

Intégrations et systèmes transversaux

Intégrations qui traversent plusieurs couches : Unity WebGL, support temps réel, IA/RAG, notifications, fichiers et services externes.

- [Unity WebGL et communication avec le Web](#)
- [Notifications et Push](#)
- [Services externes et dépendances](#)

Unity WebGL et communication avec le Web

Vue d'ensemble

Le jeu Unity WebGL est intégré au site Angular et communique avec le backend NestJS. Cette page présente les responsabilités de chaque couche et les principaux points à vérifier lorsqu'une modification touche le jeu et le Web.

Architecture générale

flowchart LR

USER[Utilisateur] --> ANG[Angular / PageWebgl]

ANG --> BUILD[Build Unity WebGL]

BUILD --> UNITY[Runtime Unity]

UNITY -->|HTTP| GAME[GameController]

GAME --> GS[GameService]

GS --> BC[BrainCloud]

GS --> REDIS[(Redis)]

GS --> STORAGE[storage / images / logs]

ANG -->|Auth / session / navigation| NEST[NestJS]

NEST --> ANG

Côté Angular

`PageWebglComponent` est l'écran qui héberge le jeu pour plusieurs routes, notamment `/game`, `/demo`, `/alpha`, `/alpha2` et `/demoLive`.

Le build Unity est livré dans les assets frontend. Le frontend Angular gère le contexte Web autour du jeu : navigation, authentification, chargement et communication avec les autres fonctions du site.

Côté backend

`GameController` dans `uniksim-back-end/src/app.controller.ts` est protégé par `AuthGuardGame`. Il délègue la logique à `GameService`.

Les endpoints couvrent notamment :

- version du build;
- informations de code produit;
- textes et traductions;
- écoles;
- token Web;
- heartbeat et présence en jeu;
- classements;
- images utilisateur;
- journalisation de crash.

Flux heartbeat

```
sequenceDiagram
    participant U as Unity WebGL
    participant G as GameController
    participant S as GameService / server.js
    participant R as Redis

    U->>G: heartBeat(simUid, userId, token)
    G->>S: heartBeat(...)
    S->>R: Vérifie session / présence
    R-->>S: État session
    S->>R: Met à jour présence + expiration
    S-->>G: Version / heartbeats / sessionTime
    G-->>U: Réponse
```

Build et fichiers statiques

Les assets Unity font partie du frontend Angular. Le build Angular de production est ensuite généré dans `uniksim-back-end/src/front`, puis servi par NestJS avec `ServeStaticModule`.

Voir **Build et déploiement Web** pour l'ordre complet de build.

Quand modifier plusieurs projets

Une modification du contrat Unity/Web nécessite souvent de vérifier plusieurs couches :

flowchart LR

CHANGE[Changement Unity / Web] --> UNITY[Code Unity]

CHANGE --> ANG[Angular / PageWebgl]

CHANGE --> API[GameController]

API --> SERVICE[GameService]

SERVICE --> DATA[BrainCloud / Redis / stockage]

UNITY --> TEST[Test WebGL]

ANG --> TEST

API --> TEST

Fichiers à connaître

- `uniksimFrontEnd/src/app/Pages/PageWebgl/`
- `uniksimFrontEnd/src/assets/unity/`
- `uniksim-back-end/src/app.controller.ts` — `GameController`
- `uniksim-back-end/src/service/game.service.ts`
- `uniksim-back-end/src/guards/authGame.guard.ts`
- `uniksim-back-end/src/node.js/server/server.js`
- `uniksim-back-end/src/node.js/server/backend_braincloud.js`

Points d'attention

- Ne pas modifier seulement le frontend si le format attendu par Unity ou l'API change.

- Vérifier le guard et le contexte d'instance (live/dev/demo) pour les endpoints du jeu.
- Les heartbeats ont des effets sur Redis et la présence; les traiter comme un flux runtime important.
- Après un nouveau build Unity, vérifier que les nouveaux fichiers WebGL sont bien ceux inclus dans les assets/build déployés.

Présence du compte et notifications après refactoring

`PageWebglComponent` transmet le jeu actif à `AccountSocketService.setGamePresence(productCodeId, gameId)` et l'efface à la sortie. Cette présence sur `/account` complète le heartbeat/session HTTP du jeu : les deux chemins ont des responsabilités distinctes. La reconnexion du compte restaure le jeu actif.

Le jeu envoie les notifications via `POST /saveNotification`; `userId` désigne le profil jeu destinataire et `extraData.simUid` la simulation. Ce contrat est traité par `NotificationController` avec `AuthGuardUser`, puis `GameNotificationService`, et non par `GameController`. Le serveur vérifie expéditeur et destinataire dans le document BrainCloud puis utilise l'inbox et le push du compte Web. Voir [Notifications et Push](#).

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](#).

