Frontend

Attente a ce stade du projet : avoir un **jeu local jouable** et **branché au backend** (API + WebSocket) sans fonctionnalités avancées.

Stack utilisée (confirmée par les fichiers du frontend)

- **React 18 + TypeScript + Vite** (dev, build, preview) — scripts `dev`, `build`, `preview`.
- **React Router** (routing SPA).
- **TailwindCSS + PostCSS + Autoprefixer** (styles utilitaires).
  Entrée HTML : `index.html` monte l'app sur `#root` et charge `/src/main.tsx`.
- **TS config** : ES2020, JSX React, strict mode, noEmit (build par Vite).

Comment le frontend parle au backend

A) Par API HTTP (pour les données "classiques")

- Le front appelle des URLs **qui commencent par `/api/...`** (le proxy Nginx route vers le bon service).
  Exemple d'usage concret dans l'app : le **compteur de visites**.
  - Lire la valeur actuelle : **GET `/api/visits`**.
  - Ajouter +1 : **POST `/api/visit`**. (via un bouton qui n'existe pour le moment pas)
- Pourquoi passer par `/api/...` ?
  ➜ Parce que le proxy (Nginx) route ensuite vers le bon service. Côté front, pas besoin de connaître les ports internes ni les noms des conteneurs.

B) WebSocket (temps réel)

- **Attention** : **même origine que le site** + chemin **`/ws`**.

  1. Si le site est visible à `https://localhost:8443`, alors l'URL WS est **`wss://localhost:8443/ws`**.
  2. Ne pas taper `wss://…` dans la barre d'adresse : ça ne s'ouvre que depuis du **JS** ou un **outil WS**.) , on peut le tester dans le terminal avec cette commande 'wscat -c wss://localhost:8443/ws'

- **À quoi ça sert ?** Événements en direct : chat, synchro de partie, ping/pong de test.

- **Comment vérifier que ça marche ?**

  1. Ouvre le site normalement (même origine que ton proxy).
  2. Utilise la **page de test** `https://localhost:8443/ws-test.html`)

3. Constater “connexion ouverte”, puis un **pong** quand on envoie un **ping** / un **ack** après un message test.

- **Utilisation de la page test : `https://localhost:8443/ws-test.html`**

La page se connecte automatiquement au serveur WebSocket.
Se connecter
Statut devient **“Connecté”**.
Message **“hello: connected”** envoyé par le backend.
Dans la zone de saisie “Message”, envoyer des commandes.

- `{ "type": "ws.ping", "requestId": "test1" }`
  recoit `{ "type": "ws.pong" }`
- `{ "type": "chat.message", "data": { "text": "coucou" }, "requestId": "test2" }`
  recoit `{ "type": "ack", "requestId": "test2" }` (accusee de reception)

Utilité de la page : Verifier que WS est fonctionnel avant de le coder

Exemple : le compteur de visites

Pour tester la chaine haîne **proxy** → **gateway** → **service** → **stockage** :

1. **L’app s’affiche** (le front est servi par le proxy).
2. Elle **demande la valeur** à `/api/visits` → on voit le chiffre à l’écran.
3. On refresh → le front appelle `/api/visit` et **ré-affiche** la nouvelle valeur.

FICHIERS

❌ A NE PAS MODIFIER

- L’**URL de base** pour communiquer avec le back : reste **`/api/...`**
- L’**URL du WebSocket** doit rester **sur la même origine que le site**, chemin **`/ws`**.
- `frontend/Dockerfile` : image de build/serveur (ne pas changer les ports/expose).
- **Convention d’URL** : utiliser **`/api/...`** (fetch) et **`/ws`** (WebSocket) pour rester **derrière le proxy** (pas d’URL absolues).
- `Makefile` (cibles globales), `proxy/nginx.conf` (routage `/api/*` & `/ws`) — **hors du frontend** mais impactent le fonctionnement.

⚠️ MODIFIABLE AVEC ATTENTION

`index.html`

- ✅ **Tu peux ajouter** : `<meta>` (SEO, responsive), `<link>` (favicon, fonts), nouveaux `<div>` si besoin pour des portails.
- ⚠️ **À conserver absolument** :
  - `<div id="root"></div>` → c'est là que React monte l'app.
  - `<script type="module" src="/src/main.tsx"></script>` → c'est ton point d'entrée.
- ❌ Ne pas supprimer ni renommer ces 2 éléments.

`package.json`

- ✅ **Tu peux ajouter** :
  - Nouvelles dépendances (`dependencies` ou `devDependencies`).
  - De nouveaux scripts dans `"scripts"` (ex. `lint`, `format`).

- ⚠️ **À conserver absolument** :
  - Les scripts de base (`dev`, `build`, `preview`).
  - Les dépendances React/Vite/Tailwind déjà présentes.
- ❌ Ne pas supprimer ou renommer les dépendances principales (sinon le projet ne démarre plus).

`package-lock.json`

- ⚠️ GénÉré automatiquement par `npm install`.
- ❌ Ne pas modifier à la main. Il peut être supprimé **seulement** si tu veux régénérer toutes les versions (`rm package-lock.json && npm install`).

`postcss.config.cjs`

- ✅ **Tu peux ajouter** des plugins PostCSS (ex. `cssnano`).
- ⚠️ **À conserver absolument** la config Tailwind et Autoprefixer déjà en place.

`tailwind.config.cjs`

- ✅ **Tu peux ajouter** : thèmes personnalisés, couleurs, breakpoints, plugins Tailwind.
  ⚠️ **À conserver absolument** : la clé `content` (sinon Tailwind ne purge pas bien les classes).

`tsconfig.json`

- ✅ **Tu peux ajuster** : options strictes (`strictNullChecks`), ajouter des libs (`ES2021` si besoin), chemins (`paths`).
- ⚠️ **À conserver absolument** :
  - `"jsx": "react-jsx"` (sinon React ne compile pas).
  - `"noEmit": true` (Vite gère le build, pas `tsc`).
  - `"include": ["src"]`.

`vite.config.ts`

- ✅ **Tu peux ajouter** :
  - Des plugins Vite (ex. alias de chemins, PWA, analyse de bundle).
  - Modifier la config du serveur (ex. proxy dev).
- ⚠️ **À conserver absolument** :
  - L'`export default defineConfig({ … })`.
  - Le plugin React déjà présent.

✅ MODIFIER LIBREMENT /SRC

`App.tsx`
⚠️ À conserver :

- l'export `default function App()` (c'est le point d'entrée monté par `main.tsx`), l'appel `<BrowserRouter>` si tu veux garder le routing.
- la logique de `useCountVisitOnceInline` peut être adaptée, mais garde en tête que c'est ton exemple de visite auto.

`main.tsx`
⚠️ À conserver :

- le montage sur `document.getElementById("root")` (c'est le lien avec `index.html`),
- l'import `App`

`index.css`
⚠️ À conserver :

- les trois lignes `@tailwind base;`, `@tailwind components;`, `@tailwind utilities;` (sinon Tailwind ne fonctionne plus).