BDD Twitch

Architecture Microservices avec Stack d'Observabilité

Documentation Technique Complète

Falilou

7 janvier 2026

Table des matières

Chapitre 1

Introduction

1.1 Vue d'Ensemble du Projet

Ce projet présente une plateforme de streaming vidéo inspirée de Twitch, construite avec une architecture microservices moderne. L'objectif principal est de démontrer les meilleures pratiques en matière de conception d'architecture distribuée, d'observabilité et de scalabilité.

1.1.1 Objectifs du Projet

— Implémenter une architecture microservices complète avec bases de données séparées

— Mettre en place une stack d'observabilité professionnelle (Prometheus, Grafana, AlertManager)

— Démontrer la scalabilité horizontale et la résilience

— Appliquer les best practices de l'industrie (Netflix, Uber, Spotify)

— Fournir une base solide pour un projet de production

1.1.2 Contexte et Motivation

Dans le développement moderne d'applications, la conteneurisation avec Docker et l'orchestration avec Docker Compose sont devenues des standards. Ce projet est né de la nécessité d'avoir une stack de monitoring complète dès le début du développement, plutôt que de l'ajouter après coup.

L'architecture microservices a été choisie pour ses avantages en termes de :

— **Isolation** : Chaque service est indépendant

— **Scalabilité** : Possibilité de scaler uniquement les services sous charge

— **Résilience** : Une panne n'affecte pas l'ensemble du système

— **Flexibilité** : Déploiements et mises à jour indépendants

Chapitre 2

Architecture Microservices

2.1 Principes de l'Architecture

2.1.1 Pattern "Database per Service"

L'architecture suit le pattern *Database per Service*, recommandé par les experts en microservices comme Martin Fowler et Sam Newman. Chaque microservice possède sa propre base de données privée, garantissant :

Avantages

- Couplage faible entre les services
- Possibilité de choisir la technologie de base de données adaptée à chaque service
- Isolation des données et sécurité renforcée
- Scalabilité indépendante de chaque base

Inconvénients

- Pas de transactions distribuées natives
- Duplication potentielle de certaines données
- Complexité opérationnelle accrue
- Nécessité de patterns comme Saga ou Event Sourcing

2.2 Les 5 Microservices

2.2.1 Users Service (Port 5432)

Responsabilité : Gestion complète des utilisateurs, authentification et autorisation.
Tables principales :

- `users` : Profils utilisateurs avec informations personnelles
- `followers` : Relations de suivi entre utilisateurs
- `user_roles` : Système de permissions et rôles
- `user_settings` : Préférences et paramètres utilisateur

— `user_sessions` : Gestion des sessions actives

Cas d'usage :

— Inscription et connexion

— Gestion du profil

— Système de followers/following

— Authentification JWT

2.2.2 Streams Service (Port 5433)

Responsabilité : Gestion des streams en direct, VODs et catégories.
Tables principales :

— `streams` : Streams en direct avec métadonnées

— `categories` : Catégories de contenu (Gaming, Music, etc.)

— `vods` : Vidéos à la demande

— `clips` : Extraits de streams

— `stream_tags` : Tags pour la découvrabilité

Cas d'usage :

— Démarrage/arrêt de stream

— Gestion des VODs

— Création de clips

— Recherche par catégorie

2.2.3 Chat Service (Port 5434)

Responsabilité : Système de chat en temps réel avec modération.
Tables principales :

— `chat_messages` : Messages du chat

— `emotes` : Emotes personnalisées

— `chat_moderators` : Modérateurs de chat

— `banned_users` : Utilisateurs bannis

— `chat_badges` : Badges de chat

Cas d'usage :

— Envoi de messages en temps réel

— Modération (timeout, ban)

— Utilisation d'emotes

— Gestion des badges

2.2.4 Subscriptions Service (Port 5435)

Responsabilité : Gestion des abonnements, donations et monétisation.
Tables principales :

