Cairo University

Faculty of Computers and Artificial Intelligence

CS251

Introduction to Software Engineering

Investment - Architech

Software Design Specifications

Version 3.0

ID	Name	Email	Mobile
20230560	Nourhan Mohammed Ahmed Fahmy	nourhanfahmy8@gmail.com	01013162696
20230440	Nada Amin Fawzy	nadaamiin00@gmail.com	01556000182
20231088	Safia Mohammed Saeid	Safiam2006@gmail.com	01009846173

April of 2025

Software Design Specification

Contents

CS251: Architech
Project: Investment

Software Design Specification

Team

ID	Name	Email	Mobile
20230560	Nourhan Mohammed Ahmed Fahmy	nourhanfahmy8@gmail.com	01013162696
20230440	Nada Amin Fawzy	nadaamiin00@gmail.com	01556000182
20231088	Safia Mohammed Saeid	Safiam2006@gmail.com	01009846173

Document Purpose and Audience

The purpose of this document is to provide a technical blueprint of the system architecture, detailing its design, component interactions, and expected behaviors. It ensures developers can accurately build the system by visualizing it through the different diagrams, testers can validate functionality against defined outputs, and project managers can track progress.

Audience

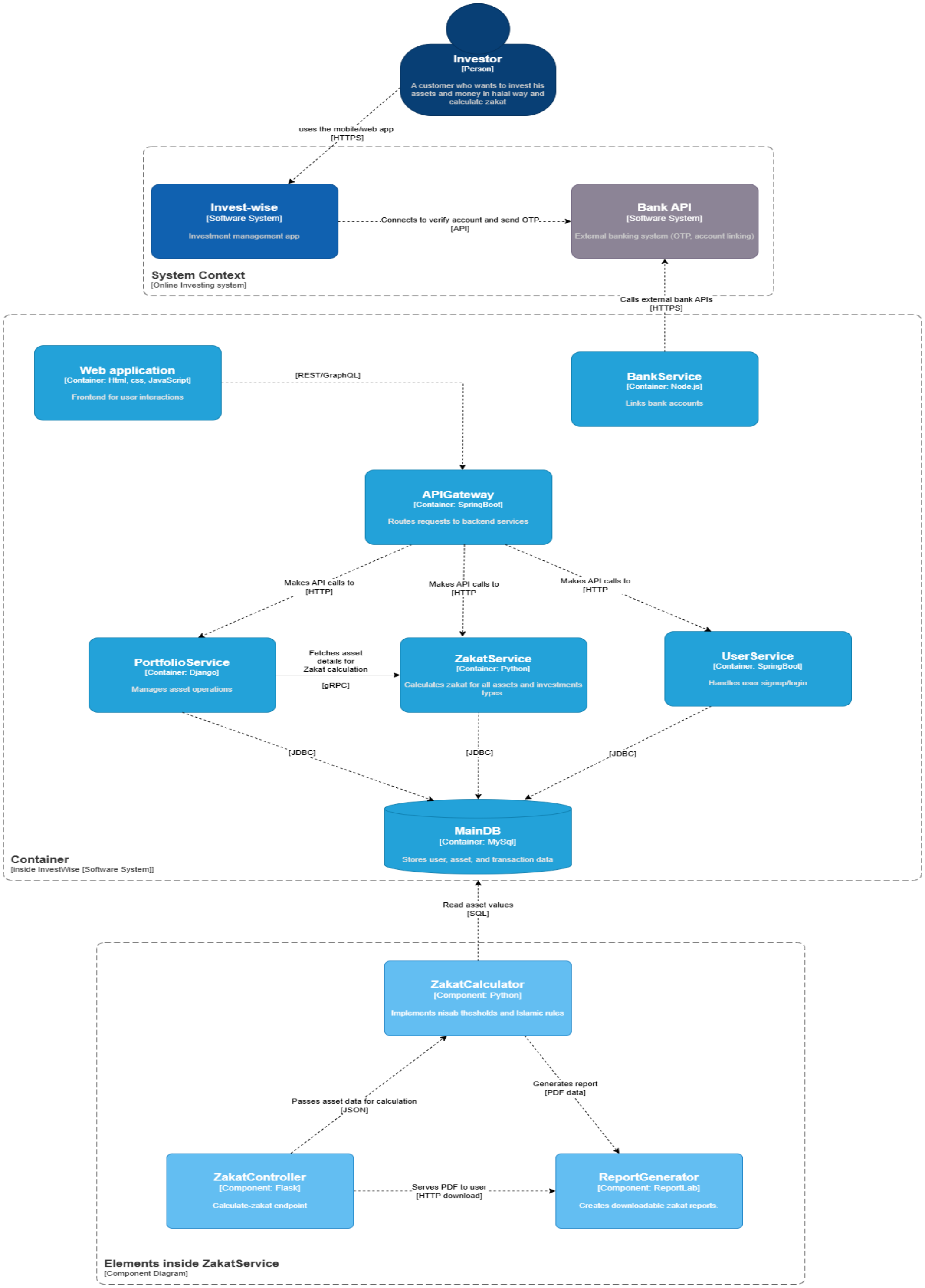
- System architects
- Project managers
- Testers
- Developers

