Lancer l’infrastructure

1. Construire et démarrer les containers

```
docker compose up -d --build
```

- `-d` : lance en mode détaché (en arrière-plan).
- `--build` : force la reconstruction des images Docker.

2. Vérifier que les containers sont sains

```
docker compose ps
```

- Les services principaux (`gateway`, `frontend`, `nginx`, `db`) devraient indiquer `healthy`.

3. Tester l’infrastructure

Un script `test.sh` est fourni pour vérifier rapidement tous les services :

```
chmod +x test.sh
./test.sh
```

Chaque service retourne le code HTTP ou un message d’erreur si inaccessible

4. Ports utilisés par les services

ATTENTION : Toujours vérifier que les ports nécessaires sont libres avant de lancer Docker Compose.

Service	Port local	Port conteneur
Auth-Service	4000	4000
Gateway	4001	4001
User-Service	4002	4002
Game-Service	4003	4003
Frontend	3000	80
API BDD	3020	3020
API Blockchain	3021	3021
API User	3022	3022
Nginx	80	80
Elasticsearch	9200	9200
Logstash	5044	5044
Kibana	5601	5601
Prometheus	9090	9090
Grafana	3010	3000

---

5. Arrêter l’infrastructure

```
docker compose down
```

Utiliser l’infrastructure

Voici un guide avec les commandes et URLs pour accéder à tous les services du projet Transcendance une fois que l'infrastructure est lancée.

✅ **Conseils de debug**

- Vérifier les logs avec `docker logs <service>`.
- Vérifier si un port est utilisé : `sudo lsof -i:<port>`.
- Rebuild d'un service : `docker compose up -d --build <service>`.

1️⃣ Backend Services (Fastify)

Tous les services exposent un endpoint `/health` pour vérifier qu'ils fonctionnent.

Service	URL locale (navigateur/curl)	Docker exec (depuis container)
Auth-Service	`http://localhost:4000/health`	`docker exec -it auth-service curl http://localhost:4000/health`
Gateway	`http://localhost:4001/health`	`docker exec -it gateway curl http://localhost:4001/health`
User-Service	`http://localhost:4002/health`	`docker exec -it user-service curl http://localhost:4002/health`
Game-Service	`http://localhost:4003/health`	`docker exec -it game-service curl http://localhost:4003/health`

2️⃣ Frontend (React/Tailwind)

Le frontend est servi par Nginx via le reverse proxy sur le port `80` ou directement sur `3000` si tu veux bypasser le reverse proxy.

Accès	URL
Via Nginx	`http://localhost/`
Direct Frontend	`http://localhost:3000/`

4️⃣ ELK Stack (logs)

Service	URL locale	Notes
Elasticsearch	`http://localhost:9200/`	Retourne info JSON
Logstash	`nc -zv localhost 5044`	Vérifie si le port TCP est ouvert
Kibana	`http://localhost:5601/`	Interface graphique pour visualiser les logs

5️⃣ Monitoring (Prometheus / Grafana / cAdvisor / Node Exporter)

Service	URL locale	Notes
Prometheus	`http://localhost:9090/`	Dashboard Prometheus
Grafana	`http://localhost:3010/`	Dashboard Grafana, user: admin, pass: admin
cAdvisor	`http://localhost:8082/`	Visualisation containers Docker
Node Exporter	`http://localhost:9100/metrics`	Métriques système

6️⃣ Vault (gestion des secrets)

En mode dev, Vault est accessible sur le port 8200 :

```
export VAULT_ADDR='http://127.0.0.1:8200'
export VAULT_TOKEN='<root_token_affiché_dans_les_logs>'
```

- Tester si Vault fonctionne :

```
vault status
vault kv put secret/test key=value
vault kv get secret/test
```

7️⃣ Reverse Proxy Nginx

- Vérifier Nginx :

```
curl -I http://localhost/
```

- Pour accéder à des services via le proxy :

Service	URL via Nginx
Gateway	`http://localhost/gateway/health`
Auth	`http://localhost/auth/`
User	`http://localhost/user/`
Game	`http://localhost/game/`

Explication de l’infrastructure

Rôle de Nginx

- Nginx est un **serveur web et reverse proxy**.
- Dans le projet :
    - Le container `frontend` sert l' application React/Vite via Nginx.
    - Le container `nginx` (reverse proxy) sert à **rediriger les requêtes externes** vers le frontend ou le backend (API gateway), selon l'URL.
- Il peut aussi gérer des règles de sécurité, cache, compression, et HTTPS si besoin.

En résumé : Nginx sert juste à **réceptionner et router le trafic HTTP vers les bons services**.

Communication entre services

- **Backend ↔ Backend** :
    - Les services Node.js (auth-service, user-service, game-service) communiquent **via l' api-gateway** en HTTP (API REST).
    - Docker crée un **réseau interne**, chaque container a un nom DNS correspondant au service (`auth-service`, `gateway`, etc.).
    - Exemple : `curl http://auth-service:4000/...` depuis un autre container fonctionnera.
- **Frontend ↔ Backend** :
    - Le frontend React appelle l'api-gateway via HTTP (`http://gateway:4001/...`) pour récupérer les données.
- **Ports exposés** :
    - Les `ports` dans `docker-compose` servent surtout à rendre les services accessibles depuis notre machine hôte (localhost).
    - Exemple : `4001:4001` → tu peux accéder à la gateway depuis ton navigateur via `http://localhost:4001`.
    - Les containers communiquent **interne réseau Docker** → pas besoin d'exposer les ports pour que la communication interne fonctionne.