— `subscriptions` : Abonnements actifs
— `subscription_tiers` : Niveaux d'abonnement (Tier 1, 2, 3)
— `donations` : Dons des viewers
— `channel_points` : Points de chaîne
— `payment_methods` : Moyens de paiement

Cas d'usage :

— Souscription à un abonnement
— Gestion des dons
— Système de points de chaîne
— Abonnements cadeaux

2.2.5 Analytics Service (Port 5436)

Responsabilité : Collecte et analyse des métriques de la plateforme.
Tables principales :

— `stream_analytics` : Statistiques de streams
— `user_analytics` : Métriques utilisateurs
— `revenue_analytics` : Analyses financières
— `engagement_metrics` : Métriques d'engagement

Cas d'usage :

— Statistiques de viewers
— Analyses de revenus
— Métriques d'engagement
— Rapports pour les streamers

2.3 Communication Entre Services

2.3.1 Approches de Communication

1. API REST : Pour les communications synchrones

```
# Exemple : Users Service appelle Subscriptions Service
GET /api/subscriptions/user/123
```

2. Message Queue : Pour les communications asynchrones (recommandé en production)

— RabbitMQ ou Apache Kafka
— Pattern Publish/Subscribe
— Event Sourcing

3. API Gateway : Point d'entrée unique (NGINX)

— Routage des requêtes
— Load balancing
— Rate limiting
— Authentification centralisée

Chapitre 3

Stack d'Observabilité

3.1 Architecture de Monitoring

L'observabilité est un pilier fondamental de cette architecture. La stack complète permet de surveiller, analyser et alerter sur l'état du système en temps réel.

3.1.1 Composants de la Stack

Composant	Port	Rôle
Prometheus	9090	Collecte et stockage des métriques
Grafana	3000	Visualisation et dashboards
AlertManager	9093	Gestion des alertes
Node Exporter	9100	Métriques système (CPU, RAM, Disk)
cAdvisor	8080	Métriques des conteneurs Docker
PostgreSQL Exporter	9187-9191	Métriques des bases PostgreSQL
Redis Exporter	9121	Métriques Redis

TABLE 3.1 – Composants de la stack d'observabilité

3.2 Prometheus

3.2.1 Configuration

Prometheus est configuré pour collecter les métriques toutes les 15 secondes. Les targets incluent :

— Les 5 bases PostgreSQL via leurs exporters

— Redis via redis_exporter

— Le système hôte via node_exporter

— Les conteneurs Docker via cAdvisor

— Prometheus lui-même

3.2.2 Métriques Clés

PostgreSQL :

```
# Etat des bases
pg_up

# Connexions actives
pg_stat_database_numbackends

# Cache Hit Ratio
rate(pg_stat_database_blks_hit[5m]) /
(rate(pg_stat_database_blks_hit[5m]) +
 rate(pg_stat_database_blks_read[5m]))

# Transactions par seconde
rate(pg_stat_database_xact_commit[5m])
```

Système :

```
# CPU Usage
100 - (avg(irate(node_cpu_seconds_total{mode="idle"}[5m])) * 100)

# Memory Usage
100 * (1 - (node_memory_MemAvailable_bytes /
            node_memory_MemTotal_bytes))
```

3.3 Grafana

3.3.1 Dashboards Préconfigurés

Deux dashboards sont automatiquement provisionnés au démarrage :

1. PostgreSQL Microservices - Vue d'Ensemble

— État des 5 bases de données (UP/DOWN)
— Connexions actives par base
— Transactions par seconde (commits + rollbacks)
— Cache Hit Ratio (doit être ¿ 95%)
— Opérations INSERT/UPDATE/DELETE
— Lignes lues par base

2. Système & Infrastructure - Vue d'Ensemble

— CPU et Memory Usage
— Conteneurs Docker actifs
— État de tous les services
— Métriques Redis (clients, mémoire, clés, commandes/sec)

