

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.

Revision #2

Created 2026-09-18 18:04:14 UTC by Mathis

Updated 2026-09-30 14:23:59 UTC by Mathis