Stockify

Dec 3, 2025

encora

Team Red

Adarsh Arora
Dev Pratap Singh
Kritika Chhabra
Rishabh Kumar
Rudraksh Chourey
Ujjwal Jain

Table of Contents

1. Project Overview

1.1 Introduction

Stockify (also known as the Stock Portfolio Tracker) is an enterprise-grade full-stack web application designed for secure stock portfolio management with role-based access control, JWT authentication, real-time news integration, and AWS SES email notifications. The application provides multi-tiered user experiences with Admin dashboards for user and stock management.

1.2 Project Objectives

- **Security:** JWT token-based security, role-based authorization, BCrypt hashing.
- **AWS SES Email Integration:** Subscription confirmation on registration, welcome-back email on login, password reset notifications.
- **Home Page:** Live financial news feed, market overview, personalized dashboards.
- **User Features:** Multi-step registration with subscription email, portfolio CRUD, performance analytics.
- **Admin Features:** User/stock management, activity monitoring, bulk operations.
- **Real-time Updates:** AWS Lambda triggered every 1 minute via EventBridge performs 10 price update cycles per invocation for smooth portfolio refresh.

1.3 Key Features

Security & Authentication

- JWT Token-Based Security: Stateless authentication with access/refresh token mechanism
- Role-Based Authorization: User and Admin roles with granular permissions
- AWS SES Email Integration:
    - Email verification on registration
    - Welcome back email on login
    - Password reset notifications

Home Page Enhancements

- Live Financial News Feed: Real-time market news and updates
- Stock Market Overview: Market indices and trending stocks
- Personalized Welcome: User-specific dashboard based on role

Admin Features

- User Management Dashboard:
    - View all registered users
    - Modify user profiles and roles
    - Delete/deactivate user accounts
    - Monitor user activity
- Stock Management Console:
    - Add new stocks to the platform
    - Modify stock metadata and symbols
    - Delete obsolete stocks
    - Bulk import/export functionality

User Features

- Multi-step registration with email verification
- Portfolio dashboard with real-time updates
- CRUD operations on stock orders
- Performance analytics and charts
- Secure profile management

1.4 Technology Stack

Frontend:

- React.js with Hooks and Context API
- JWT token management with Axios interceptors
- Protected routes with role-based rendering
- News API integration for live feeds

Backend:

- Spring Boot REST API with Spring Security
- JWT authentication filter chain
- Spring Mail + AWS SES for email services
- Role-based method security annotations
- Scheduled tasks for price updates

Database:

- AWS RDS (MySQL) - User data
- AWS DynamoDB - Stock and Stock Order data
- Hybrid architecture for optimized performance

Cloud Infrastructure:

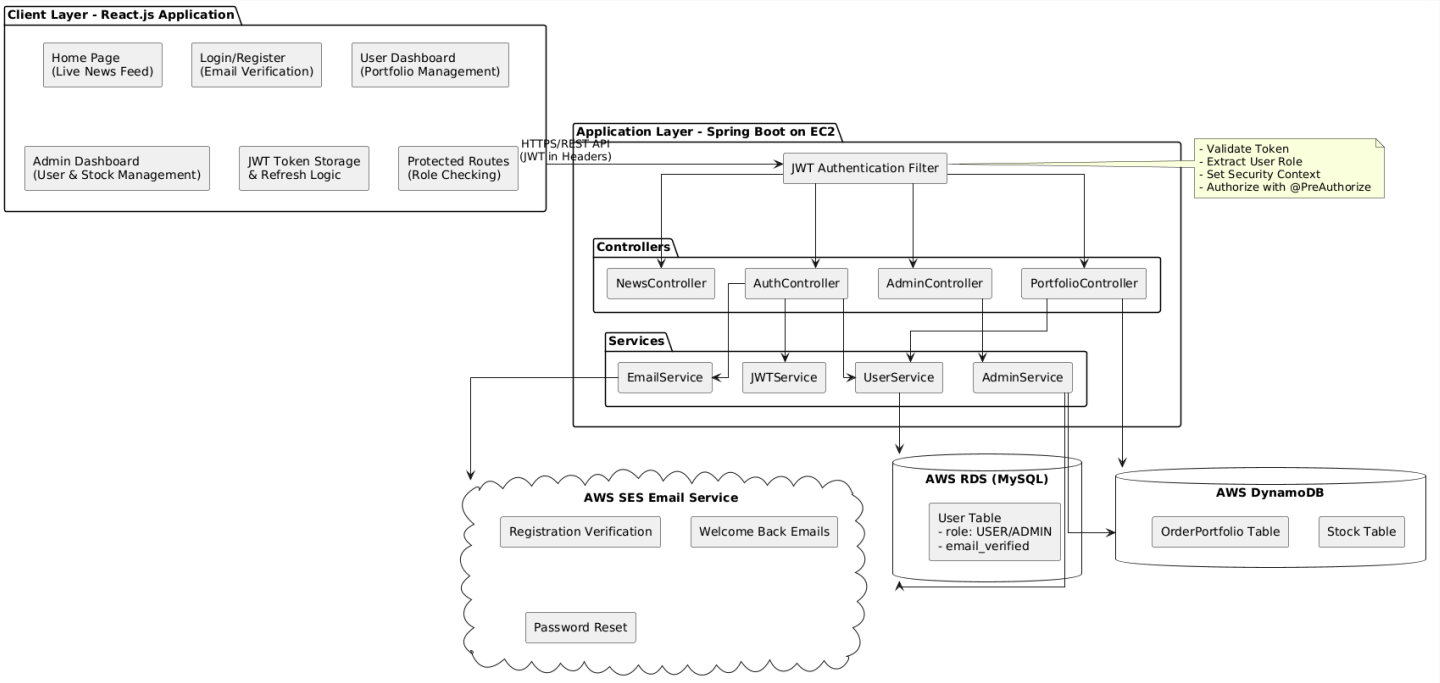
- AWS SES for transactional emails
- AWS RDS for relational data
- AWS DynamoDB for NoSQL operations
- AWS IAM for security policies
- AWS EventBridge for serverless scheduling
- AWS Lambda for price update function

Deployment or Hosting:

- AWS Elastic BeanStalk for Backend hosting
- AWS EC2 for frontend hosting

2. ARCHITECTURE

2.1 High-Level Architecture Diagram



2.2 Architecture Components

Client Layer (Presentation Tier)

- **React.js Frontend Application**: Single-page application handling UI rendering and user interactions
- **Component Library**: Reusable UI components (StockCard, PortfolioSummary, Charts)
- **State Management**: Context API (AuthContext, StockContext, RegisterUserContext)
- **HTTP Client**: Axios for REST API communication
- **Local Storage**: Session persistence and token management
- **Routing**: React Router for navigation and protected routes

Application Layer (Business Logic Tier)

- **Spring Boot REST API**: Centralized backend service exposing RESTful endpoints
- **Controller Layer**: Request handling and response formatting (portfolioController)
- **Service Layer**: Business logic implementation (portfolioService)
- **Security Module**: BCrypt password hashing and authentication validation
- **Data Access Layer**: Repository pattern for MySQL, AWS SDK for DynamoDB
- **DTO Objects**: Request/Response models (PortfolioUpdateRequest, loginUserTemplate)

Data Layer (Persistence Tier)

- **MySQL RDS Instance**: Relational database for user authentication data
  - User table: Stores credentials, personal information, investment profiles
  - ACID-compliant transactions for user management
- **AWS DynamoDB**: NoSQL database for stock market data
  - StockTable: Stock metadata, current prices, historical data
  - OrderPortfolioTable: User portfolio holdings with composite keys
  - Atomic operations for concurrent order processing

Background Services

- **Price Update Service**: Simulated real-time price updates (conceptual - every 1 minute)
- **Stock History Aggregator**: Maintains historical price data for charting
- **Portfolio Calculator**: Real-time ROI and performance calculations
- **Session Manager**: Token expiration monitoring and auto-logout

Security Layer

- JWT Filter Chain: Intercepts all requests, validates tokens, sets authentication context
- Role Authorization: `@PreAuthorize("hasRole('ADMIN')")` annotations on sensitive endpoints
- Token Refresh Mechanism: Refresh tokens with longer expiration for seamless UX
- Email Verification: Users must verify email before full access

Email Service Integration

- AWS SES Configuration: SMTP credentials stored securely
- Template Engine: HTML email templates for professional appearance
- Async Processing: Email sending happens asynchronously to avoid blocking
- Retry Logic: Failed emails queued for retry with exponential backoff

Admin Module

- User Management Service: CRUD operations on user accounts
- Stock Management Service: Add/modify/delete stocks in DynamoDB
- Analytics Dashboard: User activity metrics and system health

News Integration

- External News API: Integration with financial news providers
- Caching Layer: Redis cache for news data (5-minute TTL)
- Fallback Mechanism: Local news storage when external API fails

