Enterprise Architecture Document

Table of Contents

1. Introduction

The Investment Management System is a backend application built using Java Spring Boot. It facilitates the management of investments and associated persons, allowing users to create, update, and delete investment records. The system is designed to be scalable, maintainable, and secure, providing RESTful APIs for integration with other systems.

2. Technical Summary

Aspect	Details
Language	Java
Framework	Spring Boot
Database	Relational Database (e.g., PostgreSQL)
Security	JWT Authentication, Role-based Access
API Documentation	OpenAPI (Swagger)
Design Patterns	MVC, DTOs, Repository Pattern

3. Functional Description

The system allows users to:

- Create, modify, and delete investments.

- Manage persons associated with investments.
- Retrieve investments by person ID.
- Calculate total investments by person.
- Authenticate and authorize users.

Actors

- **Users**: Individuals who manage investments and persons.
- **Administrators**: Users with elevated privileges for managing system settings.

4. Technical Analysis

4.1 Controller and Service Structure

Controllers:

- `InvestmentController`: Manages investment-related operations.
- `PersonController`: Manages person-related operations.

Services:

- `InvestmentService`: Contains business logic for investments.
- `PersonService`: Contains business logic for persons.

4.2 Domain Model: Main Entities

- **Investment**: Represents an investment with attributes such as ID, type, amount, and date.
- **Person**: Represents a person with attributes such as ID, full name, email, and tax ID.

4.3 Repositories and Persistence Strategy

Using Spring Data JPA for persistence:

- `InvestmentRepository`: Handles operations related to investments.
- `PersonRepository`: Handles operations related to persons.

4.4 Security Configuration

- **JWT**: Used for authentication.
- **CORS**: Configured to allow access from authorized frontends.
- **Role-based Access Control**: Different roles defined for user access.

4.5 External Integrations and Communication Points

- **Feign Client**: Potential integration points for external services (not explicitly defined in the provided code).

4.6 Design Patterns and Architectural Patterns Used

- **MVC**: Separation of concerns between Model, View, and Controller.
- **DTOs**: Data Transfer Objects used to encapsulate data for transfer between layers.
- **Repository Pattern**: Abstracts data access logic.

5. Specific Design Patterns Used

- **Factory Pattern**: Used in the `InvestmentMapper` and `PersonMapper` for creating DTOs from entities.
- **Singleton Pattern**: Spring beans are managed as singletons by default.
- **Strategy Pattern**: Potentially used for different investment types (not explicitly shown in the code).

6. Considerations for QA, DevOps, and Enterprise Architecture

- **Observability**: Implement logging and monitoring for tracking application behavior.
- **CI/CD**: Set up continuous integration and deployment pipelines for automated testing and deployment.
- **Configuration Management**: Use Spring profiles for managing different environments (development, testing, production).
- **Versioning**: Maintain version control for APIs.

7. Architectural Views

- **Logical View**: Represents the high-level structure of the system, including controllers, services, and repositories.
- **Development View**: Shows the organization of code and modules.
- **Process View**: Describes the flow of data and control through the system.
- **Implementation View**: Details the deployment of components and their interactions.
- **Use Case View**: Illustrates the interactions between users and the system.

8. Non-Functional Requirements

- **Performance**: The system should handle multiple concurrent requests efficiently.
- **Availability**: The system should be available 99.9% of the time.
- **Scalability**: The system should be able to scale horizontally to accommodate increased load.

9. Architectural Decision Records (ADR)

- **Decision 1**: Use Spring Boot for rapid development and ease of integration.
- **Decision 2**: Choose JWT for stateless authentication to enhance scalability.
- **Decision 3**: Utilize Spring Data JPA for simplified data access and management.

10. Conclusion and Recommendations

The architecture is designed for scalability and maintainability, with a focus on security and quality. It is recommended to adhere to best practices in development and keep documentation up to date to facilitate collaboration and system evolution.

11. Glossary

- **DTO**: Data Transfer Object
- **JWT**: JSON Web Token

- **CORS**: Cross-Origin Resource Sharing
- **MVC**: Model-View-Controller
- **CI/CD**: Continuous Integration/Continuous Deployment

This document serves as a comprehensive overview of the architecture of the Investment Management System, providing insights into its structure, functionality, and design considerations.