SUBMISSION EXPLANATION

- **SUBMISSION 1**

– architecture diagram (done)
– uml diagram (all the classes or just 6 user stories -> will check that)
– uml class descriptions for the classes (all the classes or just 6 user stories -> will check that)
– 6 uml
– class descriptions (6/6 done)
– 3 sequence diagrams (done)

- **SUBMISSION 2**

- the architecture diagram (previously done)
- uml (previously done)
- uml class description (previously done)

- the other 3 sequence diagrams
- state diagrams
- SOLID and design patterns (describe how we apply 3 SOLID principles and use 3 different DESIGN PATTERNS)

- **SUBMISSION 3**

- THE SDS(previously done)

- implement (code) in java the 6 user stories
- add a readme file .txt, explaining the files included and the tools used
- implement the sequence diagram
- do a private repo
- documentation and clean code
- presentation
- individual task

Summary of the SRS report

Invest Wise is a Sharia-compliant investment management application designed for Egyptian users to track multi-asset portfolios (stocks, real estate, gold, crypto), automate zakat calculations, and receive Islamic finance-compliant recommendations.

Core System Overview

- **Project Name**: Invest Wise
- **Purpose**: Sharia-compliant investment app for Egyptian users to track multi-asset portfolios (stocks, real estate, gold, crypto), automate zakat calculations, and receive Islamic finance recommendations.
- **Key Features**:
    - Multi-asset tracking with real-time valuation
    - Zakat calculation (2.5% of eligible assets)
    - Halal investment screening (filters non-compliant assets)
    - Risk assessment and portfolio optimization
    - Integration with Egyptian banks (CIB/QNB) and crypto exchanges (Binance)
    - Arabic-first UI with RTL support

Critical Functional Requirements (FRs)

1. **User Management** (FR01-02):
    - Secure signup/login (email, password with complexity rules).
2. **Portfolio Management** (FR02-04, FR11):
    - Add/edit/remove assets (stocks, real estate, crypto, gold).
    - Track net worth and ROI.
3. **Financial Goals** (FR04, FR12):
    - Set targets (e.g., retirement savings) with progress tracking.
4. **Risk & Compliance** (FR05-06, FR13-14):
    - Risk scoring based on diversification.
    - Halal screening and zakat calculation.
5. **Reporting** (FR07, FR15-16):
    - Generate performance reports (PDF/Excel).
    - AI-driven insights.

Non-Functional Requirements (NFRs)

- **Performance**:
  - <5s screen load time, <20s bank balance fetch.
- **Security**:
  - OWASP-compliant auth, data encryption (transit/at rest).
    (owasp = Open Worldwide Application Security Project (OWASP) is a nonprofit foundation dedicated to improving software security.)
- **Compatibility**:
  - Android (8.0+) and iOS (12.0+).
- **Scalability**:
  - Support 100k concurrent users.

Key User Stories (for Sequence Diagrams)

1. **User Signup/Login** (US #1-2)
2. **Add/Edit Assets** (US #3-4)
3. **Risk Assessment** (US #6)
4. **Zakat Calculation** (US #8)
5. **Halal Screening** (US #9) -> the process of filtering stocks to ensure they comply with Islamic law (Shariah). This involves excluding companies involved in prohibited activities like alcohol, gambling, and interest-based financial services, and ensuring financial ratios meet Shariah standards.
6. **Bank Integration** (US #10)

External Integrations

- **Egyptian Banks**: CIB, QNB (transaction sync).
- **Market Data**: Binance (crypto), Thndr (stocks).

Exclusions

- Direct stock trading (no brokerage integration).
- Tax filing automation.

Architecture diagram

https://drive.google.com/file/d/1cwfu4YjisO5po02yG0O_-4eltqTj5UlL/view?usp=sharing

UML DIAGRAM

https://drive.google.com/file/d/1S5gznrhZDVYCBSSy2oZqxkiFexGSxtDr/view?usp=sharing

the uml on the drive in the draft report
https://drive.google.com/file/d/1eEQOwFJfSIaHXec0GdLlx2YZC-f4V7CN/view?usp=sharing

Descriptions of the classes

1. **Sign up**

2. **Log in**

3. **Add asset and**
4. **Edit/remove asset**

Class id	Class name	Description & Responsibility
	Portfolio	Represents the collection of assets owned by an investor. It is responsible for managing the list of assets, including adding new ones, updating existing ones, or removing assets from the list.
	AssetManager	Serves as the main controller for asset-related operations. It receives requests from the investor (or frontend layer), processes input, validates asset data, and delegates storage or retrieval tasks to the appropriate components like Portfolio and AssetDatabase.
	AssetDatabase	Handles the database. It is responsible for storing new asset records update asset details, Delete asset records upon request from the system's storage.
	Asset	Represents a single investment asset (e.g., stock, crypto, real estate). It holds all the data needed to describe the asset like asset name, quantity of this asset, date the investor purchased the asset and the price he bought it with.
	AssetType (Enum)	The AssetType enum defines the predefined types of assets that an investor can add to their portfolio. Using an enum ensures consistency and prevents invalid asset categories from being entered.

