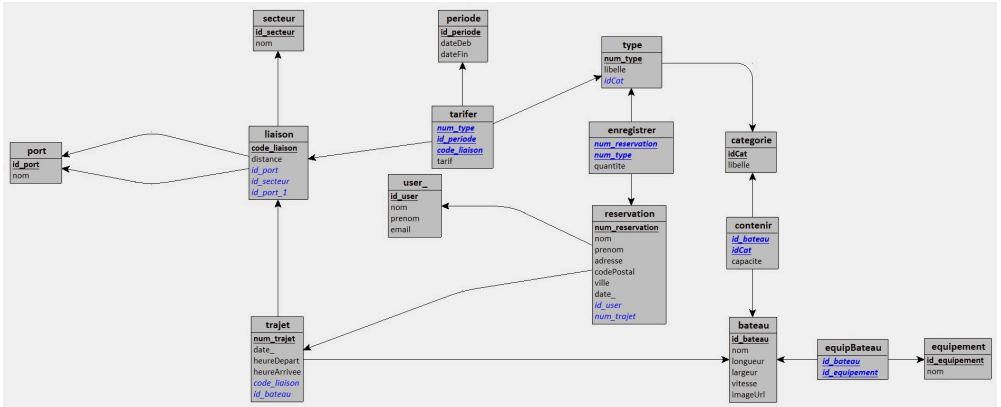

## C. Diagramme cas d’utilisation




## D. La base de données

La base de données est hébergée sur Supabase qui permet de gérer directement de manière plus moderne les données, mais aussi la solution permet d'utiliser des fonctions prédéfinies comme celle d'authentification, de récupération de rôles, de données de manière plus sécurisée. La base de données est en PostGreSQL.




# II. Fonctionnalités

## A. Authentification

Supabase gère l'authentification avec des fonctions prédéfinies. Ici par exemple on va utiliser la fonction de connexion et celle des token de session. Les token vont sécuriser les accès et attester de l'authenticité de la connexion.

```
// === AUTHENTIFICATION AVEC SUPABASE ===

// Tentative de connexion avec email/mot de passe via Supabase Auth
const { data, error } = await supabase.auth.signInWithPassword({
  email: mail,
  password: mdp,
});

// Vérification des erreurs d'authentification
if (error) {
  setError(error.message || "Erreur de connexion.");
  setLoading(false);
  return; // Arrêt de l'exécution en cas d'erreur
}

// === GESTION DU TOKEN JWT ===

// Récupération du token d'accès depuis la session Supabase
const token = await data.session.access_token;

// Stockage sécurisé du token dans un cookie
// expires: 1 = expiration après 1 jour
// path: '/' = accessible sur tout le site
Cookies.set("token", token, { expires: 1, path: "/" });

// === FEEDBACK UTILISATEUR ET REDIRECTION ===

// Affichage d'une notification de succès
showNotification("success", "Connexion réussie");
```

Par exemple avoir ce token de session permet de récupérer l'utilisateur connecté et de ce fait récupérer ses rôles et ajuster ses accès en fonction de ces derniers. Ici on va utiliser la fonction de récupération de l'utilisateur actuel de la session et par la même occasion vérifier son authenticité.

```
/**
 * Récupère les données utilisateur (token et rôle)
 * Fonction centrale pour maintenir l'état d'authentification à jour
 */
const fetchUserData = async () => {
  // Vérification côté client uniquement (Next.js SSR/SSG compatibility)
  if (typeof window !== "undefined") {
    // Récupération du token depuis les cookies
    const tokenFromCookie = Cookie.get("token");
    setToken(tokenFromCookie || null);

    // Si un token existe, récupération du rôle utilisateur
    if (tokenFromCookie) {
      try {
        // Appel à Supabase pour obtenir les données utilisateur
        const {
          data: { user },
        } = await supabase.auth.getUser();
```

```
      // Extraction du rôle depuis les métadonnées utilisateur
      const userRole = user?.user_metadata?.role;
      setRole(userRole);
     } catch (error) {
      console.error("Erreur lors de la récupération du rôle:", error);
      setRole(null);
     }
    } else {
     // Pas de token = pas de rôle
     setRole(null);
    }
  }
};
```

Quand l'user se déconnecte, de ce fait on détruit le cookie et la session est terminée

```
/**
 * Gère la déconnexion de l'utilisateur
 * - Déconnexion Supabase
 * - Suppression du cookie token
 * - Réinitialisation des états
 * - Redirection vers l'accueil
 * - Notification de succès/erreur
 */
const handleLogout = async () => {
  try {
    // Déconnexion côté Supabase
    await supabase.auth.signOut();

    // Suppression du token des cookies
    Cookie.remove("token");

    // Réinitialisation des états locaux
    setToken(null);
    setRole(null);

    // Notification de succès
    showNotification("success", "Déconnexion réussie");

    // Redirection vers la page d'accueil
    router.push("/");
  } catch (error) {
    console.error("Erreur lors de la déconnexion:", error);
    showNotification("error", "Erreur lors de la déconnexion");
  }
};
```

