

Support et IA

- [Ajouter un nouveau provider IA](#)
- [Pipeline RAG et système IA](#)
- [Support temps réel](#)

Ajouter un nouveau provider IA

1. Déclarer le provider et ses modèles

Dans `constantsVariable.js` :

```
const AI_PROVIDERS = {
  GEMINI: "gemini",
  OPENAI: "openai",
};

const AI_MODELS = {
  GEMINI_GENERATION: "gemini-3.5-flash-lite",
  GEMINI_EMBEDDING: "gemini-embedding-001",

  OPENAI_GENERATION: "gpt-example",
  OPENAI_EMBEDDING: "text-embedding-example",
};

const AI_PRICE_VERSIONS = {
  GEMINI: "vertex-global-standard-2026-09-29",
  OPENAI: "openai-standard-YYYY-MM-DD",
};
```

Configurer ensuite le provider actif dans `AI_CONFIG` :

```
const AI_CONFIG = {
  AI_PROVIDER: AI_PROVIDERS.OPENAI,
  AI_EMBEDDING_PROVIDER: AI_PROVIDERS.OPENAI,
```

```
AI_MODEL: AI_MODELS.OPENAI_GENERATION,  
AI_EMBEDDING_MODEL: AI_MODELS.OPENAI_EMBEDDING,  
  
OPENAI_API_KEY: process.env.OPENAI_API_KEY,  
};
```

Ne pas redéclarer les noms de modèles dans les fichiers du provider. Toujours importer `AI_MODELS` et `AI_PRICE_VERSIONS`.

“ **Note sur les embeddings** : si on conserve toujours le provider et le modèle d'embedding déjà existants, toute la partie « nouveau provider d'embedding » peut être ignorée. Le nouveau provider peut servir uniquement à la génération, tandis que `AI_EMBEDDING_PROVIDER` et `AI_EMBEDDING_MODEL` restent configurés sur l'existant.

2. Créer le dossier du provider

Créer la structure suivante :

```
server/ai/provider/openai/  
├─ generation_provider.js  
├─ embedding_provider.js  
├─ tokenizer.js  
├─ pricing.js  
└─ context_cache.js
```

`context_cache.js` est facultatif si le provider ne supporte pas le cache. `embedding_provider.js` est également facultatif si on garde le provider d'embedding existant.

3. Implémenter le provider de génération

Le contrat se trouve dans :

```
server/ai/contract/generation_provider_contract.js
```

Le provider doit exposer les méthodes suivantes :

```
class OpenAIProvider {
  constructor(config, dependencies = {}) {
    this.name = AI_PROVIDERS.OPENAI;
    this.config = config;
    this.client = null;
  }

  async initialize() {}

  isReady() {}

  async countTokens(messages, options) {}

  async generate(request) {}
}
```

Format normalisé des messages

```
{
  role: "user" | "assistant",
  content: [
    {
      type: "text",
      text: "Bonjour"
    },
    {
      type: "image",
      mimeType: "image/png",
      data: "base64..."
    }
  ]
}
```

Le provider doit convertir ce format vers celui de son SDK.

Requête reçue par `generate()`

```
{
  operation,
  scope,
  prefix,
  messages,
  temperature,
  maxOutputTokens,
  thinkingLevel,
  inputTokenLimit,
  tokenCountTimeoutMs,
  timeoutMs,
  usage
}
```

Réponse normalisée attendue

```
{
  text: "Réponse du modèle"
}
```

Aucun objet propre au SDK ne doit sortir du provider.

4. Implémenter le tokenizer

Créer `provider/openai/tokenizer.js` :

```
class OpenAITokenizer {
  constructor(client) {
    this.client = client;
  }

  async count(messages, options) {
    // Retourner un nombre entier de tokens.
  }
}
```

```

normalizeUsage(providerUsage) {
  return {
    inputTokens: 0,
    cachedInputTokens: 0,
    outputTokens: 0,
    reasoningTokens: 0,
  };
}
}

```

Les champs propres au SDK doivent rester dans ce fichier.

