

Architecture générale

Vue d'ensemble du système Web Uniksime : frontend Angular, backend NestJS, flux principaux, responsabilités des couches et points d'entrée.

- [Vue d'ensemble du système Web](#)
- [Structure des projets et dossiers](#)
- [Frontières entre code moderne et legacy](#)
- [Ajouter une fonctionnalité Web de bout en bout](#)

Vue d'ensemble du système Web

But de cette page

Cette page est la **carte d'entrée du système Web Uniksimpl**. Elle permet de comprendre rapidement quelles applications existent, comment elles communiquent, où sont stockées les données et dans quel chapitre chercher lorsqu'un problème survient.

Carte globale du système

flowchart LR

USER[Utilisateur] --> ANG[Frontend Angular]

ADMIN[Administrateur] --> ANG

ANG -->|HTTP| NEST[NestJS principal :3000]

ANG <-->|Socket.IO| WS[SupportGateway /support]

ANG <-->|Socket.IO| ACC[AccountGateway /account]

ANG --> UNITY[Unity WebGL]

UNITY -->|HTTP / heartbeat / données jeu| NEST

NEST --> CTRL[Controllers]

CTRL --> SVC[Services métier]

SVC --> ORM[TypeORM]

ORM --> MYSQL[(MySQL)]

SVC --> REDIS[(Redis)]

SVC --> BC[BrainCloud]

SVC --> FB[Firebase / FCM]

SVC --> MAIL[Mail]

```
SVC --> AI[Serveur IA :8106]
```

```
NEST --> ADMINEXP[Admin Server Express :8105]
```

```
ADMINEXP --> BC
```

```
ADMINEXP --> REDIS
```

```
AI --> CHROMA[(Chroma / embeddings)]
```

```
AI --> PROVIDER[Provider IA]
```

```
NEST --> STATIC[Frontend Angular compilé]
```

```
STATIC --> USER
```

Les grandes briques

Frontend Angular

Application principale côté navigateur dans `uniksimFrontEnd/`. Elle contient les pages utilisateur et admin, les composants UI, les services HTTP, l'authentification frontend, le support temps réel et l'intégration Unity.

Backend NestJS

Serveur HTTP principal dans `uniksim-back-end/`. Le point d'entrée est `src/main.ts` et l'assemblage global se fait dans `src/app.module.ts`. Il expose les APIs utilisateur, jeu, administration, support, notifications, push, mail et IA.

Admin Server Express

Serveur HTTP auxiliaire séparé dans `src/node.js/server/admin_server.js`, sur le port `8105`. Il sert principalement certaines opérations historiques/BrainCloud, Redis, copie de simulations et SSE.

Serveur IA

NestJS attend l'ouverture du listener IA sur `127.0.0.1:8106`, puis démarre. L'initialisation des providers et de Chroma se poursuit en arrière-plan; leur indisponibilité n'empêche pas NestJS ni l'endpoint de santé de démarrer. Les requêtes retiennent les dépendances manquantes. Le serveur

IA contient des contrats, factories, providers, tokenizer, pricing, `UsageMeter` et `CostGuard` sous `src/node.js/server/ai/`. Gemini utilise maintenant Vertex AI, avec génération et embeddings sélectionnés indépendamment.

Unity WebGL

Le jeu Unity est intégré au Web. Angular affiche le build WebGL et Unity communique avec le backend pour les versions, textes, données utilisateur, heartbeat, classements, images et crash logs.

Persistence

- **MySQL + TypeORM** : utilisateurs, rôles, produits, support, installations FCM, etc.
- **Redis** : présence, sessions, notifications, rankings et autres données runtime.
- **BrainCloud** : plusieurs données historiques et fonctions liées aux simulations.
- **Chroma** : recherche vectorielle du système IA/RAG.

Flux typique d'une requête Web

```
sequenceDiagram
    participant U as Utilisateur
    participant A as Angular
    participant C as Controller NestJS
    participant S as Service
    participant D as MySQL / Redis / BrainCloud

    U->>A: Action dans l'interface
    A->>C: Requête HTTP
    C->>C: Guard / validation
    C->>S: Appel métier
    S->>D: Lecture / écriture
    D-->>S: Résultat
    S-->>C: Données métier
    C-->>A: Réponse API
    A-->>U: Mise à jour de l'interface
```

Flux temps réel

Deux namespaces Socket.IO séparent les responsabilités : `/account` porte présence, télémétrie consentie, jeu actif et notifications; `/support` porte messages, tickets et typing. Le middleware JWT est commun, les rooms sont propres à chaque namespace et le frontend partage `RealtimeSocket`.

