InsightStox – System Design

1. High-level architecture

This project intentionally adopts three architecture that together define how the system is structured and how different parts of the system interact. The goal is to make the architecture clear, understandable, and aligned with how the system works. Below are three architectures on which the System is based on.

A. Monolithic Architecture

A Monolithic Architecture means the entire backend (API routes, business logic, database communication, authentication and services) is built and deployed as a single unified application.

Why this architecture:

• Our system has closely related features (auth, transactions, market data, etc), which we are keeping at one place.

• Thus it is simple to build and deploy, especially during early development.

• It keeps development fast and easy to maintain.

How it is structured in our system:

• All backend logic lives inside one Node.js server.

• All modules (auth, transactions, portfolios, market data) are placed inside one repository.

• Deployment involves running only one backend service.

Advantages:

• Simpler development and debugging.

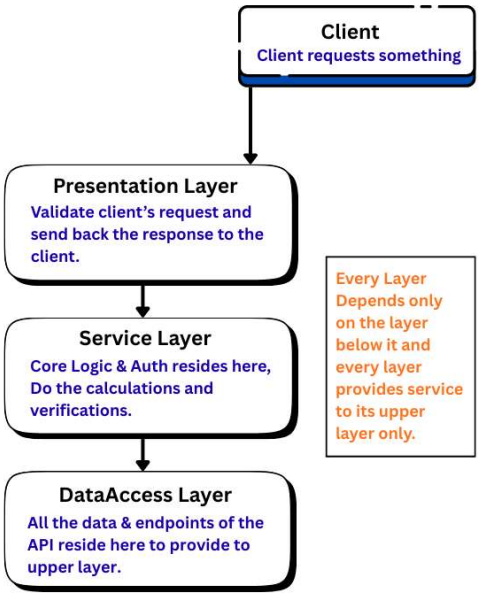
• Easier deployment for a early-stage project.

• No inter-service communication overhead.

• Lower cost and lower architectural complexity.

B. Layered Architecture (3-Tier Design)

Layered Architecture helps keep the system organized by dividing the backend into **3 logical layers**, each with specific responsibilities. As shown in the diagram here, there are mainly three layers where each one provides the service to its upper layer and is depending on layer below it.



→ Why this architecture

- Easier to understand and maintain the system.
- Each part has a clear purpose.
- Changes in one layer do not break the others.
- Encourages clean separation of concerns.

1. Presentation Layer

   - Handles HTTP requests.
   - Sends responses back to the client.
   - Defines API routes for authentication, portfolios, transactions, analytics, etc.

2. Service Layer

   - Contains the actual functional logic of the system.
   - Handles portfolio calculations, price processing, validation, rules, etc.
   - Keeps all core logic independent of HTTP or database concerns.

3. Data Access Layer

   - Talks to the PostgreSQL, MongoDB database, YahooFinance.
   - Uses SQL queries via secure, parameterized statements.
   - Stores and retrieves persistent data like users, transactions, holdings and Real Time Market Data.

C. Client – Server Architecture:

Client (Frontend)

- Built using React.
- Displays UI for portfolios, transactions, charts, analytics.
- Sends requests to the backend and receives JSON responses.

Server (Backend)

- Built with Node.js.
- Handles all processing, calculations, and database operations.
- Provides REST APIs and endpoints.
- Fetches live market data from external APIs.

Why this Architecture

- Allows frontend and backend to evolve independently.
- Supports modern, interactive UI experiences.
- Ensures secure communication and centralized data processing.
- Multiple client types (web, mobile, desktop) can use the same backend APIs.
- Backend updates don't affect the client interface and vice-versa.
- More secure because critical logic remains server-side, behind controlled endpoints.

-- So, in this way, these three architectures allow our system to be simple, understandable and usable properly.

2. Components identification and Allocation

This section identifies all major system components in the system and explains **what each component does** and **where it belongs** within our chosen architectures (Monolithic, Layered, Client–Server).

Major Component of the system:

1. Frontend Application (Client Layer)

- Technology: React
- Responsibilities:
    - User Interface for dashboards, charts, analytics, holdings, and transactions
    - Communicates with backend via REST APIs
    - Handles client-side routing, state management, and rendering

2. Backend API Server (Server Layer)

- Technology: Node.js + Express
- Responsibilities:
    - Exposes RESTful APIs for login, portfolios, transactions, analytics
    - Implements pricing endpoints and subscription handlers
    - Performs authentication and token management
    - Applies validation and request handling before passing data to services

3. Business Logic Services (Service Layer)

- Pure logic components that contain rules and computations:
    - Portfolio value calculation
    - Holdings generation from transactions
    - Realized/Unrealized P&L computation
    - Price formatting and aggregation
    - Validation logic for transaction types

These services do not talk directly to the database or HTTP—they stay independent.

