Cahier de Conception Technique : Système de Gestion de Tontine

1. Architecture du Système

L'application suit une architecture **Single Page Application (SPA)** avec une séparation claire entre la logique métier et la présentation.

1.1. Architecture logicielle (React)

- **Composants (UI)** : Découpage en composants atomiques (Boutons, Cards) et composants de page.
- **Services (Data)** : Couche d'abstraction pour les appels API ou la gestion directe de la BD.
- **Gestion d'État (State Management)** : Utilisation de useState pour le local et Context API pour les données globales (utilisateur connecté, tontine active).

2. Conception de l'Interface (UI)

2.1. Arborescence de l'Application

1. **Dashboard** (Vue d'ensemble financière)
2. **Membres** (Liste, Ajout, Détails d'un membre)
3. **Tontines** (Configuration, Liste des tontines actives)
4. **Séances** (Gestion du calendrier, Prise de présence)
5. **Caisse & Crédits** (Saisie des prêts, Remboursements, Pénalités)
6. **Projets (FIAC)** (Suivi des investissements collectifs)

2.2. Maquette de la Page "Séance" (Composant Critique)

Ce composant doit permettre une saisie ultra-rapide en réunion :

- **Header** : Date, Lieu, Statut de la séance.
- **Tableau de présence** : Liste des membres avec cases à cocher.
- **Saisie des Cotisations** : Champ numérique pré-rempli avec le montant_cotisation par défaut de la tontine.
- **Action** : Bouton "Clôturer la séance" qui génère automatiquement les pénalités pour les absents.

3. Diagrammes de Flux Logiques

3.1. Processus de Cotisation (Sequence)

1. **Utilisateur** sélectionne une séance.
2. **Interface** affiche la liste des membres inscrits à la tontine liée.

3. **Utilisateur** saisit le montant versé.
4. **Système** vérifie si montant >= (nb_parts * montant_cotisation).
5. **Système** enregistre l'entrée dans la table COTISATION.
6. **Interface** met à jour le solde global en temps réel.

3.2. Contrôle de la Tontine Optionnelle (Règle Métier)

- **Condition** : Somme(Gains_Perçus) <= Somme(Cotisations_Prévues).
- **Implémentation** : Avant de valider un membre comme bénéficiaire du tour (id_membre_beneficiaire), une fonction frontend calcule le ratio pour bloquer l'action si la condition est violée.

4. Modèle de Données Client (Mapping)

Pour faciliter le développement React, voici la structure d'objet JSON attendue pour un **Membre** (basée sur votre MLD) :

```
{
  "id_membre": "integer",
  "nom": "string",
  "prenom": "string",
  "telephone": "string",
  "email": "string",
  "date_inscription": "date",
  "stats": {
    "total_cotise": "decimal",
    "credits_en_cours": "integer",
    "penalites_dues": "decimal"
  }
}
```

5. Algorithmes Spécifiques

5.1. Calcul des Pénalités

```
Fonction calculerPenalite(membre, seance) {
  Si (membre.estAbsent && !membre.aJustifie) {
    Appliquer penalite(type: "Absence", montant: 500);
  }
  Si (cotisation.date > seance.dateLimite) {
    Appliquer penalite(type: "Retard", montant: membre.cotisation * 0.1);
  }
}
```

6. Sécurité et Validation

- **Validation des Types** : Utilisation de PropTypes ou TypeScript pour garantir l'intégrité des données passées aux composants.
- **Nettoyage des entrées** : Sanitize des inputs de recherche pour éviter les injections XSS.
- **Gestion des erreurs** : Intercepteur pour les échecs réseau (Error Boundaries).

7. Plan de Tests (Frontend)

1. **Tests Unitaires** : Calcul des intérêts de crédit, calcul du montant total distribué.
2. **Tests d'Intégration** : Création d'un membre -> Inscription à une tontine -> Paiement d'une cotisation.
3. **Tests de Performance** : Chargement de la liste des membres (> 100 membres).