ADRs

ADR-001 : Hono plutôt qu'Express ou Fastify

ADR-001 : Hono plutôt qu'Express ou Fastify

> *Date : 2026 (initial setup) — Statut : accepté*

Contexte

Au moment de choisir le framework backend pour un projet TypeScript moderne avec :

- Streaming SSE intensif (RAG)
- Validation typée (Zod inline)
- Self-hosting (image Docker légère)
- Single dev (peu de bandwidth pour gérer un écosystème complexe)

Les options évaluées : Express (mature mais legacy), Fastify (perf++ mais plugins gros), Hono (moderne, types natifs, perf comparable Fastify).

Décision

Hono, version 4.x.

Raisons principales :

- **SSE natif** via `streamSSE()` helper. Express demande des wrappers tiers, plus de friction.
- **Types end-to-end** : context typé (`AppEnv`), middleware typé, pas de `any` parasite.

- **Légèreté** : core ~10 KB, pas de plugins lourds par défaut. Image Docker plus petite.
- **Performance** : benchmarks proches de Fastify, ~2-3× Express. Suffisant pour RAG self-hosted.
- **Ergonomie** : routes composables, middleware factorisable, validation Zod naturelle.

Conséquences

Bonnes

- Le code est concis et bien typé.
- SSE marche out-of-the-box avec heartbeat helpers.
- Migrations entre versions Hono peu invasives.

Mauvaises

- Communauté plus petite qu'Express : moins de tutoriels, moins de StackOverflow.
- Pas (ou peu) de plugins prêts à l'emploi pour des trucs ésotériques. Mais on n'en a pas besoin.
- Si un jour besoin de scaler horizontalement, le pattern `streamSSE` + heartbeats est portable mais demande de la doc maison.

Alternative envisagée

- **Fastify** : aussi rapide, plus mature. Mais l'écosystème plugin est plus complexe (lifecycle hooks, fastify-types, etc.) — plus de friction pour un single dev.
- **Express** : trop legacy. Pas de types natifs, SSE = wrappers, validation = Joi/Zod custom. Peu de plaisir à coder en TS.

ADR-002 : SQLite + Drizzle plutôt que Postgres

ADR-002 : SQLite + Drizzle plutôt que Postgres

> *Date : 2026 (initial setup) — Statut : accepté*

Contexte

Stockage relationnel pour : users, games, questions, feedbacks. Volume attendu : <10K users, <100K questions sur la durée de vie du projet (usage perso + qq invités potentiels).

Options : Postgres (standard, riche), MySQL/MariaDB, SQLite (local, simple).

Décision

SQLite (via `better-sqlite3`) + **Drizzle ORM**.

Raisons :

- **Self-host minimal** : un fichier `.db` dans un volume Unraid, zéro service supplémentaire à monitorer
- **ACID suffisant** : SQLite est ACID et plus que rapide pour ce volume. WAL mode active par défaut.
- **`better-sqlite3` synchrone** : pas de Promise pour chaque query, code plus simple, perfs excellentes (< 1ms par query)

- **Drizzle ORM** : types end-to-end, migrations propres (`drizzle-kit generate` + `migrate`), syntaxe expressive sans magie
- **Zéro pool / latence réseau** : la DB tourne dans le même process que le backend

Conséquences

Bonnes

- Stack minimaliste : un container app + Qdrant, c'est tout
- Backup trivial : copier le `.db` (en mode WAL : aussi `.db-shm` et `.db-wal`)
- Migrations rapides à appliquer (single-instance, pas de coordination)
- Tests unitaires faciles : `:memory:` ou tmpdir

Mauvaises

- **Pas de scaling horizontal** : un seul process écrit. Pour scaler, il faudrait migrer vers Postgres.
- **Pas de réplication** : si le fichier corrompt, restore depuis backup obligatoire.
- **Concurrent writes** : SQLite serialize les writes. Pas un problème sur ce volume, mais à connaître.
- **Pas de full-text search avancé** : SQLite a FTS5 mais on ne l'utilise pas. Le retrieval vectoriel est sur Qdrant.

Alternative envisagée

- **Postgres** : plus robuste mais demande un container supplémentaire, configuration users/perms, backup pg_dump à scripter, tuning à apprendre. Overkill pour <100K rows.
- **PlanetScale / Neon serverless** : non self-hostable simplement. Et envoyer les données chez un cloud tiers va à l'encontre du principe self-host du projet.

