Trading Stock Portfolio Management System

Technical Report

Software Engineering Project

20th March 2025

Contents

1 Executive Summary **3**

2 System Architecture **3**
- 2.1 Architecture Overview . . . 3
- 2.2 Technology Stack . . . 3
- 2.3 Package Structure . . . 3

3 Core Features **4**
- 3.1 Portfolio Management . . . 4
- 3.2 Live Market Data Integration . . . 4
- 3.3 Advanced Calculator . . . 4
- 3.4 Real-Time Analysis (Premium Feature) . . . 5

4 Data Flow **5**
- 4.1 Portfolio Persistence Flow . . . 5
- 4.2 Real-Time Price Refresh Flow . . . 5
- 4.3 Real-Time Analysis Flow (Future) . . . 5

5 Concurrency and Multithreading **5**
- 5.1 Threading Strategy . . . 5
- 5.2 Example: Price Refresh (Async Pattern) . . . 6
- 5.3 Example: Real-Time Polling (kdb+ RDB) . . . 6

6 Database Design **6**
- 6.1 PostgreSQL Schema . . . 6
- 6.2 Future kdb+ Schema (Real-Time Analytics) . . . 7

7 API Integration **7**
- 7.1 Yahoo Finance API . . . 7
- 7.2 Alpha Vantage (Alternative) . . . 7

8 JavaFX UI Design 8
8.1 Scene Structure 8
8.2 CSS Theming 8
8.3 Reactive Data Binding 8

9 Concurrency Model 8
9.1 Threading Rules 8
9.2 Async Patterns Used 9

10 Database Design 9
10.1 PostgreSQL Schema 9
10.2 Future kdb+ Schema (Real-Time Analytics) 9

11 API Integration 10
11.1 Yahoo Finance API 10
11.2 Alpha Vantage (Alternative) 10

12 Build and Deployment 10
12.1 Maven Build Lifecycle 10
12.2 Docker Database 10
12.3 kdb+ Startup 11

13 Security and Best Practices 11
13.1 Database Security 11
13.2 API Best Practices 11
13.3 Code Quality 11

14 Testing and Troubleshooting 11
14.1 Common Issues 11
14.2 Debugging Tools 12

15 Future Enhancements 12
15.1 Short-Term (Weeks) 12
15.2 Medium-Term (Months) 12
15.3 Long-Term (Quarters) 12

16 Performance Considerations 12
16.1 Current Performance 12
16.2 Optimization Roadmap 13

17 Deployment Instructions 13
17.1 Prerequisites 13
17.2 Setup Steps 13
17.3 IntelliJ IDEA Configuration 13

18 Documentation 14

19 Conclusion 14

1 Executive Summary

This document presents the technical architecture, implementation, and future roadmap of the McGill Stock Portfolio Management System, a desktop application built with JavaFX for real-time portfolio tracking, analytics, and trade management. The system integrates:

- **JavaFX 22** for cross-platform UI with real-time data binding
- **PostgreSQL 16** (Docker) for persistent portfolio storage
- **Yahoo Finance API** for live market data integration
- **kdb+/q** (planned) for microsecond-latency time-series analytics
- **Maven** for dependency and build lifecycle management

The application follows industry-standard MVC architecture, implements safe multi-threading patterns for non-blocking UI, and provides extensibility for advanced financial analytics.

2 System Architecture

2.1 Architecture Overview

The system adopts a layered MVC (Model-View-Controller) architecture with clear separation of concerns:

1. **Presentation Layer**: JavaFX UI components (TableView, PieChart, Forms)
2. **Controller Layer**: Event handlers, scene management, UI orchestration
3. **Service Layer**: Business logic, validation, computation
4. **Repository Layer**: Data access abstraction (PostgreSQL via JDBC)
5. **Model Layer**: Domain entities with JavaFX properties for reactive UI binding

2.2 Technology Stack

Component	**Technology**
Programming Language	Java 21
UI Framework	JavaFX 22.0.2
Build Tool	Maven 3.x
Database	PostgreSQL 16 (Alpine)
Container Platform	Docker / Docker Compose
Time-Series DB (planned)	kdb+ 4.1
Market Data API	Yahoo Finance v7
JSON Parsing	Google Gson 2.10.1
JDBC Driver	PostgreSQL JDBC 42.7.1
kdb+ IPC Client	com.kx:javakdb 2.1

