Application de billetterie – Coupe du Monde 2026

Lien du repo : https://github.com/babachanakh-kateryna/coupe_du_monde_2026_react_spa.git

Démo live :
https://drive.google.com/file/d/19M7d42QGAj4r9n7FRv2yxf0rqRkmImvS/view?usp=sharing

Membres de l'équipe – Groupe 3

1. **BABACHANAKH Kateryna**
2. **TCHUENKAM Yannick**
3. **SATERIH Hamza**

Objectif

Ce projet a consisté à développer une application web front-end de type SPA (Single Page Application) en React et TypeScript. L'objectif principal était de permettre à un utilisateur de consulter les matchs de la Coupe du Monde 2026, de réserver des billets selon les catégories disponibles, de gérer un panier d'achat et de simuler le paiement. L'application communique avec une API REST distante, fournie dans le cadre du projet.

Technologies utilisées

1. React 18+ — utilisé pour le développement de l'application web en SPA (Single Page Application) avec des composants fonctionnels.
2. TypeScript — pour bénéficier du typage statique, améliorer la fiabilité du code et structurer les données échangées avec l'API REST.
3. Bun — utilisé comme environnement d'exécution pour le développement local.
4. Gestion d'état global — implémentée à l'aide de useContext et useReducer pour gérer l'état de l'utilisateur connecté et du panier de manière centralisée.
5. React Router — utilisé pour la navigation entre les différentes pages de l'application.
6. Axios — utilisé pour effectuer les requêtes HTTP vers l'API REST, avec gestion des appels asynchrones via async/await.
7. Material UI (MUI) — bibliothèque UI utilisée pour les composants graphiques (boutons, formulaires, popups...).
8. Bootstrap — utilisé pour compléter la mise en page et la responsivité via des classes utilitaires.
9. CSS personnalisé — pour certains éléments spécifiques non couverts par les bibliothèques externes.

Architecture de l'application

L'application est structurée de manière modulaire. Chaque domaine fonctionnel est organisé dans un dossier spécifique pour assurer une meilleure lisibilité, maintenabilité et évolutivité du projet.

1. Répertoire *api/* - ce dossier contient toute la logique liée aux appels à l'API REST :

- *services/* : services de communication avec l'API, séparés par domaine (*AuthService.ts*, *MatchService.ts*, *TicketService.ts*, etc.).
- *types/* : tous les types TypeScript utilisés pour typer les réponses et requêtes HTTP (*Match.ts*, *Team.ts*, *Tickets.ts*, etc.).

- *api.ts* : configuration de base d'Axios.

2. Répertoire *components/* - ce dossier regroupe tous les composants React, organisés par page ou domaine fonctionnel :

- *Accueil/* : composants de la page d'accueil.
- *Common/* : composants réutilisables comme *ToastNotification*.
- *Equipe/* : composants liés à l'affichage des équipes.
- *Groupe/* : composants pour la gestion des groupes.
- *Match/* :
    a) *FilterPopUp/* : pop-up pour filtre des matchs.
    b) *MatchCalendar/* : calendrier des matchs.
    c) *MatchDetailPopUp/* : pop-up avec les détails d'un match.
    d) *ReserveTicketPopUp/* : pop-up pour la réservation de billets.
    e) *PageMatchs.tsx* : page d'affichage des matchs.
- *Panier/* : composants pour la gestion du panier d'achat.
- *Profil/* : composants liés à l'affichage des informations du profil utilisateur.
- *Connexion.tsx* : composant pour la connexion et l'inscription.
- *NavbarWorldCup.tsx* : barre de navigation principale.
- *hooks/AuthContext.tsx* : gestion centralisée de l'état d'authentification et du panier via *useContext* et *useReducer*.

Chaque composant possède généralement son propre fichier *.css* pour isoler les styles.

3. Fichiers racine

- *App.tsx* : composant racine de l'application.
- *AppRoutes.tsx* : définition de toutes les routes avec react-router-dom.
- *main.tsx* : point d'entrée de l'application.
- *App.css*, *index.css* : styles globaux.