Exemples de champs qui ne doivent pas sortir du tokenizer :

```

promptTokenCount
candidatesTokenCount
input_tokens
output_tokens
cached_tokens

```

Le reste de l'application doit uniquement utiliser :

```

{
  inputTokens,
  cachedInputTokens,
  outputTokens,
  reasoningTokens
}

```

5. Implémenter le pricing

Créer `provider/openai/pricing.js`.

Importer les constantes partagées :

```

const {
  AI_PROVIDERS,
  AI_MODELS,

```

```
AI_PRICE_VERSIONS,  
} = require('../../../../constantsVariable');
```

Implémenter le pricing :

```
class OpenAIPricing {  
  constructor(config) {  
    this.config = config;  
    this.priceVersion = AI_PRICE_VERSIONS.OPENAI;  
  }  
  
  reservationCents(operation) {  
    // Calculer le coût maximal pouvant être facturé.  
  }  
}
```

Le fichier peut aussi exposer :

```
function generationCost(usage) {}  
  
function generationCeiling(inputTokens, outputTokens) {}  
  
// Ajouter des opérations tarifées uniquement si le provider les utilise.
```

Les calculs doivent recevoir les statistiques normalisées, jamais les données brutes du SDK.

Un modèle sans pricing explicitement validé doit être rejeté.

6. Implémenter les embeddings (facultatif)

“ Cette étape peut être ignorée si on utilise toujours le même embedding déjà existant. Dans ce cas, il n'est pas nécessaire d'implémenter un nouveau provider d'embedding ni de modifier la factory d'embedding.

Si un nouveau provider d'embedding est réellement nécessaire, le contrat se trouve dans :

```
server/ai/contract/embedding_provider_contract.js
```

Le provider doit exposer :

```
class OpenAIEmbeddingProvider {  
  constructor(config, dependencies = {}) {}  
  
  async initialize() {}  
  
  isReady() {}  
  
  assertInput(text) {}  
  
  async embed(text, usage) {}  
}
```

`embed()` doit retourner uniquement le vecteur :

```
[  
  0.012,  
  -0.034,  
  0.056  
]
```

`assertInput()` doit vérifier :

- Le modèle autorisé.
- La taille maximale du texte.
- La limite de tokens ou d'octets.
- La politique de coût applicable.

“ Changer de modèle d'embedding nécessite généralement de réindexer toute la collection Chroma.

7. Enregistrer le provider de génération

Modifier :

```
server/ai/factory/generation_provider_factory.js
```

Importer le provider :

```
const {  
  OpenAIProvider,  
} = require("../provider/openai/generation_provider");
```

Ajouter le cas dans la factory :

```
switch (config.AI_PROVIDER) {  
  case AI_PROVIDERS.GEMINI:  
    provider = new GeminiProvider(config, dependencies);  
    break;  
  
  case AI_PROVIDERS.OPENAI:  
    provider = new OpenAIProvider(config, dependencies);  
    break;  
  
  default:  
    throw new Error(  
      `Unsupported AI provider: ${config.AI_PROVIDER}`  
    );  
}
```

La factory valide automatiquement le contrat avant de retourner le provider.

8. Enregistrer le provider d'embedding (facultatif)

Ignorer cette étape si on garde le provider d'embedding existant. Le provider de génération et celui d'embedding peuvent être différents.

Modifier :

```
server/ai/factory/embedding_provider_factory.js
```

Ajouter :

```
case AI_PROVIDERS.OPENAI:
    return assertEmbeddingProviderContract(
        new OpenAIEmbeddingProvider(config, dependencies)
    );
```

Exemple si OpenAI sert à la génération mais Gemini reste utilisé pour les embeddings :

```
AI_PROVIDER: AI_PROVIDERS.OPENAI,
AI_EMBEDDING_PROVIDER: AI_PROVIDERS.GEMINI,
```