Table 1: Technology Stack

2.3 Package Structure

```
com.mcgill.application
+-- controller/        # UI event handling, scene orchestration
|   +-- PortfolioController.java
|   +-- DataManagementController.java
+-- service/           # Business logic layer
|   +-- StockService.java
|   +-- StockPriceService.java
|   +-- CalculatorService.java
|   +-- EmployeeService.java
|   +-- KdbClientService.java (kdb+ integration)
+-- repository/        # Data access layer
|   +-- StockRepository.java
|   +-- StockRepositoryPostgreSQL.java
|   +-- EmployeeRepository.java
+-- model/             # Domain entities
|   +-- Stock.java
|   +-- Person.java
|   +-- LiveTick.java
+-- database/          # DB connection management
|   +-- DatabaseConnection.java
+-- Main.java          # Application entry point
```

3 Core Features

3.1 Portfolio Management

- Add, update, delete stock holdings (symbol, shares, purchase price, current price)
- Real-time profit/loss calculation with percentage metrics
- Portfolio visualization via interactive PieChart
- Sell shares with automatic price refresh from live API
- Persistent storage in PostgreSQL with ACID guarantees

3.2 Live Market Data Integration

- Real-time price refresh via Yahoo Finance API (batch endpoint for multiple symbols)
- Automatic timestamp tracking (last_refreshed) in EDT
- Non-blocking background refresh using JavaFX Task
- Rate-limiting and User-Agent configuration to avoid API throttling

3.3 Advanced Calculator

Stock market calculators with dedicated tabs:
- Profit/Loss Calculator (absolute & percentage)
- ROI Calculator (Return on Investment)
- Break-even Price Calculator
- Position Size Calculator
- Average Purchase Price Calculator

Runs in a separate window for multi-view parallel workflows.

3.4 Real-Time Analysis (Premium Feature)

Current Status: UI window with polling architecture in place; backend integration with kdb+ in progress.

Planned Features:

- Live tick-by-tick price feed from kdb+ RDB (tickerplant/RDB/HDB stack)
- Rolling VWAP (Volume-Weighted Average Price) and spread analytics
- Asof joins for historical quote/trade correlation
- Micro-charts (candlestick, heatmaps) for intraday microstructure
- Sub-millisecond latency for tick ingestion and analytics

4 Data Flow

4.1 Portfolio Persistence Flow

1. User interacts with PortfolioController (add/update/sell)
2. Controller validates input and delegates to StockService
3. Service applies business logic (e.g., share constraints, ROI)
4. StockRepositoryPostgreSQL executes JDBC prepared statements
5. PostgreSQL Docker container persists changes
6. Controller updates UI via ObservableList binding (automatic TableView refresh)

4.2 Real-Time Price Refresh Flow

1. User clicks "Refresh Prices" button
2. PortfolioController spawns a background Task
3. Task calls StockPriceService.getCurrentPrice for each stock
4. StockPriceService makes HTTP GET to Yahoo Finance batch endpoint
5. JSON response parsed with Gson; regularMarketPrice extracted
6. Task updates Stock.currentPrice and Stock.lastRefreshed (in memory)
7. StockService.persist(...) saves to PostgreSQL
8. Platform.runLater updates UI labels and TableView

4.3 Real-Time Analysis Flow (Future)

1. kdb+ tickerplant (TP) ingests market feed (UDP multicast or vendor API)
2. TP journals ticks and publishes to RDB (in-memory) and HDB (on-disk)
3. JavaFX KdbClientService subscribes to RDB via IPC
4. RDB pushes tick updates to the client asynchronously
5. Client thread receives updates, parses via kx.c (com.kx.c.Flip)
6. Platform.runLater appends rows to Real-Time Analysis TableView

5 Concurrency and Multithreading

5.1 Threading Strategy

All long-running operations (HTTP, JDBC, sleep) are moved off the JavaFX Application Thread to prevent UI freezes.