3.3.2 Provisioning Automatique

La datasource Prometheus et les dashboards sont automatiquement configurés via :

— `monitoring/grafana/provisioning/datasources/prometheus.yml`
— `monitoring/grafana/provisioning/dashboards/dashboard.yml`
— `monitoring/grafana/dashboards/*.json`

3.4 AlertManager

3.4.1 Alertes Configurées

Alerte	Sévérité	Condition
PostgreSQL Down	Critical	Base de données inaccessible ¿ 1 min
High CPU Usage	Warning	CPU ¿ 80% pendant 5 min
High Memory Usage	Warning	RAM ¿ 85% pendant 5 min
Too Many Connections	Warning	Connexions PostgreSQL ¿ 80%
Low Cache Hit Ratio	Warning	Cache Hit ¡ 90% pendant 10 min
Disk Space Low	Critical	Espace disque ¡ 15%
High Transaction Rollback	Warning	Rollbacks ¿ 5% des transactions
Redis Down	Critical	Redis inaccessible ¿ 1 min
Container Down	Critical	Conteneur arrêté ¿ 2 min
High Response Time	Warning	Temps de réponse ¿ 1s

TABLE 3.2: Alertes configurées dans AlertManager

Chapitre 4

Schémas de Bases de Données

4.1 Users Database

4.1.1 Table : users

```sql
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    uuid UUID DEFAULT uuid_generate_v4() UNIQUE NOT NULL,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    display_name VARCHAR(100),
    bio TEXT,
    profile_image_url TEXT,
    banner_image_url TEXT,
    is_verified BOOLEAN DEFAULT FALSE,
    is_partner BOOLEAN DEFAULT FALSE,
    is_affiliate BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

4.1.2 Table : followers

```sql
CREATE TABLE followers (
    id SERIAL PRIMARY KEY,
    follower_user_id INTEGER NOT NULL,
    followed_user_id INTEGER NOT NULL,
    followed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    notifications_enabled BOOLEAN DEFAULT TRUE,
    UNIQUE(follower_user_id, followed_user_id)
);
```

4.2 Streams Database

4.2.1 Table : streams

```sql
CREATE TABLE streams (
    id SERIAL PRIMARY KEY,
```

```
    uuid UUID DEFAULT uuid_generate_v4() UNIQUE NOT NULL,
    user_id INTEGER NOT NULL,
    category_id INTEGER,
    title VARCHAR(255) NOT NULL,
    description TEXT,
    is_live BOOLEAN DEFAULT FALSE,
    viewer_count INTEGER DEFAULT 0,
    started_at TIMESTAMP,
    ended_at TIMESTAMP,
    thumbnail_url TEXT,
    language VARCHAR(10) DEFAULT 'fr',
    is_mature BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

4.3 Optimisations et Index

Tous les schémas incluent :

- Index sur les clés étrangères
- Index sur les colonnes fréquemment requêtées
- Triggers pour `updated_at` automatique
- Vues matérialisées pour les requêtes complexes
- Contraintes de clés étrangères appropriées

Chapitre 5

Déploiement et Opérations

5.1 Démarrage du Système

5.1.1 Prérequis

— Docker 20.10 ou supérieur
— Docker Compose 2.0 ou supérieur
— 8GB RAM minimum
— 20GB espace disque disponible

5.1.2 Installation

```
# Cloner le repository
git clone https://github.com/Falilou2099/ProjetTwitch.git
cd ProjetTwitch

# Copier la configuration
cp .env.example .env

# Demarrer tous les services
./start.sh
```

5.1.3 Vérification

```
# Verifier l'etat des services
./check-status.sh

# Tester les bases de donnees
./test-databases.sh

# Voir les logs
docker compose logs -f
```

5.2 Scalabilité

5.2.1 Scaling Horizontal

Chaque service peut être scalé indépendamment :

```
# Scaler le service Chat (haute charge)
docker compose up -d --scale chat-db=3

# Scaler le service Streams
docker compose up -d --scale streams-db=2
```

