System Design Document

VAMPZ - Stock Market Simulator

---

1. High-Level Description Using CRC Cards

CRC Card Template

AppUser

Class	AppUser
Responsibilities	- Represents a user entity in the system - Implements UserDetails interface for Spring Security - Stores user information (firstName, lastName, email, password, role) - Manages account state (enabled, locked) - Provides authentication-related methods
Collaborators	UserRepository, AppUserService, Spring Security framework

AppUserService

Class	AppUserService
Responsibilities	- Implements UserDetailsService for Spring Security - Handles user authentication (loadUserByUsername) - Manages user registration (signUpUser) - Performs password encoding using BCrypt - Validates user existence and credentials - Provides login functionality (loginUser)
Collaborators	UserRepository, BCryptPasswordEncoder, AppUser

UserRepository

Class	UserRepository
Responsibilities	- Provides data access layer for AppUser entities - Extends JpaRepository for CRUD operations - Finds users by email address - Persists user data to PostgreSQL database
Collaborators	AppUser, PostgreSQL Database

RegistrationController

Class	RegistrationController
Responsibilities	- Handles HTTP POST requests for user registration - Receives registration requests from frontend - Delegates registration logic to RegistrationService - Returns registration status and messages
Collaborators	RegistrationService, RegistrationRequest

RegistrationService

Class	RegistrationService
Responsibilities	- Validates email format using EmailValidator - Creates new AppUser instances - Delegates user creation to AppUserService - Returns registration results with status messages
Collaborators	EmailValidator, AppUserService, RegistrationRequest

LoginController

Class	LoginController
Responsibilities	- Handles HTTP POST requests for user login - Manages HTTP sessions for authenticated users - Validates user credentials - Creates and manages user sessions - Handles logout requests - Provides endpoint to check current user (/me)
Collaborators	UserRepository, BCryptPasswordEncoder, HttpSession

EmailValidator

Class	EmailValidator
Responsibilities	- Validates email address format - Implements Predicate interface for email validation - Uses regex pattern matching to validate emails
Collaborators	RegistrationService

WebSecurityConfig

Class	WebSecurityConfig
Responsibilities	- Configures Spring Security filter chain - Sets up CORS configuration for frontend communication - Configures authentication provider - Manages session creation policy - Disables CSRF for API endpoints
Collaborators	AppUserService, BCryptPasswordEncoder, CorsConfigurationSource

AuthContext (Frontend)

Class	AuthContext
Responsibilities	- Manages user authentication state in React application - Provides authentication methods (login, signup, logout, checkAuth) - Maintains user session state - Handles API communication with backend - Provides authentication context to all components
Collaborators	React components, Backend API endpoints

Dashboard (Frontend)

Class	Dashboard

Class	Dashboard
Responsibilities	- Displays user dashboard interface - Shows portfolio statistics and data - Manages user authentication checks - Handles logout functionality - Renders portfolio, activity, and featured stocks
Collaborators	AuthContext, useNavigate, React Router

---

2. System Interaction with Environment

Dependencies and Assumptions

Operating System

- *Assumption*: The system runs on any OS that supports Java 21 and Node.js 18+
- *Supported Platforms*: Windows, macOS, Linux

Backend Dependencies

- *Java Version*: Java 21 (as specified in pom.xml)
- *Framework*: Spring Boot 3.4.10
- *Build Tool*: Apache Maven
- *Database*: PostgreSQL 13+ (configured in application.properties)
- *Database Connection*:
  - URL: jdbc:postgresql://localhost:5432/stockdb
  - Username and password configured in application.properties
- *Spring Security*: For authentication and authorization
- *Spring Data JPA*: For database operations
- *Hibernate*: As JPA implementation (configured with PostgreSQL dialect)
- *Lombok*: For reducing boilerplate code
- *BCrypt*: For password encryption

Frontend Dependencies

- *Node.js*: Version 18+
- *Package Manager*: npm
- *Framework*: React 19.1.1
- *Build Tool*: Vite 7.1.7
- *Routing*: React Router DOM 6.28.0
- *Styling*: Tailwind CSS 4.1.16, Custom CSS
- *Icons*: Lucide React
- *Animations*: Framer Motion