Migration future possible

Si on doit scaler :

1. Drizzle supporte Postgres natif → migrer le schéma `src/schema.ts` (changer le driver)
2. Migrations SQL traduites manuellement (SQLite et Postgres ont des syntaxes proches mais quelques diffs : `INTEGER PRIMARY KEY AUTOINCREMENT` vs `SERIAL`)

3. Sessions in-memory → table `sessions` dans la même DB
4. Container Postgres + volume + healthcheck dans `docker-compose.yml`

L'effort est borné : Drizzle abstrait l'essentiel. C'est faisable en 1-2 jours quand le besoin se présente.

ADR-003 : Claude via SSH plutôt qu'API Anthropic directe

ADR-003 : Claude via SSH plutôt qu'API Anthropic directe

> *Date : 2026 (premiers tests RAG) — Statut : accepté*

Contexte

L'app a besoin d'appeler Claude (Opus + Haiku) pour : génération RAG, HyDE, decompose-query, contextual retrieval, hierarchy LLM, conflict detection, intent classification, deckbuilding.

Volume estimé : 1000-5000 appels Claude par jour en pic d'usage (notamment pendant les ingestions).

Options :

1. **API Anthropic directe** (`https://api.anthropic.com/v1/messages`)
2. **Claude Code CLI via SSH** vers une VM `oracle` (compte personnel Pro/Team)

Décision

Claude Code CLI via SSH vers une VM dédiée.

Raisons :

- **Quota Pro/Team** : compte personnel Anthropic Pro/Team paie un forfait fixe (~$20/mois) avec quota glissant 5h. Beaucoup plus économique que payer à l'API ($/M tokens) pour du volume moyen-élevé.
- **Outil `Read` natif** : Claude Code a un outil `Read` qui lui permet de lire les PNG des règles directement. Pour la "vision inline" (cf. ADR-004), c'est la fonctionnalité-pivot.
- **No-cost marginal** : un appel de plus = pas de coût supplémentaire (dans la limite quota). Permet d'expérimenter (Contextual Retrieval B = 1 appel par chunk = potentiellement 1000s d'appels par PDF) sans angoisse de facturation.
- **CLI maintenu par Anthropic** : pas de risque que la lib bouge sous mes pieds.

Conséquences

Bonnes

- Coût stable et prévisible
- Vision inline gratuite (lecture PNG ad libitum dans la limite quota)
- Outil `Read` directement accessible — pas besoin d'inliner les images en base64 dans les prompts
- Possibilité d'utiliser des system prompts riches sans facturer chaque token

Mauvaises

- **Quota saturable** : si on dépasse, pause obligatoire (cf. `services/claude-quota.ts` + ADR-007 implicite sur la pause/reprise auto). À l'API ce serait juste $$.
- **Latence baseline ~5s** : tunnel SSH + cold start CLI. Pour les calls courts (HyDE, classify), c'est non-négligeable. D'où le timeout 25s configuré pour absorber cette latence.
- **Sécurité SSH critique** : la VM oracle exécute du code Claude. Verrouillage multi-couche obligatoire (cf. ADR sécurité ssh-oracle).
- **Single point of failure** : si la VM oracle tombe, plus de RAG. Pas de fallback API direct (faisable mais pas implémenté).
- **Pas de batching natif** : chaque appel = un SSH. Pour les pools (Contextual B 10 parallèles), on multiplie les SSH simultanés.

Alternative envisagée

- **API Anthropic directe** : plus rapide (latence ~500ms vs 5s SSH), mais coûte 5-10× plus pour le volume actuel. Non retenu pour des raisons économiques.
- **Hybrid** : API pour les calls courts (HyDE, classify) + SSH pour Opus/Read. Plus complexe, gain marginal. Reporté.

Si on doit migrer un jour

- Implémenter un fallback `services/claude-api.ts` qui wrappe `@anthropic-ai/sdk`
- Mettre les credentials dans une env var `ANTHROPIC_API_KEY`
- Toggle via env (`CLAUDE_BACKEND = 'ssh' | 'api' | 'hybrid'`)
- Garder la vision via une upload image base64 dans le prompt (latence + coût supérieurs)

Effort : 1-2 jours si urgent.