9. Enregistrer le pricing

Modifier :

```
server/ai/factory/pricing_factory.js
```

Ajouter :

```
case AI_PROVIDERS.OPENAI:
    return assertPricingContract(
        new OpenAIPricing(config)
    );
```

10. Ajouter les tests

Vérifier au minimum :

- L'initialisation du SDK.
- Le contrat de génération.
- Le contrat d'embedding, uniquement si un nouveau provider d'embedding est ajouté.
- La conversion des messages normalisés.
- La conversion des images.
- Le comptage des tokens.
- La normalisation des statistiques d'utilisation.
- Le calcul des coûts.
- Les limites d'entrée.
- Les timeouts.
- Les réponses sans métadonnées.
- Le rejet des modèles sans pricing.
- Le rejet des providers inconnus.

Exécuter les tests AI :

```
npm run test:ai
```

Checklist finale

- Provider ajouté dans `AI_PROVIDERS`.
- Modèles ajoutés dans `AI_MODELS`.
- Version tarifaire ajoutée dans `AI_PRICE_VERSIONS`.
- Dossier créé sous `ai/provider/<provider>/`.
- Contrat de génération respecté.
- Tokenizer spécifique isolé.

- Statistiques d'utilisation normalisées.
- Pricing explicitement validé.
- Contrat d'embedding respecté uniquement si un nouveau provider d'embedding est nécessaire.
- Provider enregistré dans les factories nécessaires.
- Aucun objet SDK ne sort du provider.
- Aucun champ propre au SDK n'est utilisé par le serveur générique.
- Tests AI réussis.
- Collection vectorielle réindexée uniquement si le modèle d'embedding change.

Provider actuellement implémenté et règles de coût

Seul Gemini est enregistré dans les factories. Les exemples OpenAI ci-dessus décrivent un ajout possible, pas un provider déjà disponible. Le modèle de génération configuré est `gemini-3.5-flash-lite`, l'embedding reste `gemini-embedding-001`. Gemini utilise `@google/genai` via Vertex AI et `vertex_client.js`; `GOOGLE_CLOUD_PROJECT` est requis et la politique tarifaire actuelle impose `GOOGLE_CLOUD_LOCATION=global`.

`generation_provider.js` adapte les messages normalisés et le SDK; `generator.js` compte l'entrée, applique la limite et enregistre l'usage; `tokenizer.js` normalise `input/cachedInput/output/reasoning`. Le cache actuel est implicite : il n'y a plus de gestionnaire de cache de contexte explicite ni de coût de stockage associé dans ce provider.

Une nouvelle intégration doit enregistrer chaque opération payante avec `UsageMeter.begin()`, puis `complete()` lorsque l'usage est exploitable ou `failed()` sur erreur. `embed(text, usage)` accepte le meter à la recherche; l'ingestion peut appeler `embed(text)`. Les plafonds doivent inclure les tokens de raisonnement facturables. L'usage absent ou non pris en charge reste estimé au plafond réservé; les coûts ne doivent jamais être inventés à partir d'un usage manquant.

Les tests sont maintenant sous `test/ai/`. Depuis le backend : `npm run test:ai`, `npm run test:ai:coverage` ou `npm test -- --runInBand` pour toutes les suites.

Pipeline RAG et système IA

Vue d'ensemble

Le système IA du support UnikSim est basé sur une architecture RAG (*Retrieval-Augmented Generation*). Le backend NestJS orchestre la conversation de support, délègue la recherche et la génération au serveur IA interne, puis décide si une réponse IA peut être retournée ou si la conversation doit être escaladée vers un humain.

Les providers de génération et d'embedding sont volontairement indépendants. Cela permet de changer le modèle de génération sans modifier l'espace vectoriel existant dans Chroma.

