Spécification Technique Détaillée - Plateforme de Trading Crypto Mid-Frequency AVD (Architecture .NET 9 Microdistribuée en Mémoire)

1. Objectif Technique

Fournir une architecture performante, modulaire et 100% .NET 9 pour un moteur de trading mid-frequency orienté AVD (achat/vente directionnel), traitant les données en mémoire, avec une persistance minimale pour les comptes clients, les paramètres des bots et l'historique d'opérations.

2. Architecture Générale

- **Langage / Plateforme** : .NET 9
- **Librairie tierce unique** : Binance.Net
- **Architecture** : Microservices déployables séparément
- **Communication interservices** : API REST native (Minimal API), SignalR optionnel
- **Event-driven en mémoire** : EventHub local par domaine regroupé
- **Modularité** : Regroupement logique avec découplage interne

3. Modules Regroupés

3.1 MarketContextAPI

Contient :

- `MarketDataCollector`
- `ContextScorer`
- `SignalEngine`

Rôle : Collecter les données WebSocket Binance, stocker en mémoire les bougies, analyser les patterns comportementaux, calculer les scores, générer des signaux d'achat.

Endpoints REST :

- `GET /symbols/contexts` : retourne liste des symboles analysés avec leur score, dernière action, état du signal

Event Hub local :

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

Composants internes :

- CircularBuffer pour stockage glissant
- EventDispatcher
- AnalyzerService : détection accumulation, breakout, etc.
- SignalEvaluator : si seuil franchi, génère signal

3.2 TradeEngineAPI

Contient :

- `TradeExecutor`
- `RiskManager`

Rôle : Exécuter les trades via Binance.Net, appliquer les règles de gestion du capital, SL dynamique, blocage auto.

Endpoints REST :

- `POST /trades/execute` : créer une position
- `GET /trades/active` : consulter les positions ouvertes

Event Hub local :

- `TradePlaced`
- `SLAdjusted`
- `TradeClosed`
- `DrawdownLimitReached`

Composants internes :

- TradeService : gère appel Binance.Net (limit / market)
- SLMonitorService : ajuste le SL selon les pics
- RiskLimiter : pause journalière, levier max, blocage

3.3 ClientAPI

Contient :

- `ClientService`

Rôle : Authentification, gestion des comptes clients, des clés API, frais prépayés, historique opérations

Endpoints REST :

- `GET /client/me` (auth)
- `POST /client/deposit` (simulateur paiement)
- `GET /client/funds`
- `GET /client/parameters`
- `GET /client/internal/balance` (interne pour TradeEngine)

Persistance : EF Core locale (SQLite, SQL Server embedded ou file-based .NET storage)

3.4 BacktestRunner

Type : ConsoleApp .NET **Rôle** : Charger un historique local (CSV/JSON), simuler l'exécution du moteur de décision pour générer les stats

3.5 Fronts Blazor

- **ClientUI** : tableau de bord client temps réel, suivi positions, historique, paiements, config bots
- **AdminUI** : gestion clients, stats bots, journal, paramètres globaux

4. Mécanismes Transverses

4.1 EventHub local

Chaque module regroupe un `EventDispatcher` maison qui distribue en mémoire les événements sous forme de delegate ou interface (`IEventHandler<T>`). Cela remplace tout broker externe.

Exemple dans MarketContextAPI :

```
public class EventHub {
  public Action<KlineEvent>? OnKlineReceived;
  public Action<ContextScore>? OnContextUpdated;
  public Action<TradeSignal>? OnSignalGenerated;
}
```

4.2 Communication interservice

- `HttpClient` étiqueté via `IHttpClientFactory`
- Appels REST explicites
- SignalR entre UI et MarketContextAPI si besoin de flux temps réel

4.3 Gestion du temps et des tâches

- `BackgroundService` pour polling, timers, check SL
- `Timer` ou `PeriodicTimer` pour fréquences précises (5min, 15min, etc.)

4.4 Authentification et sécurité

- JWT Token signé en local
- Chiffrement clés API via `IDataProtector`
- 2FA optionnel

5. Modèles de Données

Exemple : ContextScore

```
public record ContextScore {
  public string Symbol { get; init; }
  public decimal Score { get; init; }
  public string Pattern { get; init; }
  public DateTime Timestamp { get; init; }
}
```

Exemple : Signal

```
public record TradeSignal {
  public string Symbol { get; init; }
  public decimal Confidence { get; init; }
  public DateTime GeneratedAt { get; init; }
  public string Reason { get; init; }
}
```

6. Déploiement

- Docker éventuellement si nécessaire, sinon exécutables
- Pas d'orchestrateur obligatoire
- Fichiers de conf JSON par module

7. Tests et Logs

- xUnit pour les services
- Mocks de BinanceNet via interfaces
- `ILogger` + fichier local en JSON
- Event Replay en dev depuis un dossier d'events mockés

8. Performances

- 100% async/await
- En mémoire seulement pour 90% du runtime
- Chaque module peut tourner sur un thread CPU dédié
- Channels / ConcurrentQueue pour backpressure

9. Roadmap Technique

- V1 : MarketContextAPI, ClientAPI, TradeEngineAPI (+ UI minimale)
- V2 : Backtest + dashboard complet
- V3 : IA légère (ML.NET) optionnelle pour scoring

10. Conclusion

Cette architecture en .NET pur est à la fois performante, déployable module par module, et extensible. Elle combine in-memory processing, événementiel local, REST, et possibilité de SignalR pour un écosystème de trading temps réel robuste, sans aucun outil tiers imposé.