Il faut aussi noter que Supabase demande à l'utilisateur de vérifier son compte via un mail envoyé qui contient un lien d'activation. Sans quoi l'utilisateur ne peux pas accèder au site web.
Lors de l'inscription, le mot de passe doit être assez fort pour être accepté, des directives sont indiquées à l'utilisateur. Sans un mot de passe valide le compte ne peut pas être créé. Ici, on considère un mot de passe valide s'il correspond aux prérequis de la CNIL, à savoir une longueur minimale de 13 caractères avec au moins une majuscule, une minuscule, un chiffre, un caractère spécial.

```
/**
 * Valide le mot de passe en temps réel et affiche les critères
 * @param {string} mdp - Mot de passe à valider
 * @returns {boolean} - True si le mot de passe est valide
 */
const verifMdp = (mdp) => {
  setMdp(mdp);

  // Critères de validation du mot de passe
  const minLength = 13;
  const hasUpperCase = /[A-Z]/.test(mdp); // Au moins une majuscule
  const hasLowerCase = /[a-z]/.test(mdp); // Au moins une minuscule
  const hasSpecialChar = /[!@#$%^&*(),.?":{}|<>]/.test(mdp); // Au moins un caractère spécial

  // Si tous les critères sont respectés
  if (
    mdp.length >= minLength &&
    hasLowerCase &&
    hasUpperCase &&
    hasSpecialChar
  ) {
    setGuidelines(
      <p className="text-green-600 font-medium pt-4">Mot de passe valide</p>
    );
    return true;
  } else {
    // Affichage dynamique des critères non respectés
    setGuidelines(
      <div className="pt-4">
        <p className="mb-1">Un mot de passe valide doit contenir :</p>
        <ul className="list-disc pl-5 space-y-1">
          {mdp.length < minLength && <li>13 caractères minimum</li>}
          {!hasUpperCase && <li>Une majuscule minimum</li>}
          {!hasLowerCase && <li>Une minuscule minimum</li>}
          {!hasSpecialChar && <li>Un caractère spécial minimum</li>}
        </ul>
      </div>
    );
    return false;
  }
};
```

Une fois le compte validé, il peut se connecter, s'il n'est pas vérifié Supabase refusera la connexion en retournant une erreur.

## B. Réservation

D'abord pour réserver, l'utilisateur doit être connecté. S'il ne l'est pas il sera redirigé vers la page de connexion et sera invité à se connecter.

```
/**
 * Gestionnaire de clic sur le bouton "Réserver"
 * Vérifie si l'utilisateur est connecté avant de permettre la réservation
 */
const handleReservationClick = (e, trajetNum) => {
  const tokenFromCookie = Cookies.get("token");
```

```
  // Vérifie si le token est présent (utilisateur connecté)
  if (!tokenFromCookie) {
    e.preventDefault(); // Empêche le lien de fonctionner immédiatement

    // Enregistre l'ID du trajet dans les cookies pour après connexion
    Cookies.set("resTrajet", trajetNum, { expires: 1, path: "/" });

    // Redirige l'utilisateur vers la page de connexion
    router.push("/connexion");
  }
  // Si token présent, le lien fonctionne normalement
};
```

Ici on vérifie si le token de session est présent pour être sûr que l'utilisateur est log, sinon il est redirigé vers la page de connexion. Aussi, le numéro de trajet est pris en cookie, de cette manière a sa connexion l'utilsateur sera redirigé directement sur le trajet qu'il voulait réserver.

On effectue cette vérification dans la page Connexion:

```
const reserveTrajet = Cookie.get("resTrajet");

if (reserveTrajet) {
  // Si une réservation était en cours, redirection vers la page de réservation
  router.push(`/reservation/${reserveTrajet}`);
  Cookie.remove("resTrajet"); // Nettoyage du cookie temporaire
} else {
  // Sinon, redirection vers la page d'accueil
  router.push("/");
}
```

Sur la page, l'utilisateur peut rechercher un trajet via les champs, ces derniers sont non remplissables manuellement et sont des champs à sélections qui prend en base de données les données disponibles. D'abord il choisit un secteur, le site retourne les liaisons disponibles dans ce secteur et affiche le champ pour en sélectionner une et l'utilisateur choisi une date. A noter quel le champ de date ne peut sélectionner de date antérieure à celle du jour.

Une fois les champs remplis, l'utilisateur clique sur rechercher et choisi le trajet qu'il souhaite, il peut voir les places disponibles et choisir de réserver ce trajet.

En cliquant sur réserver ce trajet, le code va récupérer l'id du trajet et le transmettre à la page suivante, depuis cette page s'effectueront toutes les récupérations de données associées au trajet, comme les tarifs, les places disponibles, etc.

Ce bouton permet la redirection, ici handleReservationClick permet de vérifier si l'user est log, sinon la redirection se fait vers la page /reservation/[trajetNum]. Où trajetNum est un argument passé en paramètre afin d'être récupéré de l'autre côté. Ici on donne celui du trajet que l'user a sélectionné.

```
<Link
  href={`/reservation/${selectedTrajet.num}`}
  onClick={(e) => handleReservationClick(e, selectedTrajet.num)}
  className="p-3 bg-sky-900 rounded-xl text-white w-[80%] m-auto block text-center"
>
  Réserver ce trajet
</Link>;
```