Seuls les sockets du compte sont enregistrés dans `PresenceService`. Voir [Support temps réel](#) et [Notifications et Push](#).

Toutes les notifications sont stockées dans une inbox Redis du compte, diffusées par `AccountGateway` puis poussées vers toutes ses installations FCM. Les notifications envoyées par le jeu passent par `GameNotificationService`, qui vérifie les membres de la simulation BrainCloud. Les subscriptions FCM par `productCode` ont été retirées.

Build et hébergement

Le build Angular de production est généré dans `uniksim-back-end/src/front`, puis NestJS est buildé à son tour. Le backend sert ensuite le frontend compilé avec `ServeStaticModule`. L'instance de développement hébergée est `sim.dev.uniksim.com`. Voir **Build et déploiement Web**.

Où chercher selon le problème

- **Écran, navigation ou composant** → chapitre Frontend Angular.
- **Endpoint, logique backend, auth** → Serveurs et API.
- **Donnée SQL ou Redis** → Base de données et persistance.
- **Constante, port, environnement** → Configuration, environnements et constantes.
- **Chat, WebSocket, RAG, provider IA** → Support et IA.
- **Unity, notifications, Firebase, services externes** → Intégrations et systèmes transversaux.
- **Déploiement / instance demo** → Build et déploiement Web.

Points d'entrée essentiels

- `uniksimFrontEnd/src/main.ts`
- `uniksimFrontEnd/src/app/app.module.ts`
- `uniksim-back-end/src/main.ts`
- `uniksim-back-end/src/app.module.ts`
- `uniksim-back-end/src/app.controller.ts`

- `uniksim-back-end/src/database.providers.ts`
- `uniksim-back-end/src/node.js/server/constantsVariable.js`
- `uniksim-back-end/src/node.js/server/admin_server.js`
- `uniksim-back-end/src/node.js/server/ai/`

Structure des projets et dossiers

Objectif

Cette page indique **où chercher dans le code** selon le type de changement ou de bug. Elle ne remplace pas les pages détaillées des sous-systèmes.

Vue des deux projets principaux

flowchart TD

ROOT[uniksimweb] --> FE[uniksimFrontEnd]

ROOT --> BE[uniksim-back-end]

FE --> FEA[src/app]

FEA --> P[Pages]

FEA --> C[Components]

FEA --> POP[Popup]

FEA --> FS[service]

FEA --> FM[Models]

BE --> BES[src]

BES --> CTRL[app.controller.ts]

BES --> MOD[app.module.ts]

BES --> SV[service]

BES --> ENT[entities]

BES --> GW[gateway]

BES --> G[guards]

BES --> NODE[node.js/server]

BES --> EMB[embedding]

Frontend : uniksimFrontEnd

- `src/app/Pages/` : écrans complets et routes principales.
- `src/app/Components/` : composants réutilisables.
- `src/app/Popup/` : popups et dialogues.
- `src/app/service/` : logique frontend partagée, accès API, auth, support, notifications, présence, etc.
- `src/app/Models/` : modèles TypeScript.
- `src/app/constants/` : constantes frontend.
- `src/assets/` : ressources statiques, incluant le build Unity WebGL.
- `src/environments/environment.ts` : URLs/backend disponibles côté Angular.
- `angular.json` : build Angular et destination du build production.

Backend : uniksim-back-end

- `src/main.ts` : bootstrap NestJS.
- `src/app.module.ts` : enregistrement global des modules, contrôleurs, services et intégrations.
- `src/app.controller.ts` : contrôleurs HTTP principaux.
- `src/service/` : logique métier.
- `src/entities/` : entités TypeORM.
- `src/gateway/` : WebSockets/Socket.IO.
- `src/guards/` : autorisation/authentification backend.
- `src/mapper/` : conversion entre données internes et réponses API.
- `src/dto/` : DTOs et contrats d'entrée/sortie.
- `src/node.js/server/` : couche historique/auxiliaire : BrainCloud, constantes, Admin Server, IA, Redis, etc.
- `src/embedding/` : ingestion et documentation destinée au RAG.
- `src/migrations/` : changements de schéma/versionnement DB lorsque utilisés.
- `src/front/` : **frontend Angular compilé**; ne pas modifier manuellement.

