ARCHITECTURAL DESIGN

COMPONENT EXPLANATION

1. USER INTERFACE (UI)

Layer: Presentation Layer

Responsibility:
Handles user interaction and initiates the "compare two tickers" action. It collects input and renders output (currently console output; later this would be graphical output in a web interface).

Requests Data From:
IComparisonService

Interface Role:
Uses (requires) the IComparisonService port.

Why Abstraction is Used:
The UI depends only on the interface, not on a specific implementation of the service. This allows the presentation layer to change (e.g., console → web controller) without modifying the application logic.

How it Supports the Web Application:
In a full web application, this component would become the web controller and would handle HTTP requests and rendering charts while delegating business logic to the service layer.

---

2. COMPARISON SERVICE

Layer: Application Layer

Responsibility:
Implements the core use case: comparing two tickers. It coordinates retrieval of price data and returns the result in a structured format.

Requests Data From:
IPriceRepository

Interface Role:
Provides IComparisonService
Uses IPriceRepository

Why Abstraction is Used:
The service does not know how or where data is stored or retrieved. It only knows that a repository can supply price data. This ensures the application logic is independent of infrastructure.

How it Supports the Web Application:
Acts as the central business logic layer. Web requests are directed here, ensuring clean separation between user interaction and business processing.

---

3. PRICE REPOSITORY

Layer: Data Access / Infrastructure Adapter

Responsibility:
Implements the data retrieval policy:

- Attempt to retrieve cached data first
- If not found, fetches it from external provider
- Caches the fetched data

Requests Data From:
ILocalPriceStore
IMarketDataProvider

Interface Role:
Provides IPriceRepository
Uses ILocalPriceStore and IMarketDataProvider

Why Abstraction is Used:
The repository coordinates between storage and external data without knowing implementation details. Storage technology and external APIs can be replaced independently.

How it Supports the Web Application:
Improves efficiency and scalability by preventing repeated external calls and preparing the system for offline/persistent storage support.

---

4. LOCAL PRICE STORE

Layer: Infrastructure (Storage Adapter)

Responsibility:
Stores and retrieves price series data using a key-value structure (currently in-memory cache).

Requests Data From:
None

Interface Role:
Provides ILocalPriceStore

Why Abstraction is Used:
The repository should not depend on a specific storage mechanism (HashMap, JSON, SQLite). The abstraction allows future replacement with persistent storage.

How it Supports the Web Application:
Enables caching and later persistent storage, supporting performance optimisation and offline functionality.

---

5. MARKET DATA PROVIDER

Layer: Infrastructure (External Data Adapter)

Responsibility:
Retrieves market data from an external source through an API abstraction.

Requests Data From:
IExternalAPI

Interface Role:
Provides IMarketDataProvider
Uses IExternalAPI

Why Abstraction is Used:
Separates application logic from direct interaction with external systems. The specific external API can change without affecting repository or service layers.

How it Supports the Web Application:
Handles integration with external data sources, allowing the system to retrieve real-time market data when connected to the internet.

---

6. EXTERNAL API

Layer: Infrastructure (External System Adapter)

Responsibility:
Simulates an external stock data provider (currently returns mock data). In future sprints, this would connect to a real external service.

Requests Data From:
In Sprint 1: none
In later versions: remote external service

Interface Role:
Provides IExternalAPI

Why Abstraction is Used:
Allows substitution of the dud implementation with a real API integration without modifying higher-level components.

How it Supports the Web Application:
Provides the external data integration point required for retrieving real stock prices.

ALIGNMENT WITH SOLID AND SIMPLE ARCHITECTURE PRINCIPLES

Single Responsibility Principle

Each component has one clearly defined purpose (UI displays, service coordinates, repository manages retrieval policy, store caches, provider integrates externally).

Open/Closed Principle

New storage systems or external APIs can be introduced by implementing existing interfaces without modifying higher-level logic.

Liskov Substitution Principle

Any implementation of ILocalPriceStore, IMarketDataProvider, or IPriceRepository can replace the current one without breaking the system.