Une fois sur la page réservation, toute la récupération des données s'effectue, voilà par exemple comment on retourne toutes les informations du trajet sélectionné à l'affichage:

```
/**
 * Fonction getServerSideProps - Exécutée côté serveur avant le rendu de la page
 * Récupère toutes les données nécessaires pour afficher le formulaire de réservation
 * @param {Object} context - Contexte Next.js contenant les paramètres de la route
 * @returns {Object} Props à passer au composant
 */
export async function getServerSideProps(context) {
  const { trajetNum } = context.params; // Récupération du numéro de trajet depuis l'URL

  console.log(trajetNum);

  // Récupération des informations de base du trajet
  const { data: trajet, error } = await supabase
    .from("trajet")
    .select("num, heureDepart, heureArrivee, idBateau, idLiaison, date")
    .eq("num", trajetNum)
    .single();

  if (trajet) {
    console.log("trajet trouvé");
  } else {
    console.log("trajet non trouvé");
  }

  // Récupération du nom du bateau
  const { data: bateau } = await supabase
    .from("bateau")
    .select("nom")
    .eq("id", trajet.idBateau)
    .single();

  // Récupération des informations de liaison (ports de départ/arrivée)
  const { data: liaison } = await supabase
    .from("liaison")
    .select("code, depart_id, arrivee_id")
    .eq("code", trajet.idLiaison)
    .single();

  const { data: portDepart } = await supabase
    .from("port")
    .select("nom")
    .eq("id", liaison.depart_id)
    .single();

  const { data: portArrivee } = await supabase
    .from("port")
    .select("nom")
    .eq("id", liaison.arrivee_id)
    .single();

  // Récupération de toutes les réservations existantes pour ce trajet
  const { data: reservation, error: errorReservation } = await supabase
```

```
  .from("reservation")
  .select("num")
  .eq("idTrajet", trajet.num);

// Initialisation des compteurs de places réservées
let placePassagerReserv = 0; // Total des passagers déjà réservés
let placePetitVehReserv = 0; // Total des petits véhicules déjà réservés
let placeGrandVehReserv = 0; // Total des grands véhicules déjà réservés

// Calcul des places déjà réservées pour chaque catégorie
await Promise.all(
 reservation.map(async (res) => {
  const reservationNum = res.num;

  // Comptage des passagers (adultes, juniors, enfants - types 1, 2, 3)
  const { data: passagerReserv, error: errorPassagerReserv } =
   await supabase
    .from("enregistrer")
    .select("quantite")
    .eq("reservation_num", reservationNum)
    .or("type_num.eq.1,type_num.eq.2,type_num.eq.3");

  if (!errorPassagerReserv) {
   placePassagerReserv += passagerReserv.reduce(
    (sum, row) => sum + row.quantite,
    0
   );
  }

  // Comptage des petits véhicules (voitures, camionnettes - types 4, 5)
  const { data: petitVehiculeReserv, error: errorPetitVehiculeReserv } =
   await supabase
    .from("enregistrer")
    .select("quantite")
    .eq("reservation_num", reservationNum)
    .or("type_num.eq.4,type_num.eq.5");

  if (!errorPetitVehiculeReserv) {
   placePetitVehReserv += petitVehiculeReserv.reduce(
    (sum, row) => sum + row.quantite,
    0
   );
  }

  // Comptage des grands véhicules (camping-cars, camions - types 6, 7)
  const { data: grandVehiculeReserv, error: errorGrandVehiculeReserv } =
   await supabase
    .from("enregistrer")
    .select("quantite")
    .eq("reservation_num", reservationNum)
    .or("type_num.eq.6,type_num.eq.7");

  if (!errorGrandVehiculeReserv) {
   placeGrandVehReserv += grandVehiculeReserv.reduce(
    (sum, row) => sum + row.quantite,
```

```
        0
      );
    }
  })
);

// Récupération des capacités maximales du bateau pour chaque type de place
const { data: placePassager } = await supabase
  .from("contenir")
  .select("capacite")
  .eq("idBateau", trajet.idBateau)
  .eq("idPlace", "A") // 'A' correspond aux places passagers
  .single();

const { data: placePetitVehicule } = await supabase
  .from("contenir")
  .select("capacite")
  .eq("idBateau", trajet.idBateau)
  .eq("idPlace", "B") // 'B' correspond aux petits véhicules
  .single();

const { data: placeGrandVehicule } = await supabase
  .from("contenir")
  .select("capacite")
  .eq("idBateau", trajet.idBateau)
  .eq("idPlace", "C") // 'C' correspond aux grands véhicules
  .single();

console.log(trajet.date);

// Détermination de la période tarifaire pour ce trajet
// La période est déterminée en fonction de la date du trajet
const { data: periode, error: periodeError } = await supabase
  .from("periode")
  .select("id")
  .lte("dateDeb", trajet.date) // dateDeb <= trajet.date
  .gte("dateFin", trajet.date) // dateFin >= trajet.date
  .single();

if (periodeError || !periode) {
  console.error(
    "Erreur lors de la récupération de la période:",
    periodeError
  );
  // Retour d'une erreur 404 si aucune période n'est trouvée
  return {
    notFound: true,
  };
}

console.log("Période trouvée:", periode);

// Récupération des tarifs en fonction de la liaison et de la période
const { data: prix, error: prixError } = await supabase
  .from("tarifer")
```

```
  .select("tarif, type")
  .eq("liaison_code", liaison.code)
  .eq("idPeriode", periode.id)
  .order("type", { ascending: true }); // Tri par type pour un ordre cohérent

if (prixError) {
  console.error("Erreur lors de la récupération des tarifs:", prixError);
  return {
    notFound: true,
  };
}

console.log("Tarifs trouvés:", prix);

// Formatage des heures pour l'affichage (HH:MM vers HHhMM)
const heureDepartFormat =
  trajet.heureDepart.substring(0, 2) +
  "h" +
  trajet.heureDepart.substring(3, 5);
const heureArriveeFormat =
  trajet.heureArrivee.substring(0, 2) +
  "h" +
  trajet.heureArrivee.substring(3, 5);
// Formatage de la date pour l'affichage (YYYY-MM-DD vers DD/MM/YYYY)
const dateFormat =
  trajet.date.substring(8, 10) +
  "/" +
  trajet.date.substring(5, 7) +
  "/" +
  trajet.date.substring(0, 4);

// Calcul de la durée du trajet
const { hours, minutes } = (() => {
  const [heureD, minuteD] = trajet.heureDepart.split(":");
  const [heureA, minuteA] = trajet.heureArrivee.split(":");

  // Création d'objets Date pour calculer la différence
  const depart = new Date();
  depart.setHours(heureD, minuteD, 0, 0);

  const arrivee = new Date();
  arrivee.setHours(heureA, minuteA, 0, 0);

  // Calcul de la différence en millisecondes puis conversion en minutes
  const differenceInMillis = arrivee - depart;
  const differenceInMinutes = Math.floor(differenceInMillis / 60000);

  // Conversion en heures et minutes
  return {
    hours: Math.floor(differenceInMinutes / 60),
    minutes: differenceInMinutes % 60,
  };
})();

// Retour des props pour le composant
```

```
  // Toutes les données sont formatées et les places disponibles sont calculées
  return {
    props: {
      trajet: {
        ...trajet,
        num: trajetNum,
        nomBateau: bateau.nom,
        tempsTrajet: `${hours}h ${minutes}m`,
        portDepart: portDepart.nom,
        portArrivee: portArrivee.nom,
        // Calcul des places disponibles = capacité totale - places déjà réservées
        placePassager: placePassager.capacite - placePassagerReserv,
        placePetitVehicule: placePetitVehicule.capacite - placePetitVehReserv,
        placeGrandVehicule: placeGrandVehicule.capacite - placeGrandVehReserv,
        heureDepartFormat: heureDepartFormat,
        heureArriveeFormat: heureArriveeFormat,
        dateFormat: dateFormat,
        prix: prix, // Tableau des tarifs par type
      },
    },
  };
}
```

Une fois les champs remplis, l'utilisateur valide en cliquant sur réserver et aussi en validant une deuxième fois via une boite de dialogue.