Patterns Applied:
- **Task¡T¿**: background work with progress/message binding for UI feedback
- **Platform.runLater**: async enqueue to FX thread for safe UI updates
- **ScheduledExecutorService**: fixed-rate polling (e.g., kdb+ query every 1s)
- **Daemon threads**: background listeners (kdb+ IPC) that exit with the app

5.2 Example: Price Refresh (Async Pattern)

```java
Task<Void> task = new Task<>() {
    @Override protected Void call() {
        for (Stock s : stocks) {
            double price = stockPriceService.getCurrentPrice(s.
    getSymbol());
            s.setCurrentPrice(price);
            s.setLastRefreshed(LocalDateTime.now());
            stockService.persist(s);
        }
        Platform.runLater(() -> showSuccess("Prices updated"));
        return null;
    }
};
new Thread(task, "price-refresh").start();
```

Listing 1: refreshStockPrices in PortfolioController

5.3 Example: Real-Time Polling (kdb+ RDB)

```java
ScheduledExecutorService ses = Executors.
    newSingleThreadScheduledExecutor();
ses.scheduleAtFixedRate(() -> {
    try {
        Object obj = kdbClient.exec("select sym,px from quote");
        if (obj instanceof com.kx.c.Flip f) {
            String[] sym = (String[]) f.y[0];
            double[] px = (double[]) f.y[1];
            Platform.runLater(() -> {
                for (int i = 0; i < sym.length; i++) {
                    LiveTick lt = new LiveTick();
                    lt.setSymbol(sym[i]);
                    lt.setPrice(px[i]);
                    data.add(0, lt);
                }
            });
        }
    } catch (Exception ignored) {}
}, 0, 1, TimeUnit.SECONDS);
```

Listing 2: Polling kdb+ quote table

6 Database Design

6.1 PostgreSQL Schema

Table: portfolio

Column	**Type**	**Constraint**	**Description**
id	SERIAL	PRIMARY KEY	Auto-increment ID
symbol	VARCHAR(10)	NOT NULL	Stock ticker (e.g., AAPL)
company	VARCHAR(255)	NOT NULL	Company name
shares	INTEGER	NOT NULL, ≥ 0	Number of shares owned
purchase_price	NUMERIC(10,2)	NOT NULL	Original buy price
current_price	NUMERIC(10,2)	NOT NULL	Latest market price
last_refreshed	TIMESTAMP	NULL	Last API update time

Table 2: PostgreSQL Portfolio Schema

6.2 Future kdb+ Schema (Real-Time Analytics)

Table: quote (in-memory RDB)

Column	**Type**	**Description**
sym	symbol	Stock ticker
t	timespan	Nanosecond timestamp
px	float	Last trade price
sz	long	Trade size

Table 3: kdb+ Quote Table (Tick Data)

Architecture: Tickerplant (TP) $\rightarrow$ RDB (today in RAM) $\rightarrow$ HDB (historical on disk). JavaFX subscribes to RDB for push updates or polls for batch retrieval.

7 API Integration

7.1 Yahoo Finance API

Endpoint: `https://query1.finance.yahoo.com/v7/finance/quote`

Parameters:

- `symbols`: comma-separated ticker list (e.g., AAPL,MSFT,GOOGL)
- User-Agent header required to avoid 401/403

Response: JSON with `quoteResponse.result[].regularMarketPrice`

Rate Limits: Public endpoint; recommend 1-2 requests/second max; batch symbols to minimize calls.

Implementation: StockPriceService uses Java HttpClient, Gson for parsing, chunked batch requests to avoid rate limits.

7.2 Alpha Vantage (Alternative)

Endpoint: `https://www.alphavantage.co/query?function=GLOBAL_QUOTE`

Rate Limits: Free tier 5 calls/min; premium tiers for higher throughput.

Usage: kdb+ RDB can fetch via `.Q.hg` with rotation through symbols (12s interval per symbol for 5/min compliance).

8 JavaFX UI Design

8.1 Scene Structure

1. **Login Scene**: Basic auth placeholder (username/password)
2. **Main Menu**: Navigation hub (Portfolio, Calculator, Logout)
3. **Portfolio Scene**: Stock table, add/sell forms, PieChart button, refresh button, real-time analysis button
4. **Calculator Window**: Separate stage with tabbed calculators
5. **Real-Time Analysis Window**: Separate stage with live tick TableView (polling kdb+ RDB)

8.2 CSS Theming

Professional color palette inspired by iOS design:

- Primary: `#007AFF` (iOS blue)
- Success: `#34C759`
- Danger: `#FF3B30`
- Neutral grays: `#F2F2F7`, `#8E8E93`

Modular CSS: `theme.css`, `common.css`, `portfolio.css`, `calculator.css`.