Où commencer selon le besoin

- Nouvelle page Angular → `Pages/` + routing dans `app.module.ts`.
- Nouveau composant réutilisable → `Components/`.
- Nouvel appel API → service Angular approprié puis controller/service NestJS.
- Nouvelle donnée SQL → `entities/`, service/repository et schéma DB.
- Temps réel → `gateway/` côté backend et service Socket.IO côté frontend.
- Comportement IA → `node.js/server/ai/`, `embedding/` et services support concernés.

- Paramètre global → `constantsVariable.js` ou fichier d'environnement.

Règle pratique

Avant d'ajouter un nouveau fichier ou une nouvelle abstraction, vérifier si le comportement appartient déjà à un dossier/service existant. Le projet contient plusieurs couches historiques; dupliquer une responsabilité peut rendre les flux beaucoup plus difficiles à suivre.

Organisation des sous-systèmes

- Backend : `gateway/account.gateway.ts` pour le compte, `gateway/support.gateway.ts` pour le support, `guards/socketAuth.middleware.ts` pour leur authentification commune.
- Notifications : `service/notification.service.ts` est le pipeline générique; `service/game-notification.service.ts` valide le destinataire du jeu; `dto/notification.dto.ts` définit l'enveloppe.
- Angular : `service/realtime/` pour le transport, `service/account/` pour le temps réel du compte, `service/support/` pour les conversations et tickets; contrats séparés dans `Models/Notification.ts` et `Models/Presence.ts`.
- IA : `node.js/server/ai/contract/`, `factory/`, `guard/`, `usage/` et `provider/gemini/`. Le SDK Vertex et son adaptation restent dans le provider.
- Tests backend : `test/ai/` et `test/notification/`; configuration partagée `test/jest.config.cjs`, configuration IA `test/jest.ai.config.cjs`. Exécuter `npm test -- --runInBand` ou `npm run test:ai` depuis le backend.

Frontières entre code moderne et legacy

Pourquoi cette page existe

Le projet Web combine une architecture NestJS/Angular moderne avec une couche plus ancienne sous `src/node.js/server/`. Les deux sont encore actives. Comprendre cette frontière évite de réécrire une fonctionnalité déjà existante ou de modifier la mauvaise couche.

Carte simplifiée

flowchart LR

ANG[Angular] --> NEST[NestJS moderne]

NEST --> CTRL[Controllers]

CTRL --> SVC[Services TypeScript]

SVC --> ORM[TypeORM / MySQL]

SVC --> REDIS[RedisService]

SVC --> LEGACY[src/node.js/server]

LEGACY --> BC[BrainCloud]

LEGACY --> REDIS

LEGACY --> ADMIN[Admin Server Express]

LEGACY --> AI[Serveur IA]

Couche moderne

La majorité du nouveau code backend devrait naturellement passer par :

- `app.controller.ts` pour les endpoints;

- `service/*.service.ts` pour la logique métier;
- `entities/` et TypeORM pour MySQL;
- `guards/` pour l'autorisation;
- `gateway/` pour le temps réel.

Couche historique / auxiliaire

`src/node.js/server/` contient encore plusieurs responsabilités importantes :

- `server.js` : fonctions historiques de jeu, présence, rankings, images, etc.;
- `backend_braincloud.js` : intégration BrainCloud;
- `admin_server.js` : serveur Express admin séparé;
- `constantsVariable.js` : configuration globale et constantes;
- `ai/` : serveur IA et providers.

Quand utiliser quelle couche

flowchart TD

```

NEW[Nouveau changement backend] --> EXIST{Fonction existante déjà dans node.js/server ?}
EXIST -- Oui --> CONTRACT{Peut-on garder le contrat et l'encapsuler ?}
CONTRACT -- Oui --> WRAP[Appeler depuis un service NestJS]
CONTRACT -- Non --> MIGRATE[Planifier migration explicite]
EXIST -- Non --> MODERN[Implémenter dans Controller + Service NestJS]

MODERN --> DATA{Persistance ?}
DATA -- SQL --> ORM[TypeORM]
DATA -- Runtime --> REDIS[RedisService]
```

Règle générale

Pour une nouvelle fonctionnalité, privilégier NestJS et les services TypeScript. Ne déplacer du code historique que lorsqu'il y a une raison claire : simplification, suppression d'une dépendance legacy ou correction d'un problème architectural. Une migration opportuniste au milieu d'une feature augmente le risque.