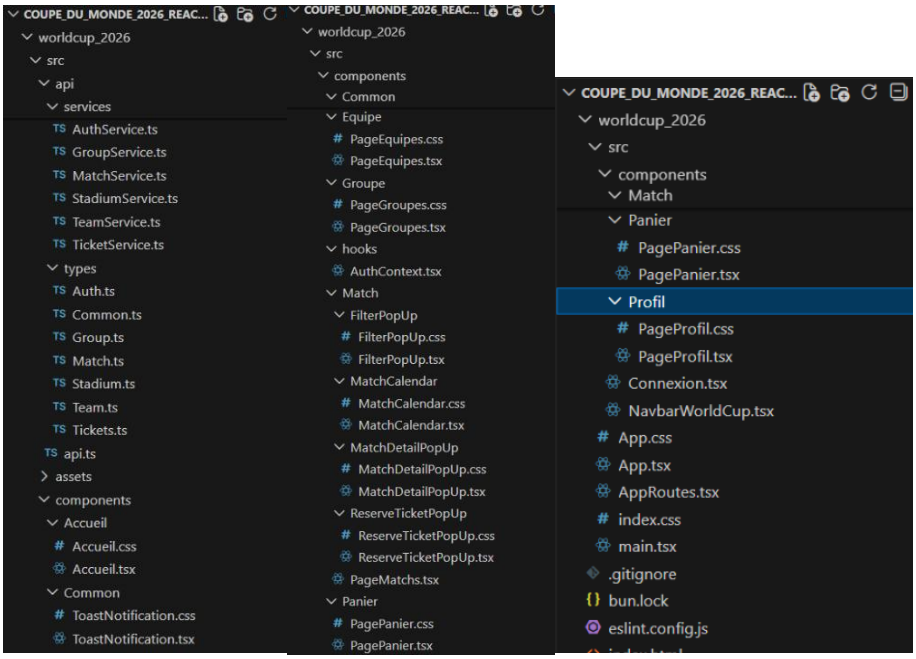

Figure 1. Architecture de l'application

Cette architecture assure :

1. Une séparation claire des responsabilités.
2. Une réutilisabilité des composants.
3. Une extensibilité pour de futures fonctionnalités.
4. Une cohérence dans la gestion des données via un typage strict avec TypeScript.

Fonctionnalités implémentées

1. Liste des matchs (calendrier)	Consultation des matchs planifiés via un calendrier visuel interactif. Affichage détaillé pour chaque match (pop-up) : 1) Équipes (drapeaux, noms, codes) 2) Date et heure (format français) 3) Lieu (ville, pays, stade) 4) Icônes contextuelles (stade, calendrier, horloge) 5) Statut du match (upcoming, ongoing, finished) avec couleur dynamique 6) Phase de la compétition (groupes, finale…) 7) Liste des catégories de billets (disponibilité, prix, places restantes) 8) Bouton « Réserver maintenant » qui ouvre le popup de réservation. Si l'utilisateur n'est pas connecté, un toast d'alerte lui indique qu'il doit se connecter pour réserver. Filtres avancés : 1) Par date (sélecteur de date) 2) Par équipe (recherche avec autocomplétion + drapeau) 3) Par groupe (ID du groupe) Feedback utilisateur (toast si aucun match trouvé) Design responsive (grille adaptative avec défilement horizontal)
2. Liste des équipes	Affichage en grille de toutes les équipes Carte par équipe avec drapeau, nom complet, code, continent, confédération (couleur codée), groupe Filtre par continent (liste déroulante) Tri alphabétique Chargement avec spinner + gestion d'erreur
3. Liste des groupes	Affichage des groupes Carte par groupe avec lettre du groupe en grand et liste des 4 équipes (drapeau + nom + code) Filtre par nom de groupe (sélecteur) Mise en page équilibrée (grille responsive) Chargement avec spinner + gestion d'erreur
4. Système de panier	Ajout de billets (popup de réservation) : 1) Sélection de catégorie (liste déroulante avec prix) 2) Choix de la quantité (1 à 6 billets) 3) Validation en temps réel (places disponibles, bouton désactivé si non valide) 4) Ajout au panier et mise à jour du panier global 5) Toast de confirmation : « X billets ajoutés au panier ! » 6) Gestion des erreurs (ex: places épuisées) Page Panier : 1) Pour chaque billet ajouté, les informations suivantes s'affichent : match (équipes, date, lieu), catégorie sélectionnée et prix du billet.

	2) Possibilité de supprimer un billet du panier à tout moment. 3) Affichage du sous-total, frais (simulés à 0 €), et total général. 4) Calcul en temps réel des montants en fonction du contenu du panier. 5) Un bouton permet de valider la commande (« Payer »). 6) Une confirmation visuelle (toast) s'affiche après un paiement réussi, indiquant le nombre de billets confirmés. 7) Le panier est ensuite vidé et l'utilisateur redirigé vers sa page de profil. 8) Une date d'expiration de la réservation est affichée. 9) Gestion des erreurs (ex : le nombre de places demandées dépasse la disponibilité, ou un billet ne peut pas être supprimé ou le paiement échoue) 10) Le nombre total de billets dans le panier est affiché dans l'interface (*navbar*). Ce nombre est synchronisé grâce à un contexte global (*useContext + useReducer*).
5. Persistance du panier	Sauvegarde à chaque modification (ajout, suppression, paiement) Chargement local si API indisponible Nettoyage après paiement ou déconnexion
6. Authentification	Un formulaire d'authentification avec deux onglets : « Se connecter » et « Créer un compte ». Les champs obligatoires sont l'adresse e-mail et le mot de passe (ainsi que le prénom, le nom et la date de naissance pour l'inscription). Connexion à l'API REST : 1) Les données sont envoyées à l'API via *axios.post*. 2) L'API répond avec un cookie HTTP-only contenant le JWT access token. 3) En cas de succès, récupération de l'utilisateur authentifié. Récupération et gestion du token JWT : même si le token n'est pas lisible en JavaScript (car sécurisé en HTTP-only), il est automatiquement utilisé par le navigateur : Toutes les requêtes ultérieures avec axios sont faites avec *withCredentials: true*, ce qui permet au navigateur d'envoyer automatiquement le token. Cela respecte les bonnes pratiques de sécurité : le token n'est pas stocké manuellement dans le *localStorage*, mais géré par le navigateur via un cookie sécurisé. Un système de gestion d'état global avec useContext et useReducer permet de : 1) Savoir si l'utilisateur est authentifié (isAuthenticated) 2) Récupérer les infos utilisateur (user) 3) Mettre à jour automatiquement l'interface après connexion/déconnexion Des notifications (toasts) informent l'utilisateur en cas d'erreur de connexion ou de succès.
7. Passage et consultation de commandes	Validation du panier auprès de l'API REST : 1) L'utilisateur clique sur « Payer » dans la page panier. 2) Une requête POST */tickets/pay-pending* est envoyée à l'API. 3) L'API simule le paiement, met à jour le statut des billets de *pending_payment* à *confirmed* et renvoie le nombre de billets confirmés ainsi que le montant total. Simulation du paiement :