4. Database Layer (Data Storage Layer)

- Technologies: PostgreSQL / MongoDB (where applicable)
- Responsibilities:
    - Store persistent system data such as:
        - Users
        - Portfolios
        - Transactions
        - Holdings
        - Market Price Cache
    - Use SQL queries via the sql tagged template
    - Ensure data integrity, indexing, and relations

5. Market Data Provider Integration

- External API: YahooFinance API
- Responsibilities:
    - Fetch live stock prices
    - Fetch historical candlestick data
    - Provide symbol metadata (sector, name, exchange)
    - Supply corporate actions (splits/dividends)

6. Background Worker / Scheduler

- Technology: Node worker process / cron jobs
- Responsibilities:
    - Periodically fetch and refresh live prices
    - Cache the latest prices in Redis
    - Generate portfolio snapshots
    - Recalculate holdings after new transactions
    - Avoid heavy tasks running inside the main API request cycle

7. Redis Cache Layer

- Responsibilities:
    - Store frequently accessed data such as live prices
    - Reduce load on external APIs
    - Act as a job queue for workers

8. Authentication

- JWT-based authentication
- Password hashing using bcrypt
- Role-based access controls
- Protects all sensitive endpoints

Component Allocation in the Architecture

Below is how each component maps to the system's overall architecture.

A. In the Monolithic Architecture

All backend modules—API routes, services, DB access, job workers exist in **one Node.js codebase** and therefore it simplifies:

- Deployment
- Development workflow
- Inter-module communication

B. In the Layered Architecture

Components are grouped into layers like below:

Presentation – Routes, Controllers, Express API, WebSocket handlers
Service – Portfolio services, transaction services, price and info services
Data Access – DB connection, SQL queries, models, price cache storage

C. In the Client-Server Architecture

- **Client:** React Frontend Application
- **Server:** Node.js backend (API server + worker)
- **Communication:** HTTPS + WebSockets

3. Database Storage Strategy

This section describes **how InsightStox stores, manages, optimizes, and protects data** across databases and caches. Instead of focusing on size, this section explains *how* data flows through the system, *where* it is stored, and *what strategies* are used to ensure reliability, scalability, and performance.

The goal is to ensure that our system stores data **efficiently, safely, and in a scalable manner** as the platform grows.

Types of Data Stored in the System

1. User Data
   - Account details: email, name, password hash.
   - Preferences and watchlists.
   - Authentication tokens.
   - **Storage Strategy:** Relational table (PostgreSQL), indexed for fast lookup.

2. Portfolio Data
   - Portfolio metadata such as name and currency.
   - **Storage Strategy:** Stored relationally; indexed by user.

3. Transaction Data
   - Buy/sell trades with quantity, price, fees, timestamps.
   - Storage Strategy: Stored in PostgreSQL with strong consistency and relational links.
   - Optimization: Indexed by portfolio and symbol.

4. Holdings Data
   - Computed values: quantity, avg_price, realized P&L.
   - Storage Strategy: Materialized table for fast retrieval.
   - Frequently recalculated by a worker.

5. Market Data (Real-Time)
   - Latest stock prices.
   - High-frequency refresh.
   - Storage Strategy: Redis cache → volatile but extremely fast.

Storage Technologies used

1. PostgreSQL (Primary Database)

Used for all mission-critical, structured, relational data:

- Users
- Portfolios
- Transactions
- Holdings

Why PostgreSQL?

- Strong consistency
- ACID transactions
- Excellent indexing support

2. Cache

Used for:

- Live stock prices
- Frequently accessed computed data

Why cache?

- In-memory → extremely low latency
- Great for temporary/frequently refreshed data
- Removes load from PostgreSQL

3. MongoDB (fast retrieval)

Used for:

- User Portfolio Valuations
- User Suggestions and Queries.

Why MongoDB?

- No need for complex schema migration.
- Flexible structure.

Core Database Strategies

A. Normalization for critical data

User, portfolio, and transaction tables follow **3rd normal form**: - No duplication. - Clear relationships. - Referential integrity via foreign keys. This ensures clean and consistent data.

B. Denormalization where needed

Some tables are optimized for speed: - Holdings table stores pre-computed values. - Portfolio summaries may be cached. This improves read performance for dashboards.

C. Caching Strategies

To avoid overloaded APIs and DB: - Live quotes cached with TTL - Portfolio summary cached for quick dashboard access - Prevents repeated expensive calculations.

D. Consistency Strategy

- PostgreSQL ensures ACID for critical data.
- Redis used only for non-critical or ephemeral data.
- Strong consistency for user transactions & holdings.