2.3 Architecture Patterns

1. Hybrid Database Architecture Pattern

- Polyglot persistence strategy combining relational and NoSQL databases
- MySQL for structured user data requiring ACID properties
- DynamoDB for high-throughput stock operations and flexible schema

2. Three-Tier Architecture

- Clear separation: Presentation (React) → Application (Spring Boot) → Data (MySQL + DynamoDB)
- Loose coupling enables independent scaling and technology swaps

3. Repository Pattern

- Abstraction layer between business logic and data access
- portfolioRepository for MySQL JPA operations
- Direct AWS SDK integration for DynamoDB (alternative to traditional repository)

4. Context API Pattern (Frontend)

- Global state management without prop drilling
- Centralized authentication, stock data, and registration state

5. RESTful API Design

- Resource-based endpoints following REST principles
- Stateless communication using JWT/session tokens
- Standard HTTP methods (GET, POST) for CRUD operations

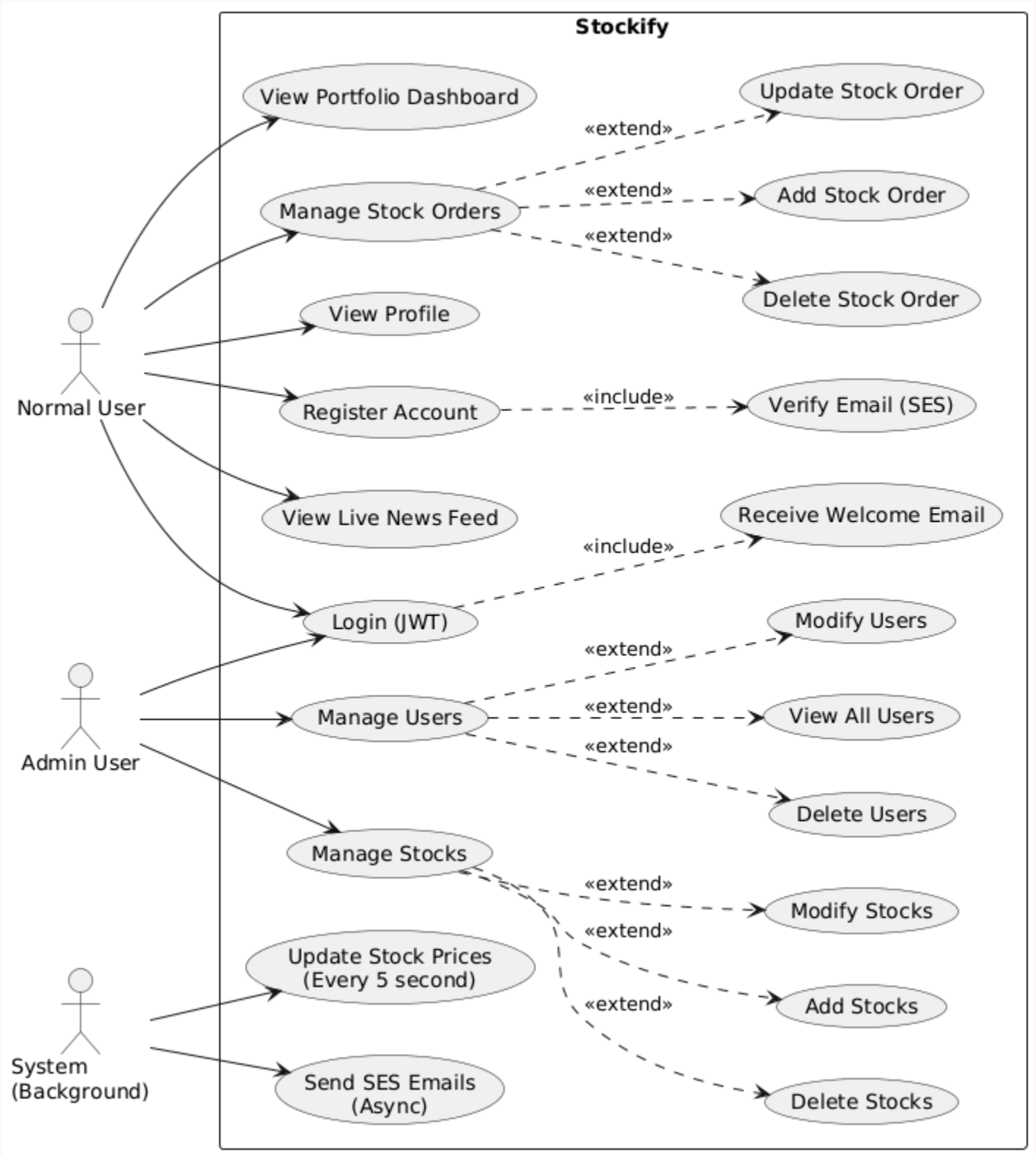
6. Service Layer Pattern

- Business logic encapsulation in portfolioService
- Transactions, validations, and complex calculations isolated from controllers

7. DTO Pattern

- Data Transfer Objects (PortfolioUpdateRequest) separate API contracts from internal models
- Prevents over-exposure of domain entities

3. Use Case Diagram



4. Use Case Descriptions

UC-01: User Registration with Subscription email

Use Case ID: UC-01

Actor: New User

Precondition: User has valid email address

Postcondition: User account created, verification email sent via AWS SES

Main Flow:

1. User completes 4-step registration form
2. System validates input and creates account in RDS with role=USER, email_verified=false
3. System generates JWT verification token
4. System sends subscription email via AWS SES
5. User clicks link in email
6. System validates token and sets email_verified=true
7. System displays success message

Alternative Flows:

- A1: Email already exists → Display error
- A2: SES email fails → Queue for retry, notify user
- A3: Verification token expires (24 hours) → User must request new link

UC-02: User Login with JWT & Welcome Email

Use Case ID: UC-02

Actor: Registered User

Precondition: User account exists

Postcondition: JWT tokens issued, welcome email sent

Main Flow:

1. User enters username/email and password
2. System validates credentials against RDS
3. System generates JWT access token (15 min expiry) and refresh token (7 days)
4. System sends "Welcome Back" email via AWS SES
5. System returns tokens to client
6. Client stores tokens (localStorage/sessionStorage)
7. User redirected to appropriate dashboard (User/Admin based on role)

Alternative Flows:

- A1: Invalid credentials → Display error, increment failed attempt counter
- A2: Account locked (5 failed attempts) → Display lockout message
- A3: Email not verified → Prompt to verify email first

UC-03: View Portfolio Dashboard

Use Case ID: UC-03

Use Case Name: View Portfolio Dashboard

Actor: Authenticated User

Precondition: User is logged in

Postcondition: Portfolio information is displayed

Main Flow:

1. User accesses portfolio dashboard
2. System retrieves user's stock orders from DynamoDB
3. System retrieves current stock prices from DynamoDB
4. System calculates portfolio metrics:
    - Total investment amount
    - Current portfolio value
    - Total gain/loss
    - Percentage change
5. System displays portfolio summary, holdings list, and charts
6. System automatically updates prices every 5 seconds

Alternative Flows:

- A1: If user has no stock orders, system displays empty portfolio message

Business Rules:

- Prices update automatically every 5 seconds
- Calculations are performed in real-time
- Display shows top holdings by value

UC-04: Add Stock Order

Use Case ID: UC-04

Use Case Name: Add Stock Order

Actor: Authenticated User

Precondition: User is logged in and viewing portfolio

Postcondition: New stock order is added to portfolio

Main Flow:

1. User clicks "Add Stock" button
2. System displays stock order form
3. User selects stock symbol from available stocks
4. User enters quantity and purchase price
5. User submits form
6. System validates input data
7. System creates new stock order record in DynamoDB
8. System refreshes portfolio dashboard
9. System displays success message

Alternative Flows:

- A1: If stock symbol is invalid, system displays error
- A2: If quantity is not positive, system displays validation error

Business Rules:

- Quantity must be positive integer
- Purchase price must be positive decimal
- Stock must exist in Stock table

UC-05: Update Stock Order

Use Case ID: UC-05

Use Case Name: Update Stock Order

Actor: Authenticated User

Precondition: User has existing stock orders

Postcondition: Stock order is updated in DynamoDB

Main Flow:

1. User clicks edit icon on stock order
2. System displays pre-filled form with current values
3. User modifies quantity or purchase price
4. User submits changes
5. System validates input
6. System updates stock order in DynamoDB
7. System recalculates portfolio metrics
8. System refreshes dashboard
9. System displays success message

Alternative Flows:

- A1: User cancels edit operation
- A2: Validation fails, system displays errors

UC-06: Delete Stock Order

Use Case ID: UC-06

Use Case Name: Delete Stock Order

Actor: Authenticated User

Precondition: User has existing stock orders

Postcondition: Stock order is removed from portfolio

Main Flow:

1. User clicks delete icon on stock order
2. System displays confirmation dialog
3. User confirms deletion
4. System deletes stock order from DynamoDB
5. System recalculates portfolio metrics
6. System refreshes dashboard
7. System displays success message

Alternative Flows:

- A1: User cancels deletion

UC-07: Real-time Stock Price Update

Use Case ID: UC-07

Use Case Name: Automatic Stock Price Update

Actor: System

Precondition: Stock table exists in DynamoDB

Postcondition: Stock prices are updated

Main Flow:

1. EventBridge triggers Lambda every 1 minute.
2. Lambda loop: Update all stock prices 10 times.
3. Write to DynamoDB currentPrice.
4. Clients refresh portfolios.