S'il valide et que les vérifications faites du côté du code, pour les saisies et la validité des champs sont bonnes, alors le code prend en compte les informations et va enregistrer la réservation en base de données. D'abord il va créer un identifiant unique composé du timestamp du moment de réservation et l'id du compte, ainsi on peut s'assurer que chaque réservation a un identifiant unique.

```
/**
 * Génère un numéro de réservation unique basé sur l'ID utilisateur et un timestamp
 * @param {string} idCompte - ID du compte utilisateur
 * @returns {string} Numéro de réservation unique
 */
const generateUniqueReservationNum = (idCompte) => {
  const timestamp = Date.now();
  return `${idCompte}-${timestamp}`;
};
```

Si tout est bon alors on va créer la réservation avec confirmReservation:

```
/**
 * Fonction de confirmation finale de la réservation
 * Exécutée après validation par l'utilisateur dans le dialogue
 */
const confirmReservation = async () => {
  setIsSubmitting(true);
  setLoading(true);
  setShowConfirmDialog(false);

  try {
    // Récupération de l'utilisateur authentifié
    const {
```

```
  data: { user },
} = await supabase.auth.getUser();
const idUser = user.id;
const numReservation = generateUniqueReservationNum(idUser);
const today = new Date().toISOString().split("T")[0]; // Date du jour au format YYYY-MM-DD

// Insertion de la réservation principale dans la base de données
const { error } = await supabase.from("reservation").insert([
  {
    num: numReservation,
    idTrajet: trajet.num,
    idCompte: idUser,
    nom: formData.nom,
    prenom: formData.prenom,
    adr: formData.adresse,
    cp: formData.codePostal,
    ville: formData.ville,
    date: today,
  },
]);

if (error) throw error;

// Préparation des insertions pour chaque type de place réservée
// Chaque condition vérifie s'il y a des places de ce type à insérer
const insertPromises = [];
if (formData.adulte > 0) {
  insertPromises.push(
    supabase.from("enregistrer").insert([
      {
        type_num: 1, // ID du type "adulte"
        reservation_num: numReservation,
        quantite: formData.adulte,
      },
    ])
  );
}
if (formData.junior > 0) {
  insertPromises.push(
    supabase.from("enregistrer").insert([
      {
        type_num: 2, // ID du type "junior"
        reservation_num: numReservation,
        quantite: formData.junior,
      },
    ])
  );
}
if (formData.enfant > 0) {
  insertPromises.push(
    supabase.from("enregistrer").insert([
      {
        type_num: 3, // ID du type "enfant"
        reservation_num: numReservation,
        quantite: formData.enfant,
```

```
      },
    ])
  );
}
if (formData.voiture > 0) {
  insertPromises.push(
    supabase.from("enregistrer").insert([
      {
        type_num: 4, // ID du type "voiture"
        reservation_num: numReservation,
        quantite: formData.voiture,
      },
    ])
  );
}
if (formData.camionnette > 0) {
  insertPromises.push(
    supabase.from("enregistrer").insert([
      {
        type_num: 5, // ID du type "camionnette"
        reservation_num: numReservation,
        quantite: formData.camionnette,
      },
    ])
  );
}
if (formData.campingCar > 0) {
  insertPromises.push(
    supabase.from("enregistrer").insert([
      {
        type_num: 6, // ID du type "camping-car"
        reservation_num: numReservation,
        quantite: formData.campingCar,
      },
    ])
  );
}
if (formData.camion > 0) {
  insertPromises.push(
    supabase.from("enregistrer").insert([
      {
        type_num: 7, // ID du type "camion"
        reservation_num: numReservation,
        quantite: formData.camion,
      },
    ])
  );
}

// Exécution de toutes les insertions en parallèle
await Promise.all(insertPromises);

// Récupération des détails complets de la réservation créée
const reservation = await recupRes(numReservation);
setSelectedReservation(reservation);
```

```
    setDone(true); // Passage à l'écran de confirmation
    showNotification("success", "Réservation effectuée avec succès !");
  } catch (error) {
    console.error(error);
    showNotification(
      "error",
      "Une erreur est survenue lors de la réservation."
    );
  } finally {
    setIsSubmitting(false);
    setLoading(false);
  }
};
```

Une fois crée, la réservation est récupérée et un récapitulatif est affiché à l'utilisateur.