Services externes et dépendances

Vue d'ensemble

Plusieurs fonctions du Web dépendent de services externes ou de couches historiques. Cette page sert de carte pour comprendre **quelle dépendance intervient où** et quels fichiers vérifier lorsqu'une intégration tombe en panne.

Carte des intégrations

flowchart LR

NEST[NestJS principal] --> MYSQL[(MySQL)]

NEST --> REDIS[(Redis)]

NEST --> BC[BrainCloud]

NEST --> FCM[Firebase / FCM]

NEST --> MAIL[Mail]

NEST --> AI[Serveur IA :8106]

ADMIN[Admin Server :8105] --> BC

ADMIN --> REDIS

AI --> CHROMA[(Chroma)]

AI --> PROVIDER[Provider IA]

UNITY[Unity WebGL] --> NEST

ANG[Angular] --> NEST

ANG <-->|Socket.IO| NEST

BrainCloud

BrainCloud est encore utilisé par plusieurs fonctions historiques liées aux simulations et au jeu.

Fichiers importants :

- `src/node.js/server/backend_braincloud.js`
- `src/node.js/server/server.js`
- `src/node.js/server/admin_server.js`
- `src/service/game.service.ts`

Lorsqu'un bug concerne une simulation, une copie de simulation, certains documents de jeu ou des opérations historiques admin, vérifier si le flux passe par BrainCloud avant de modifier MySQL.

Redis

Redis est utilisé pour les données runtime : présence, sessions, rankings, notifications et autres informations temporaires.

Fichier principal : `src/service/redis.service.ts`. Certaines couches historiques l'utilisent également depuis `src/node.js/server/`.

Firebase / FCM

Firebase Cloud Messaging assure les push notifications vers les appareils enregistrés.

Fichiers principaux :

- `src/service/firebase.service.ts`
- `src/service/notification.service.ts`
- `UserFcmToken` dans les entités TypeORM.

Voir **Notifications et Push** pour le flux complet.

IA / Chroma

Le serveur IA est séparé logiquement sous `src/node.js/server/ai/` et utilise un provider de génération, un provider d'embeddings et Chroma pour la recherche vectorielle.

Fichiers importants :

- `src/service/ai.service.ts`

- `src/node.js/server/ai/server.js`
- `src/node.js/server/ai/provider/`
- `src/node.js/server/ai/factory/`
- `src/node.js/server/ai/guard/`
- `src/embedding/`

Voir le chapitre **Support et IA** pour le pipeline RAG et l'ajout de providers.

Mail

Le backend possède une couche mail utilisée notamment pour le reset de mot de passe et les courriels applicatifs.

- `src/service/mail.service.ts`
- `src/mail/`
- `MailController` dans `app.controller.ts`

Comment diagnostiquer une intégration

flowchart TD

```

BUG[Erreur d'intégration] --> LOCAL{Le backend reçoit la requête ?}
LOCAL -- Non --> FE[Frontend / réseau / auth]
LOCAL -- Oui --> SERVICE[Identifier le service backend]
SERVICE --> EXT{Appel externe ?}
EXT -- Non --> INTERNAL[Logique interne / DB]
EXT -- Oui --> WHICH{Quelle dépendance ?}
WHICH --> BC[BrainCloud]
WHICH --> R[Redis]
WHICH --> F[Firebase]
WHICH --> AI[IA / Chroma]
WHICH --> M[Mail]
BC --> LOG[Logs + config + credentials]
R --> LOG
F --> LOG
AI --> LOG
  
```

Points d'attention

- Ne jamais documenter de clé API, mot de passe ou credential en clair.
- Vérifier la configuration d'environnement avant de conclure que le service externe est en panne.
- Plusieurs intégrations possèdent une couche historique et une couche NestJS moderne; identifier le chemin réellement utilisé avant de modifier le code.
- Lorsqu'une dépendance externe échoue, distinguer l'échec réseau/provider d'un problème de mapping ou de logique interne.

Contrats d'intégration actuels — septembre 2026

Gemini utilise `@google/genai` en mode Vertex AI via `provider/gemini/vertex_client.js`. La configuration utilise `GOOGLE_CLOUD_PROJECT`, `GOOGLE_CLOUD_LOCATION=global` et les credentials Google par défaut de l'environnement. Chroma utilise `CloudClient` avec `CHROMA_API_KEY`, `CHROMA_TENANT` et `CHROMA_DATABASE`; la collection configurée est `game`.

Firebase envoie les notifications à toutes les installations du compte stockées dans `UserFcmToken`. La validation des membres d'une simulation pour les notifications du jeu relève de `GameNotificationService` et de BrainCloud; les installations ne sont plus sélectionnées par subscription produit.

Socket.IO sépare `/account` (présence et notifications) et `/support` (conversations et tickets). L'initialisation IA en arrière-plan permet au serveur principal de démarrer même si un provider ou Chroma est temporairement indisponible.