Network Configuration

- *Backend Server*: Runs on http://localhost:8080 (default Spring Boot port)
- *Frontend Server*: Runs on http://localhost:5173 (Vite default) or http://localhost:3000
- *CORS Configuration*:
  - Allowed origins: http://localhost:5173, http://localhost:3000
  - Allowed methods: GET, POST, PUT, DELETE, OPTIONS
  - Credentials: Enabled (for session management)
- *Session Management*: Uses HTTP session cookies with 1-hour timeout
- *API Communication*: RESTful API with JSON payloads

Database Configuration

- *Database Name*: stockdb (configurable via application.properties)
- *Hibernate DDL*: create-drop (for development - will be changed for production)
- *SQL Logging*: Enabled for development (spring.jpa.show-sql=true)

Assumptions

- PostgreSQL database is running and accessible on localhost:5432
- Database user has sufficient privileges to create/drop tables
- Frontend and backend are running on the same machine or network during development
- Browser supports modern JavaScript features (ES6+)
- Browser supports session cookies and CORS with credentials

Note: The configurations above reflect the development setup of the VAMPZ system (localhost-based). For deployment, the backend will be hosted on a cloud platform (e.g., AWS/Render) with a managed PostgreSQL database, and the frontend will be hosted on a static site platform (e.g., Vercel/Netlify). CORS, HTTPS, and environment variables will be adjusted accordingly for secure production communication.

---

3. System Architecture

Architecture Overview

The VAMPZ system follows a *three-tier architecture* pattern:

1. *Presentation Tier* (Frontend - React)
2. *Application Tier* (Backend - Spring Boot)
3. *Data Tier* (PostgreSQL Database)

Component Relationships

1. *Frontend → Backend*:
    - React components make HTTP requests to Spring Boot REST endpoints
    - Uses fetch API with credentials for session management
    - Communication via JSON
2. *Backend Controllers → Services*:
    - Controllers handle HTTP requests and delegate business logic to services
    - Services return Map<String, String> or Map<String, Object> for responses
3. *Services → Repositories*:
    - Services use repositories for data persistence
    - Repositories extend JpaRepository for CRUD operations
4. *Repositories → Database*:
    - JPA/Hibernate handles SQL generation and execution
    - Direct connection via JDBC to PostgreSQL
5. *Security Layer*:
    - WebSecurityConfig wraps all requests
    - AppUserService implements UserDetailsService for authentication
    - Session management via HttpSession

---

4. System Decomposition

Component Breakdown

Frontend Components (React)

Authentication Module

- AuthContext.jsx: Global authentication state management
- LoginPage.jsx: User login interface
- SignupPage.jsx: User registration interface
- ForgotPasswordPage.jsx: Password recovery interface

Main Application Module

- App.jsx: Main application router and layout
- Dashboard.jsx: User dashboard with portfolio, stats, and activity

UI Components

- Navbar.jsx: Navigation bar
- Hero.jsx: Hero section with featured content
- DataWidget.jsx: Reusable stock data display widget
- Features.jsx, Pricing.jsx, CTA.jsx, Footer.jsx: Marketing pages

Role in Architecture: Presentation layer - handles user interaction and displays data

Backend Components (Spring Boot)

Controller Layer (REST API Endpoints)

- RegistrationController: /api/signup - Handles user registration
- LoginController: /api/login, /api/login/me, /api/login/logout - Handles authentication

Service Layer (Business Logic)

- RegistrationService: Validates registration requests and creates users
- AppUserService: Manages user authentication and registration logic
- EmailValidator: Validates email format

Data Access Layer

- UserRepository: JPA repository for AppUser entity

- AppUser: JPA entity representing users in database

Security Layer

- WebSecurityConfig: Configures Spring Security
- PasswordEncoder: BCrypt password encoding configuration

Role in Architecture: Application layer - handles business logic, authentication, and data access

Database Schema

app_user Table

- id (Long, Primary Key, Auto-generated)
- firstName (String)
- lastName (String)
- email (String, Unique)
- password (String, Encrypted)
- role (Enum: USER, ADMIN)
- enabled (Boolean)
- locked (Boolean)

Role in Architecture: Data persistence layer - stores user information

Interaction Flow

Registration Flow:

1. User submits registration form → SignupPage
2. AuthContext.signup() → HTTP POST to /api/signup
3. RegistrationController → RegistrationService
4. EmailValidator validates email
5. RegistrationService → AppUserService.signUpUser()
6. AppUserService → UserRepository.save()
7. UserRepository → PostgreSQL (via JPA)
8. Response flows back through layers