1. Architecture des composants

Ce diagramme montre les responsabilités principales et les frontières entre le support, le serveur IA et les providers.

flowchart LR

U[Utilisateur] --> FE[Frontend Angular / App]

FE --> SS[SupportService - NestJS]

SS --> AIS[AIService - NestJS]

AIS -->|POST /ai/query| AI[Serveur IA interne]

AI --> CG[CostGuard]

AI --> UM[UsageMeter]

AI --> EP[Embedding Provider]

EP --> CH[(Chroma)]

AI --> GP[Generation Provider]

GP --> LLM[Gemini / autre LLM]

SS --> DB[(MySQL)]

SS -->|answered| FE

SS -->|insufficient_context| ESC[Clarification puis suggestion d'escalade humaine]

Responsabilités

- **SupportService** : persistance de la conversation, historique, contexte de récupération, pièces jointes et décision de réponse/escalade.
- **AIService** : client NestJS du serveur IA interne, validation et gestion des erreurs/timeouts.
- **Serveur IA** : recherche vectorielle, construction du contexte, génération, comptabilisation des coûts et garde-fous.
- **Embedding Provider** : transforme une requête ou un chunk documentaire en vecteur.
- **Chroma** : stocke et retourne les chunks sémantiquement proches.
- **Generation Provider** : encapsule le SDK du fournisseur LLM et retourne une réponse normalisée.

2. Pipeline d'ingestion documentaire

Les documents de support sont préparés à l'avance par le script d'ingestion. Le même provider/modèle d'embedding doit être utilisé à l'ingestion et à la recherche.

flowchart TD

```
DOC[Documents PDF / Markdown] --> META[Lecture des métadonnées héritées]
META --> VALID[Validation game / scope / audience / language]
VALID --> PARSE[Parsing du document]
PARSE --> CHUNK[Découpage en chunks avec overlap]
CHUNK --> EMB[Embedding de chaque chunk]
EMB --> IDS[IDs stables SHA-256]
IDS --> UPSERT[Upsert par batch dans Chroma]
UPSERT --> CLEAN[Suppression des anciens chunks devenus obsolètes]
CLEAN --> ORPHAN[Nettoyage des documents supprimés ou déplacés]

UPSERT -->|échec| ROLLBACK[Rollback des nouveaux IDs]
```

Le chunking privilégie les paragraphes et titres, puis découpe les blocs trop longs en phrases et en mots. Un chevauchement est conservé entre les chunks afin de limiter la perte de contexte.

3. Séquence complète d'une requête RAG

sequenceDiagram

actor U as Utilisateur

participant S as SupportService

participant A as AIService

participant R as Serveur IA

participant E as Embedding Provider

participant C as Chroma

participant G as Generation Provider

U->>S: Envoie un message

S->>S: Sauvegarde message utilisateur

S->>S: Charge historique + langue + retrievalContexts

S->>A: queryRAG(...)

A->>R: POST /ai/query + token interne

R->>R: Réserve budget (CostGuard)

R->>R: Construit la requête de recherche

R->>E: embed(searchQuery)

E-->>R: vecteur

R->>C: query(vector, filter, nResults)

C-->>R: chunks + distances

R->>R: Filtre selon MAX_DISTANCE et construit le contexte

R->>G: generate(prefix, history, question, image?)

G-->>R: texte normalisé

R->>R: UsageMeter + règlement du budget

R-->>A: answered / insuffisant_context

A-->>S: résultat RAG

alt réponse exploitable

S->>S: Sauvegarde message IA

S-->>U: Réponse IA

else contexte insuffisant

S->>S: Sauvegarde clarification ou suggestion d'escalade

S-->>U: Propose le support humain

end

Construction du contexte de récupération

Le navigateur ne décide pas directement du scope RAG. Le backend dérive les `retrievalContexts` à partir des produits possédés par l'utilisateur. Les filtres peuvent inclure le jeu, le scope, l'audience et d'autres métadonnées documentaires.

L'historique envoyé à l'IA contient uniquement les tours USER et AI, dans une fenêtre bornée. L'image, lorsqu'elle existe, est attachée uniquement au tour courant.

