SPÉCIFICATION TECHNIQUE – MODULE MARKETCONTEXTAPI (AVEC GESTION DES SYMBOLES INTÉGRÉE)

1. Objectif technique

Le module MarketContextAPI vise à fournir une infrastructure complète, autonome et performante permettant de surveiller, analyser et réagir en temps réel à l'évolution des marchés crypto. Cette spécification intègre la **gestion complète des symboles** (source Binance), la **collecte de données temps réel**, l'**analyse comportementale prix + volume**, ainsi que la **génération d'événements de signal d'achat** à destination du moteur d'exécution.

L'objectif principal est de garantir : - Une **connexion fiable** aux flux Binance (WebSocket et REST) - Un **système de scoring en mémoire** précis et rapide - Une **architecture modulaire et découplée**, prête pour le scaling et la haute disponibilité - Une **intégration native** avec les autres modules via EventHub local

Ce module regroupe donc toutes les responsabilités liées à l'observation du marché, à la qualification des symboles, à l'analyse comportementale, et à l'émission de signaux exploitables.

---

2. Architecture logicielle

- **Technologie** : .NET 9
- **Langage** : C# moderne, full async/await
- **Librairie unique tierce** : Binance.Net
- **Communication inter-module** : REST Minimal APIs + EventHub en mémoire
- **Communication inter-processus** (optionnel) : SignalR si besoin de flux vers UI
- **Architecture** : microservice isolé (pas de dépendance forte), conteneurisable
- **Persistences** : aucune base de données, uniquement fichiers JSON fallback en local

Chaque composant fonctionnel est construit autour de principes SOLID, testable unitairement, sans effet de bord, et interagit uniquement via des contrats internes (interfaces, events, endpoints).

---

3. Composants intégrés du module (vision unifiée)

3.1 Gestion complète des symboles (module interne partagé)

Fonctionnalités :

- Téléchargement initial des symboles Binance Spot via `GetExchangeInfoAsync`
- Extraction des métadonnées critiques :
- minQty, stepSize, tickSize, quoteAsset, baseAsset
- statuts (active, maintenance, etc.)
- types de paires (spot seulement)
- Rafraîchissement automatique toutes les 60 minutes (ou configurable)
- Mise en cache mémoire (`ConcurrentDictionary<string, SymbolMetadata>`, par exemple)
- Sérialisation locale du cache pour usage fallback (`symbols.json`)

- Notification interne lors de mises à jour via EventHub
- Exposition via REST : `GET /symbols`

Intégrations internes :

- MarketDataCollector interroge ce service pour savoir quels symboles observer
- AnalyzerService peut consulter les métadonnées symboles (volatilité, minQty...)

Sécurité :

- Endpoints protégés par authentification (JWT / API Key)
- Logs explicites sur les chargements, erreurs, inconsistances Binance

---

3.2 Collecte temps réel des bougies (MarketDataCollector)

Objectif :

Établir des connexions WebSocket persistantes vers Binance Spot, et recevoir les bougies clôturées M1, M5, M15 pour chaque symbole actif.

Fonctionnement :

- Abonnement WebSocket multi-symboles et multi-timeframes (en série ou via multiplexing si dispo)
- Chaque bougie reçue déclenche un événement `OnKlineReceived`
- Les données sont envoyées directement vers les buffers circulaires associés
- Reconnexion automatique en cas de perte de stream ou timeout
- Un watchdog surveille les délais entre messages (timeout configurable)

Limites techniques :

- Max symboles simultanés : 20 (configurable)
- Sélection des timeframes active : configurable dans appsettings

Données utilisées :

- Open, High, Low, Close, Volume, OpenTime
- Symbol, Interval (M1, M5, M15)

3.3 Buffers circulaires de marché

Objectif :

Conserver en mémoire les N dernières bougies par symbole et par timeframe, sans surcharge mémoire.

