# **Architecture Logicielle**

# **Introduction**

La conception de l’architecture logicielle est une étape clé dans la réussite du projet OnlyFlick. Elle permet de structurer les différentes couches techniques de l’application, d’assurer une bonne séparation des responsabilités et de garantir la performance, la sécurité et la maintenabilité du système. Le choix retenu pour ce projet est une **architecture client-serveur basée sur une API REST en Go (backend)** et un **frontend Flutter** pour les plateformes **Web et Mobile**.

## **Objectifs de l’architecture**

* Favoriser la **modularité** : séparation claire entre client et serveur
* Permettre une **scalabilité** horizontale (ajout de services ou serveurs)
* Assurer la **sécurité** des données (authentification, droits, paiements)
* Maintenir une bonne **performance** (temps de réponse, consommation mémoire)
* Faciliter les **tests**, la **CI/CD** et les déploiements

## **Description de l’architecture choisie**

### **1. Frontend – Flutter**

* Développé en **Dart** avec Flutter
* Génère une application Web (SPA) + Android (APK)
* Consomme l’API REST via **HTTP/JSON**
* Responsif (tablette, smartphone, navigateur)
* Gestion de session (token JWT stocké en mémoire sécurisée ou cache local)
* Composants dynamiques, interactions utilisateur (UI/UX modernes)

#### 

#### Arborescence Complète

Thinkshare/frontend/

├── lib/

│ ├── main.dart # Bootstrap de l'application

│ ├── config/

│ │ ├── app\_theme.dart # Thèmes, couleurs, typographie

│ │ └── app\_routes.dart # Définition des routes (GoRouter)

│ │

│ ├── core/ # Composants partagés

│ │ ├── widgets/ # Boutons, badges, avatars

│ │ ├── utils/ # Validators, constants, network helper

│ │ └── models/ # Entités partagées (UserModel, PostModel)

│ │

│ ├── features/ # Organisation par fonctionnalité

│ │ ├── auth/

│ │ │ ├── data/

│ │ │ │ ├── auth\_api.dart # Appels HTTP (login, register)

│ │ │ │ └── auth\_repository.dart

│ │ │ └── presentation/

│ │ │ ├── providers/

│ │ │ │ └── auth\_provider.dart

│ │ │ ├── screens/

│ │ │ │ ├── login\_screen.dart

│ │ │ │ └── register\_screen.dart

│ │ │ └── widgets/

│ │ │ ├── login\_form.dart

│ │ │ └── register\_form.dart

│ │ │

│ │ ├── home/

│ │ │ ├── data/

│ │ │ │ ├── home\_api.dart

│ │ │ │ └── home\_repository.dart

│ │ │ └── presentation/

│ │ │ ├── providers/

│ │ │ │ └── home\_provider.dart

│ │ │ ├── screens/

│ │ │ │ ├── feed\_screen.dart

│ │ │ │ └── home\_detail\_screen.dart

│ │ │ └── widgets/

│ │ │ ├── post\_card.dart

│ │ │ └── comment\_tile.dart

│ │ │

│ │ ├── message/

│ │ │ ├── data/

│ │ │ │ ├── message\_api.dart

│ │ │ │ └── message\_repository.dart

│ │ │ └── presentation/

│ │ │ ├── providers/

│ │ │ │ └── message\_provider.dart

│ │ │ ├── screens/

│ │ │ │ └── chat\_screen.dart

│ │ │ └── widgets/

│ │ │ └── message\_bubble.dart

│ │ │

│ │ ├── payment/

│ │ │ ├── data/

│ │ │ │ ├── payment\_api.dart

│ │ │ │ └── payment\_repository.dart

│ │ │ └── presentation/

│ │ │ ├── providers/

│ │ │ │ └── payment\_provider.dart

│ │ │ ├── screens/

│ │ │ │ ├── subscription\_screen.dart

│ │ │ │ └── checkout\_screen.dart

│ │ │ └── widgets/

│ │ │ └── subscription\_card.dart

│ │ │

│ │ └── admin/

│ │ ├── data/

│ │ │ └── admin\_repository.dart

│ │ └── presentation/

│ │ ├── providers/

│ │ │ └── admin\_provider.dart

│ │ ├── screens/

│ │ │ └── admin\_dashboard\_screen.dart

│ │ └── widgets/

│ │ ├── report\_tile.dart

│ │ └── user\_list.dart

│ │

│ ├── providers/ # Providers globaux (ThemeProvider, Connectivity)

│ └── services/ # Services transverses (ApiService, NotificationService)

│ ├── api\_service.dart # Configuration Dio/baseURL, interceptors

│ └── notification\_service.dart

│

├── assets/

│ ├── images/

│ └── fonts/

│

├── test/ # Tests unitaires et widget tests

│

├── pubspec.yaml

└── README.md # Documentation du frontend

**Points clés :**

* **Feature-first** : isolation des responsabilités par domaine fonctionnel.
* **Clean layers** : data, domain, presentation pour chaque feature.
* **Services** : centralisation de la communication HTTP, gestion des tokens.
* **Providers** : ChangeNotifier ou Riverpod pour gérer l’état.