Login Flow:

1. User submits login form → LoginPage
2. AuthContext.login() → HTTP POST to /api/login
3. LoginController → UserRepository.findByEmail()
4. Password verification with BCryptPasswordEncoder
5. Session creation (HttpSession)
6. Response with user data → AuthContext updates state
7. Redirect to Dashboard

Authentication Check Flow:

1. Component mounts → AuthContext.checkAuth()
2. HTTP GET to /api/login/me
3. LoginController checks session
4. Returns user data if authenticated
5. AuthContext updates user state

---

5. Error Handling and Exception Strategy

Error Handling Approach

Backend Error Handling

1. User Registration Errors

- *Invalid Email Format*:
    - Detected by: EmailValidator
    - Response: {"status": "error", "message": "Invalid email address"}
    - HTTP Status: 200 (business logic error, not HTTP error)
- *Duplicate Email*:
    - Detected by: AppUserService.signUpUser() checking UserRepository.findByEmail()
    - Response: {"status": "error", "message": "User with email already exists!"}
    - HTTP Status: 200

2. User Authentication Errors

- *Invalid Credentials*:
    - Detected by: LoginController or AppUserService.loginUser()
    - Response: {"status": "error", "message": "Invalid email or password"}
    - HTTP Status: 401 (UNAUTHORIZED) for login endpoint
    - HTTP Status: 200 for service layer (with error status in body)

- *User Not Found*:
    - Detected by: UserRepository.findByEmail() returns empty Optional
    - Response: {"status": "error", "message": "Invalid email or password"}
    - HTTP Status: 401 (UNAUTHORIZED)

3. Session Management Errors

- *No Active Session*:
    - Detected by: LoginController.me() checking HttpSession
    - Response: {"status": "error", "message": "Not authenticated"}
    - HTTP Status: 401 (UNAUTHORIZED)

4. Database Errors

- *Connection Failures*:
    - Handled by: Spring Data JPA exception handling
    - Expected Behavior: Application fails to start or throws runtime exceptions
    - Mitigation: Ensure database is running and accessible
- *Data Integrity Violations*:
    - Handled by: JPA constraint violations
    - Expected Behavior: Transaction rollback, exception thrown
    - Response: HTTP 500 (Internal Server Error) - should be caught and handled

5. General Exception Handling

- *Unhandled Exceptions*:
    - Currently: No global exception handler implemented
    - Expected Behavior: Spring Boot default error handling
    - Recommendation: Implement @ControllerAdvice for centralized exception handling

Frontend Error Handling

1. Network Errors

- *Connection Failures*:
    - Detected by: fetch() throws error
    - Handling: catch blocks in AuthContext methods
    - User Experience: Returns {success: false, message: "Login failed. Please try again."}
    - Display: Error messages shown in UI components

2. Authentication Errors

- *Failed Login*:
    - Detected by: Response contains status: "error"
    - Handling: AuthContext.login() returns {success: false, message: data.message}
    - User Experience: Error message displayed on login page

3. Invalid Input

- *Form Validation*:
    - Currently: Basic HTML5 validation
    - Recommendation: Implement client-side validation for better UX

4. Session Expiration

- *Expired Session*:
    - Detected by: /api/login/me returns error
    - Handling: AuthContext.checkAuth() sets user to null
    - User Experience: Redirect to login page (via Dashboard useEffect)

Error Response Format

Standard Success Response: json { "status": "success", "message": "Operation successful", "user": { ... } // Optional, for user data }

Standard Error Response: json { "status": "error", "message": "Error description" }

Exception Handling Strategy Summary

Current Implementation:

- ✅ Business logic errors return error status in response body
- ✅ HTTP status codes used appropriately (401 for unauthorized)
- ✅ Frontend catches network errors and displays user-friendly messages
- ❌ No global exception handler for unexpected server errors
- ❌ No validation framework for request validation
- ❌ Limited client-side input validation

Recommended Improvements:

1. Implement @ControllerAdvice for centralized exception handling
2. Add request validation using @Valid and validation annotations
3. Implement proper logging for errors
4. Add client-side form validation
5. Implement retry logic for network failures

6. Add timeout handling for API calls
7. Implement proper error boundaries in React

Security Considerations

Current Security Measures:

- ✅ Password encryption using BCrypt
- ✅ Session-based authentication
- ✅ CORS configuration for allowed origins
- ✅ SQL injection prevention (via JPA parameterized queries)
- ⚠ CSRF disabled (acceptable for API-only backend)
- ⚠ No rate limiting (should be added for production)

Error Information Disclosure:

- Current: Error messages are user-friendly and don't expose system internals
- Recommendation: Implement different error messages for development vs. production

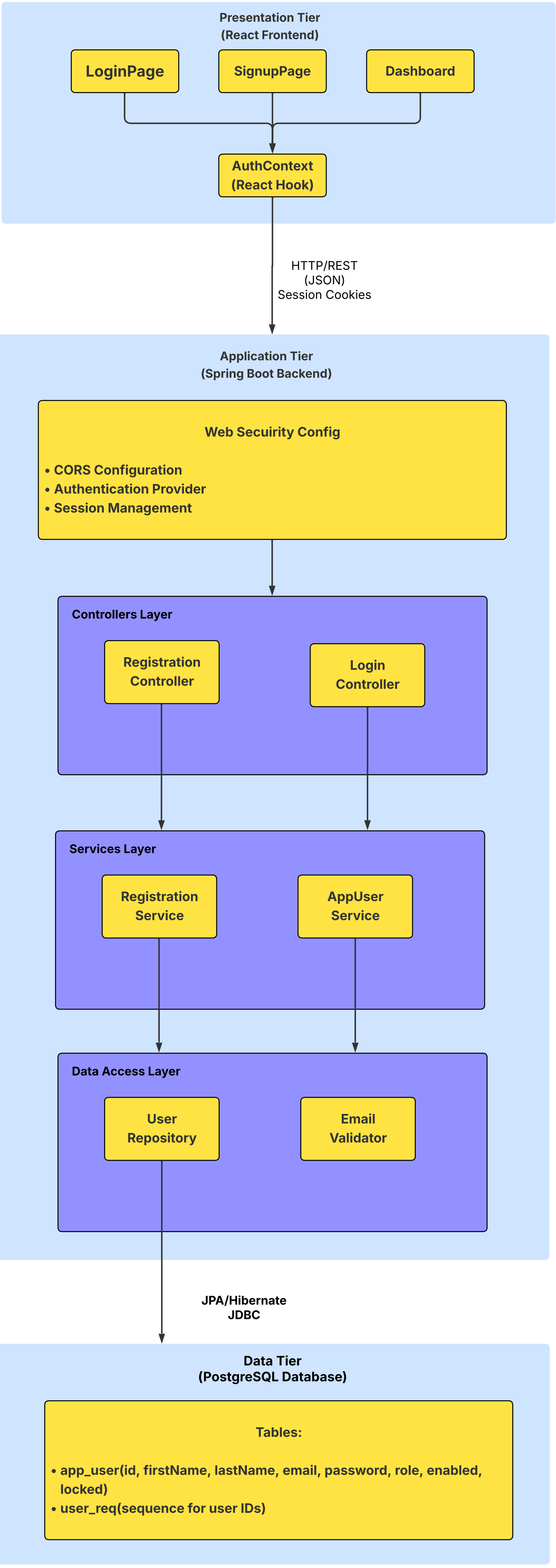
---

Document Revision History

Version	Date	Author	Changes
1.0	Sprint 1	Team	Initial system design document

Notes

- This document reflects the current state of the system as of Sprint 1
- The architecture may evolve as additional features are implemented (portfolio management, stock trading, etc.)
- Database schema is currently minimal (users only) and will expand with new features
- Error handling strategy should be enhanced as the system matures
- Security considerations should be reviewed before production deployment


Presentation Tier
(React Frontend)
LoginPage
SignupPage
Dashboard
AuthContext
(React Hook)
HTTP/REST
(JSON)
Session Cookies
Application Tier
(Spring Boot Backend)
Web Secuirity Config
• CORS Configuration
• Authentication Provider
• Session Management
Controllers Layer
Registration
Controller
Login
Controller
Services Layer
Registration
Service
AppUser
Service
Data Access Layer
User
Repository
Email
Validator
JPA/Hibernate
JDBC
Data Tier
(PostgreSQL Database)
Tables:
• app_user(id, firstName, lastName, email, password, role, enabled, locked)
• user_req(sequence for user IDs)