ADR-004 : Vision inline (lue au moment Q) plutôt que pré-ingestion

ADR-004 : Vision inline (lue au moment Q) plutôt que pré-ingestion

> *Date : 2026-04-11 — Statut : accepté*

Contexte

Les règles de jeux contiennent souvent du contenu visuel important : icônes, schémas, plateau, tuiles, couleurs. Le retrieval texte seul rate ces aspects.

Premier prototype 2026-Q1 : vision **à l'ingestion** — chaque page était envoyée à un modèle vision qui produisait une description textuelle, ingérée comme un chunk supplémentaire.

Problèmes constatés :

- **Coût élevé** : 1 appel vision par page × N pages × tous les jeux ingérés = facture importante
- **Sur-description** : le modèle décrivait des détails inutiles, pollution du retrieval
- **Latence d'ingestion** : ajoutait 5-10 min par jeu

- **Description figée** : si la question demande "que représente la zone rouge en haut à gauche", la description ingérée n'a peut-être pas mentionné cette zone (subjective)

Décision

Vision inline au moment de la question. Claude lit le PNG de la page la plus pertinente directement via son outil `Read` côté VM SSH, **uniquement** quand la question a une dimension visuelle.

Détails dans `pipeline-rag/vision-inline.md`.

Conséquences

Bonnes

- **Coût zéro à l'ingestion** : juste `pdftoppm` qui rend les PNG localement
- **Précision** : Claude regarde l'image avec la question en tête → réponse ciblée, pas de description générique
- **Latence d'ingestion réduite** de 5-10 min
- **Fonctionne grâce à** `--append-system-prompt` qui préserve le contrat outils Claude Code
- **Pas de dépendance à un modèle vision tiers** : Claude Code suffit

Mauvaises

- **Latence par question +20-30s** quand l'image est lue (Read tool sur PNG 300 DPI)
- **1 image max** : si la question concerne 2 pages, on en perd une. Compromis assumé (passer à 2-3 doublait/triplait la latence pour gain marginal)
- **Granularité** `(livret, page)` **obligatoire** : oubli a déjà coûté un bug — chunk page 4 base + chunk page 4 extension donnent la mauvaise image
- **Dépend du chemin SSH côté oracle** : `permissions.additionalDirectories` doit pointer sur le volume PNG côté VM
- **Si le PNG manque** (rendu raté à l'ingestion), Claude ne peut pas lire — silently dégradé

Alternative envisagée

- **Vision pré-ingérée + cache** : décrit chaque page une fois, stocke la description, ré-utilise. Coût initial élevé, problème de description figée non résolu.
- **Vision par chunk** : décrit la zone autour du chunk plutôt que la page entière. Trop fin, pas implémentable simplement avec PDF natif.
- **OCR seulement** : voir ADR sur OCR (Phase 1 livrée 2026-05-06). Complémentaire mais pas substitut — l'OCR récupère du texte, pas l'analyse visuelle.

Variante future (Phase 2 OCR)

Roadmap : Phase 2 OCR (cf. mémoire `project_ocr_phase2.md`) prévoit Claude vision en fallback de Tesseract (page-by-page si confidence basse). Différent du flow actuel : fallback OCR vs lecture à la question. Les deux peuvent coexister.

ADR-005 : Fusion RRF v2 (pondération question×2 + blending position-aware)

ADR-005 : Fusion RRF v2 (pondération question×2 + blending position-aware)

> *Date : 2026-Q1 — Statut : accepté*

Contexte

Le retrieval combine plusieurs sources :

- Vecteur dense de la question brute (TEI bge-m3)
- Vecteur dense du passage HyDE (Haiku-généré)
- Sparse BM25 sur la question brute
- (Selon intent) chunks META, chunks synergy

Toutes les sources retournent un top-K avec rank et score. Il faut les fusionner en une seule liste classée.

Premier essai (RRF v1, classique) : `score = Σ 1/(60 + rank)` pour chaque chunk présent dans une passe. Inspiré de la littérature standard.

Problèmes constatés :

- **HyDE dilue la précision** : les exact-matches de la question brute sont noyés par les paraphrases HyDE. Si rank 1 dans question + rank 50 dans HyDE → score combiné moyen, pas top-1.
- **Reranker peu confiant** sur du contenu technique abstrait (notes de mécaniques, conditions de victoire) → score reranker ~0 → enterre des bons chunks.