4. Décision de réponse ou d'escalade

flowchart TD

```
Q[Requête utilisateur] --> SEARCH[Recherche Chroma]
SEARCH --> REL{Assez de chunks pertinents ?}
REL -->|Non et aucune image| IC[insuffisant_context]
REL -->|Oui| CTX[Construire le contexte]
REL -->|Image présente| CTX
CTX --> GEN[Génération LLM]
GEN --> OUT{Réponse vide ou marqueur unsupported ?}
OUT -->|Oui| IC
OUT -->|Non| ANS[Retourner answer + confidence]
IC --> SUG[SupportService clarifie puis propose l'escalade]
ANS --> SAVE[Sauvegarder le message IA]
SUG --> HUMAN[Utilisateur peut escalader vers un humain]
```

Contrôle des coûts et garde-fous

Chaque opération protégée du serveur IA passe par une réservation de budget avant l'appel au modèle. Un `UsageMeter` accumule ensuite l'usage réellement observé et permet de régler la réservation dans un bloc `finally`, même si l'opération échoue.

- Plafonds d'entrée, de sortie et de taille de recherche.
- Timeouts côté NestJS et côté providers.
- Tarification associée au provider/modèle réellement utilisé.
- Refus des modèles non tarifés ou inconnus.

- Endpoint de santé séparant l'état du provider de génération, de l'embedding et de Chroma.

Fichiers principaux

- `uniksim-back-end/src/service/support.service.ts`
- `uniksim-back-end/src/service/ai.service.ts`
- `uniksim-back-end/src/node.js/server/ai/server.js`
- `uniksim-back-end/src/embedding/ingest.js`
- `server/ai/provider/*` pour les providers de génération, embedding, tokenizer et pricing.

Points d'attention

- Changer uniquement le provider de génération ne nécessite pas de réindexer Chroma.
- Changer le modèle d'embedding modifie l'espace vectoriel et implique normalement une réindexation.
- Les objets SDK spécifiques à un fournisseur ne doivent pas sortir de son provider.
- La réponse IA ne doit être considérée valide que si le contexte est suffisamment pertinent et que le provider retourne une réponse supportée.

Frontend et expérience utilisateur

Le frontend ne connaît pas directement Chroma, les embeddings ni les règles internes de retrieval. Il envoie le message utilisateur au backend et affiche le résultat normalisé retourné par le système de support.

flowchart LR

UI[UI Support / Chat]

HTTP[Service frontend HTTP]

SS[SupportService]

AI[Pipeline RAG]

SOCK[SupportSocketService]

STATE[Etat local / conversation]

UI -->|message + attachment| HTTP

HTTP --> SS

SS --> AI

```
AI --> SS
SS -->|persistance| DB[(MySQL)]
SS -->|MESSAGE_CREATED| SOCK
SOCK --> STATE
STATE --> UI

SS -->|insufficient_context| ESC[Clarification puis suggestion d'escalade]
ESC --> UI
```

Responsabilités du frontend

- collecter le texte saisi et la pièce jointe éventuelle;
- envoyer l'action utilisateur au backend;
- afficher les messages USER, AI, SYSTEM et ADMIN selon leur type;
- afficher les clarifications et, au seuil configuré, la suggestion de support humain décidées par le backend;
- mettre à jour l'interface à partir des événements temps réel plutôt que d'exécuter lui-même la logique RAG.

Ce que le frontend ne décide pas

Le navigateur ne choisit ni les documents recherchés ni le scope documentaire. Les `retrievalContexts` sont construits côté serveur à partir des droits et produits de l'utilisateur. Cela évite qu'un client modifie directement le jeu, l'audience ou le scope envoyés à Chroma.

Historique et pièces jointes

L'historique RAG est reconstruit côté backend à partir des messages persistés. Le frontend n'a donc pas à fabriquer le prompt conversationnel complet. Lorsqu'une image est jointe, elle est validée et chargée côté backend avant d'être transmise au serveur IA, et n'est attachée qu'au tour utilisateur courant.

Réponse IA vs expérience de support

Pour le frontend, le RAG produit des messages IA dans la conversation de support. Une réponse exploitable apparaît comme un message AI. Si le contexte est insuffisant, le backend sauvegarde d'abord des demandes de clarification AI, puis une suggestion de support humain après deux clarifications consécutives. L'utilisateur peut alors demander l'escalade.