- **Reverse proxy Nginx** :
    - Il agit comme un **point d'entrée unique** pour l'extérieur.
    - Exemple : tout ce qui arrive sur `http://localhost/` peut être redirigé vers `frontend` ou `gateway`.

On va voir comment les services communiquent dans l' infra actuelle, en mettant l'accent sur le **frontend** → **gateway** → **backend** et la manière dont Docker gère ça.

1️⃣ Flux principal de communication

Explication pas-à-pas :

1. **Frontend (React + Nginx)**
    - L'utilisateur ouvre le navigateur → Nginx sert les fichiers statiques (`index.html`, JS, CSS).
    - Pour récupérer des données dynamiques, le frontend fait des appels HTTP vers la **gateway** (ex: `http://gateway:4001/api/...`).
2. **Gateway**
    - Sert de **point central** pour toutes les requêtes API.
    - Redirige les appels vers le service backend correspondant :
        - `auth-service` pour l'authentification
        - `user-service` pour gérer les utilisateurs
        - `game-service` pour le jeu
    - Cela permet au frontend de ne connaître qu'une seule URL (`gateway`) au lieu de plusieurs services.

3. **Backend Services**
    - Chaque microservice fait son boulot (CRUD, logique métier, DB access).
    - Ils exposent leurs endpoints sur le réseau Docker interne.
4. **Communication via Docker réseau interne**
    - Chaque service peut appeler un autre **par son nom de service** (`gateway`, `auth-service`) sans passer par les ports exposés à l'extérieur.

Exemple :

```
curl http://auth-service:4000/login
```

2️⃣ Points importants à retenir

Élément	Rôle / Astuce
Nginx frontend	Sert le build React statique (`index.html`) et peut faire du reverse proxy si nécessaire.
Gateway	Centralise les appels API, gère l'authentification et redirige vers le bon backend.

3️⃣ Exemple concret d'un appel API

Disons que le frontend veut afficher le profil utilisateur :

1. Frontend React envoie une requête HTTP :
   `fetch("http://gateway:4001/api/users/me")`

2. Gateway reçoit la requête, vérifie le token, et envoie :

   ```
   curl http://user-service:4002/me
   ```

3. User-service consulte la DB et renvoie les infos à la gateway.
4. Gateway renvoie la réponse finale au frontend.
5. Nginx frontend sert le JSON au navigateur.

💡 **Astuce pratique** : tu peux tester les appels entre containers avec :

```
docker exec -it frontend sh
curl http://gateway:4001/health
curl http://user-service:4002/me
```

Ça te permet de vérifier que le réseau interne fonctionne avant de chercher un problème côté code.

Très bonne remarque 👌 C'est une question que beaucoup se posent quand ils découvrent les microservices.

En théorie, oui : ton **frontend pourrait directement appeler** `auth-service`, `user-service`, `game-service` etc. Mais en pratique, ça pose plusieurs problèmes. C'est exactement pour ça qu'on introduit une **API Gateway**.

🎯 Rôles et intérêts d'une API Gateway

1. Point d'entrée unique

- Sans gateway : ton frontend doit connaître **toutes les URLs** des microservices (`/auth/login`, `/users/42`, `/game/start`, etc.).

- Avec gateway : le frontend n'appelle qu'une seule URL (`/api/...`), et la gateway **redirige vers le bon service**.

👉 Résultat : simplicité côté frontend et plus facile de changer l'architecture derrière.

---

2. Sécurité

- La gateway sert de **barrière de sécurité** :
    - **WAF (ModSecurity)** intégré.
    - Vérification des **JWT / 2FA** avant de laisser passer une requête.
    - Limitation des appels (rate limiting, anti-DDOS).
    - Centralisation de la gestion **CORS**.

👉 Ça évite d'exposer directement tes microservices sensibles (comme `auth-service`) sur Internet.

---

3. Centralisation de l'authentification

- Si tu n'as pas de gateway : chaque microservice doit **vérifier lui-même les tokens JWT** ou la 2FA.
- Avec gateway : une seule vérification, puis la gateway transfère la requête aux services internes.

👉 Gain énorme en **simplicité et cohérence**.

---

4. Flexibilité & évolutivité

- Tu peux faire évoluer l'architecture **sans casser le frontend**.
  Exemple :
    - Tu changes `game-service` → `game-service-v2`.

- Le frontend ne change rien, c'est la gateway qui reroute `/api/game` vers le bon service.

👉 Cela permet de déployer **plusieurs versions** d'un service en parallèle (utile en prod).

---

5. Observabilité

- Tu peux centraliser :
  - **Logs des requêtes**
  - **Metrics (Prometheus/Grafana)**
  - **Monitoring des erreurs**
    → directement au niveau de la gateway.