Structure :

- Un buffer circulaire par (symbol, timeframe)
- Capacité : 20 bougies par défaut
- Écrasement automatique des plus anciennes

Rôle :

- Fournir à l'analyse comportementale un historique glissant à jour
- Permettre l'accès rapide sans I/O ni recalcul

3.4 Analyse comportementale prix + volume (AnalyzerService)

Objectif :

Évaluer le comportement récent d'un actif (via ses bougies), détecter les patterns prédéfinis, et calculer un score de contexte entre 0 et 100.

Patterns reconnus :

- **Accumulation** : prix stables + volume croissant
- **Breakout haussier** : pic de prix + pic de volume
- **Distribution** : prix haut + volume stagnant ou baissant
- **Rejet** : mèche longue + volume élevé

Algorithmes internes :

- Analyse sur les 10–20 dernières bougies
- Calcul de :
- variance du prix
- pente du volume
- position du dernier close dans le range local
- Attribution d'un score (pondéré, normalisé)
- Sortie : objet `ContextScore { Symbol, Timeframe, Pattern, Score, Timestamp }`

Optimisations :

- Traitement asynchrone
- Un worker par timeframe ou batch

3.5 Génération de signaux (SignalEvaluator)

Objectif :

Déclencher un signal de trading lorsqu'un contexte favorable est détecté (score > seuil).

Conditions :

- Score > 75 (configurable)
- Aucun trade actif sur le symbole (appel à TradeEngine ou lecture d'état partagé)

Événement généré :

- `TradeSignal` { Symbol, Pattern, Confidence, GeneratedAt }
- Transmis via EventHub aux modules abonnés (ex : TradeExecutor)

Logging :

- Signal déclenché / bloqué / ignoré est systématiquement loggé

3.6 EventHub local

Architecture :

- Délégation en mémoire (delegate, events, IEventHandler<T>)
- Aucun broker externe requis
- Faible latence, haute testabilité

Événements supportés :

- `KlineReceived`
- `ContextScoreCalculated`
- `SignalGenerated`
- `SymbolListUpdated`

Usage :

- Diffusion entre composants du module
- Subscription possible depuis l'extérieur si exposé

---

4. API REST exposée

4.1 `GET /symbols`

- Description : liste complète des symboles suivis (nom, base, quote, minQty...)
- Cache : réponse générée depuis cache mémoire
- Sécurité : JWT requis si exposé publiquement

4.2 `GET /symbols/contexts`

- Description : score actuel, pattern détecté et timestamp par symbole
- Usage : observabilité, UI, debug, stratégie IA future

---

5. Sécurité et robustesse

5.1 Accès API sécurisé

- Middleware d'authentification obligatoire sur toutes les routes REST
- Authentification supportée :
- JWT signé localement
- Clé API (paramétrable)

5.2 Résilience

- Système de reconnexion WebSocket automatique

• Surveillance de l’activité de chaque stream
• Sauvegarde régulière des symboles (fallback JSON)

5.3 Logs techniques

• Log minimal en stdout ou fichier JSON
• Inclut :
• erreurs WebSocket
• signaux générés / bloqués
• changements de symboles
• seuils franchis

---

6. Performances attendues

• **Latence** : < 1s pour traitement complet post-réception bougie
• **Scalabilité** : 10–20 symboles simultanés sur instance unique
• **Usage mémoire** : stable (< 100 Mo pour 20 symboles / 3 TF / 20 bougies)
• **Traitement async** : full awaitable, sans blocage

---

7. Livrables V1

• Module .NET exécutant les responsabilités décrites ci-dessus
• Documentation technique des endpoints REST
• Fichier `symbols.json` auto-maintenu en fallback
• Logique de collecte, analyse et signalisation en production-ready
• Tests unitaires sur chaque sous-composant
• Possibilité de simulation locale via events mockés

Ce module constitue une brique centrale de la plateforme de trading. Il peut être étendu dans les prochaines versions pour : - Ajouter la gestion IA des patterns (ML.NET) - Supporter le backtesting direct - Fournir des indicateurs secondaires dérivés (ex : momentum, squeeze) - Injecter les données dans un canal SignalR vers UI