5.2.2 Optimisations de Performance

En développement :

— Ressources limitées par conteneur

— Logs en mode debug

— Hot reload activé

En production :

— Connection pooling (PgBouncer)

— Read replicas pour les bases très lues

— Cache Redis pour les données fréquentes

— CDN pour les assets statiques

— Load balancing avec NGINX

5.3 Maintenance

5.3.1 Backup des Bases de Données

```
# Backup d'une base
docker exec twitch-users-db pg_dump -U twitch_user users_db >
    backup_users.sql

# Backup de toutes les bases
./backup-all-databases.sh

# Restauration
docker exec -i twitch-users-db psql -U twitch_user users_db <
    backup_users.sql
```

5.3.2 Mises à Jour

```
# Mettre a jour les images
docker compose pull

# Redemarrer avec les nouvelles images
docker compose up -d

# Verifier les logs
docker compose logs -f
```

Chapitre 6

Sécurité

6.1 Bonnes Pratiques Implémentées

6.1.1 Isolation Réseau

— Réseau Docker privé (`twitch-network`)
— Pas d'exposition directe des bases de données
— API Gateway comme point d'entrée unique

6.1.2 Gestion des Secrets

— Variables d'environnement dans `.env`
— Fichier `.env` dans `.gitignore`
— Rotation régulière des mots de passe
— Utilisation de Docker Secrets en production

6.1.3 Authentification et Autorisation

— JWT pour l'authentification
— RBAC (Role-Based Access Control)
— Rate limiting sur l'API Gateway
— Protection CSRF

6.2 Recommandations pour la Production

Checklist Sécurité

- [ ] Activer HTTPS avec certificats SSL/TLS
- [ ] Configurer un firewall (UFW, iptables)
- [ ] Mettre en place un WAF (Web Application Firewall)
- [ ] Activer l'audit logging
- [ ] Configurer fail2ban
- [ ] Mettre à jour régulièrement les images Docker
- [ ] Scanner les vulnérabilités (Trivy, Clair)
- [ ] Limiter les ressources par conteneur
- [ ] Utiliser des utilisateurs non-root dans les conteneurs
- [ ] Chiffrer les backups

Chapitre 7

Dépannage

7.1 Problèmes Courants

7.1.1 Prometheus n'affiche pas de métriques

Symptômes : Grafana est vide, pas de données dans Prometheus
Solutions :

1. Vérifier que les targets sont UP dans Prometheus (`http://localhost:9090/targets`)
2. Attendre 30 secondes pour la première collecte
3. Redémarrer Prometheus : `docker compose restart prometheus`
4. Vérifier les logs : `docker logs twitch-prometheus`

7.1.2 Base de données inaccessible

Symptômes : Erreur de connexion à PostgreSQL
Solutions :

1. Vérifier que le conteneur est UP : `docker compose ps`
2. Vérifier les logs : `docker logs twitch-users-db`
3. Tester la connexion : `docker exec twitch-users-db pg_isready`
4. Redémarrer la base : `docker compose restart users-db`

7.1.3 Conteneur qui redémarre en boucle

Symptômes : Un conteneur se relance continuellement
Solutions :

1. Voir les logs : `docker logs [conteneur] --tail 100`
2. Vérifier les ressources disponibles : `docker stats`
3. Vérifier les health checks dans `docker-compose.yml`
4. Augmenter les limites de ressources si nécessaire

7.2 Commandes de Diagnostic

```
# Verifier l'etat de tous les conteneurs
docker compose ps

# Voir les logs en temps reel
docker compose logs -f

# Verifier l'utilisation des ressources
docker stats

# Inspecter un conteneur
docker inspect [conteneur]

# Executer une commande dans un conteneur
docker exec -it [conteneur] bash

# Tester la connectivite reseau
docker exec [conteneur] ping [autre-conteneur]
```