Business Rules:

- Updates occur 10 times every 1 minute consistently
- Price changes are realistic (± 2% variation)
- All active clients receive updates

UC-08: User Logout

Use Case ID: UC-08

Use Case Name: User Logout

Actor: Authenticated User

Precondition: User is logged in

Postcondition: User session is terminated

Main Flow:

1. User clicks logout button
2. System invalidates session token
3. System clears client-side authentication data
4. System redirects user to login page

UC-09: View Live News Feed

Use Case ID: UC-09

Actor: Any User (Authenticated)

Precondition: User is logged in

Postcondition: Latest financial news displayed

Main Flow:

1. User navigates to home page
2. System fetches news from external API or cache
3. System displays news articles with titles, summaries, and timestamps
4. News auto-refreshes every 5 minutes
5. User can click to read full articles

UC-10: Admin - Manage Users

Use Case ID: UC-10

Actor: Admin User

Precondition: User has ADMIN role, JWT token valid

Postcondition: User account modified/deleted

Main Flow:

1. Admin navigates to User Management dashboard
2. System displays all users with filters (role, status, registration date)
3. Admin selects user and action (Edit/Delete/Change Role)
4. System validates Admin JWT token and role
5. System performs operation on RDS User table
6. System logs admin action for audit trail
7. System displays success message

Business Rules:

- Admin cannot delete their own account
- At least one Admin must exist in system
- Deleting user cascades to OrderPortfolio in DynamoDB

UC-11: Admin - Manage Stocks

Use Case ID: UC-11

Actor: Admin User

Precondition: Admin authenticated with valid JWT

Postcondition: Stock added/modified/deleted in DynamoDB

Main Flow:

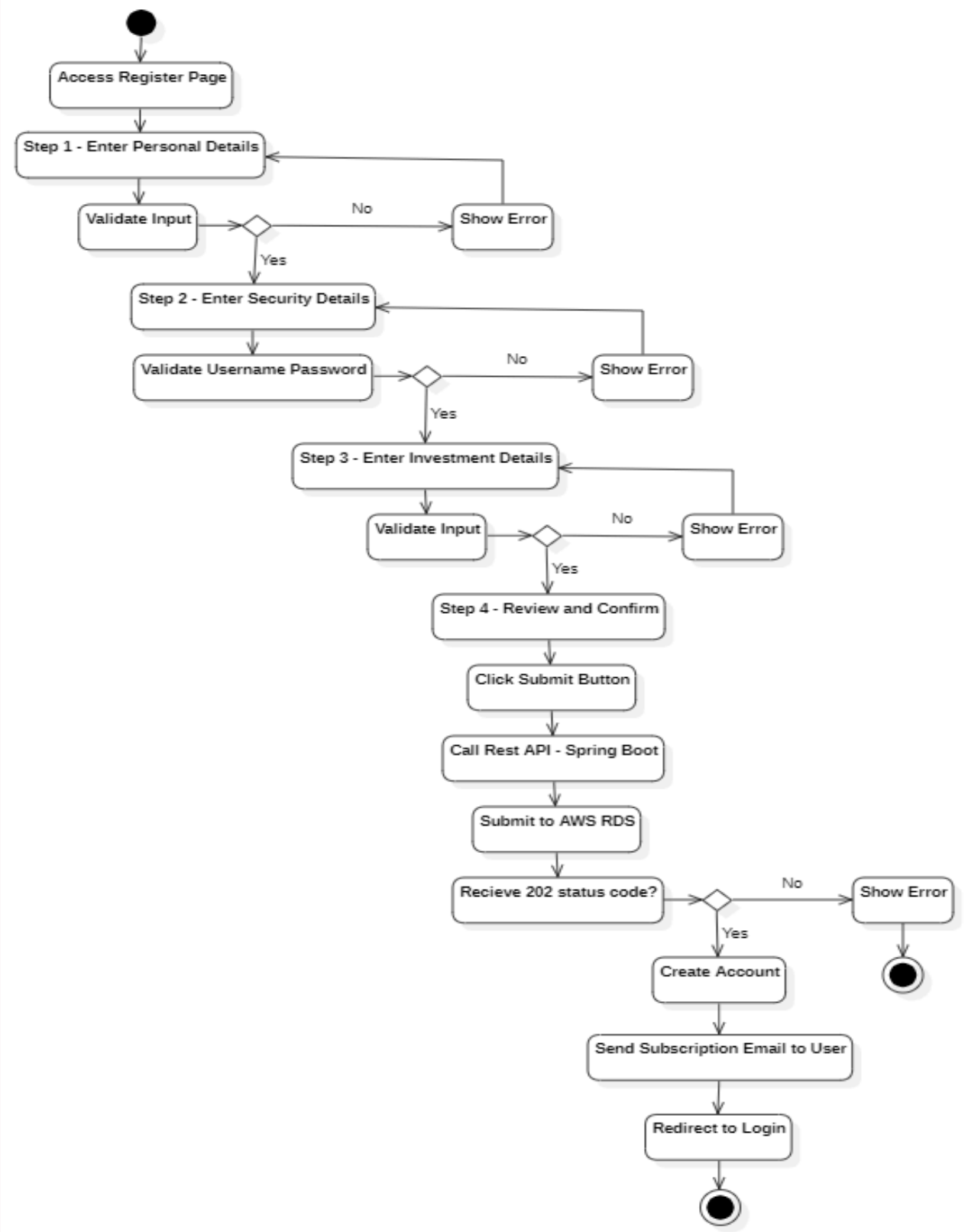
1. Admin accesses Stock Management console
2. System displays all stocks with current prices
3. Admin selects Add/Edit/Delete stock
4. Admin enters stock details (Symbol, Name, Initial Price)
5. System validates Admin authorization
6. System updates DynamoDB Stock table
7. System initializes price history for new stocks
8. System displays confirmation

Business Rules:

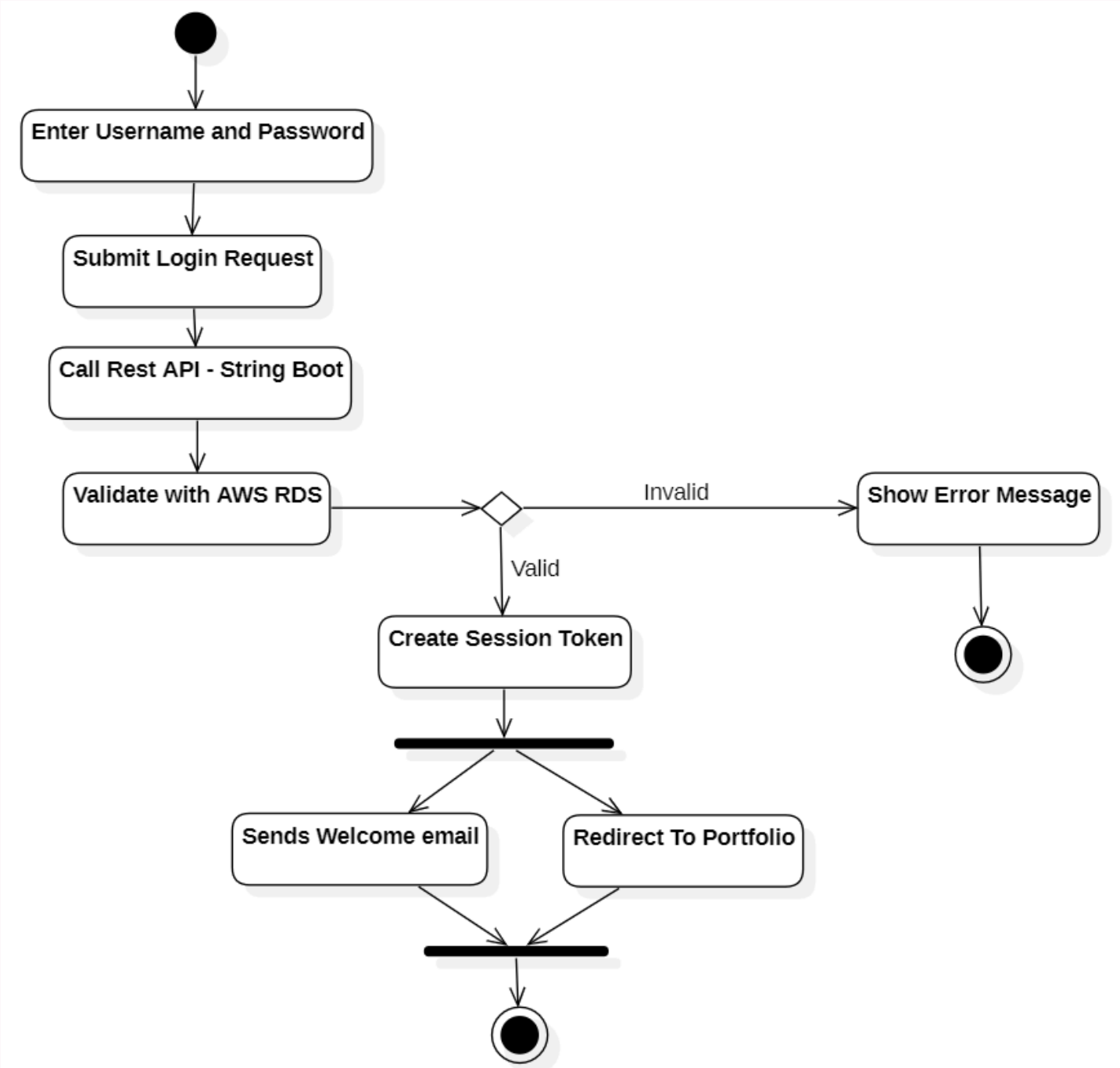
- Stock symbol must be unique (e.g., AAPL, TSLA)
- Cannot delete stock with active user orders (display warning)
- Price changes logged in history array