On la récupère avec la fonction recupRes:

```
/**
 * Fonction pour récupérer les détails complets d'une réservation
 * @param {string} numRes - Numéro de la réservation
 * @returns {Object|null} Objet contenant tous les détails de la réservation ou null en cas d'erreur
 */
async function recupRes(numRes) {
  try {
    // Récupération de la réservation principale
    const { data: reservation, error: reservationError } = await supabase
      .from("reservation")
      .select("*")
      .eq("num", numRes)
      .single();

    if (reservationError) throw reservationError;

    // Récupération des informations du trajet associé
    const { data: trajet } = await supabase
      .from("trajet")
      .select("*")
      .eq("num", reservation.idTrajet)
      .single();

    // Formatage de la date et des heures pour l'affichage
    const date =
      trajet.date.substring(8, 10) +
      "/" +
      trajet.date.substring(5, 7) +
      "/" +
      trajet.date.substring(0, 4) +
      " de " +
      trajet.heureDepart.substring(0, 2) +
      "h" +
      trajet.heureDepart.substring(3, 5) +
      " à " +
      trajet.heureArrivee.substring(0, 2) +
```

```
    "h" +
    trajet.heureArrivee.substring(3, 5);

  // Récupération des informations de la liaison (ports de départ/arrivée)
  const { data: liaison } = await supabase
    .from("liaison")
    .select("*")
    .eq("code", trajet.idLiaison)
    .single();

  const { data: depart } = await supabase
    .from("port")
    .select("nom")
    .eq("id", liaison.depart_id)
    .single();

  const { data: arrivee } = await supabase
    .from("port")
    .select("nom")
    .eq("id", liaison.arrivee_id)
    .single();

  // Récupération des places réservées
  const { data: placesReserves } = await supabase
    .from("enregistrer")
    .select("*")
    .eq("reservation_num", reservation.num);

  // Récupération de la période tarifaire correspondant à la date du trajet
  const { data: periode, error: periodeError } = await supabase
    .from("periode")
    .select("id")
    .lte("dateDeb", trajet.date) // dateDeb <= trajet.date
    .gte("dateFin", trajet.date) // dateFin >= trajet.date
    .single();

  if (periodeError || !periode) {
    console.error(
      "Erreur lors de la récupération de la période:",
      periodeError
    );
    return {
      notFound: true,
    };
  }

  // Enrichissement des données de places avec les types et tarifs
  const detailedSeats = await Promise.all(
    placesReserves.map(async (place) => {
      // Récupération du type de place (adulte, junior, etc.)
      const { data: seatType } = await supabase
        .from("type")
        .select("*")
        .eq("num", place.type_num)
        .single();
```

```
      // Récupération du tarif correspondant
      const { data: seatPrice } = await supabase
        .from("tarifer")
        .select("*")
        .eq("liaison_code", liaison.code)
        .eq("type", place.type_num)
        .eq("idPeriode", periode.id)
        .single();

      // Calcul du prix total pour cette catégorie de places
      const totalPrice = seatPrice.tarif * place.quantite;

      return {
        ...place,
        type: seatType,
        tarif: seatPrice.tarif,
        total: totalPrice,
      };
    })
  );

  // Retour de l'objet complet avec toutes les informations formatées
  return {
    ...reservation,
    depart_nom: depart.nom,
    arrivee_nom: arrivee.nom,
    date: date,
    places: detailedSeats,
  };
 } catch (err) {
  console.error("Erreur lors de la récupération de la réservation:", err);
  return null;
 }
}
```

## C. Mon compte

La page mon compte permet à l'utilisateur de visualiser ses données et ses réservations. Il peut modifier les informations de con compte et afficher les détails de toutes ses réservation en cliquant sur le ticket présent à côté de son numéro de réservation.

Supabase permet de récupérer un utilisateur qui est connecté avec cette fonction:

```
const { data: { user }, error } = await supabase.auth.getUser();
```

En le récupérant, on va avoir accès à son id et avec son id on récupère toutes les informations sur ses réservations, s'il en a, pour les afficher. On fait toutes ces actions dans la méthode fetchUser:

```
/**
 * Fonction principale pour récupérer toutes les données de l'utilisateur
 *
 * Processus complexe en plusieurs étapes :
 * 1. Récupération des informations de base de l'utilisateur
 * 2. Récupération de ses réservations
```

```
 * 3. Pour chaque réservation : enrichissement avec détails du trajet, liaison, ports
 * 4. Calcul des prix selon les périodes tarifaires
 */
const fetchUser = async () => {
 try {
  // === RÉCUPÉRATION DE L'UTILISATEUR CONNECTÉ ===
  const {
   data: { user },
   error,
  } = await supabase.auth.getUser();
  if (error) throw error;

  // === RÉCUPÉRATION DES RÉSERVATIONS DE BASE ===
  const { data: reservation, error: reservationError } = await supabase
   .from("reservation")
   .select("*")
   .eq("idCompte", user.id); // Filtre par ID de l'utilisateur connecté

  if (reservationError) throw reservationError;

  // === ENRICHISSEMENT DES RÉSERVATIONS ===
  /**
   * Pour chaque réservation, on va chercher toutes les informations détaillées :
   * - Détails du trajet (date, heures, liaison)
   * - Informations des ports (départ/arrivée)
   * - Places réservées avec types et tarifs
   */
  const detailedReservations = await Promise.all(
   reservation.map(async (res) => {
    // --- Récupération des détails du trajet ---
    const { data: trajet } = await supabase
     .from("trajet")
     .select("*")
     .eq("num", res.idTrajet)
     .single();

    // Formatage de la date et heures pour affichage lisible
    // Format : "DD/MM/YYYY de HHhMM à HHhMM"
    const date =
     trajet.date.substring(8, 10) +
     "/" +
     trajet.date.substring(5, 7) +
     "/" +
     trajet.date.substring(0, 4) +
     " de " +
     trajet.heureDepart.substring(0, 2) +
     "h" +
     trajet.heureDepart.substring(3, 5) +
     " à " +
     trajet.heureArrivee.substring(0, 2) +
     "h" +
     trajet.heureArrivee.substring(3, 5);

    // --- Récupération des détails de la liaison ---
    const { data: liaison } = await supabase
```

```
    .from("liaison")
    .select("*")
    .eq("code", trajet.idLiaison)
    .single();

  // --- Récupération des noms des ports ---
  const { data: depart } = await supabase
    .from("port")
    .select("nom")
    .eq("id", liaison.depart_id)
    .single();

  const { data: arrivee } = await supabase
    .from("port")
    .select("nom")
    .eq("id", liaison.arrivee_id)
    .single();

  // --- Récupération des places réservées ---
  const { data: placesReserves } = await supabase
    .from("enregistrer")
    .select("*")
    .eq("reservation_num", res.num);

  // --- Recherche de la période tarifaire active ---
  /**
   * Trouve la période tarifaire correspondante à la date du trajet
   * Une période est active si : dateDeb <= date_trajet <= dateFin
   */
  const { data: periode, error: periodeError } = await supabase
    .from("periode")
    .select("id")
    .lte("dateDeb", trajet.date) // dateDeb <= trajet.date
    .gte("dateFin", trajet.date) // dateFin >= trajet.date
    .single();

  if (periodeError || !periode) {
    console.error(
      "Erreur lors de la récupération de la période:",
      periodeError
    );
    return {
      notFound: true,
    };
  }

  // --- Enrichissement détaillé de chaque place réservée ---
  /**
   * Pour chaque place, récupération :
   * - Du type de place (libellé, caractéristiques)
   * - Du tarif applicable (selon liaison, type, période)
   * - Calcul du prix total (tarif × quantité)
   */
  const detailedSeats = await Promise.all(
    placesReserves.map(async (place) => {
```

```
        // Récupération du type de place
        const { data: seatType } = await supabase
          .from("type")
          .select("*")
          .eq("num", place.type_num)
          .single();

        // Récupération du tarif spécifique
        const { data: seatPrice } = await supabase
          .from("tarifer")
          .select("*")
          .eq("liaison_code", liaison.code)
          .eq("type", place.type_num)
          .eq("idPeriode", periode.id)
          .single();

        // Calcul du prix total pour cette catégorie de places
        const totalPrice = seatPrice.tarif * place.quantite;

        return {
          ...place,
          type: seatType,
          tarif: seatPrice.tarif,
          total: totalPrice,
        };
      })
    );

    // Retour de la réservation enrichie avec toutes les informations
    return {
      ...res,
      depart_nom: depart.nom,
      arrivee_nom: arrivee.nom,
      date: date,
      places: detailedSeats,
    };
  })
);

// === MISE À JOUR DE L'ÉTAT AVEC LES DONNÉES COMPLÈTES ===
setUserData({
  display_name: user.user_metadata?.display_name || "Nom inconnu",
  prenom: user.user_metadata?.prenom || "",
  nom: user.user_metadata?.nom || "",
  email: user.email,
  reservation: detailedReservations,
});
} catch (err) {
  setError(err.message);
}
};
```