Chapitre 8

Évolutions Futures

8.1 Améliorations Prévues

8.1.1 Phase 1 : Optimisations

— Implémentation de PgBouncer pour le connection pooling
— Ajout de Read Replicas pour les bases très lues
— Migration du Chat vers Redis pour de meilleures performances temps réel
— Mise en place de Kubernetes pour l'orchestration

8.1.2 Phase 2 : Fonctionnalités

— API REST complète pour chaque service
— Frontend React/Next.js
— Streaming vidéo avec WebRTC
— Système de notifications en temps réel
— Application mobile (React Native)

8.1.3 Phase 3 : Infrastructure

— CI/CD avec GitHub Actions
— Déploiement sur AWS/GCP/Azure
— CDN pour la distribution de contenu
— Elasticsearch pour la recherche
— Apache Kafka pour l'event streaming

8.2 Technologies à Explorer

Technologie	**Usage**
Kubernetes	Orchestration de conteneurs
Istio	Service mesh
Elasticsearch	Recherche full-text
Apache Kafka	Event streaming
gRPC	Communication inter-services
Consul	Service discovery
Vault	Gestion des secrets
Jaeger	Distributed tracing
Fluentd	Log aggregation
MinIO	Stockage objet (S3-compatible)

TABLE 8.1 – Technologies pour les évolutions futures

Chapitre 9

Conclusion

9.1 Résumé du Projet

Ce projet démontre une architecture microservices complète et professionnelle, incluant :

- 5 microservices indépendants avec bases de données séparées
- Stack d'observabilité complète (Prometheus, Grafana, AlertManager)
- Dashboards préconfigurés et alertes automatiques
- Documentation exhaustive et scripts d'automatisation
- Bonnes pratiques de sécurité et scalabilité

9.2 Apprentissages Clés

9.2.1 Architecture

- Le pattern "Database per Service" offre une excellente isolation mais nécessite une gestion attentive des transactions distribuées
- L'observabilité doit être intégrée dès le début, pas ajoutée après coup
- La documentation est aussi importante que le code

9.2.2 Opérations

- Docker Compose est excellent pour le développement et les petits déploiements
- Les health checks sont essentiels pour la résilience
- Les scripts d'automatisation facilitent grandement les opérations

9.2.3 Monitoring

- Prometheus + Grafana forment une stack puissante et flexible
- Les dashboards préconfigurés accélèrent le time-to-value
- Les alertes doivent être pertinentes pour éviter la fatigue d'alerte

9.3 Remerciements

Ce projet s'inspire des meilleures pratiques de l'industrie, notamment :

— **Netflix** : Architecture microservices et résilience

— **Uber** : Scalabilité et gestion de la charge

— **Spotify** : Organisation des équipes et des services

— **Google SRE** : Observabilité et fiabilité

Merci à la communauté open-source pour les outils exceptionnels utilisés dans ce projet.

Annexe A

Annexes

A.1 Références

- Martin Fowler - *Microservices* : https://martinfowler.com/articles/microservices.html
- Sam Newman - *Building Microservices*
- Chris Richardson - *Microservices Patterns*
- Google - *Site Reliability Engineering*
- Prometheus Documentation : https://prometheus.io/docs/
- Grafana Documentation : https://grafana.com/docs/

A.2 Glossaire

Microservice Architecture où l'application est composée de services indépendants

Observabilité Capacité à comprendre l'état interne d'un système via ses outputs

Prometheus Système de monitoring et d'alerting open-source

Grafana Plateforme de visualisation et d'analytics

Docker Plateforme de conteneurisation

PostgreSQL Système de gestion de base de données relationnelle

Redis Base de données en mémoire pour le cache

NGINX Serveur web et reverse proxy

API Gateway Point d'entrée unique pour les APIs

Health Check Vérification de l'état de santé d'un service