5. Activity Diagrams

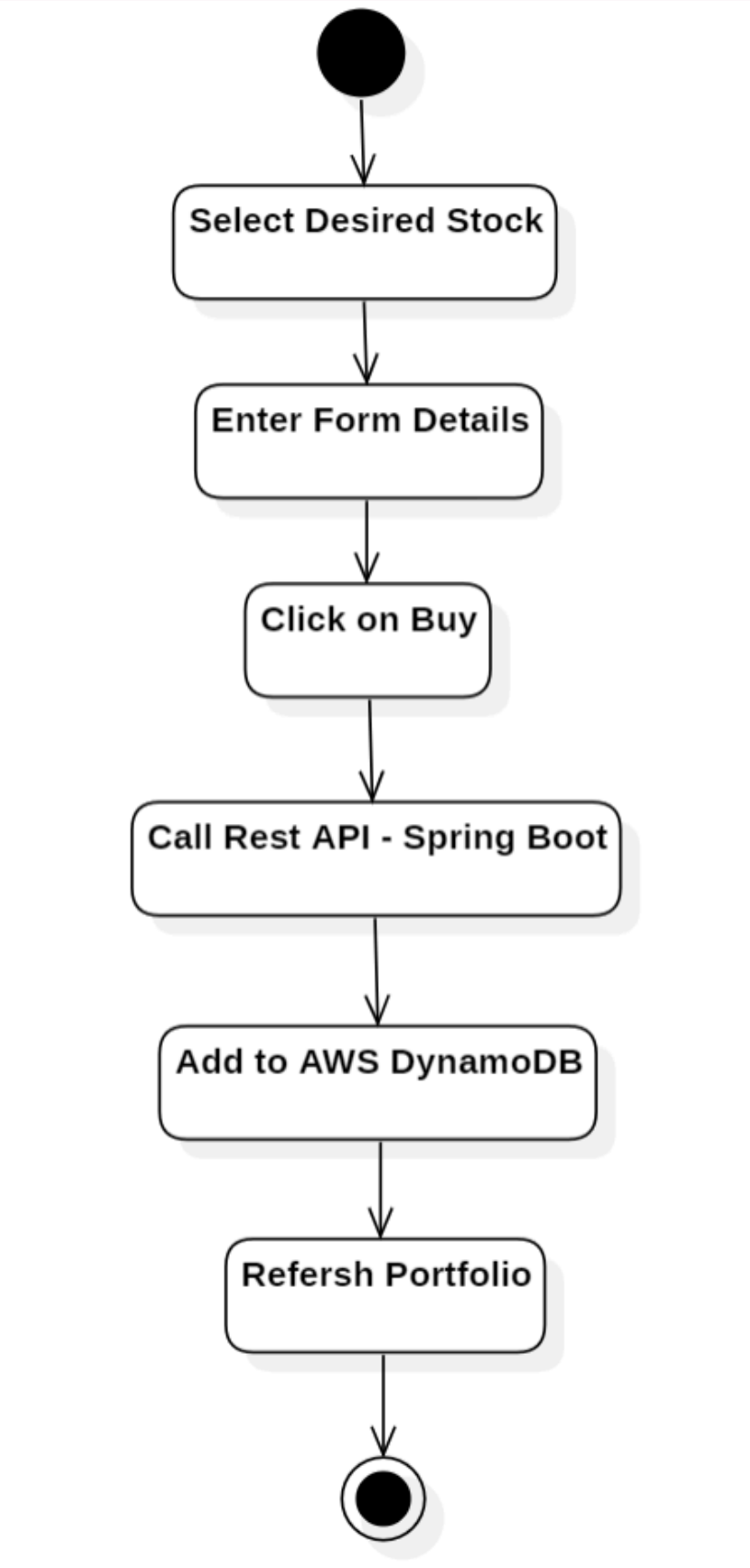
5.1 User Registration Activity Diagram



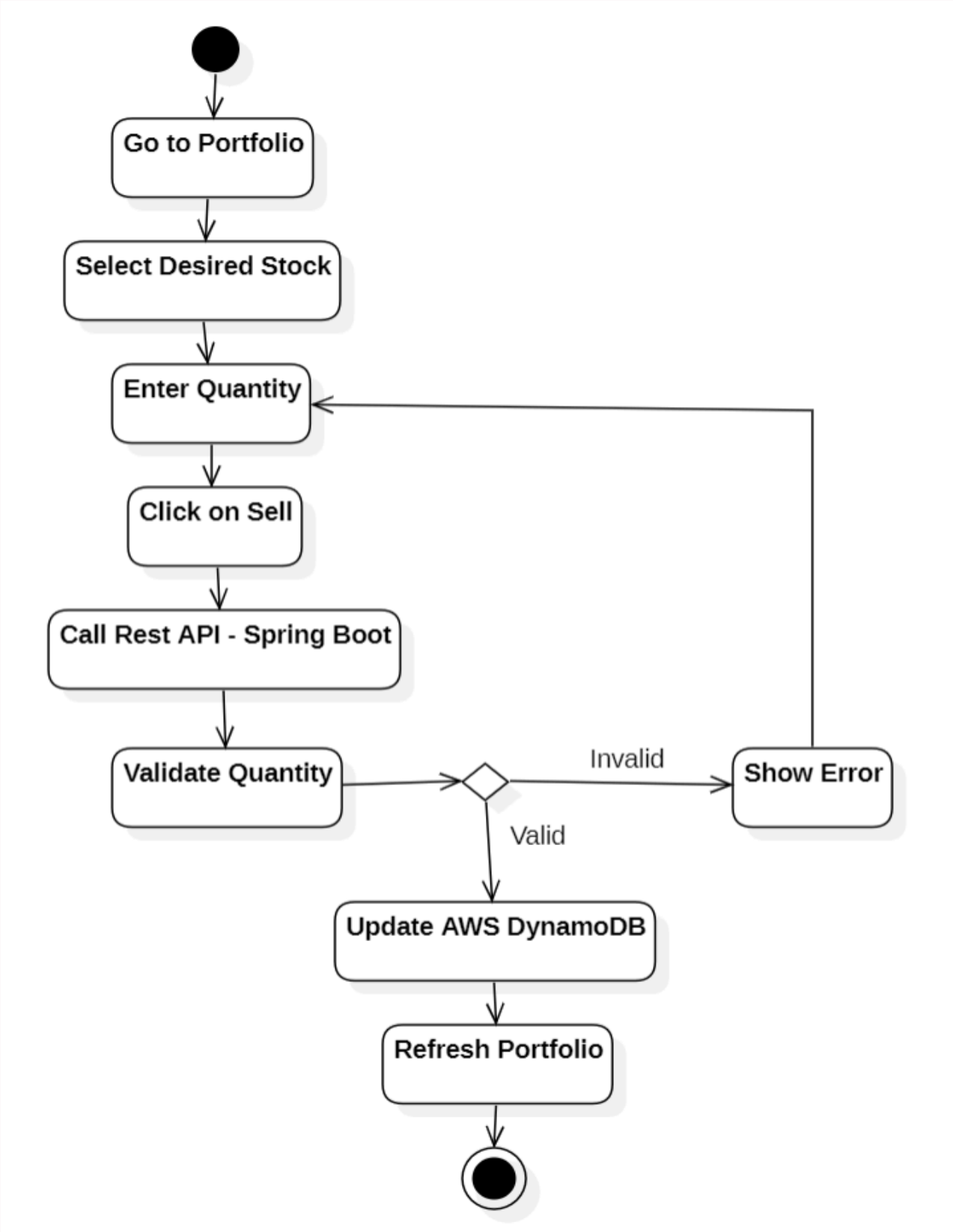
5.2 User Login Activity Diagram



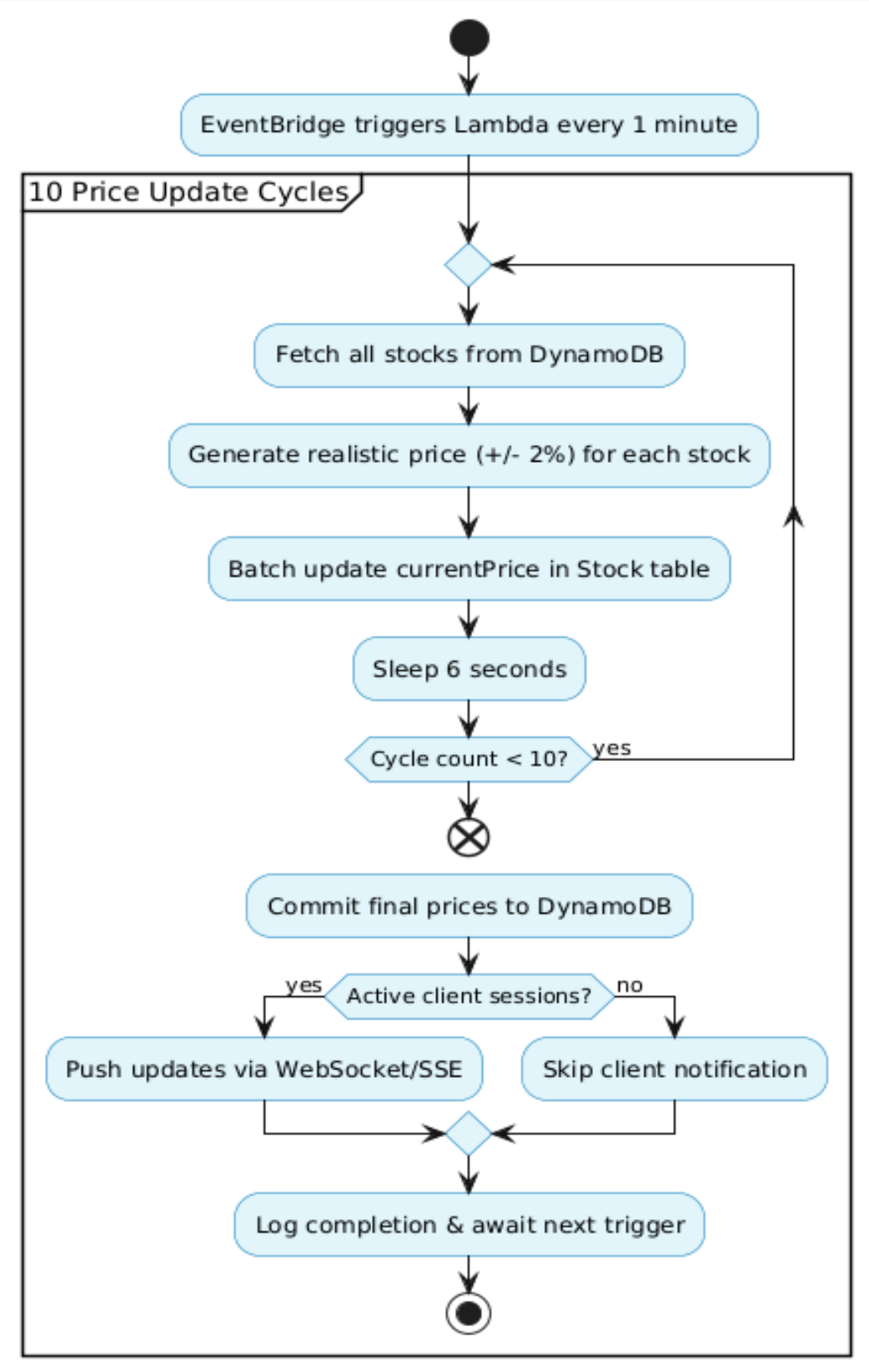
5.3 Buy Stock Activity Diagram



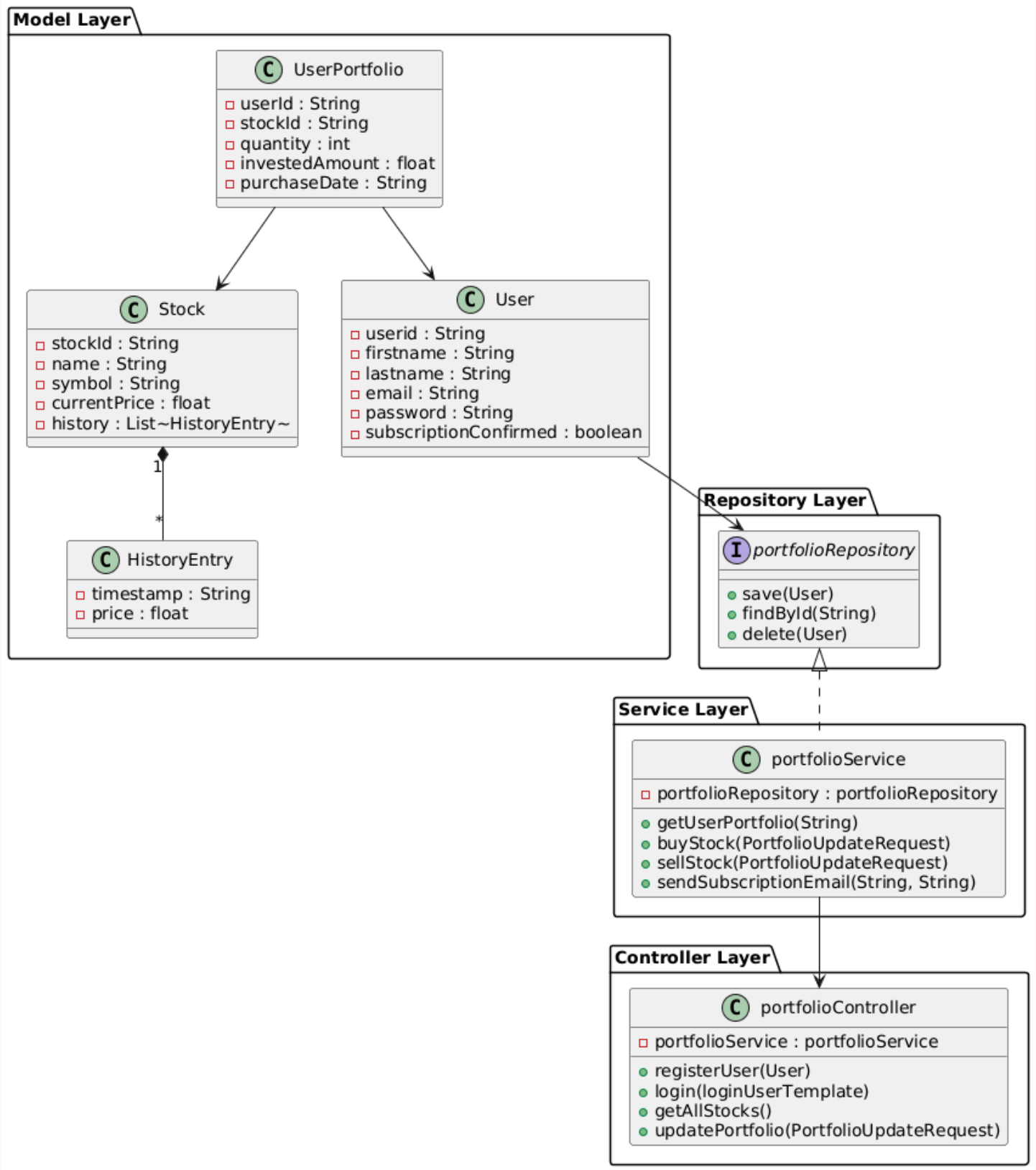
5.4 Sell Stock Activity Diagram



5.5 Real-time Price Update Activity Diagram



6. Class Diagram

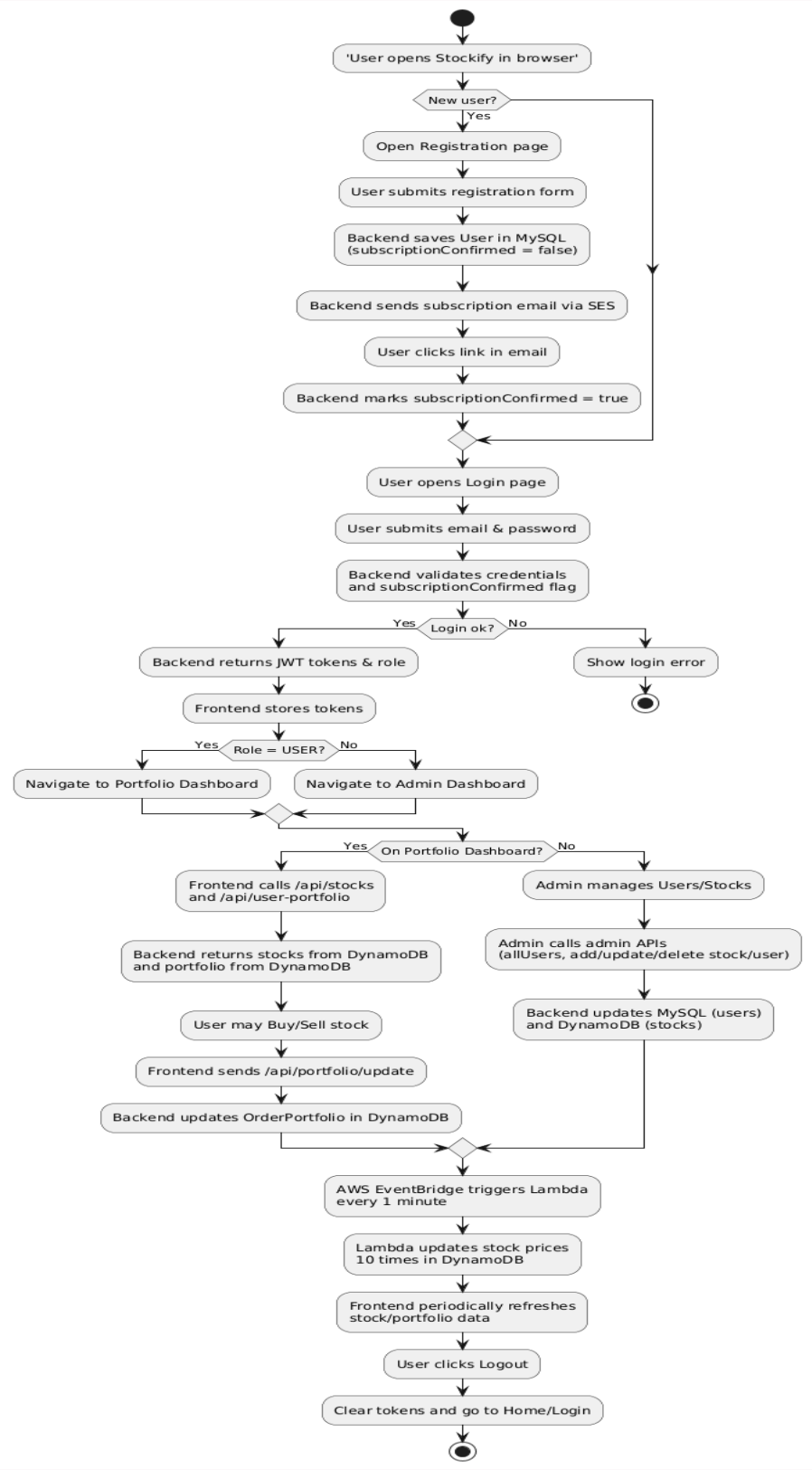


Explanation:

- **Model Layer (Entities):**
  - User: Contains user profile fields (user_id, first_name, last_name, email, password, risk_appetite, investment_goal, etc.).