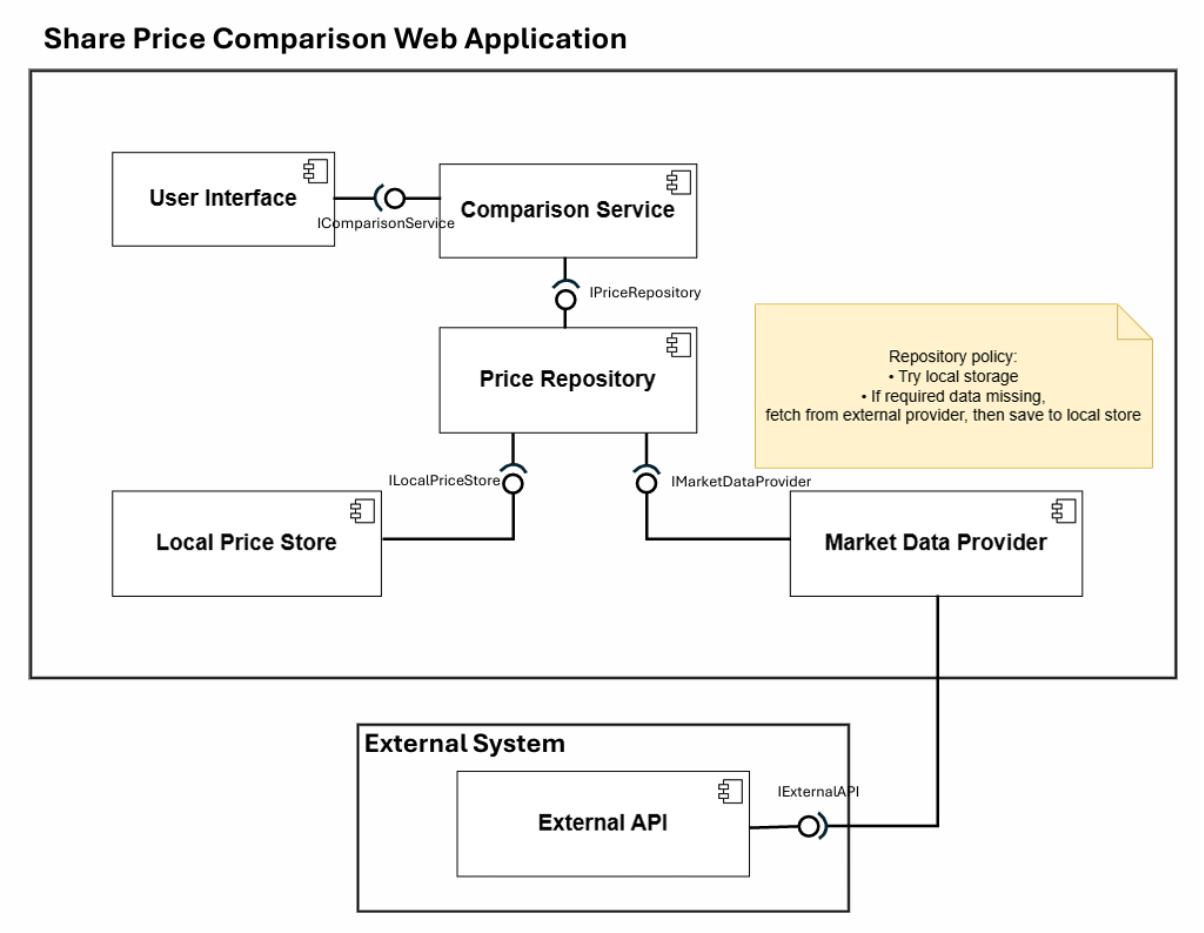
Interface Segregation Principle

Interfaces are small and focused. Components only implement the methods they require.

Dependency Inversion Principle

High-level components (UI and service layer) depend on abstractions, not concrete infrastructure. Concrete implementations are injected during application wiring.

ON THE NEXT PAGE IS A SIMPLE COMPONENT DIAGRAM TO ILLUSTRATE.



COMPONENT COHESION PRINCIPLES

Common Reuse Principle (CRP)

Classes that are reused together should belong to the same component.

In this system, domain objects (Ticker, PricePoint, PriceSeries, ComparisonResult) are grouped together because they are always used together when handling price data. Similarly, the repository-related classes operate together to retrieve and manage data.

This prevents unrelated classes from being forced into the same dependency group and reduces unnecessary coupling between parts of the system.

Common Closure Principle (CCP)

Classes that change for the same reason should be grouped together.

Each component in the architecture changes for a single reason:

- UI changes if presentation requirements change
- Service changes if comparison logic changes
- Repository changes if retrieval policy changes
- Store changes if storage technology changes

- Provider changes if external data integration changes

Because responsibilities are separated, modifying one concern does not require changes across multiple components.

COMPONENT COUPLING PRINCIPLES

Acyclic Dependencies Principle (ADP)

The dependency graph of components should contain no cycles.

The architecture forms a one-directional flow:

UI → Service → Repository → Storage/Provider → External API

No component depends on a higher-level component, preventing circular dependencies and making the system easier to understand and maintain.

Stable Dependencies Principle (SDP)

A component should depend only on components that are more stable than itself.

Higher-level policy components (service and domain) depend only on interfaces, not concrete infrastructure. Infrastructure components depend inward toward abstractions, meaning unstable parts (external API, storage technology) do not affect stable core logic.

Stable Abstractions Principle (SAP)

Stable components should be abstract; unstable components should be concrete.

Core components define interfaces (IPriceRepository, IMarketDataProvider, etc.), while changeable infrastructure components provide implementations (LocalPriceStore, ExternalAPI).

This ensures the parts most likely to change are concrete implementations, while stable components remain abstract and protected.