8.3 Reactive Data Binding

Stock model uses JavaFX properties:

```java
private final StringProperty symbol = new SimpleStringProperty();
private final DoubleProperty currentPrice = new SimpleDoubleProperty();
private final ObjectProperty<LocalDateTime> lastRefreshed = new
    SimpleObjectProperty<>();

public DoubleProperty profitLossProperty() {
    return new SimpleDoubleProperty(
        (currentPrice.get() - purchasePrice.get()) * shares.get()
    );
}
```

Listing 3: Stock.java properties

TableView columns bind to these properties; UI updates propagate automatically when values change.

9 Concurrency Model

9.1 Threading Rules

1. All UI mutations occur on the JavaFX Application Thread
2. HTTP, JDBC, and sleep operations run on background threads (Task, ExecutorService)
3. Platform.runLater marshals results back to the FX thread
4. ScheduledExecutorService handles periodic tasks (polling, timers)

9.2 Async Patterns Used

Background Task with Progress:

```
Task<Void> refreshTask = new Task<>() {
    @Override protected Void call() {
        updateMessage("Fetching prices...");
        // HTTP/JDBC work here
        updateProgress(n, total);
        return null;
    }
};
new Thread(refreshTask, "bg-refresh").start();
```

Scheduled Polling:

```
ScheduledExecutorService ses = Executors.
   newSingleThreadScheduledExecutor();
ses.scheduleAtFixedRate(() -> {
    var rows = kdbClient.exec("select sym,px from quote");
    Platform.runLater(() -> updateTable(rows));
}, 0, 1, TimeUnit.SECONDS);
```

Daemon Threads: kdb+ IPC listener runs as daemon to exit cleanly with the app.

10 Database Design

10.1 PostgreSQL Schema

Table: portfolio

Column	Type	Constraint	Description
id	SERIAL	PRIMARY KEY	Auto-increment ID
symbol	VARCHAR(10)	NOT NULL	Stock ticker (e.g., AAPL)
company	VARCHAR(255)	NOT NULL	Company name
shares	INTEGER	NOT NULL, ≥ 0	Number of shares owned
purchase_price	NUMERIC(10,2)	NOT NULL	Original buy price
current_price	NUMERIC(10,2)	NOT NULL	Latest market price
last_refreshed	TIMESTAMP	NULL	Last API update time

Table 4: PostgreSQL Portfolio Schema

10.2 Future kdb+ Schema (Real-Time Analytics)

Table: quote (in-memory RDB)

Column	Type	Description
sym	symbol	Stock ticker
px	float	Current quote price

Table 5: kdb+ Quote Table (Current Prices)

Architecture: Tickerplant (TP) → RDB (today in RAM) → HDB (historical on disk). JavaFX subscribes to RDB for push updates or polls for batch retrieval.

11 API Integration

11.1 Yahoo Finance API

Endpoint: `https://query1.finance.yahoo.com/v7/finance/quote`

Parameters:

- `symbols`: comma-separated ticker list (e.g., AAPL,MSFT,GOOGL)
- User-Agent header required to avoid 401/403

Response: JSON with `quoteResponse.result[].regularMarketPrice`

Rate Limits: Public endpoint; recommend 1-2 requests/second max; batch symbols to minimize calls.

Implementation: StockPriceService uses Java HttpClient, Gson for parsing, chunked batch requests to avoid rate limits.

11.2 Alpha Vantage (Alternative)

Endpoint: `https://www.alphavantage.co/query?function=GLOBAL_QUOTE`

Rate Limits: Free tier 5 calls/min; premium tiers for higher throughput.

Usage: kdb+ RDB can fetch via `.Q.hg` with rotation through symbols (12s interval per symbol for 5/min compliance).

