System Design Task: Automated Investment System

Problem Statement

Design a system that allows users to deposit funds into an account, which are then automatically invested into appropriate financial assets based on predefined criteria or user preferences.

Requirements

Functional Requirements

1. **User Onboarding:** Allow new users to create an account.
2. **Deposit Funds:** Users must be able to deposit money into their account through various methods (e.g., bank transfer, card payment).
3. **Investment Strategy:** The system needs a mechanism to determine the correct assets for investment. This could be based on user risk profiles, pre-defined portfolios, or other criteria.
4. **Automated Investment:** Once funds are deposited, the system should automatically execute trades to invest the money according to the determined strategy.
5. **Account Viewing:** Users should be able to view their account balance, holdings, and transaction history.

Non-Functional Requirements

1. **Reliability:** The system must reliably process deposits and execute investments. Financial transactions must be accurate and atomic.
2. **Scalability:** The system should handle a growing number of users and transactions.
3. **Security:** Protect user data, financial information, and prevent unauthorized access. Ensure secure handling of transactions.
4. **Performance:** Deposits and investment executions should occur within a reasonable timeframe. Users should experience low latency when viewing their account information.
5. **Compliance:** The system must adhere to relevant financial regulations and reporting requirements.

Task

1. Describe the high-level architecture of the system.
2. Detail the key components and their responsibilities.
3. Explain the data models required.
4. Discuss the technologies you would choose and justify your choices.
5. Outline how you would address the non-functional requirements, particularly security, reliability, and scalability.
6. Consider potential failure points and how the system would handle them.

Answers

The architecture can be broadly divided into the following layers:

Question 1

The system follows a layered, microservices-based architecture as follows:

1. **Presentation Layer (Client Interfaces - Web/Mobile app):**
    - A responsive interface for users to access their accounts, manage funds, set preferences, and view dashboards. (FR.1, FR.2, FR.5)
2. **API Gateway Layer:**
    - Acts as a single entry point for all client requests.
    - Responsibilities include:
        - **Request Routing:** Directing incoming requests to the appropriate backend microservice.
        - **Authentication & Authorization:** Verifying user identity and permissions before forwarding requests.
        - **Rate Limiting & Throttling:** Protecting backend services from abuse and overload.
        - **Load Balancing:** Distributing traffic across multiple instances of services.
        - **SSL Termination:** Handling HTTPS and encrypting/decrypting traffic.
        - **Request/Response Transformation:** Adapting requests and responses between clients and services if needed.
3. **Application Layer (Microservices):**
    - **User Service:** Manages user onboarding (account creation, verification processes), profile management, and authentication credentials.
    - **Account Service:** Handles user financial accounts, including balance management, deposit/withdrawal tracking, and transaction history.
    - **Payment Service:** Integrates with various payment gateways (bank transfers, card processors etc.) to facilitate fund deposits and withdrawals.
    - **Investment Strategy Service:** Manages the logic for determining appropriate investment strategies. This includes user risk profiling, selection of predefined portfolios, and potentially algorithmic strategy execution.
    - **Trading/Execution Service**: Executes trades by interacting with broker APIs
    - **Portfolio Service**: Maintains user holdings, current valuations, and performance tracking.
    - **Notification Service**: Sends alerts/updates via in-app notifications or user-specified (email, sms, whatsapp etc.)
    - **Compliance and Audit Service**: Logs all financial transactions and ensures regulatory reporting.

4. **Data Storage Layer:**
    - **Relational Databases:** For structured user data and transactions
    - **Time-Series Databases:** Optimized for storing and querying market data and portfolio performance over time.
5. **Integration Layer (Third-Party Services):**
    - **Payment Gateways:** For processing deposits and withdrawals.
    - **Brokerage APIs** For executing trades.
    - **Market Data Providers:** For fetching financial asset prices and information.
    - **KYC Verification Services:** For identity verification during user onboarding.

Question 2

Breakdown of the major microservices found in the above system design

1. **User Service:**
    - User registration, login and authentication
    - Stores user profiles and their preferences
    - Manages KYC verification
2. **Payment Service**
    - Manages fund deposits through supported payment methods (bank transfer, card payment etc.)
    - Tracks deposit status, handles payment failures and updates wallet balances
3. **Investment Strategy Service**
    - Determines asset allocation using algorithms based on user-selected strategies or risk profiles (low-risk, bold, etc.)
4. **Trading/Execution Service**
    - Executes investment orders through brokerage APIs
    - Converts strategy into executable trades and ensures best execution and compliance with trading constraints
5. **Portfolio Service:**
    - Maintains records of all user holdings, returns, asset distribution, performance metrics and history. (Also possibly support download/export of history functionality)
6. **Notification Service:**
    - Sends trade confirmations, app reminders, portfolio updates etc. via multiple channels (in-app, email, sms etc.)
7. **Compliance/Audit Service:**
    - Logs all financial activity and api calls to ensure adherence to financial regulations.

