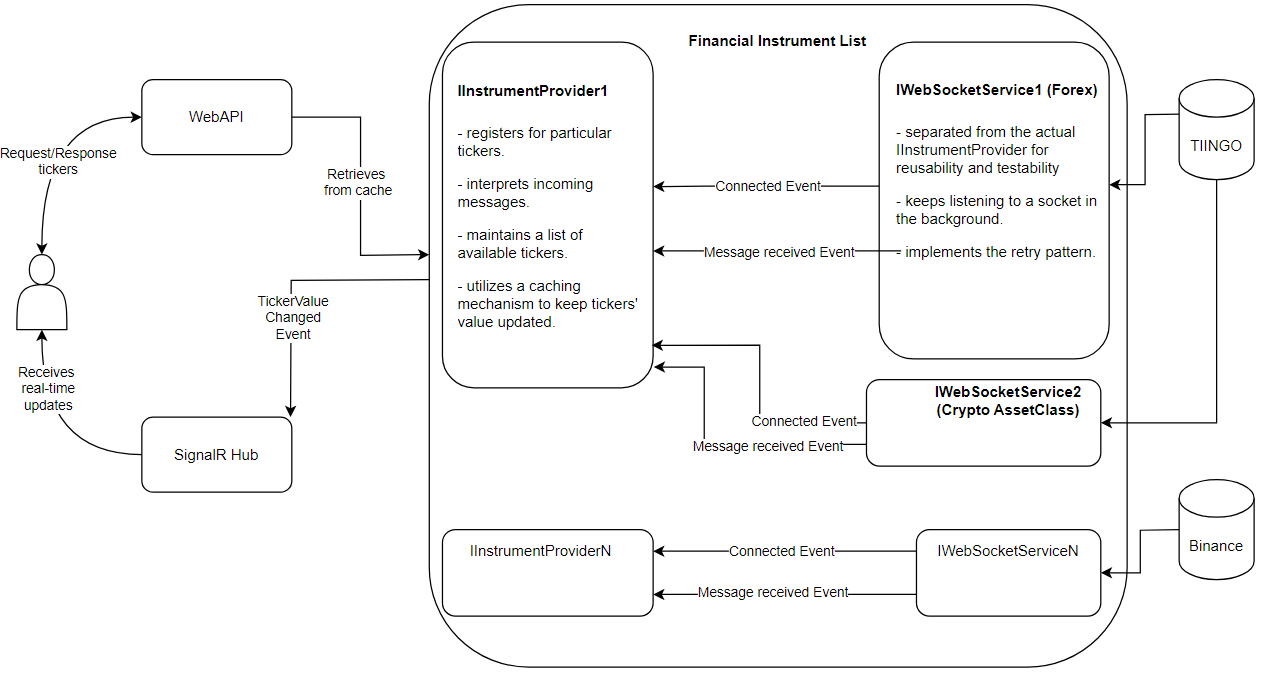
**Financial Instrument Price Service**

**Project Overview**

This project implements a service providing REST API and WebSocket endpoints for live financial instrument prices. The service is designed to efficiently handle over 1,000 subscribers while sourcing data from a public data provider (Tiingo).



**Project Structure**

The solution consists of a WebApi application with the following layout:

* Controllers
* Hubs
* Models
* Infrastructure
* Services

This simplified structure maintains readability and agility while allowing for separation of concerns. The Infrastructure folder contains interfaces that would typically be in the Application layer (port) and their implementations in the Infrastructure layer (adapter).

![Project Structure Diagram]

**Key Components**

1. REST API:
   * Endpoint to get a list of available financial instruments (e.g., EURUSD, USDJPY, BTCUSD)
   * Endpoint to get the current price of a specific financial instrument
2. WebSocket Service:
   * Allows subscription to live price updates for specific financial instrument(s)
   * Broadcasts price updates to all subscribed clients
3. Data Source:
   * Utilizes Tiingo API for fetching live price data and instrument details
4. Performance Optimization:
   * Designed to efficiently manage 1,000+ WebSocket subscribers with a single connection to the data provider
5. Logging and Error Reporting:
   * Implements comprehensive event and error logging

**Key Design Decisions**

1. **Decoupled Instrument Providers**: The system is designed to allow easy substitution of instrument providers through Dependency Injection. This approach enhances testability and allows for potential implementation of a strategy provider pattern in the future.
2. **WebSocket Management**: Utilizes a BackgroundService (IHostedService) for running WebSocket connections. This approach provides better lifecycle management, including graceful shutdown functionality.
3. **Resilience**: Implements retry pattern using the Polly library to make WebSocket connections resilient.
4. **Caching**: Utilizes DistributedMemoryCache to keep REST API endpoints responsive. This can be easily swapped with Redis or other IDistributedCache mechanisms in production.
5. **Logging**: Implements trace-level logging for detailed diagnostics, with the recommendation to use information-level and above in production for performance reasons.
6. **Polymorphic Serialization**: Utilizes Newtonsoft.Json for polymorphic serialization of quote objects, allowing for proper serialization and deserialization of derived classes.

**Implementation Details**

**WebSocket Service**

The WebSocketService class, implementing IWebSocketService, handles the core WebSocket functionality:

* Manages connection, reconnection, and message handling
* Implements retry logic for resilience
* Provides events for message receipt and connection status

**Instrument Provider**

The TiingoInstrumentProvider class implements IInstrumentProvider:

* Manages available instruments
* Handles WebSocket message parsing using specialized parsing factories for different asset types (e.g., ForexQuoteParsingFactory, CryptoQuoteParsingFactory)
* Updates cache with serialized quote objects
* Provides methods for fetching current prices and subscribing to updates

**Quote Parsing**

The system uses a factory pattern for parsing quotes:

* IQuoteParsingFactory interface defines methods for parsing from REST API and WebSocket
* Specialized factories (e.g., ForexQuoteParsingFactory, CryptoQuoteParsingFactory) implement parsing logic for different asset types
* This approach allows for easy extension to new asset types and maintains separation of concerns

**Security**

API keys are secured using the user secret mechanism. To set up, add the following to your secrets.json:

{

"AmegaSettings:WebSocketServiceSettingsList:0:APIKey": "xxx"

}

**Testing Approach**

The project includes unit tests and integration tests with an Application-Layer-Centric focus. This approach emphasizes testing the business logic, which forms the core value of the project.

1. Unit Tests: These focus on testing individual components in isolation, such as the TiingoInstrumentProvider, WebSocketService, and various parsing strategies.
2. Integration Tests: These test the interaction between multiple components, particularly how the TiingoInstrumentProvider interacts with the PriceUpdateService and TickerController.

**Areas for Future Improvement**

1. Implement resilience and retry patterns in the ticker's registration process
2. Optimize try/catch/finally blocks in WebSocketService.ExecuteAsync() for better performance
3. Enhance data modeling for more structural raw response materialization
4. Implement a state management mechanism for IInstrumentProviders
5. Increase test coverage to examine false positives and negatives

**Running the Project**

1. Clone the repository
2. Set up user secrets as described in the Security section
3. Ensure you have the latest .NET SDK installed
4. Navigate to the project directory and run dotnet run

**Conclusion**

This implementation provides a scalable and resilient solution for providing live financial instrument prices. It's designed with extensibility in mind, allowing for easy adaptation to changing requirements or data sources. The focus on core functionality and performance optimization makes it suitable for handling a large number of concurrent users while maintaining responsiveness.