	1) Aucun système de paiement réel n'est intégré (pas de Stripe, PayPal…). 2) Le paiement est simulé côté serveur : le backend valide immédiatement la transaction et renvoie une réponse positive. 3) Le panier est vidé (côté client et serveur) après confirmation. Consultation d'un historique des commandes passées : 1) La page */profil* est accessible uniquement aux utilisateurs connectés. 2) Chargement simultané des données utilisateur et de tous les billets. 3) Les billets sont regroupés par statut : en attente (*pending_payment*), payées (*confirmed*), utilisées (*used*) 4) Affichage sous forme d'onglets avec icônes et compteurs dynamiques. 5) Pour chaque billet : match (équipes, date/heure format français), stade, catégorie, prix, QR Code (affiché uniquement si *status === 'confirmed'*), statut visuel avec couleur 6) Statistiques globales en haut de page (total des billets, billets payés, billets utilisés) 7) Gestion des états de chargement et d'erreur (spinner, message d'échec). 8) Génération du QR Code côté client : L'API renvoie uniquement une chaîne unique (qrCode: "QR-Q8NT53NKL"), donc le QR est généré en temps réel via la bibliothèque *qrcode.react*
8. Fonctionnalité libre	Intégration de l'API OpenWeatherMap pour afficher la météo actuelle de la ville du stade directement dans le pop-up de détail du match. Utilisation d'API tierce : Appel à *https://api.openweathermap.org/data/2.5/weather* avec la ville et le code pays du stade.

Toutes les tâches obligatoires et facultatives ont été réalisées.

Gestion d'état global

Pour cette application, on a mis en place une gestion d'état global, car plusieurs informations importantes doivent être accessibles depuis n'importe quel composant :

- si l'utilisateur est connecté,
- le nombre de billets dans le panier.

Au lieu de gérer ces données séparément dans chaque page avec *useState* et de les transmettre via de nombreux props, on a choisi *useContext* + *useReducer*. Cette solution permet de centraliser tout l'état dans un seul endroit et de rendre les données accessibles partout dans l'application. Il assure des mises à jour cohérentes après la connexion, la déconnexion ou les modifications du panier, et simplifie le code et empêche la duplication.

Problèmes rencontrés

1. Lors de l'appel GET /matches, tous les matchs retournés par l'API avaient un nombre de *availableSeats* strictement supérieur à 0. Aucun match n'était donc marqué comme indisponible. Cela

a compliqué la partie design, car il était impossible de tester ou d'afficher correctement l'état visuel d'un match sold out (bouton désactivé, message d'indisponibilité, etc.).

2. Pendant le projet, j'ai constaté que les informations affichées dans Swagger ne correspondaient pas toujours aux réponses réelles obtenues via Postman. Certains champs et formats étaient différents. Cela montre que la documentation Swagger n'était pas mise à jour, ce qui a entraîné plusieurs difficultés :

- perte de temps à vérifier chaque requête via Postman,
- incompréhensions sur certains types de données,
- risque d'intégrer un format incorrect côté front-end.

Pour un développeur front-end, il est essentiel d'avoir une documentation Swagger à jour, car elle sert de référence pour comprendre la structure des données, construire les interfaces et éviter des erreurs d'intégration.

Retour d'expérience

Ce projet nous a permis d'améliorer plusieurs compétences techniques et organisationnelles. Nous avons appris à :

- manipuler une API REST complexe avec filtres, paramètres et modèles imbriqués,
- gérer l'état global d'une application avec *useContext* et *useReducer*,
- organiser une architecture React modulaire et maintenable,
- intégrer des cookies HTTP-only pour l'authentification sécurisée,
- gérer les erreurs réseau, les loaders et le feedback utilisateur,
- construire une interface responsive et professionnelle,
- manipuler des données dynamiques en temps réel (panier, billets, statut des matchs).
- s'adapter aux incohérences entre documentation (Swagger) et API réelle,

En résume, nous avons appris à gérer un projet proche d'un vrai cas professionnel : API distante, authentification, panier, états globaux, UI complexe, erreurs réelles et documentation incomplète.