Backend

Attente à ce stade

Avoir un **backend minimal et fiable** qui :

- expose les **APIs** nécessaires (ex. compteur de visites),
- fournit un **canal WebSocket** pour le temps réel (ping/pong, chat),
- reste **observé** (santé + métriques),
- s'intègre avec le **proxy** (façade unique).

Stack

- **Node.js + TypeScript + Fastify** (HTTP + WS).
- **SQLite** pour le stockage (fichier local, pas de conteneur DB).
- **Découpage logique en microservices** (chacun a ses routes), exposés via une **gateway** :
    - `auth`, `chat`, `game`, `tournament`, `visits ( pour test)`
    - **gateway** = point d'entrée (reçoit `/api/*` et `/ws` depuis Nginx, et route vers les services).

Exposition

Via le proxy (façade publique)

- **HTTP API** : toutes les requêtes passent par **`/api/...`**
    - Exemples :
        - `/api/visits` (lecture du compteur)
        - `/api/visit` (incrémenter)
        - `/api/users/ping` / `/api/games/ping` / `/api/chat/ping` / `/api/tournaments/ping` (sanity)

- **WebSocket** : **même origine que le site** + **chemin `/ws`**
    - Le proxy route `/ws` vers la **gateway** qui gère le WS et redistribue aux services si besoin.

Microservices

- **Porte d'entrée unique : `gateway`**
  - Le proxy envoie **tout** vers la gateway :
    - HTTP → `/api/...`
    - WebSocket → `/ws`

- **Pas de communication directe entre services**
  - Les services **ne se parlent pas entre eux**.
  - La gateway **oriente** chaque requête vers le service concerné, puis **répond** au client.

- **Stateless côté services** autant que possible
  - Toute logique transversale (auth, journalisation, métriques, accès data) est **gérée en amont** (gateway) ou via des couches partagées.

Communication frontend

API (HTTP)

- **GET `/api/visits`** → renvoie le total (ex. `{ "total": 42 }`).
- **POST `/api/visit`** → incrémente le total et renvoie l'état (ex. `{ "ok": true, "total": 43 }`).
- **Pings** (lecture seule) :

  - `/api/users/ping`, `/api/games/ping`, `/api/chat/ping`, `/api/tournaments/ping` → `{ "ok": true }`.

  Ces réponses sont utilisées par les tests et la doc front. **Ne change pas leur forme**.

WebSocket

- Le serveur **attend des messages JSON** (pas de texte brut).
- **Exemples de messages supportés** :

  - **Ping** : `{ "type": "ws.ping", "requestId": "…" }` → réponse **`{ "type": "ws.pong" }`**
  - **Chat** : `{ "type": "chat.message", "data": { "text": "…" }, "requestId": "…" }`
    → réponse **`{ "type": "ack", "requestId": "…" }`** (accusee de reception)

- Le test WS (page `/ws-test.html`) s'appuie sur ces formats.

Données (SQLite) - Acces unique

Ce qu'on utilise

- **SQLite** (fichier local, pas de conteneur DB).
- **Mode WAL** activé : `PRAGMA journal_mode = WAL;`
  ➜ Meilleure robustesse et concurrence lecture/écriture.

Où vit la base

- **Fichier** : stocké côté backend (ex. volume monté ou chemin local du service).
- **Sauvegardes** : il suffit de copier le fichier de base (et, si présent, les fichiers `-wal` / `-shm` quand l'app tourne).

Initialisation au démarrage

Au start, le backend :

1. **ouvre la base**, applique les **PRAGMA** (dont WAL),
2. **exécute `schema.sql`** pour garantir l'état minimal,
3. effectue, si besoin, des **mises à niveau** simples (ajouts de colonnes, index, données par défaut).

Règle d'architecture : un seul point d'accès à la BDD

- Seule la gateway (ou une couche data/ partagée appelée par la gateway) lit/écrit dans SQLite.

- Les services ne manipulent pas la DB directement : ils passent par la logique exposée par la gateway (ou par des fonctions de la couche data/ appelées depuis la gateway).

Fichiers modifiables sans problème (dans `backend/src/`)

Ajoute/ajuste librement le comportement. Évite juste de **casser** les contrats ci-dessus.

- `modules/**/http.ts`
  - **Garde** : les **préfixes** (ex. `/api/visits`, `/api/users/ping`, etc.) et les **formats de réponse** utilisés par le front/testeur.
- `ws-raw.ts` (gestion WS côté gateway)
  - **Garde** : l'acceptation **JSON**, le **`ws.ping`** → **`ws.pong`**, et l'**ACK** pour `chat.message`.
- `common/metrics.ts`
  - **Garde** : les métriques déjà exploitées par Prometheus/Grafana (noms et labels).
- `data/schema.sql`
  - **Compteur de visit a valeur d'exemple : tu peux supprimer**
- `data/index.ts`
  - **Garde** : l'initialisation SQLite et les exports utilisés par les modules.
- `types/**`
  - **Garde** : les types partagés référencés par plusieurs modules.

Fichiers à modifier avec ATTENTION (ajouts OK, pas de suppression)

- `backend/package.json`
  - **Garde** : Fastify/WS/TS et les scripts de base (`build`, `start`) utilisés par le Dockerfile.

`tsconfig.json` (backend)

  - **Garde** un paramétrage compatible avec la build (module, target).

❌ À NE PAS MODIFIER (côté architecture)

- `proxy/nginx.conf` : `/api/*` et **`/ws`**.
- **Port / interface** de la **gateway** attendus par le proxy (si tu changes, il faut changer Nginx en même temps).
- **Noms et formats** des endpoints **utilisés par le front et le testeur** (`/api/visits`, `/api/visit`, pings).

Noms de métriques déjà branchées dans Prometheus/Grafana (sinon dashboards/alertes cassent).

- **Dockerfile** (ports, user, CMD)