

Étude de cas détaillée · 8 min de lecture

Spotitube

Une app sociale de musique, films et séries pour Android (façon YouTube Music), avec une console d'administration web testée et un back-end durci.

RÔLE

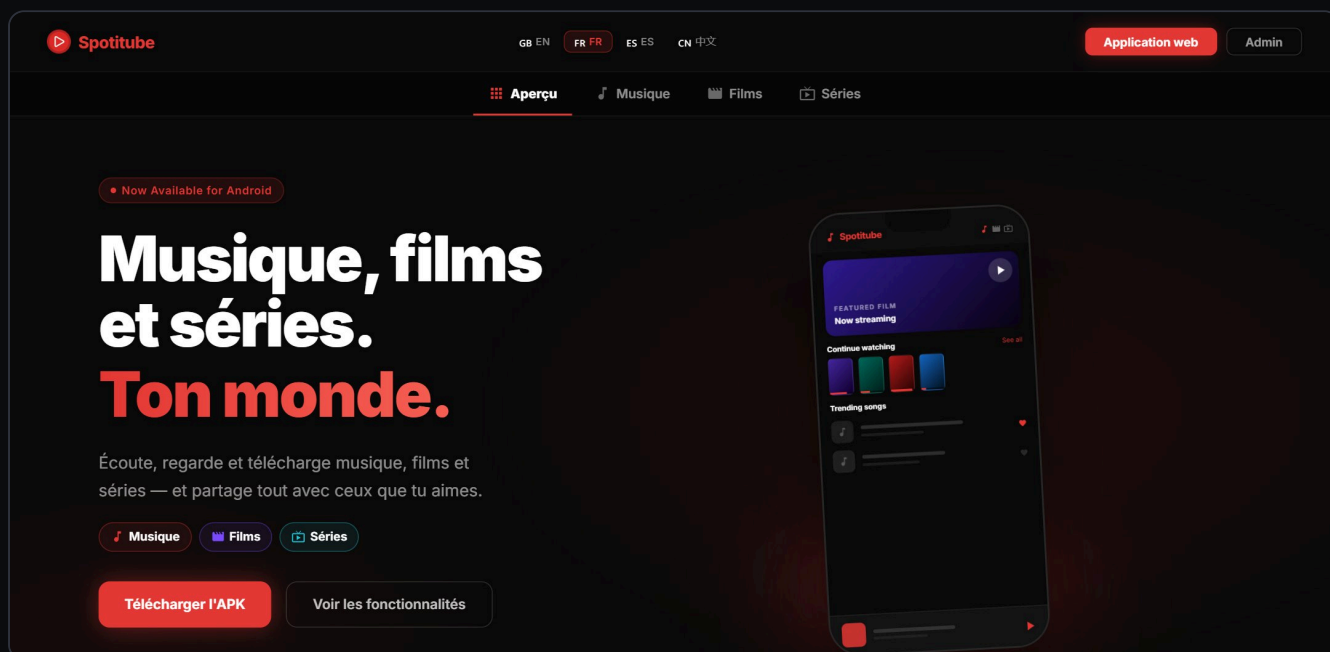
Conception · architecture ·
développement

PLATEFORME

Android · Admin web

ÉTAT

v1.4.3 · ~136 jours de dev · testé



Spotitube web — page d'accueil

Le produit

Spotitube est une application sociale de **musique, films et séries** : écouter et télécharger, créer des playlists, suivre ses amis, voir ce qu'ils écoutent et regarder ensemble. Elle existe en application Android et en application web React, avec une **console d'administration** web pour gérer le catalogue et un backend Firebase avec Cloud Functions.

C'est mon produit le plus abouti — **v1.4.3**, environ **920 commits** et une vraie suite de tests — et celui qui m'a le plus appris sur l'exploitation d'un produit, pas seulement sa construction.

Architecture

- **Android** : une application en couches données / domaine / UI — Kotlin, Jetpack Compose, Hilt pour l'injection de dépendances, Room, Retrofit et un lecteur Media3/ExoPlayer.
- **Web** : React 18, TypeScript et Vite, avec MUI et TanStack Query ; la console d'administration ajoute Recharts, le mode PWA et l'i18n.
- **Backend** : Firebase Auth, Firestore, Storage et FCM, plus des Cloud Functions pour tout ce qu'un client ne doit pas pouvoir faire.
- **Médias** : l'audio et la vidéo sont servis depuis Cloudflare R2 et un CDN plutôt que Firebase Storage.