  - Stock: Represents a stock (StockId, Name, Symbol, CurrentPrice, History list of price entries).
  - StockPriceHistoryEntry: An entry in the History list (price, timestamp).
  - OrderPortfolio: Represents a user's holding (UserId, StockId, investedAmount, purchaseDate, quantity).
- **Repository Layer:**
  - portfolioRepository: JPA repository for User entity in MySQL.
  - Direct AWS SDK calls for DynamoDB (Stock and OrderPortfolio tables).
- **Service Layer:**
  - portfolioService: Handles business logic like:
    - User registration/login and email sending.
    - Portfolio CRUD (add/update/delete stock orders).
    - Portfolio metric calculations (total investment, current value, gain/loss).
- **Controller Layer:**
  - portfolioController: Exposes REST endpoints (e.g., /register, /login, /portfolio, /stocks) and delegates to portfolioService.

7. Activity Flow Diagrams

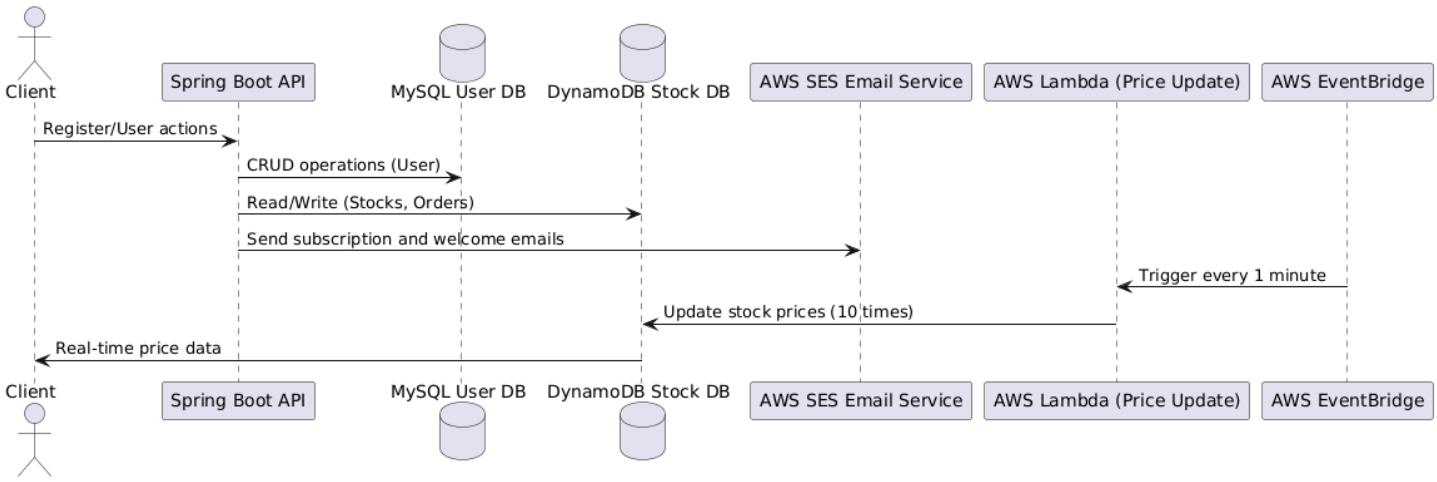
7.1 Complete Application Flow Diagram



Explanation:

- Shows the full journey:
    - User registers → verifies email → logs in → views dashboard → buys/sells stocks → admin manages users/stocks → background Lambda updates prices → UI reflects changes.
- Highlights how frontend, backend, databases, and AWS services interact over time.

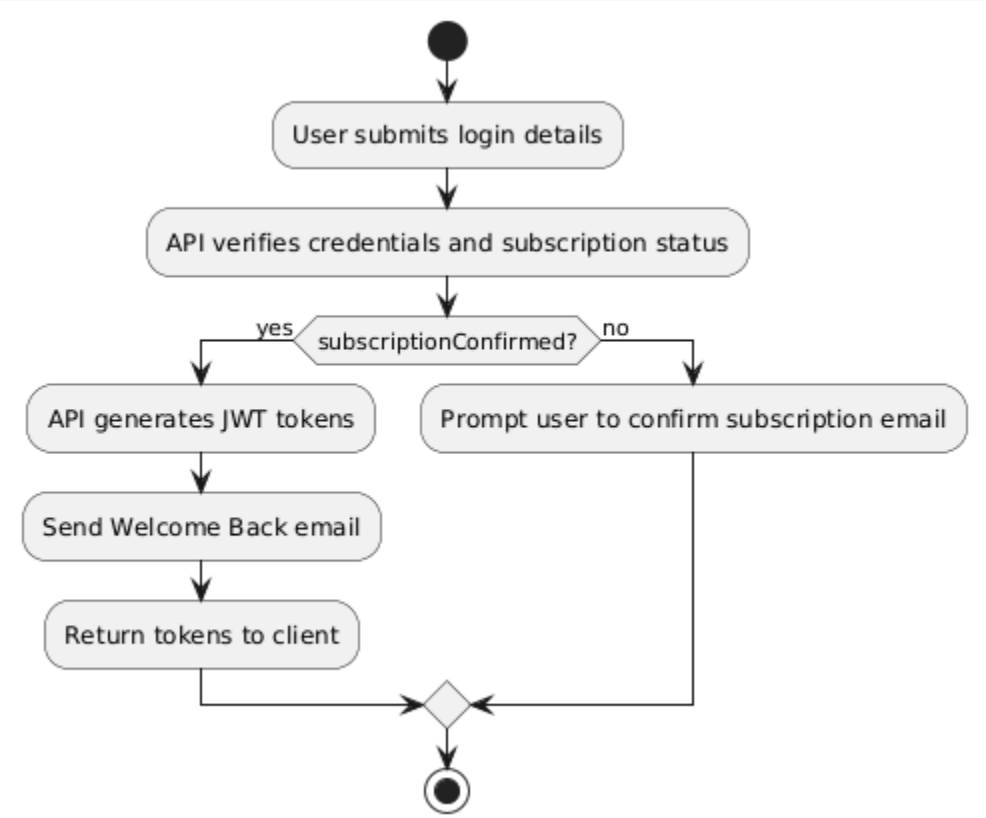
7.2 Data Flow Diagram



Explanation:

- Focuses on data movement:
    - User input (registration, login, stock order) → frontend → REST API → backend → MySQL (for user data) / DynamoDB (for stock/portfolio).
    - Background: Lambda reads stock data → updates prices in DynamoDB → frontend polls or receives updated prices → UI refreshes.
- Shows separation of concerns between relational and NoSQL data.

7.3 Authentication Flow Diagram



Explanation:

- User sends credentials → backend validates against MySQL → generates access/refresh tokens → returns tokens.
- Subsequent requests include JWT in header → JWT filter validates token → sets authentication context → allows access to protected endpoints.
- Refresh token flow for token renewal is also shown.

8. Database Design

8.1 Database Architecture Overview

The application uses a **hybrid database architecture** combining AWS RDS (MySQL) for relational user data and AWS DynamoDB for high-performance stock operations.

Rationale for Hybrid Approach:

- **AWS RDS (MySQL)**: User authentication data requires ACID properties, complex joins for user profiles, and traditional relational integrity
- **DynamoDB**: Stock prices and orders require high throughput, low latency reads/writes, and can handle eventual consistency for real-time updates

8.2 AWS RDS (MySQL) Schema

User Table

Table Name: `User`

Field	Type	Null	Key	Default	Description
user_id	INT	NO	PRI	NULL	Unique ID for each user
first_name	VARCHAR(255)	YES		NULL	User's first name
last_name	VARCHAR(255)	YES		NULL	User's last name
date_of_birth	DATE	YES		NULL	Date of birth
email	VARCHAR(255)	NO	UNI	NULL	User's email (unique)
phone	VARCHAR(255)	YES		NULL	Contact number
username	VARCHAR(255)	YES		NULL	Login username
password	VARCHAR(255)	YES		NULL	Hashed password
risk_appetite	VARCHAR(255)	YES		NULL	Risk profile
experience	VARCHAR(255)	YES		NULL	Trading experience

investment_goal	VARCHAR(255)	YES		NULL	Purpose of investment
created_at	DATE	YES		NULL	Account creation date
updated_at	DATE	YES		NULL	Account last updated

8.3 AWS DynamoDB Schema

Stock Table

Table Name: `Stock`

Primary Key: `StockId` (String, Partition Key)

Attribute	Type	Key	Description
StockId	String (S)	PK	Unique ID of the stock
Name	String (S)		Company name
Symbol	String (S)		Stock market symbol
CurrentPrice	Number (N)		Current market price
History	List<Map>		List of price entries

History Item Structure

Field	DynamoDB Type	Description
price	Number (N)	Price at given time
timestamp	String (S)	ISO-8601 timestamp

OrderPortfolio Table

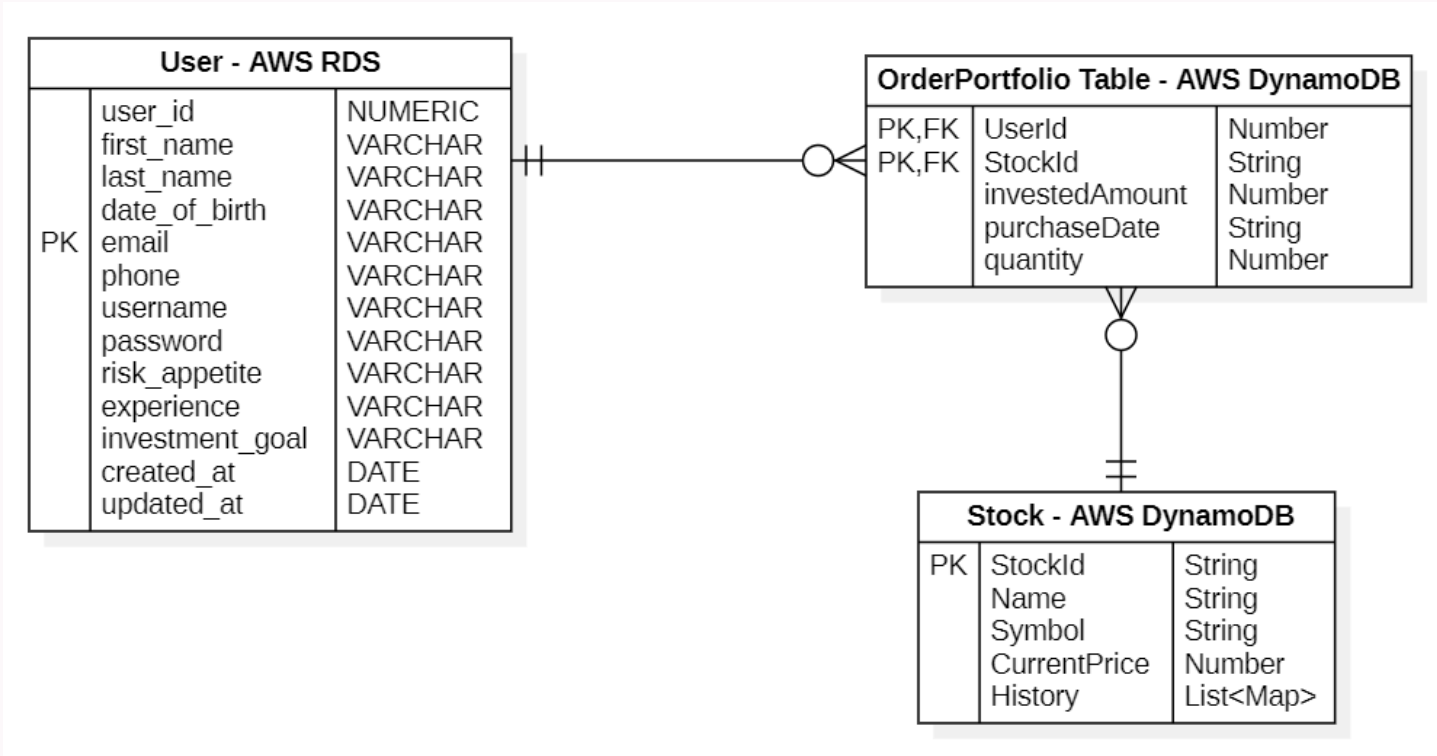
Table Name: `OrderPortfolio`

Primary Key:

- **Partition Key**: `UserId` (String)
- **Sort Key**: `StockId` (String)

Attribute	Type	Key	Description
UserId	String (S)	PK (Partition)	Links order to user
StockId	String (S)	SK (Sort Key)	Identifies which stock was ordered
investedAmount	Number (N)		Total amount invested
purchaseDate	String (S)		Purchase timestamp (ISO format)
quantity	Number (N)		Units purchased

8.4 Database Relationships (ER Diagram)



Cardinality:

- One User can have Many OrderPortfolio entries (1:N)

- One Stock can be in Many OrderPortfolio entries (1:N)

- OrderPortfolio links User and Stock (Many-to-Many relationship resolved)

Note: UserId in OrderPortfolio references user_id from User table

(logical foreign key, not enforced at database level)

8.5 Data Consistency Model

User Data (RDS)

- **Consistency:** Strong consistency (ACID properties)
- **Isolation Level:** READ_COMMITTED
- **Transactions:** Supported for multi-step operations
- **Suitable For:** User authentication, profile updates

Stock Data (DynamoDB)

- **Consistency:** Eventual consistency for reads (configurable)
- **Write Consistency:** Strong consistency guaranteed
- **Suitable For:** Real-time price updates where 2-second latency is acceptable
- **History Management:** Limit history array to last 100 entries to prevent item size issues

OrderPortfolio Data (DynamoDB)

- **Consistency:** Strong consistency for writes
- **Read Consistency:** Strongly consistent reads for portfolio calculations
- **Reason:** User portfolio must show accurate holdings, no tolerance for stale data

9. Module Breakdown

9.1 Frontend Modules (React.js)

9.1.1 Authentication Module

- App.js: Main entry point; routes and context providers.
- AuthContext: Manages authentication states, login/logout, token expiration.
- Login.js: Email/password login form.
- Register.js: Multi-step registration wizard with steps for personal info, account security, investment profile, and final review.
- RegisterUserContext: Manages complex state and validation of registration flow.