### **2. Backend – Go (Golang)**

* API REST sécurisée avec JWT et middlewares (auth, logging, CORS, etc.)
* Organisation possible : architecture **hexagonale** ou **MVC avec services/dépôts**
* Serveur HTTP (Gin ou Fiber recommandé)
* Routage : /auth, /posts, /users, /payments, /admin, etc.
* Gestion centralisée des erreurs, logs formatés, vérifications d’accès

#### Arborescence Complète

Thinkshare/backend/

├── cmd/

│ └── server/

│ └── main.go # Point d'entrée : configuration et lancement du serveur

│

├── internal/ # Code métier interne

│ ├── auth/ # Authentification & autorisation

│ │ ├── handler.go # Handlers HTTP (/login, /logout, /refresh)

│ │ ├── service.go # Logique : génération JWT, 2FA

│ │ ├── repository.go # Accès DB pour tokens, sessions  
  
│ │ ├── middleware.go # Middleware pour sécuriser les routes

│ │ └── model.go # Structures Auth (Credentials, Token)

│ │

│ ├── user/ # Gestion des profils

│ │ ├── handler.go # Handlers (/users, /users/:id)

│ │ ├── service.go # Logique métier (CRUD, désactivation)

│ │ ├── repository.go # Requêtes SQL/PostgreSQL

│ │ └── model.go # User, ProfileDTO, UpdateUserDTO

│ │

│ ├── post/ # Gestion des publications

│ │ ├── handler.go # Handlers (/posts, /posts/:id)

│ │ ├── service.go # Règles (visibilité, filtrage)

│ │ ├── repository.go # Accès DB pour posts, comments

│ │ └── model.go # Post, Comment, PostDTO

├── comment/ # Gestion des commentaires (attachés à un post)

│ │ ├── handler.go # POST /comments, GET /posts/:id/comments

│ │ ├── service.go

│ │ ├── repository.go

│ │ ├── model.go

├── like/ # Gestion des likes (post ou commentaire)

│ │ ├── handler.go # POST /likes, DELETE /likes/:id

│ │ ├── service.go

│ │ ├── repository.go

│ │ ├── model.go

│ │

│ ├── message/ # Messagerie privée

│ │ ├── handler.go # Handlers (/users/:id/messages)

│ │ ├── service.go # Logique d’envoi, modification

│ │ ├── repository.go # Accès DB pour messages

│ │ └── model.go # Message, Conversation

│ │

│ ├── payment/ # Paiements & abonnements

│ │ ├── handler.go # Handlers (/payments, /subscriptions)

│ │ ├── service.go # Intégration Stripe, webhooks

│ │ ├── repository.go # Transactions, abonnements

│ │ └── model.go # Payment, Subscription

│ │

│ ├── admin/ # Administration & modération

│ │ ├── handler.go # Handlers (/admin/users, /admin/reports)

│ │ ├── service.go # Modération, rapports

│ │ ├── repository.go # Logs, signalements

│ │ └── model.go # Report, AdminAction

│ │

│ └── stats/ # Statistiques & dashboard

│ ├── handler.go # Handlers (/stats/creator, /stats/admin)

│ ├── service.go # Génération KPI, graphes

│ ├── repository.go # Agrégations SQL

│ └── model.go # StatsDTO

│

├── pkg/ # Bibliothèques réutilisables

│ ├── config/ # Chargement env, YAML

│ ├── logger/ # Logs formatés (zap, logrus)

│ ├── middleware/ # CORS, rate limiting

│ └── utils/ # Helpers (validation, hash)

│

├── api/ # Spécification OpenAPI/Swagger

│ └── openapi.yaml

│

├── migrations/ # Migrations SQL (Flyway / Goose)

│ └── V1\_\_init.sql

│

├── scripts/ # Scripts utilitaires (seed, test)

│ └── seed\_users.go

│

├── test/

│ ├── unit/ # Tests unitaires avec mocks

│ │ ├── user\_test.go

│ │ └── auth\_test.go

│ └── integration/ # Tests d’intégration avec vraie DB

│ ├── user\_integration\_test.go

│ └── auth\_integration\_test.go

│

├── go.mod

├── go.sum

└── Dockerfile # Construction de l'image backend

**Description** :

* Chaque dossier **internal/** module contient handler.go, service.go, repository.go, model.go pour une séparation claire controller/business/data/model.
* **pkg/** regroupe la configuration, middleware et outils partagés.
* **api/** centralise la doc OpenAPI.
* **migrations/** et **scripts/** pour la gestion de la base.

## **Exemple de flux d'interaction (workflow utilisateur)**

### **Exemple : un abonné consulte un contenu premium**

1. L’utilisateur se connecte via Flutter → envoie les credentials à /auth/login
2. L’API Go vérifie et renvoie un token JWT
3. L’utilisateur clique sur un post premium → Flutter appelle /posts/:id
4. L’API Go vérifie les droits d’accès (a-t-il payé ?)
5. Si autorisé → renvoie le contenu ; sinon, invite à souscrire via /payments
6. En cas de paiement → redirection vers Stripe Checkout → callback backend
7. Le backend valide, enregistre dans PostgreSQL, renvoie confirmation à Flutter
8. Le contenu est débloqué pour l’abonné

## **Schéma architectural**

[Flutter Web/Mobile] → [API REST Go] → [PostgreSQL]

↑ ↓

[Redis] [Stripe]

* Tous les échanges sont en HTTPS
* L’authentification se fait par JWT
* Les rôles (Admin, Abonné, Créateur) sont gérés côté backend via une matrice de droits

## **Conclusion**

Cette architecture client-serveur avec une API REST Go et un frontend Flutter permet de construire une plateforme performante, évolutive et adaptée aux exigences modernes d'une application à fort trafic et riche en fonctionnalités. Elle garantit une séparation claire des responsabilités, facilite le développement modulaire et prépare le terrain pour une éventuelle migration vers une architecture microservices plus poussée si nécessaire.