Une interface de lecture, deux moteurs

Les films viennent de trois sources — le CDN, un téléchargement local ou YouTube — et chaque plateforme les lit avec **deux moteurs** : un lecteur natif pour le CDN et les fichiers locaux, et un lecteur IFrame/WebView pour YouTube. La règle sur laquelle repose tout le lecteur est inscrite dans son contrat : *l'interface ne demande jamais quel moteur elle utilise*. Chaque moteur s'adapte à une interface d'état et de commandes commune, et l'interface ne consomme que celle-ci.

Avant d'étendre le lecteur, j'ai audité les deux moteurs **à partir du code**, et non de mémoire, puis corrigé le plan là où il se trompait :

Capacité	CDN / local	YouTube
Recherche précise	Oui	Approximative — cale sur les images clés
Mises à jour du temps	Événements	Interrogées — l'API IFrame n'en a pas
Image dans l'image	Oui	Non — c'est une API de <video>
Lecture en arrière-plan (Android)	Oui	Non — elle tourne dans une WebView
Hors ligne	Fichiers locaux seulement	Non

Un abonnement, plusieurs appareils

Un abonnement peut servir sur plusieurs appareils : il faut donc décider qui garde une place et qui est déconnecté. Cette règle décide qui paie ; je l'ai donc **extraite du point d'entrée des Cloud Functions dans son propre module** pour la tester sans Firebase.

Un détail illustre la réflexion nécessaire : l'app se synchronise à chaque lancement et reprise, et chaque écriture sur un abonnement relance l'écouteur sur tous les autres appareils. Le signal de présence d'une session n'est donc enregistré **qu'une fois par heure au plus** — bien plus fin que la fenêtre d'inactivité de plusieurs jours qu'il alimente, sans tempête d'écritures entre appareils.

Protéger un vrai budget

Certains outils d'administration appellent l'API YouTube Data, qui accorde **10 000 unités par jour** et en facture **100 pour une seule recherche**. Un après-midi de recherches enthousiastes peut l'épuiser, et l'erreur ressemble alors à une clé d'API cassée. Les limites de débit de ces fonctions rendent cet épuisement impossible par accident — et échouent avec une phrase qui explique ce qui s'est passé, au lieu de lancer quelqu'un à la recherche d'un bug dans une clé qui fonctionne.

Qualité et sécurité

- **139 fichiers de tests** entre l'application web (Vitest), les Cloud Functions (test runner de Node) et Android (JUnit).
- `npm run ci` enchaîne vérification des types, tests et build de production ; les **tests des règles de sécurité** Firestore tournent sur l'émulateur.
- Une **revue de sécurité** (juin 2026) a couvert les règles Firestore, l'authentification des fonctions et les règles Storage. Elle a révélé que les règles Storage n'existaient que dans la console — pas dans le code versionné — et l'a corrigé avec des règles propriétaire uniquement, images seulement, limitées à 5 Mo. Elle a aussi signalé un point d'accès de migration protégé par un jeton dans son URL, avec des options concrètes pour le supprimer.
- Un **blocage des utilisateurs appliqué côté serveur** a remplacé un simple filtre client : des Cloud Functions répliquent chaque blocage, désabonnent dans les deux sens et retirent les collaborations ; les règles Firestore l'appliquent aux commentaires, salons et mentions ; des tests de règles le couvrent. Planifié et livré en 7 jours.

La suite

Aujourd'hui, l'abonnement protège les **applications**, pas les fichiers audio eux-mêmes. La solution est entièrement conçue et planifiée avant le lancement public : une Cloud Function émet des URL signées HMAC de courte durée après vérification de l'abonnement et de la place de l'appareil, et un Worker Cloudflare devant un bucket privé les vérifie — avec une clé de cache limitée au chemin, pour que tous les utilisateurs partagent le même cache. Elle n'est pas encore construite, et je ne prétends pas le contraire.

Ce que j'en retiens

- Isoler les règles qui coûtent de l'argent ou bloquent des utilisateurs, et les tester à part.
- Auditer ce que fait réellement le code avant de planifier par-dessus.
- Écrire les messages d'erreur pour la personne qui les lira.

BENOIT PRESLY NDONG OKE

✉ oke.benoit@gmail.com 🌐 okebenoithub.web.app <> github.com/OkeBenoitHub

🌐 linkedin.com/in/presly-benoit-ndong-oke-27987b117