## D. Administration

Si un utilisateur est un admin alors il verra dans la barre de navigation l'option Dashboard s'afficher.

Pour procéder à la vérification des droits on vérifie les droits via ceux dans Supabase. A savoir qu'un utilisateur crée via le site n'est que User et pas Admin, il faut passer par l'interface Supabase pour donner le rôle admin à un utilisateur.

```
const checkAdminRole = async () => {
 try {
  // Récupération de l'utilisateur connecté depuis Supabase Auth
  const {
   data: { user },
  } = await supabase.auth.getUser();
  const userRole = user?.user_metadata?.role;

  // Vérification du rôle admin
  if (userRole !== "admin") {
   router.push("/"); // Redirection si pas admin
   return;
  }

  setIsAuthorized(true);
 } catch (error) {
  console.error("Erreur lors de la vérification du rôle:", error);
  router.push("/"); // Redirection en cas d'erreur
 } finally {
  setIsChecking(false);
 }
};
```

En cliquant dessus il sera redirigé vers le dashboard admin qui permet de visualiser les données sur une période donnée, aussi d'accéder à la partie gestion des liaisons.

Pour récupérer toutes ces informations, on fait de la manière suivante:

```
/**
 * Fonction principale pour récupérer et calculer toutes les données du dashboard
 *
 * Logique :
 * 1. Récupère les revenus basés sur la date de réservation
 * 2. Récupère les passagers basés sur la date du trajet
 * 3. Calcule les tarifs en croisant avec les périodes et tarifications
 */
const fetchRevenus = async () => {
 setLoading(true);
 setError(null);

 // Variables temporaires pour les calculs
 let prixTotalCalculé = 0;
 let revenusTemp = {};
 let passagerTemp = {};
 let totalPassagerTemp = { catA: 0, catB: 0, catC: 0 };

 try {
  // === FORMATAGE DES DATES ===
  /**
   * Fonction utilitaire pour formater une date en YYYY-MM-DD
   * Format requis par Supabase pour les comparaisons de dates
   */
  const formatDate = (date) => {
```

```
  const year = date.getFullYear();
  const month = String(date.getMonth() + 1).padStart(2, "0");
  const day = String(date.getDate()).padStart(2, "0");
  return `${year}-${month}-${day}`;
};

const startDateFormatted = formatDate(startDate);
const endDateFormatted = formatDate(endDate);

// === RÉCUPÉRATION DES REVENUS ===
/**
 * Récupération des réservations avec leurs trajets et enregistrements
 * Filtre basé sur la date de réservation (pas la date du trajet)
 *
 * Structure de données :
 * - reservation : informations de base de la réservation
 * - trajet : informations du trajet associé (liaison, date)
 * - enregistrer : détails des passagers (quantité, type)
 */
const { data: reservationsData, error: errorReservations } = await supabase
  .from("reservation")
  .select(
    `
            num,
            date,
            idTrajet,
            trajet:trajet (
              idLiaison,
              date
            ),
            enregistrer (
              quantite,
              type_num
            )
          `
  )
  .gte("date", startDateFormatted) // >= date de début
  .lte("date", endDateFormatted); // <= date de fin

if (errorReservations) throw errorReservations;

// === RÉCUPÉRATION DES PASSAGERS ===
/**
 * Récupération des trajets avec leurs réservations
 * Filtre basé sur la date du trajet (pas la date de réservation)
 *
 * Utilisé pour calculer le nombre de passagers par jour de voyage
 */
const { data: trajetsData, error: errorTrajets } = await supabase
  .from("trajet")
  .select(
    `
            num,
            date,
            idLiaison,
```

```
            reservation!reservation_idTrajet_fkey (
              num,
              enregistrer (
                quantite,
                type_num
              )
            )
          `
  )
  .gte("date", startDateFormatted)
  .lte("date", endDateFormatted);

if (errorTrajets) throw errorTrajets;

// === TRAITEMENT DES REVENUS ===
/**
 * Pour chaque réservation :
 * 1. Trouve la période tarifaire correspondante à la date du trajet
 * 2. Récupère les tarifs pour cette période et liaison
 * 3. Calcule le montant total (quantité × tarif)
 * 4. Accumule par date de réservation
 */
for (const reservation of reservationsData) {
  let montantTotal = 0;

  // Recherche de la période tarifaire active pour la date du trajet
  const { data: periode, error: periodeError } = await supabase
    .from("periode")
    .select("id")
    .lte("dateDeb", reservation.trajet.date) // dateDeb <= trajet.date
    .gte("dateFin", reservation.trajet.date) // dateFin >= trajet.date
    .single();

  if (periodeError || !periode) {
    console.error(
      "Erreur lors de la récupération de la période:",
      periodeError