Décision

Fusion RRF v2 avec deux modifications :

1. Pondération question × 2

```
score = 2 × (1 / (60 + rank_question + 1))
      + 1 × (1 / (60 + rank_hyde + 1))
      + 1 × (1 / (60 + rank_bm25 + 1))
```

La question brute pèse double — sa précision compte plus que celle du HyDE qui est une paraphrase.

2. Top-rank bonus

```
if (rank === 0) score += 0.05;
else if (rank === 1 || rank === 2) score += 0.02;
```

Préserve les exact-matches contre la dilution.

3. Blending position-aware (post-rerank)

Au lieu d'écraser le score RRF avec le score reranker :

```
const rrf_position = candidate.rrf_rank;
let weight_rrf, weight_rerank;
if (rrf_position <= 2)        { weight_rrf = 0.75; weight_rerank = 0.25; }
else if (rrf_position <= 9)   { weight_rrf = 0.60; weight_rerank = 0.40; }
else                          { weight_rrf = 0.40; weight_rerank = 0.60; }

final_score = weight_rrf * rrf_score + weight_rerank * rerank_score;
```

Si la fusion RRF a déjà classé un chunk dans le top, on garde son signal. Si le reranker est confiant et qu'il détrône un mauvais top, son signal compte plus.

Activable / désactivable via `RAG_FUSION_V2_ENABLED` (défaut `true`).

Conséquences

Bonnes

- Les exact-matches de la question survivent au passage HyDE
- Le reranker peu confiant sur du contenu abstrait n'enterre plus systématiquement les bons chunks
- Kill-switch via env var pour A/B test si nécessaire
- Inspiré de tobi/qmd, pattern documenté

Mauvaises

- **Magic numbers** : 0.05, 0.02, 75/25, 60/40, 40/60. Les ratios ont été tunés empiriquement, pas théoriquement.
- **Plus de paramètres à comprendre** pour un dev qui débarque
- Toute modification doit être validée sur le banc d'éval RAG (sinon régression silencieuse)

Tuning

Si tu veux ajuster :

- **Ratios blending** : monter à 85/15 sur top-3 si tu veux encore plus protéger le signal RRF
- **Top-rank bonus** : monter à +0.10 sur rank=0 si exact-matches cruciaux

Toujours `npm run test:rag` avant et après pour comparer la qualité.

Source d'inspiration

- RRF original paper (Cormack 2009)
- tobi/qmd — implémentation référence pour les ratios pondérés
- Banc d'éval RAG interne (boardgame-referee `tests/rag/`)

ADR-006 : Repository pattern (Phase 2)

ADR-006 : Repository pattern (Phase 2)

> *Date : 2026-Q1 (Phase 2 refactoring) — Statut : accepté*

Contexte

Au début du projet, les routes Hono importaient directement `db` et `drizzle-orm` :

```
// routes/games.ts
import { db } from '../db.js';
import { games } from '../schema.js';

app.get('/api/games', async (c) => {
  const allGames = await db.select().from(games).all();
  return c.json(allGames);
});
```

Au fil du temps, le même `db.select().from(games)` se retrouvait dans 20 fichiers (routes, services, scripts, crons). Conséquences :

- Quand on voulait ajouter un index sur une colonne, il fallait grep tous les `from(games)` pour comprendre l'impact
- Les requêtes se dupliquaient (3 endroits avec `where(eq(games.id, ...))`)
- Pas de naming métier — juste du Drizzle qui parlait techniquement

Décision

Repository pattern. Tout accès DB passe par `src/repositories/*.repo.ts`. Aucune autre couche n'importe `db`, `drizzle-orm` ou les tables du schéma.

```
// repositories/games.repo.ts
import { db } from '../db.js';
import { games } from '../schema.js';

export async function getById(id: string): Promise<Game | null> {
  return db.select().from(games).where(eq(games.id, id)).get() ?? null;
}

export async function listByParent(parentId: string): Promise<Game[]> {
  return db.select().from(games).where(eq(games.parentGameId, parentId)).all();
}

// + une trentaine de fonctions par repo
```

```
// routes/games.ts
import * as gamesRepo from '../repositories/games.repo.js';

app.get('/api/games/:id', async (c) => {
  const game = await gamesRepo.getById(c.req.param('id'));
  if (!game) return c.json({ error: 'not found' }, 404);
  return c.json(game);
});
```