Signaux qu'on touche du legacy

- appel direct à `backend_braincloud.js`;
- fonction exportée depuis `server.js`;
- utilisation du serveur Express admin;
- globals comme `NodeJsServer`, `NodeJsUtils` ou `NodeJsBackendBraincloud` initialisés dans `database.providers.ts`.

Avant de refactorer

- Identifier tous les appels entrants vers la fonction legacy.
- Vérifier les effets Redis, BrainCloud et fichiers locaux.
- Conserver le contrat API existant si possible.
- Séparer migration et changement fonctionnel lorsqu'ils sont suffisamment gros.

Frontière des notifications après refactoring

Les anciennes fonctions d'inbox/notification ont été retirées de `server.js`. La responsabilité appartient à `NotificationService` (stockage Redis, événement du compte et push) et à `GameNotificationService` (autorisation BrainCloud et résolution profil jeu → compte). Ne pas réintroduire une seconde inbox dans la couche historique. `AccountGateway` diffuse les notifications; `SupportGateway` conserve les conversations et tickets.

Ajouter une fonctionnalité Web de bout en bout

Objectif

Cette page donne un chemin pratique pour ajouter une fonctionnalité Web sans oublier une couche. Le principe est de suivre le flux Angular → API → service → données → retour UI.

Flux type

flowchart LR

```
UI[Page / Component Angular] --> FES[Service Angular]
FES --> AUTH[AuthInterceptor]
AUTH --> CTRL[Controller NestJS]
CTRL --> GUARD[Guard]
GUARD --> BES[Service backend]
BES --> DATA{Données}
DATA --> ORM[TypeORM / MySQL]
DATA --> REDIS[Redis]
DATA --> BC[BrainCloud]
BES --> CTRL
CTRL --> FES
FES --> UI
```

Étapes

1. Définir l'action utilisateur, les données entrantes/sortantes, le niveau d'accès et le type de persistance.
2. Côté Angular, placer l'écran dans `Pages/` ou le composant dans `Components/`, puis passer par un service Angular pour les appels backend.

3. Ajouter ou réutiliser la route dans `app.module.ts` et appliquer le guard frontend adapté.
4. Côté backend, ajouter l'endpoint au controller logique existant. Garder le controller mince.
5. Placer la logique métier dans `src/service/`.
6. Choisir le stockage : TypeORM/MySQL pour le relationnel persistant, Redis pour le runtime/temporaire, BrainCloud pour les données historiques de simulation déjà gérées là.
7. Ajouter Socket.IO/EventEmitter seulement si la mise à jour doit être temps réel.
8. Appliquer le guard backend approprié et valider les IDs et permissions côté serveur.
9. Centraliser les paramètres dans `constantsVariable.js` ou les fichiers `.env`.
10. Tester le cas nominal, les entrées invalides, les accès interdits et les erreurs de dépendances.

Choisir le stockage

flowchart TD

```
D[Besoin de stockage] --> P{Persistant relationnel ?}
P -- Oui --> SQL[Entity TypeORM + MySQL]
P -- Non --> RT{Temporaire / temps réel ?}
RT -- Oui --> REDIS[RedisService]
RT -- Non --> OLD{Déjà géré dans BrainCloud ?}
OLD -- Oui --> BC[BrainCloud]
OLD -- Non --> REVIEW[Revoir le modèle avant d'ajouter un stockage]
```

Checklist rapide

- UI dans le bon dossier.
- Service Angular réutilisé ou créé au bon endroit.
- Route et guards corrects.
- Controller mince.
- Logique dans un service backend.
- Stockage choisi selon le type de donnée.
- Temps réel ajouté seulement si nécessaire.
- Cas d'erreur et permissions testés.
- Configuration centralisée.
- Build/deploy vérifié.
- Documentation mise à jour si l'architecture change.