    );
    // Gestion d'erreur : période non trouvée
    return {
      notFound: true,
    };
  }

  // Récupération de tous les tarifs pour cette période
  const { data: tarifs, error: errorTarifs } = await supabase
    .from("tarifer")
    .select("*")
    .eq("idPeriode", periode.id);

  if (errorTarifs) throw errorTarifs;

  // Création d'un index des tarifs pour un accès rapide
  // Structure : {"liaison_code-type": tarif}
  const tarifsMap = {};
```

```
    tarifs.forEach((tarif) => {
      tarifsMap[`${tarif.liaison_code}-${tarif.type}`] = tarif.tarif;
    });

    // Calcul du montant pour chaque enregistrement de la réservation
    for (const enregistrement of reservation.enregistrer) {
      const tarifKey = `${reservation.trajet.idLiaison}-${enregistrement.type_num}`;
      const tarif = tarifsMap[tarifKey];
      montantTotal += enregistrement.quantite * tarif;
    }

    // Accumulation des totaux
    prixTotalCalculé += montantTotal;
    revenusTemp[reservation.date] =
      (revenusTemp[reservation.date] || 0) + montantTotal;
  }

  // === TRAITEMENT DES PASSAGERS ===
  /**
   * Classification des passagers par catégories :
   * - Cat A : types 1-3
   * - Cat B : types 4-5
   * - Cat C : types 6+
   *
   * Regroupement par date de trajet (jour de voyage effectif)
   */
  for (const trajet of trajetsData) {
    // Initialisation des compteurs pour cette date si nécessaire
    if (!passagerTemp[trajet.date]) {
      passagerTemp[trajet.date] = { catA: 0, catB: 0, catC: 0 };
    }

    // Parcours des réservations de ce trajet
    for (const reservation of trajet.reservation || []) {
      for (const enregistrement of reservation.enregistrer) {
        // Classification par catégorie selon le type_num
        if (enregistrement.type_num <= 3) {
          passagerTemp[trajet.date].catA += enregistrement.quantite;
          totalPassagerTemp.catA += enregistrement.quantite;
        } else if (enregistrement.type_num <= 5) {
          passagerTemp[trajet.date].catB += enregistrement.quantite;
          totalPassagerTemp.catB += enregistrement.quantite;
        } else {
          passagerTemp[trajet.date].catC += enregistrement.quantite;
          totalPassagerTemp.catC += enregistrement.quantite;
        }
      }
    }
  }

  // === MISE À JOUR DES ÉTATS ===
  setPrixTotal(prixTotalCalculé);
  setRevenusParDate(revenusTemp);
  setPassagerParDate(passagerTemp);
  setTotalPassagers(totalPassagerTemp);
```

```
  } catch (err) {
    setError("Une erreur est survenue lors du calcul des revenus.");
    console.error("Erreur:", err);
  } finally {
    setLoading(false);
  }
};
```

Sur cette page on peut modifier la période souhaitée, consulter les revenus et le voir sous forme de graphique, consulter les passages transportés sur la période ainsi que les catégories concernées.

On récupère les liaisons existantes avec fetchLisaions:

```
/**
 * Récupère toutes les liaisons avec leurs informations détaillées
 * Joint les données des ports de départ/arrivée et du secteur
 */
const fetchLiaisons = async () => {
  setLoading(true);
  setError(null);

  try {
    // Récupération des liaisons de base
    const { data: liaisons, error: errorLiaisons } = await supabase
      .from("liaison")
      .select("*");

    if (errorLiaisons) throw errorLiaisons;

    // Enrichissement de chaque liaison avec les détails des ports et secteur
    const liaisonsPorts = await Promise.all(
      liaisons.map(async (liaison) => {
        // Récupération du port de départ
        const { data: depart, error: errorDepart } = await supabase
          .from("port")
          .select("*")
          .eq("id", liaison.depart_id)
          .single();

        // Récupération du port d'arrivée
        const { data: arrivee, error: errorArrivee } = await supabase
          .from("port")
          .select("*")
          .eq("id", liaison.arrivee_id)
          .single();

        // Récupération du secteur
        const { data: secteur, error: errorSecteur } = await supabase
          .from("secteur")
          .select("*")
          .eq("id", liaison.secteur_id)
          .single();

        if (errorDepart || errorArrivee) {
```

```
          throw errorDepart || errorArrivee;
        }

        // Retour de la liaison enrichie avec toutes les informations
        return {
          ...liaison,
          depart: depart,
          arrivee: arrivee,
          secteur: secteur,
        };
      })
    );

    setLiaisons(liaisonsPorts);
  } catch (error) {
    setError("Erreur lors de la récupération des liaisons.");
    console.error("Erreur:", error);
  } finally {
    setLoading(false);
  }
};
```