Règle stricte :

- ❌ `db.select(...)` dans `routes/`, `services/`, `middleware/`, `cron/`, `scripts/`, `index.ts`
- ✅ `db.select(...)` uniquement dans `src/repositories/*.repo.ts`

Repos actuels

- `repositories/games.repo.ts` — listAll, getById, getByName, search, upsert, setIngestStatus, listByParent, etc.
- `repositories/questions.repo.ts` — CRUD questions + feedback

- `repositories/users.repo.ts` — auth + permissions

Conséquences

Bonnes

- **Lecture facilitée** : pour comprendre toutes les opérations sur `games`, j'ouvre `games.repo.ts` et c'est tout.
- **Refactor simple** : ajouter un index, changer un nom de colonne → un seul endroit à modifier (sauf si la modif change le contrat de retour, mais ça c'est attendu).
- **Naming métier** : `getByBggId(id)` parle, `db.select().from(games).where(eq(games.bggId, id)).get()` ne parle pas.
- **Tests** : mock du repo avec `vi.mock('../repositories/games.repo.js')` est trivial — pas besoin de mock toute la couche Drizzle.

Mauvaises

- **Indirection supplémentaire** : pour ajouter une nouvelle requête, il faut éditer le repo + le consommateur (au lieu d'un seul fichier).
- **Pas une vraie abstraction** : si on migre Drizzle → autre ORM, les repos restent à réécrire. Mais leur API reste stable, donc les consommateurs ne bougent pas.
- **Conventions à appliquer** : un nouveau dev pourrait être tenté de mettre `db.select()` directement dans une route. Documenter (cf. CONTRIBUTING.md) + grep régulier pour vérifier.

Vérification

```
# Doit retourner zéro résultat
grep -rn "from 'drizzle-orm'" src/ --include="*.ts" \
  | grep -v src/repositories/ \
  | grep -v src/db.ts \
  | grep -v src/schema.ts \
  | grep -v drizzle.config.ts
```

Alternative envisagée

- **Active Record style** (ex. Prisma) : modèles auto-générés, requêtes natives sur les modèles. Demande de changer d'ORM. Drizzle est bien, on garde + repo pattern manuel.
- **Service layer mais sans repo** : services qui contiennent leurs propres queries Drizzle. Mais le service mélange logique métier + accès DB, retombe dans le même problème (DB queries éparpillées).

Évolution future

Si on ajoute un cache (Redis) entre repo et consommateurs :

- Le repo reste l'API stable
- L'implémentation du repo lit/écrit Redis en surcouche
- Les consommateurs ne changent pas

C'est précisément le bénéfice de l'abstraction.

ADR-007 : Handlers MVC (Phase 4)

ADR-007 : Handlers MVC (Phase 4)

> *Date : 2026-Q2 (Phase 4 refactoring) — Statut : en cours / accepté*

Contexte

Avec le repo pattern (ADR-006), les routes étaient devenues plus propres :

```
// routes/games.ts (avant Phase 4)
app.post('/api/games/ingest', requireAuth, requireConfirmed, requireCanAddGames, async (c) =>
{
  const body = await c.req.parseBody();
  const validated = gameIngestMetadataSchema.parse(body);

  // 50 lignes de logique : upload PDF, validation extension parent, démarrage ingest...

  const game = await gamesRepo.upsert(...);
  // … encore 30 lignes

  return c.json({ gameId: game.id });
});
```

Mais les routes contenaient encore beaucoup de **logique métier** :

- Validation cross-table (parent existe ?)
- Décisions (ingest immédiat vs scheduled ?)
- Manipulation fichiers (PDF upload sur disque)
- Orchestration services (startIngestJob)

Conséquences :

- Routes de 100+ lignes
- Difficile à tester sans démarrer Hono
- Logique métier mélangée avec préoccupations HTTP