12 Build and Deployment

12.1 Maven Build Lifecycle

```
mvn clean compile       # compile Java sources
mvn package             # build JAR
mvn javafx:run          # run JavaFX app via plugin
```

pom.xml highlights:

- Java 21 source/target
- JavaFX dependencies (controls, graphics, base) 22.0.2
- PostgreSQL JDBC 42.7.1
- Gson 2.10.1
- com.kx:javakdb 2.1 (kdb+ IPC)
- javafx-maven-plugin for running without module-path fuss

12.2 Docker Database

Service: PostgreSQL 16 Alpine

Configuration (`docker-compose.yml`):

- Container name: `mcgill-stock-postgres`
- Port: `5433:5432` (host:container)
- Volume: `stock_portfolio_data` for persistence
- Init script: `init.sql` creates schema and test data

- Healthcheck: `pg_isready` at 10s intervals

Commands:

```
docker-compose up -d                # start DB
docker exec -it mcgill-stock-postgres psql -U mcgill_user -d
   stock_portfolio
docker-compose down                 # stop (data persists)
docker-compose down -v              # stop and delete volume
```

12.3 kdb+ Startup

Scripts:

- `JavaFx/scripts/kdb-start.sh`: starts RDB (and feed if enabled) in background
- `JavaFx/scripts/kdb-stop.sh`: gracefully stops processes via PIDs
- Logs: `JavaFx/kdb/logs/*.log`
- PIDs: `JavaFx/kdb/run/*.pid`

Startup:

```
Q_BIN=/Users/priv/q/m64/q JavaFx/scripts/kdb-start.sh
lsof -i :5012   # verify RDB is listening
```

13 Security and Best Practices

13.1 Database Security

- Credentials in docker-compose.yml (dev mode); use secrets manager for prod
- JDBC prepared statements to prevent SQL injection
- Connection pooling (future: HikariCP) for scalability

13.2 API Best Practices

- User-Agent header for Yahoo Finance compliance
- Batch requests to minimize API calls
- Rate-limiting logic (12s per symbol for Alpha Vantage free tier)
- Graceful degradation on network/API failures (catch, log, show user message)

13.3 Code Quality

- Separation of concerns (MVC layers)
- Repository pattern for swappable data sources (in-memory $\leftrightarrow$ PostgreSQL)
- Service layer encapsulates business logic (no DB logic in controllers)
- JavaFX properties for reactive UI (no manual listeners)

14 Testing and Troubleshooting

14.1 Common Issues

1. **UI freezes**: long operation on FX thread $\rightarrow$ move to Task

2. **IllegalStateException (not on FX thread)**: UI touched from background → wrap in Platform.runLater
3. **PostgreSQL connection refused**: Docker not running → docker-compose up -d
4. **Module javafx.controls not found**: IntelliJ VM options misconfigured → remove all VM options; let Maven handle module-path
5. **kdb+ IPC subscription empty**: broadcast syntax error in TP/RDB → use polling interim; revisit push after validating q scripts

14.2 Debugging Tools

- IntelliJ Debugger: set breakpoints in controllers, services
- PostgreSQL psql: `docker exec -it mcgill-stock-postgres psql ...`
- kdb+ q console: connect via `hopen `:5012` and query tables
- Logs: `JavaFx/kdb/logs/*.log` for q process errors
- lsof/netstat: verify ports (5012 for kdb+, 5433 for Postgres)

15 Future Enhancements

15.1 Short-Term (Weeks)

- Finalize kdb+ subscription (push model) for Real-Time Analysis
- Add User-Agent rotation/proxy for Yahoo API resilience
- Implement rolling VWAP and spread analytics in q
- Add micro-charts (line/candlestick) in Real-Time Analysis window

15.2 Medium-Term (Months)

- HDB integration: persist intraday ticks to disk; enable backtesting queries
- Multi-user support: authentication, role-based access control
- Portfolio analytics: Sharpe ratio, beta, drawdown metrics
- Export/import: CSV, Excel for portfolio snapshots
- Alerts: threshold-based notifications (price drops, profit targets)

15.3 Long-Term (Quarters)

- Production kdb+ deployment: bare-metal or tuned VMs for sub-ms latency
- Integration with broker APIs (FIX, OUCH) for order execution
- Advanced risk analytics: Greeks (options), VaR, stress testing
- Web UI (Spring Boot + React) for mobile/cross-platform access
- Machine learning: anomaly detection, predictive signals

