

# Notifications et Push

## Vue d'ensemble

Toutes les notifications passent par `NotificationService.notify(userId, notificationType, payload, isImportant)`. Le compte reçoit une entrée dans son inbox Redis, un événement Socket.IO sur `/account` et une tentative de push FCM vers ses installations. Le Web et l'application mobile lisent l'inbox; le jeu produit des notifications via l'API.

## Flux de création et responsabilités

```
flowchart LR
    S[Service métier ou GameNotificationService] --> N[NotificationService.notify]
    N --> R[(Inbox Redis du compte)]
    N --> E[EventEmitter notification.created]
    E --> G[AccountGateway /account]
    G --> A[AccountSocketService]
    A --> U[UserNotificationService et interface]
    N --> F[FirebaseService]
    F --> T[(UserFcmToken)]
    F --> D[FCM vers installations du compte]
```

1. Vérifier que le compte destinataire existe.
2. Construire l'enveloppe avec UID unique, puis sauvegarder l'inbox.
3. Émettre `notification.created`; `AccountGateway` relaie la notification vers `user_<userId>` dans le namespace `/account`.
4. Construire le titre et le corps localisés, puis tenter le push. Une erreur de push est journalisée sans annuler l'entrée déjà sauvegardée.

## Contrat partagé

`NotificationDto` côté backend et `UserNotification` côté Angular partagent les champs `uid`, `userId`, `notificationType`, `timestamp`, `isRead`, `isImportant`. Les champs métier, par exemple

`ticketId` ou `simUid`, sont placés directement à côté de cette enveloppe. Les champs d'enveloppe définis par le serveur écrasent ceux du payload.

L'UID combine le compte, le timestamp et quatre octets aléatoires pour distinguer plusieurs notifications créées dans la même milliseconde. Le texte push utilise les clés `uniksim.push_<type>_title/body`, les placeholders du payload et un texte de secours. Les valeurs FCM sont converties en chaînes; `notificationUid` conserve la compatibilité avec les clients historiques.

## Inbox Redis et API du compte

La clé est `user_<id>_notifications`. `LPUSH` conserve les plus récentes en premier, `LTRIM` limite l'inbox à 100 entrées et `EXPIRE` fixe 10 519 200 secondes de rétention. Chaque insertion renouvelle le TTL de toute la liste; ce n'est pas une expiration indépendante par notification.

| Action      | Endpoint                                               | Comportement                                         |
|-------------|--------------------------------------------------------|------------------------------------------------------|
| Charger     | <code>GET /notifications</code>                        | Retourne l'inbox du compte authentifié.              |
| Lu / non lu | <code>POST /notifications/:notificationUid/read</code> | Met à jour l'entrée par UID avec <code>LSET</code> . |
| Supprimer   | <code>DELETE /notifications/:notificationUid</code>    | Retire l'entrée avec <code>LREM</code> .             |

Ces routes utilisent `AuthGuardUser` et l'identité `request.user.sub`.

## Frontend et temps réel

`AccountSocketService` reçoit `notification.created`, déclenche le son et expose l'observable. `AppComponent` transmet la notification à `UserNotificationService.upsert()`. Ce service maintient le `BehaviorSubject`, déduplique par UID et dérive le nombre de notifications non lues. Le chargement initial et les actions lu/suppression utilisent HTTP. `SupportSocketService` gère les conversations et tickets; la réception des notifications appartient au service du compte.

## Installations et livraison FCM

`PUT /push/installations/:installationId` enregistre `token` et `platform` pour le compte authentifié. Dans une transaction, le service supprime les inscriptions périmées portant le même token puis crée ou met à jour `UserFcmToken`. `DELETE /push/installations/:installationId` retire l'installation de ce compte.

La livraison charge toutes les installations du compte, déduplique les tokens et les envoie par lots via `sendEachForMulticast`. Les tokens invalides ou désenregistrés entraînent la suppression de leurs installations. Le modèle actuel ne contient plus `UserFcmSubscription` ni de filtrage des appareils par `productCode`. Si les variables Firebase sont absentes, le push est désactivé mais l'inbox reste utilisable.

# Notifications envoyées par le jeu

`POST /saveNotification` reçoit `userId` (profil jeu du destinataire), `notificationType`, `extraData` et éventuellement `isImportant`. `GameNotificationService` traduit le profil actif `ProductsCode.game0id` en compte Web et vérifie la simulation BrainCloud indiquée par `extraData.simUid`.

L'expéditeur doit posséder un profil actif membre de cette simulation; le destinataire doit aussi en être membre. Le serveur utilise `teacherUid` et les clés de `students` du document pour cette vérification. Les types réservés au support sont refusés, le payload JSON est limité à 2 048 caractères et les identifiants à 255 caractères. La vérification BrainCloud expire après dix secondes. Une fois validée, la notification suit exactement le flux commun `notify()`.

## Fichiers principaux

- `src/service/notification.service.ts`, `game-notification.service.ts`, `firebase.service.ts`
- `src/dto/notification.dto.ts`, `src/gateway/account.gateway.ts`,  
`src/guards/socketAuth.middleware.ts`
- `src/app.controller.ts` et `NOTIFICATION_CONFIG` dans `src/node.js/server/constantsVariable.js`
- Angular : `service/account/account-socket.service.ts`, `service/realtime/realtime-socket.ts`,  
`service/notifications/user-notification.service.ts`, `Models/Notification.ts`
- Tests : `test/notification/notification.service.test.ts` et `game-notification.service.test.ts`

Voir aussi [Support temps réel](#).

---

Revision #4

Created 2026-09-17 21:03:11 UTC by Mathis

Updated 2026-09-30 14:41:41 UTC by Mathis