Décision

Architecture en 4 couches : `routes` → `handlers` → `services` → `repositories`.

```
// routes/games.ts (Phase 4)
import { ingestGameHandler } from '../handlers/games/ingest.js';

app.post('/api/games/ingest', requireAuth, requireConfirmed, requireCanAddGames, async (c) =>
{
  const formData = await c.req.parseBody();
  const validated = gameIngestMetadataSchema.parse(formData);
  const pdfBuffer = await (formData['pdf'] as File).arrayBuffer();
  const userId = c.get('user').id;

  const result = await ingestGameHandler(validated, Buffer.from(pdfBuffer), userId);

  if (!result.ok) return c.json({ error: result.error }, result.status);
  return c.json({ gameId: result.gameId, scheduled: result.scheduled });
});
```

```
// handlers/games/ingest.ts
type IngestResult =
  | { ok: true; gameId: string; scheduled?: boolean; scheduledStartAt?: string }
  | { ok: false; status: 400 | 404; error: string };

export async function ingestGameHandler(
  metadata: GameIngestMetadata,
  pdfBuffer: Buffer,
```

```
  userId: string,
): Promise<IngestResult> {
  // Validation cross-table
  if (metadata.parentGameId) {
    const parent = await gamesRepo.getById(metadata.parentGameId);
    if (!parent) return { ok: false, status: 404, error: 'parent game not found' };
  }

  // Manipulation fichiers
  const sourceFile = `/app/pdfs/${slug}-${Date.now()}.pdf`;
  await fs.writeFile(sourceFile, pdfBuffer);

  // Décision ingest immédiat vs scheduled
  if (metadata.scheduledStartAt) {
    const game = await gamesRepo.upsert({ ..., ingestStatus: 'scheduled' });
    scheduleIngestStart(game.id, new Date(metadata.scheduledStartAt));
    return { ok: true, gameId: game.id, scheduled: true, scheduledStartAt };
  }

  // Ingest immédiat
  const game = await gamesRepo.upsert({ ..., ingestStatus: 'idle' });
  startIngestJob(game.id);
  return { ok: true, gameId: game.id };
}
```

Règles du handler

1. **Pas d'import Hono** : un handler ne connaît pas le framework. Pas de `c.json`, pas de `c.req`, pas de `Context`.
2. **Reçoit des données validées** : le handler suppose que l'input est déjà passé par Zod côté route.
3. **Reçoit les dépendances explicites** : `pdfBuffer` (pas `formData`), `userId` (pas le cookie session).
4. **Retourne un Result discriminé** : `{ ok: true; ... } | { ok: false; status: ...; error: string }` pour les erreurs **attendues**. Pour les bugs / timeouts / erreurs non-récupérables, on `throw` normalement (capturé par le handler global Hono).
5. **Status HTTP dans le Result** : la route mappe le `status` au `c.json(..., status)`. Garde la connaissance HTTP côté route.

Conséquences

Bonnes

- **Routes plus courtes** : 5-15 lignes, juste parsing + délégation
- **Handlers testables sans Hono** : `await ingestGameHandler(meta, buf, 'user-123')` directement en Vitest
- **Logique métier centralisée** : ouvrir `handlers/games/ingest.ts`, voir tout ce qui se passe
- **Result discriminé** : TypeScript force à gérer tous les cas (ok / 400 / 404)
- **Réutilisabilité** : un handler peut être appelé par plusieurs routes ou par un script CLI

Mauvaises

- **Plus de fichiers** : `handlers/games/{ingest,delete,update,...}.ts`. Friction au refactor (~50% plus de fichiers).
- **Verbosité du Result** : pour des erreurs simples, le Result discriminé est plus verbeux que `throw new HTTPError(404, 'not found')`.
- **Migration en cours** : pas tous les routes sont passées au pattern. Cohabitation `routes-style` ancien et `routes + handlers` nouveau pendant la transition.

État de la migration (2026-05-10)

Route	Statut
`POST /api/games/ingest`	Phase 4 (handler)
`DELETE /api/games/:id`	Phase 4 (handler)
`POST /api/ask/stream`	encore en pattern routes-direct (logique RAG complexe, refactor reporté)
`POST /api/auth/*`	pattern routes-direct (logique simple, pas urgent)
`POST /api/admin/*`	mixte

À chaque nouveau besoin sur une route, l'extraire en handler en passant.

Workflow "ajouter une route"

1. Ajouter le schéma Zod dans `src/lib/schemas.ts` (si réutilisable)
2. Créer le handler dans `src/handlers/<domaine>/<action>.ts`

3. Définir le type `Result` discriminé
4. Implémenter la logique en utilisant les services + repos
5. Écrire le test Vitest du handler (mock services / repos)
6. Brancher la route Hono qui parse + valide + délègue + map le Result en réponse HTTP