---

6. Support des features cross-services

Certaines fonctionnalités nécessitent de croiser plusieurs microservices :

- Exemple :
  - `/api/me` → récupère les infos utilisateur (user-service) + ses scores (game-service) + son historique de tournois (tournament-service).
- Sans gateway : le frontend doit faire **3 appels différents** et recoller les morceaux.
- Avec gateway : la gateway orchestre ça et ne renvoie **qu'une seule réponse**.

👉 Ça améliore l'**expérience utilisateur** (moins de requêtes, plus rapide).

⚖️ Comparatif

Approche	Avantages	Inconvénients

Frontend → Services directs	Simple à mettre en place, pas d'intermédiaire	Multiplication des endpoints à gérer, sécurité plus difficile, duplication du code d'auth, changement d'URLs casse tout
Frontend → API Gateway → Services	Point d'entrée unique, sécurité centralisée, simplification du frontend, évolutif, monitoring centralisé	Ajoute une couche supplémentaire à gérer (infra + config)

✅ En résumé :

- **Sans API Gateway** = faisable pour un petit projet, mais vite ingérable si tu grossis.
- **Avec API Gateway** = plus robuste, plus pro, exactement ce qui est attendu dans une archi **microservices** (surtout pour ton module DevOps).

Veux-tu que je t'écrive un **exemple concret de config Fastify Gateway** pour ton projet (genre `fastify-gateway.config.js`) afin que tu voies comment ça route les requêtes entre frontend et services ?

Avantage et Inconvénient d'une infrastructure multi-services

Avoir beaucoup de containers et de microservices apporte **des avantages**, mais aussi des **difficultés**, surtout pour la communication et la maintenance.

1️⃣ Avantages d'une architecture multi-containers / microservices

1. **Isolation et modularité**
    - Chaque service est dans son propre container → il peut être démarré, mis à jour ou redémarré **sans affecter les autres**.
    - Facilite le développement, les tests et le déploiement indépendant de chaque service.
2. **Scalabilité**
    - Tu peux monter ou descendre individuellement chaque service selon la charge.
    - Exemple : si `frontend` ou `game-service` est très sollicité, tu peux lancer plusieurs instances sans toucher au reste.
3. **Technologies différentes**
    - Chaque microservice peut utiliser **une stack adaptée** (Node.js, Python, Go…) sans casser l'architecture globale.
4. **Maintenance et mise à jour plus facile**
    - Les bugs ou mises à jour sont limités à un seul service → moins de risques de casse.
5. **Déploiement et CI/CD simplifiés**
    - Les pipelines peuvent rebuild / tester un service sans rebuild l'ensemble.

2️⃣ Difficultés et challenges

1. **Complexité du réseau**
    - Les services doivent communiquer via un réseau Docker interne ou via API.
    - Mauvaise configuration DNS ou ports → services "unreachable".
2. **Orchestration et dépendances**
    - Certains services doivent démarrer dans un certain ordre (ex. `frontend` dépend de `gateway` → dépendance saine via healthcheck).

- Si un service échoue, il peut casser toute la chaîne.

3. **Monitoring et debugging**
    - Avec 15 containers, suivre logs, métriques et erreurs devient compliqué.
    - Tu dois centraliser logs (ELK) et métriques (Prometheus/Grafana).
4. **Performance et ressources**
    - Chaque container consomme CPU / RAM → beaucoup de services peuvent saturer la machine.
    - Optimiser Dockerfiles et limiter les ressources devient important.
5. **Sécurité**
    - Plus de services = plus de surfaces d'attaque.
    - Il faut gérer correctement les secrets, certificats, et isolation réseau.

3️⃣ Gestion efficace de la communication entre services

1. **Réseau Docker interne**
    - Chaque service peut appeler un autre via son **nom de service** (`auth-service`, `gateway`, etc.) → pas besoin de passer par les ports exposés.
    - Exemple : `curl http://user-service:4002/api/users`.
2. **API Gateway / Reverse proxy**
    - Centralise les appels externes → simplifie le frontend.
    - Permet aussi de gérer **authentification, throttling, logs et sécurité**.
3. **Healthchecks et dépendances**
    - Docker healthchecks assurent qu'un service n'est utilisé que s'il est prêt.
    - `depends_on` + `condition: service_healthy` dans docker-compose est utile mais limité → pour prod, un orchestrateur comme **Kubernetes** est plus

robuste.

4. **Message broker / Queue (optionnel)**
    - Pour les services qui doivent communiquer **asynchrone**, utiliser RabbitMQ, Kafka, ou Redis peut aider à dé-coupler les services.
5. **Documentation & conventions**
    - Définir des **API contract** clairs (ex. Swagger/OpenAPI) pour que chaque service sache comment communiquer.

💡 **Résumé** :

- Les microservices et containers donnent **modularité, scalabilité et indépendance**, mais ajoutent de la **complexité réseau et orchestration**.
- Pour gérer la communication : réseau interne Docker, API Gateway, healthchecks, et éventuellement des queues pour l'asynchrone.