16 Performance Considerations

16.1 Current Performance

- UI responsiveness: <16ms per frame (60 FPS target via non-blocking I/O)
- Price refresh latency: 200–500ms per stock (Yahoo API response time)
- PostgreSQL query latency: <10ms for portfolio CRUD (local Docker)

- kdb+ query latency (polling): <5ms for in-memory select (RDB)

16.2 Optimization Roadmap

- Replace HTTP polling with WebSocket for sub-second API updates
- Connection pooling (HikariCP) for PostgreSQL under load
- kdb+ push subscriptions to reduce polling overhead
- Compute analytics (VWAP, spreads) server-side in q; send aggregates to JavaFX
- Cache frequently-accessed data (e.g., company names) to reduce DB round-trips

17 Deployment Instructions

17.1 Prerequisites

- Java 21+ JDK
- Maven 3.9+
- Docker Desktop (for PostgreSQL)
- kdb+ 4.1+ (for Real-Time Analysis; optional for basic portfolio)

17.2 Setup Steps

1. Clone repository and navigate to `JavaFx/`
2. Start PostgreSQL:
   ```
   docker-compose up -d
   ```
3. Build project:
   ```
   mvn clean package
   ```
4. Run app (IntelliJ or Maven):
   ```
   mvn javafx:run
   ```
5. (Optional) Start kdb+ for Real-Time Analysis:
   ```
   Q_BIN=/path/to/q JavaFx/scripts/kdb-start.sh
   ```

17.3 IntelliJ IDEA Configuration

- Project SDK: Java 21
- Main class: `com.mcgill.application.Main`
- VM options: (leave empty; Maven handles JavaFX module-path)
- Use classpath of module: JavaFx
- Before launch: Build Project

18 Documentation

Comprehensive docs in `JavaFx/docs/`:

- `README.md`: Project overview and doc navigation
- `GETTING_STARTED.md`: Setup and installation
- `ARCHITECTURE_GUIDE.md`: MVC layers and data flow
- `DATABASE.md`: PostgreSQL schema and Docker commands
- `LIVE_PRICES.md`: Yahoo Finance API integration
- `UI_GUIDE.md`: Screen-by-screen user guide
- `COMMANDS.md`: Quick reference for Docker, psql, Maven
- `KDB_PREFERRED.md`: Why kdb+ for real-time finance, comparison table
- `THREADING_ASYNC.md`: Concurrency patterns, Task vs ExecutorService
- `TROUBLESHOOTING.md`: Common errors and solutions

19 Conclusion

The McGill Stock Portfolio Management System demonstrates industry-standard patterns for building responsive, data-intensive desktop applications:

- Clean MVC architecture with layered separation
- Non-blocking UI via background threading and Platform.runLater
- Persistent storage with PostgreSQL and Docker orchestration
- Live market data integration with rate-limit-aware HTTP clients
- Extensibility for advanced time-series analytics via kdb+ (in progress)

The system is production-ready for basic portfolio management and serves as a solid foundation for real-time trading analytics, risk management, and multi-user SaaS deployment.

Appendix A: Key Technologies

JavaFX Cross-platform UI toolkit with scene graph, CSS theming, and property bindings.

PostgreSQL ACID-compliant RDBMS for transactional portfolio storage.

kdb+ Columnar time-series database with q language for low-latency tick analytics.

Maven Build automation, dependency management, plugin ecosystem.

Docker Container orchestration for reproducible PostgreSQL deployment.

Yahoo Finance API Public HTTP endpoint for real-time stock quotes.

Appendix B: Glossary

Asof Join Time-aware join matching each fact row with the latest reference row at or before that timestamp.

Daemon Thread Background thread that doesn't prevent JVM shutdown.

HDB Historical Database (kdb+); on-disk partitioned data.

Platform.runLater JavaFX API to enqueue UI updates on the Application Thread.

RDB Real-time Database (kdb+); today's data in memory.

Task JavaFX background job with progress/message properties.

TP Tickerplant (kdb+); ingests feed, journals, republishes to subscribers.

VWAP Volume-Weighted Average Price; time-series metric.