E. Inconsistent Data Format

- Daily User Valuation and User Suggestions and User

Data Lifecycle Strategy

1. Live Data
    - Quotes updated every few seconds.
    - Stored temporarily in Redis.
    - NOT stored long-term.

2. Frequently Used Data

   - Cached for fast UI rendering.
   - Invalidated and refreshed as needed.

3. Daily User Performance and User Queries

   - Stored in MongoDB.
   - It is consistent in format and changes very frequently.
   - Stored Long Term.

4. Interface Communication Definition

This section explains **how different components communicate** within InsightStox — between client and server, between services inside the backend, and with external systems like market data providers.

Interface communication defines:

- How modules interact
- What protocols they use
- What types of data are exchanged
- The format and structure of requests/responses

Client-Server Communication

Protocol: HTTPS (REST APIs)
Data Format: JSON

Client → Server examples:

- Login request
- Fetch user portfolios
- Add a transaction
- Fetch portfolio analytics
- Fetch historical chart data

Server → Client examples:

- Authentication tokens
- Portfolio summaries
- Holdings details
- Analytics outputs (daily P&L, allocations)
- Error responses with codes

Real-Time Communication (YahooFinance – API, Frankfurter)

Used for:

- Live stock price updates
- Real-time portfolio value refresh
- Live Exchange Rate

Internal Backend Communication (Layered Architecture)

Controller → Service Layer

- Calls pure functions for logic
- Sends validated inputs

Service Layer → Data Access Layer

- Executes SQL queries
- Fetches/updates database records
- Sends results back to controllers

Service Layer → External Market API Layer

- Uses YahooFinance adapter
- Fetches live quotes
- Fetches historical OHLCV

Worker → Cache → API

- Worker updates Redis with fresh quotes
- API retrieves cached data for fast responses

Third-Party API Communication

Protocol: HTTPS
Provider: YahooFinance API

Used for:

- Live market quotes
- Historical data
- Stock metadata

Provider: Frankfurter

Used for:

- Live Market Rates

Provider: GoogleOauth

Used for:

- Getting User through Access_Token.

Responses are normalized before being stored or processed inside the system.

5. Low Level Design (LLD)

This section explains:

- Detailed internal design
- Algorithms used
- Data structures
- End-to-end flows for important operations

Low-Level Design Components

A. Controllers
   - Handle requests
   - Validate input
   - Pass structured data to services
   - Return responses or errors

B. Service
   - Contain calculation logic
   - Encapsulate business rules

C. DAL (Data Access Layer)
   - SQL query builders
   - Repository-like functions

D. Worker
   - Runs background tasks
   - Updates cache & historical data

Important Algorithms

1. Holdings Calculation Algorithm

Goal: Convert a list of user transactions into final holdings.
Input: Array of transactions sorted by date
Output: quantity, avg_price, realized_pnl
Algorithm: FIFO-based cost calculation

```
for each transaction in transactions:
    if BUY:
        total_cost += quantity * price
        total_quantity += quantity
        avg_price = total_cost / total_quantity

    if SELL:
        realized_pnl += (sell_price - avg_price) * quantity
        total_quantity -= quantity
```

2. Portfolio Value Computation Algorithm

```
portfolio_value = sum( holding.quantity * latest_market_price(symbol) )
```

3. Real-Time Price Update Algorithm

```
 while true:
    symbols = getAllActiveSymbols()
    quotes = fetchFromMarketAPI(symbols)
    for each quote:
        saveToCache(symbol, price)
        notifyWebSocketClients(symbol, price)
    wait(refresh_interval)
```

Data Structure Used

Holdings Structure

```
{
  "symbol": "AAPL",
  "quantity": 50,
  "avg_price": 172.5,
  "realized_pnl": 450.0
}
```

Transaction Structure

```
{
  "id": "uuid",
  "symbol": "AAPL",
  "type": "BUY",
  "quantity": 20,
  "price": 170.0,
  "timestamp": "2024-02-01"
}
```

Price Cache Structure

```
{
  "symbol": "AAPL",
  "price": 172.45,
  "timestamp": 1708930500
}
```

Sequence Flows (LLD Flows)

1. Add Transaction Flow
   `Client → API → Controller → Service → DB → Service → API → Client`

2. Live Price Update Flow
   `Worker → Redis → API → WebSocket → Client`

3. Portfolio Page Load
   `Client → API → Cache (for latest prices)`
   `Client → API → DB (for holdings)`
   `API → Combine → Send to Client`

Summary of LLD

Low-Level Design ensures:

- Predictable behavior
- Clear code separation
- Efficient use of caching and background jobs
- Reliable financial calculations