Question 3

The system would make use of several data models for each service. Brief explanations of the types of fields and relationships of these models are explained below.

1. **User:** This would contain relevant details pertaining to each individual's unique account, including: unique_id, name, email, phone number etc.
2. **Wallet:** This would track the user's account's cash balance available for investment including it's currency and wallet id etc.
3. **Deposit Action:** This would represent a deposit method action made by a user and would include the amount, payment method, status and reference_id etc.
4. **Investment Portfolio:** This would define a portfolio based on a user-defined strategy (low-risk, bold etc.)
5. **Holding:** Would represent actual assets/stock owned by the user, including the symbol, quantity, current value etc.
6. **Trade/Execution Record:** This would capture trades on behalf of the user and would either be buy or sell, and include the asset/stock symbol, quantity, price per stock, total cost and status etc.
7. **Audit Log:** This will track every "Trade/Execution Record" and other critical events for compliance to financial regulations.

Question 4

The system would make use of python flask for backend API services, Vue.js for the web interface, and Flutter/Dart for the (mobile) app. The reasons why are listed below:

Backend: Python Flask

- Lightweight and flexible for building REST APIs (micro-framework)
- Popular and well documented/maintained
- Ideal for integrating with data science, trading algorithms etc. as python is the most popular ecosystem for those things.
- Easy integration with third-party services.

Web Interface: Vue.js

- Vue is a modern, lightweight framework alternative to React/Angular
- It makes it easy to manage user sessions, investment data and real-time portfolio views.
- It is fast and efficient for dashboards and data-rich pages like transaction history or portfolio charts.

Mobile App: Flutter/Dart

- Flutter is cross-platform (android/ios), saving time and cost.
- Flutter has smooth ui performance, which is preferential in financial apps with charts, animations and real-time data.

- Strong ecosystem for authentication, networking, charts etc.

Question 5

1. **Security**
    - Authentication/Authorisation with secure login (JWT etc.) and 2-factor authorisation for sensitive actions.
    - Encrypt sensitive data
    - Use input validation to prevent injection attacks etc.
2. **Reliability**
    - Atomic Transactions to ensure consistency (deposits/investment execution)
    - Failover systems (avoid single points of failure)
    - Monitoring & Logging
3. **Scalability**
    - Microservice architecture allows for each service to scale independently (less issues)
    - Database optimisation for scalable data access
    - Stateless services (horizontal scaling) would perform well especially under high user or trading load
    - Load balancing would distribute requests evenly to improve performance
4. **Performance**
    - Caching to store frequently-accessed user information and computed data
    - Async processing would offload long-running tasks
    - Optimise database queries etc.
5. **Compliance**
    - KYC/AML third-party verification
    - Audit logging of all user and admin actions in case it is needed

Question 6

1. **Potential Failure 1: Payment Processing Failures (umbrella term)**
    - **Cause:** Network Issues, API downtime, invalid card details etc.
    - **System Handling:**
        i. First try to use implemented retry mechanisms.
        ii. Log failures and notify users.
        iii. Implement systems to prevent duplicate charges.
2. **Potential Failure 2: Trade Execution Failures**
    - **Cause:** Brokerage API downtime, market closed, insufficient funds etc.
    - **Handling:**
        i. First try to use implemented retry mechanisms
        ii. Notify users of failed trade after a set amount of time
        iii. Rollback partial transactions
3. **Potential Failure 3: Data Inconsistency**
    - **Cause:** Crashes during fund transfer or trade execution.
    - **Handling:** Atomic Transactions

4. **Potential Failure 4: Service/Service downtime**
    - **Cause:** Infrastructure failure or deployment bugs/issues.
    - **Handling:**
        i. Use scheduled health checks and auto-restart mechanisms
        ii. Notify users of downtime
5. **Potential Failure 5: Security Breach**
    - **Cause:** User/admin credential leakage, exposed API keys etc.
    - **Handling:**
        i. Enforce 2FA
        ii. Rotate admin credentials regularly
        iii. Increase encryption security
6. **Potential Failure 6: Scalability Bottlenecks**
    - **Cause:** Sudden urge in requests volume
    - **Handling:**
        i. Cache hot data (market prices etc)
        ii. Auto-scale containers