État actuel du pipeline — 30 septembre 2026

`GeminiProvider` et `GeminiEmbeddingProvider` utilisent Vertex AI via `vertex_client.js`. Génération : `gemini-3.5-flash-lite` avec `THINKING_LEVEL=LOW`; embeddings : `gemini-embedding-001`. Chroma est un `CloudClient` avec collection `game`.

Le listener IA démarre sur `127.0.0.1:8106` avant l'initialisation des dépendances. NestJS attend ce listener, pas la disponibilité du modèle et de Chroma. `/ai/health` retourne 503 si une dépendance manque; les requêtes retentent son initialisation.

Clarifications avant support humain

Le statut positif du contrat HTTP est `answered`; sans contexte, il est `insufficient_context`. `SupportService` compte les clarifications IA consécutives : les deux premières réponses sauvegardent `uniksim.support_ai_clarify`, puis la suivante sauvegarde `uniksim.support_suggest_escalation`. Ces messages ont le type AI. L'utilisateur demande ensuite l'escalade; le statut insuffisant ne crée pas automatiquement un ticket.

Un salut simple sans image reçoit une réponse locale sans embedding ni génération. Une image validée peut permettre la génération sans passage assez proche; le prompt interdit d'inventer des faits sur la plateforme. Pour le texte, le seuil est au moins un chunk de distance $\leq 0,5$. La confiance est la moyenne de `clamp(1 - distance, 0, 1)`; elle indique la proximité documentaire, pas une probabilité de justesse.

Usage, cache et réservation

Instructions et contexte précèdent l'historique et l'image courante pour permettre le cache implicite Gemini. Le cache de contexte explicite a été retiré. Le tokenizer normalise input, cachedInput, output et reasoning; le pricing facture le raisonnement comme sortie. La réservation utilise le plafond du modèle de 65 536 tokens facturables de sortie, et non les seules limites visibles de 4 000 tokens par réponse ou 256 par résumé.

`UsageMeter` conserve les coûts en nanodollars entiers puis les convertit en cents. L'usage absent, incohérent ou d'un tier non pris en charge reste estimé au plafond. Le `finally` règle la réservation; si la réconciliation échoue, la réservation durable est conservée. Version : `vertex-global-standard-2026-09-29`. Voir [Ajouter un nouveau provider IA](#).

Support temps réel

Vue d'ensemble

Le support utilise HTTP pour valider et persister les conversations, messages et tickets; Socket.IO diffuse les changements après leur sauvegarde. Le refactoring du 30 septembre 2026 sépare le temps réel du compte de celui du support.

Deux namespaces et un transport partagé

Responsabilité	Compte : /account	Support : /support
Backend	AccountGateway + PresenceService	SupportGateway + SupportService
Frontend	AccountSocketService	SupportSocketService
Événements reçus	user.status, user.presence, admin.status, notification.created	message.created, ticket.updated, typing
Actions émises	updateStatus, updateTelemetry, updateGamePresence	joinConversation, leaveConversation, typing, checkOnline

Les services frontend utilisent `RealtimeSocket`, qui centralise l'authentification, les observables, la reconnexion et les acknowledgements. Les namespaces d'un même serveur partagent la connexion sous-jacente, mais leurs rooms sont indépendantes.

```
flowchart LR
  UI[Interface] --> HTTP[API HTTP NestJS]
  HTTP --> S[SupportService]
  S --> DB[(MySQL)]
  S --> E[EventEmitter]
  E --> SG[SupportGateway /support]
  SG --> SS[SupportSocketService]
  SS --> UI
  AS[AccountSocketService] <--> AG[AccountGateway /account]
  AG --> P[PresenceService]
```

```
N[NotificationService] --> AG
```

```
AS --> UI
```

Authentification, rooms et présence

Chaque gateway installe `socketAuthMiddleware(jwtService)` à l'initialisation du namespace. Le JWT est vérifié avant les handlers; le payload est placé dans `client.data.user`. Le token est accepté depuis `handshake.auth.token`, la query ou l'en-tête Authorization.