		Possible Values: - Gold - Crypto - Real Estate - Stocks

5. Zakat calculation and

6. Manage/create bank account

Class id	Class name	Class description
	ZakatCalcStrategy	• The common interface for all concrete zakat calculators • Provides type-checking capability and Calculation logic Pattern: stragety
	CryptoZakatCalc	Calculates zakat for cryptocurrency holdings (concrete strategy)
	GoldZakatCalc	Calculates zakat for gold investments (concrete strategy)
	StockZakatCalc	Calculates zakat for equity investments (concrete strategy)
	EstateZakatCalc	Calculates zakat for real estate assets. (concrete strategy)
	ZakatCalculator	It uses a ZakatCalcStrategy instance to perform the actual calculation. (Context class)
	ZakatCalculatorFactory	Returns the appropriate ZakatCalcStrategy based on asset type. Pattern: factory
	InvalidAssetExp	Raised when asset data is incomplete.
	ZakatReport	Holds calculated zakat data and generates a PDF report.
	Bank	Represents a bank and stores data like bank name, bank ID and check if the bank exists in the list of supported banks
	BankAccManager	Acts as the main controller class that interacts with all the components: card verification, OTP, and account linking. Pattern: Facade
	CardInfo	Holds and validates the user's card data.
	BankAccount	Represents a successfully linked account.
	User	Represents a user with one linked BankAccount. Stores user information and associated account.

Sequence diagrams

- **Sign up**
  https://drive.google.com/file/d/1UGs1Z0G2cSsgsSzpmD06SfgO1U5GYRpW/view?usp=sharing

- **log in**
  https://drive.google.com/file/d/1ZNl1jksWXE9jrCw85dxIkCTA4tnAEuJP/view?usp=sharing

- **Add asset**
  https://drive.google.com/file/d/10zBMZtV0sHEG2YdRYkqqnxxA9ggMAGhn/view?usp=sharing

- **Edit/remove asset**
  https://drive.google.com/file/d/1esMsYt7ZcfWmU9p_lVc4zOdw7UiNhDzH/view?usp=sharing

- **Zakat calculation**
  https://drive.google.com/file/d/1_J2qYR3bnbpjiP2RM7Z2ombcOynllxF/view?usp=sharing

- **manage/create bank account**
  https://drive.google.com/file/d/1Qwf_hcDrj16YWj9etGjyofP0lTcqfhYG/view?usp=sharing

State diagram

https://drive.google.com/file/d/16R_AKJ9k0uvp8wKFsJQ8NnmqBTy4ziAx/view?usp=sharing

Design patterns

1. Log-In System (Decorator Pattern)

The log-in system uses the **Decorator Pattern** to add security checks step by step. The main Login_operation handles basic login, by verifying the username and password. These are done by adding "decorators" around it. Each decorator does one specific check. First the username decorator checks "does the username exist?", if yes, it passes the control to the other step to password validation which checks "Does the password match the username in the database?"). So the validation is done by wrapping the controls. This makes it easy to add or remove checks without changing the core login logic.

Key Benefits:

✓ **Flexible** – New checks (like 2FA) can be added without rewriting code.

✓ **Clean** – Each decorator handles just one task (Single Responsibility).

✓ **Extensible** – Decorators can be stacked in any order for different security needs.

---

2. Sign-Up System (Strategy Pattern)

The sign-up system uses the **Strategy Pattern** to handle different validations (name, username, email, password). Instead of putting all checks in one big method, each check is in its own class. The Sign_up_operation picks the right strategy for each field, making it easy to update or add new rules later.

Key Benefits:

✓ **Modular** – Each validation rule is separate and reusable.

✓ **Easy to Maintain** – Changing one rule (like password strength) doesn't break others.

✓ **Scalable** – New rules (e.g., phone number validation) can be added without rewriting old code.