Pour créer une liaison, c'est dans handleSubmit que tout se passe:

```
/**
 * Gère la soumission du formulaire pour créer une nouvelle liaison
 * @param {Event} e - Événement de soumission du formulaire
 */
const handleSubmit = async (e) => {
  e.preventDefault();
  setLoadingCreate(true);

  // Vérification que tous les champs sont remplis
  if (
    selectedSecteur !== "" &&
    selectedPortArrivee !== "" &&
    selectedPortDepart !== "" &&
    distance !== ""
  ) {
    // === VALIDATIONS MÉTIER ===

    // Vérification que les ports de départ et d'arrivée sont différents
    if (selectedPortArrivee === selectedPortDepart) {
      showNotification(
        "error",
        "Le port d'arrivée ne peut pas être le même que celui de départ"
      );
      setLoadingCreate(false);
      return;
    }

    // Vérification que la distance est positive
    if (distance <= 0) {
      showNotification(
```

```
        "error",
        "Une liaison ne peut pas avoir une distance inférieure à 0"
      );
      setLoadingCreate(false);
      return;
    }

    try {
      // Vérification de l'unicité de la liaison
      const exist = await isExisting(
        selectedSecteur,
        selectedPortDepart,
        selectedPortArrivee
      );

      if (exist) {
        showNotification("error", "La liaison existe déjà");
      } else {
        // === INSERTION EN BASE DE DONNÉES ===

        const { data, error } = await supabase.from("liaison").insert([
          {
            secteur_id: selectedSecteur,
            depart_id: selectedPortDepart,
            arrivee_id: selectedPortArrivee,
            distance: Number(distance), // Conversion en nombre
          },
        ]);

        if (error) {
          showNotification("error", `Erreur d'insertion : ${error.message}`);
        } else {
          // === SUCCÈS DE LA CRÉATION ===

          showNotification("success", "Liaison créée avec succès");
          fetchLiaisons(); // Rechargement de la liste
          resetForm(); // Reset du formulaire
          setIsAdding(false); // Sortie du mode ajout
        }
      }
    } catch (error) {
      showNotification("error", error.message);
    } finally {
      setLoadingCreate(false);
    }
  } else {
    showNotification("error", "Veuillez remplir tous les champs");
    setLoadingCreate(false);
  }
};
```

Lorsqu’on modifie une liaison, les champs de modifications se remplissent automatiquement avec les informations de la liaison existante et en prenant son id, puis c’esdt handleUpdate qui est concerné ici:

```
/**
 * Gère la soumission du formulaire pour mettre à jour une liaison
 * @param {Event} e - Événement de soumission du formulaire
 */
const handleUpdate = async (e) => {
 e.preventDefault();
 setLoadingUpdate(true);

 // Même logique de validation que pour la création
 if (
  selectedSecteur !== "" &&
  selectedPortArrivee !== "" &&
  selectedPortDepart !== "" &&
  distance !== ""
 ) {
  // === VALIDATIONS MÉTIER ===

  if (selectedPortArrivee === selectedPortDepart) {
   showNotification(
    "error",
    "Le port d'arrivée ne peut pas être le même que celui de départ"
   );
   setLoadingUpdate(false);
   return;
  }

  if (distance <= 0) {
   showNotification(
    "error",
    "Une liaison ne peut pas avoir une distance inférieure à 0"
   );
   setLoadingUpdate(false);
   return;
  }

  try {
   // Vérification de l'unicité (en excluant la liaison en cours d'édition)
   const exist = await isExisting(
    selectedSecteur,
    selectedPortDepart,
    selectedPortArrivee
   );

   if (exist && exist.code != editingLiaison.code) {
    showNotification("error", "La liaison existe déjà");
   } else {
    // === MISE À JOUR EN BASE DE DONNÉES ===

    const { data, error } = await supabase
     .from("liaison")
     .update({
      secteur_id: selectedSecteur,
      depart_id: selectedPortDepart,
      arrivee_id: selectedPortArrivee,
```

```
          distance: Number(distance),
        })
        .eq("code", editingLiaison.code); // Condition sur l'ID de la liaison

      if (error) {
        showNotification("error", `Erreur de mise à jour : ${error.message}`);
      } else {
        // === SUCCÈS DE LA MISE À JOUR ===

        showNotification("success", "Liaison mise à jour avec succès");
        fetchLiaisons(); // Rechargement de la liste
        resetForm(); // Reset du formulaire
        setIsEditing(false); // Sortie du mode édition
        setEditingLiaison(null); // Réinitialisation de la liaison en édition
      }
    }
  } catch (error) {
    showNotification("error", `Erreur de mise à jour : ${error.message}`);
  } finally {
    setLoadingUpdate(false);
  }
 } else {
  showNotification("error", "Veuillez remplir tous les champs");
  setLoadingUpdate(false);
 }
};
```

Il y a une double confirmation pour la suppression afin d'éviter les erreurs.

La suppression est gérée de cette manière:

```
/**
 * Confirme et exécute la suppression d'une liaison
 * Vérifie d'abord si la liaison n'est pas utilisée dans des trajets
 */
const confirmDelete = async () => {
 setLoadingDelete(true);
 setShowDeleteDialog(false);

 // === VÉRIFICATION DES DÉPENDANCES ===

 // Vérification si la liaison est utilisée dans des trajets
 const { data: trajets, error: errorTrajets } = await supabase
  .from("trajet")
  .select("*")
  .eq("idLiaison", liaisonToDelete.code);

 if (trajets && trajets.length > 0) {
  showNotification(
   "error",
   "Impossible de supprimer cette liaison : elle est utilisée dans un ou plusieurs trajets"
  );
  setLoadingDelete(false);
  return;
```

```
  }

  // === SUPPRESSION EN BASE DE DONNÉES ===

  const { error } = await supabase
    .from("liaison")
    .delete()
    .eq("code", liaisonToDelete.code);

  if (error) {
    showNotification("error", `Erreur de suppression : ${error.message}`);
  } else {
    showNotification("success", "Liaison supprimée avec succès");
    fetchLiaisons(); // Rechargement de la liste
  }

  // Nettoyage des états de suppression
  setLoadingDelete(false);
  setLiaisonToDelete(null);
};
```