9.1.2 Portfolio Module

- PortfolioTracker.js: Main dashboard container.
- PortfolioSummary.js: Calculates portfolio metrics.
- StockCard.js: Displays individual stock holdings.
- NewStockCard.js: Shows general stocks with mini chart.
- PriceChart.js: SVG-based stock price chart.

9.1.3 Stock Operations

- StockDetails.js: Detailed stock view (includes buy/sell options).
- OrderStock.js: Form validating stock buy/sell orders.
- StockHistoryChart.js: Reusable trend chart.

9.1.4 Admin Modules

- ManageStocks.js: Admin dashboard for stock management.
- AddStockForm.js, ModifyStockForm.js: Add/edit stock forms.
- ManageUser.js: User management dashboard.
- ModifyUser.js: Update user profile/role.

9.1.5 General

- Navbar.js: Conditional links based on role and login.
- Home.js: Landing page with stock list.
- Profile.js: User account details.
- News.js: Business news from external API.
- ProtectedRoutes.js: Route protection with role-based access.
- PortfolioCalculator.js: Utility for portfolio computations.

9.2 Backend Modules (Spring Boot)

- Security Module: JWT validation filter, Spring Security config.
- EmailService: AWS SES integration for subscription and welcome emails.
- User Management Service: CRUD for users with MySQL via portfolioRepository.
- Portfolio Service: Business logic for stocks/orders using DynamoDB.
- Controllers: REST endpoints for registration, login, portfolio, admin operations.

9.3 AWS Infrastructure Modules

- DynamoDB: Stock, OrderPortfolio tables for high throughput data.
- AWS Lambda: Price update function triggered every 1 minute by EventBridge performing 10 price updates per run.
- AWS SES: Sends subscription confirmation and notification emails.
- AWS EventBridge: Serverless scheduler triggering Lambda.

9.4 Integration

- Frontend communicates with the backend strictly via REST API with JWT security.
- Backend unifies relational and NoSQL databases for data management and scalability.
- AWS cloud services used for notifications, scheduling, and scalability.

10. Unique Selling Proposition (USP)

10.1 Why This Application?

This application brings together portfolio tracking, real-time price simulation, and admin management into one cohesive platform instead of separate tools or spreadsheets. It is designed as an “enterprise-style” learning project that demonstrates how a realistic trading dashboard could be built end-to-end: secure onboarding, hybrid data storage, background price updates, and cloud deployment on AWS.

10.2 Technology Choices

10.2.1 Why React.js for frontend?

- React’s component-based model fits the UI needs of dashboards such as PortfolioTracker, StockCard, and admin tables, where only parts of the screen need to refresh when prices or holdings change.
- Hooks and Context (AuthContext, StockCon, RegisterUserContext) make it simple to share auth and stock state across pages and to implement protected routes and role-based navigation without heavy state libraries.

10.2.2 Why Spring Boot for backend?

- Spring Boot offers a mature ecosystem for REST APIs, security, validation, and integrations, which matches the needs of JWT auth, MySQL via JPA, and DynamoDB via the AWS SDK in this project.
- The layering with portfolioController, portfolioService, and portfolioRepository clearly separates HTTP concerns, business logic, and persistence, resulting in maintainable, testable backend code.

10.2.3 Why Hybrid databases (RDS + DynamoDB)?

- MySQL (RDS) is ideal for user registration, credentials, and profile data because those require strong consistency, relational constraints, and typical SQL queries.
- DynamoDB is better suited for high-volume stock and portfolio operations where data is key-value style and must be updated frequently (e.g., current prices, user holdings) with low latency and elastic scaling; using both lets each workload use the most suitable engine.

10.2.4 Why Real-time current stock price update 10 times in every 1 minute?

- Running the stock price updater once per minute via a scheduled Lambda keeps infrastructure simple and predictable while still feeling "live enough" for learning and demo purposes.
- Performing 10 internal price update cycles inside each 1-minute Lambda invocation simulates more granular market movement without paying for a very high trigger frequency or heavy continuous processing.

10.2.5 Why AWS cloud Infrastructure?

- AWS provides managed building blocks for each concern: RDS for relational data, DynamoDB for NoSQL, SES for transactional emails, Lambda for serverless compute, and EventBridge for scheduling, reducing the need to manage servers manually.
- Because these services are pay-as-you-go and many have generous free tiers at low traffic, the app can run at minimal cost for demos, while still illustrating patterns that scale to higher loads with the same architecture.

10.3 Competitive Analysis

- Compared to simple personal spreadsheets or single-page calculators, this app adds secure multi-user support, portfolio persistence, and simulated live pricing with automatic refresh.
- Compared to many basic tutorials that only show monolithic CRUD on one database, this project showcases a more realistic architecture with hybrid storage, JWT, and background services, closer to what production trading and analytics systems use.

10.4 Future Enhancements Potential

- The simulated price engine can be replaced with real market data APIs, extended with more instruments (ETFs, currencies), and enhanced with advanced charting (candlestick, moving averages, volume).
- The platform can add alerting (email/push notifications on thresholds), richer analytics (sector exposure, risk metrics), and collaboration features like shared portfolios or advisor views.

10.5 Learning Outcomes & Skills Demonstration

- Frontend skills: Designing a React SPA with routing, protected routes, context-based state, and reusable components for dashboards and admin screens.
- Backend and cloud skills: Implementing JWT security, layered Spring Boot services, mixing RDS and DynamoDB, and wiring AWS SES, Lambda, and EventBridge into a coherent solution.

11. Summary

11.1 Project Recap

The application enables users to register via subscription email, log in securely with JWT, manage their stock portfolio, and view live-updating performance metrics. Admins can manage both users and available stocks, while background services keep prices fresh in DynamoDB and the UI reacts to changes in near real time.

11.2 Key Achievements

- Implemented a full authentication flow with subscription email, JWT tokens, and role-based access (USER/ADMIN) across frontend and backend.
- Delivered an integrated portfolio experience with dashboards, charts, real-time price simulation, and admin modules on top of RDS + DynamoDB.

11.3 Architecture Highlights

- Clear three-layer architecture (client, application, data) plus a background services layer for price updates and email, making responsibilities easy to reason about.
- Hybrid persistence model and serverless price update loop show how to combine traditional web APIs with event-driven cloud components in a single system.

11.4 Tech Stack Summary

- React frontend with Context API, Axios, routing, and dedicated modules for auth, registration, portfolio, admin, and news.
- Spring Boot backend with security configuration, controller–service–repository layering, MySQL via JPA, DynamoDB via AWS SDK, SES integration, and scheduled/Lambda-style background jobs.

11.5 Business Value

- For end users, it centralizes portfolio monitoring and reduces manual updates by continuously syncing holdings with computed prices and metrics.
- For stakeholders (recruiters, mentors, or teams), it demonstrates practical ability to design and implement a realistic financial web application on a modern cloud stack.

11.6 Scalability & Performance

- DynamoDB's design and the stateless REST APIs make it straightforward to scale reads/writes and application instances as usage grows.
- Offloading periodic price computation to a scheduled background process (Lambda-style) avoids overloading user-facing APIs and keeps response times predictable.

11.7 Security Measures

- Uses hashed passwords, JWT-based stateless sessions, and role-based authorization checks on sensitive endpoints and routes.
- Separates admin capabilities (user/stock management) from normal user actions, reducing the blast radius of misconfigurations and ensuring least-privilege access.

11.8 Lessons Learned

- Splitting responsibilities between MySQL and DynamoDB based on access patterns leads to simpler, more efficient data models than forcing everything into a single store.
- Designing flows around background events (subscription email, periodic price updates) improves user experience without complicating the main HTTP request paths.

11.9 Future Roadmaps

- Technical: introduce CI/CD, comprehensive test suites, observability (logs, metrics, traces), and potential decomposition into microservices for auth, pricing, and portfolios.
- Product: add richer analytics, personalization (e.g., goals, risk-based suggestions), and possibly mobile or desktop clients reusing the same backend APIs.

11.10 Conclusions

The project successfully ties together frontend UX, backend engineering, data design, and cloud services into a coherent stock portfolio tracker that mirrors many patterns from real trading dashboards. It stands as both a practical tool for experimentation and a strong portfolio artifact to demonstrate full-stack and cloud-native development skills.