`AccountGateway` enregistre et retire les connexions dans `PresenceService`. Seuls les sockets du namespace compte comptent pour la présence. `SupportGateway` rejoint `user_<id>` et, pour les administrateurs, `admins`; il ne crée pas une seconde connexion de présence.

Pour `joinConversation`, un utilisateur doit posséder la conversation; les admins peuvent joindre les conversations de support. `typing` exige l'appartenance à `conversation_<id>` et ne vise que les autres sockets de cette room. `checkOnline` interroge `PresenceService`; un non-admin est limité aux admins assignés à ses tickets.

Message et réponse IA

1. L'interface envoie le message et la pièce jointe par HTTP.
2. `SupportService` valide la conversation et la pièce jointe puis sauvegarde le message USER.
3. `MESSAGE_CREATED` est émis; le gateway mappe le message avec `SupportMapper.toMessageDto` et diffuse `message.created` vers sa conversation.
4. Dans une conversation non escaladée, le service construit l'historique, la langue et les droits documentaires puis appelle `AIService.queryRAG`. La réponse ou le message système est persisté et diffusé par le même chemin.

Voir [Pipeline RAG et système IA](#) pour les clarifications et l'escalade lorsque le contexte est insuffisant.

Escalade et cycle de vie des tickets

L'escalade protège la transition `conversation.escalated: false → true` par une transaction atomique. Elle crée le ticket OPEN, sauvegarde un message SYSTEM, émet `MESSAGE_CREATED` et `TICKET_UPDATED`, puis demande un résumé asynchrone pour l'administration. Le premier admin qui répond peut devenir l'assigné.

```
stateDiagram-v2
[*] --> OPEN
OPEN --> IN_PROGRESS: assignTicket ou première réponse admin
OPEN --> RESOLVED: resolveTicket
IN_PROGRESS --> RESOLVED: resolveTicket
RESOLVED --> IN_PROGRESS: reopenTicket avec assigné
RESOLVED --> OPEN: reopenTicket sans assigné
```

Lecture et notifications

Les marqueurs `lastUserReadAt / lastUserReadMessageId` et `lastAdminReadAt / lastAdminReadMessageId` sont indépendants. L'utilisateur compte les messages ADMIN non lus; l'admin compte les messages USER non lus. Pour `ticket.updated`, le gateway obtient deux vues via `getTicketUpdates` et envoie la vue utilisateur à sa room, la vue admin à `admins`.

Les notifications de compte suivent `NotificationService → AccountGateway → AccountSocketService`. Elles sont distinctes des mises à jour de ticket. Voir [Notifications et Push](#).

Reconnexion et état frontend

`SupportSocketService` conserve l'ID de conversation active et rejoint sa room à la reconnexion. `AccountSocketService` remet l'utilisateur en ligne, réémet la télémétrie si le consentement est accordé et restaure le jeu actif. Le retrait du consentement envoie une télémétrie nulle; quitter le jeu efface sa présence. Les composants support restent centrés sur les messages, tickets et leur état local.

`RealtimeSocket` encapsule `connect/disconnect`, `on`, `emit`, `emitWithAck` et `close`. L'HTTP charge l'état initial et persiste les actions; les événements synchronisent l'interface.

Fichiers principaux

- Backend : `src/gateway/account.gateway.ts`, `support.gateway.ts`,
`src/guards/socketAuth.middleware.ts`, `src/service/presence.service.ts`, `support.service.ts`

- Angular : `service/realtime/realtime-socket.ts`, `service/account/account-socket.service.ts`, `service/support/support-socket.service.ts` et `support-state.service.ts`.
- Contrats : `Models/Presence.ts`, `Models/Notification.ts`, `Models/Support.ts`; constantes : `account.constants.ts`, `realtime.constants.ts`, `support.constants.ts`.