Software Design Specification

System Models

I. Architecture Diagram

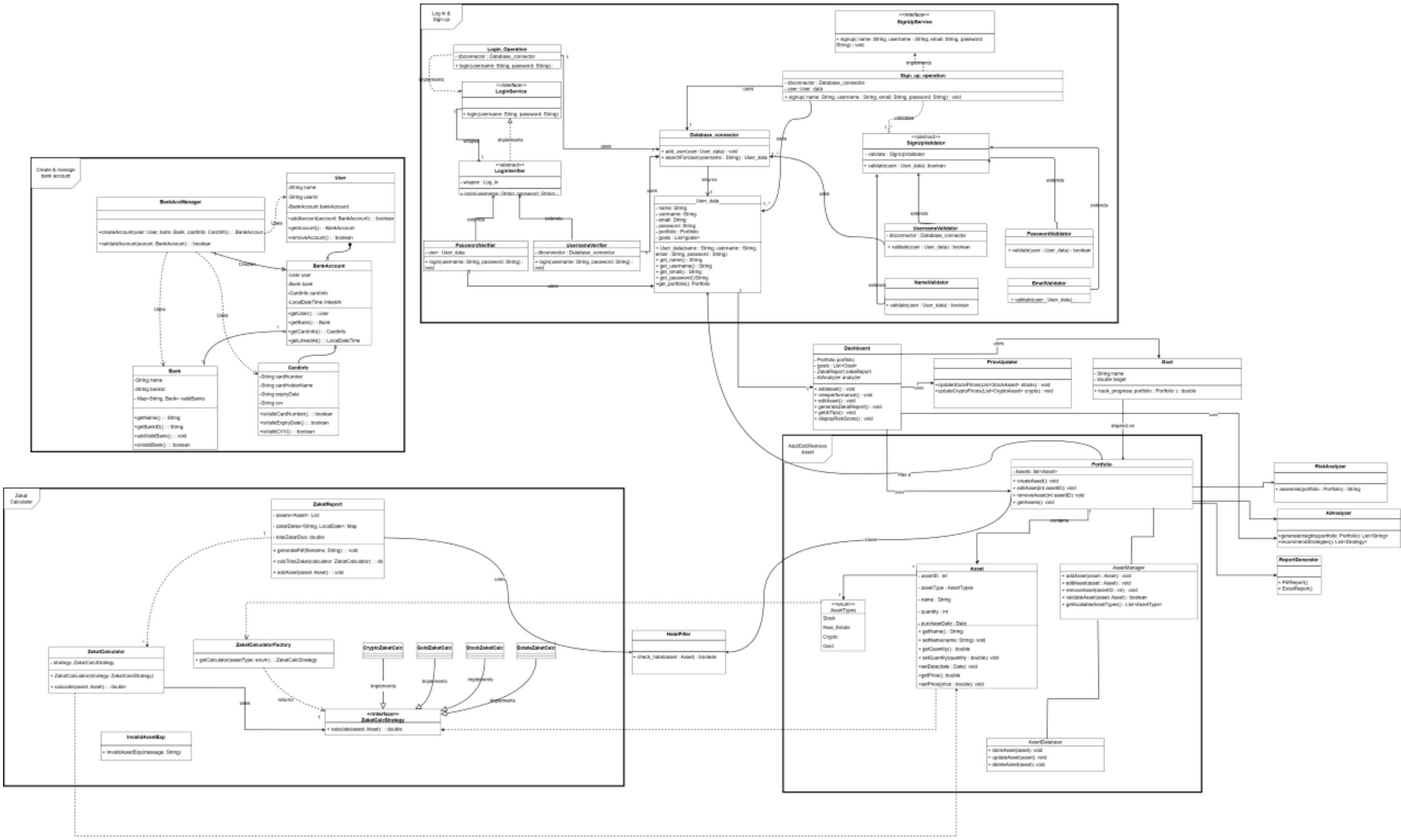
- **The Design We're Using:**
  We're building the app using separate mini-apps (called microservices) that all talk to each other, with one main doorway (API Gateway) controlling access.
- **What This Means:**
  - There's one main entrance (API Gateway) that checks who you are before letting you in
  - Different parts of the app handle different jobs:
    - User accounts (built with Java and stored into a database)
    - Bank connections (built with Node.js)
    - Zakat calculations (built with Python)
  - All parts share the same database (MySQL) to remember everything
- **Why This Works Best:**
  - **It's Flexible:**
    - We can update the zakat calculator without touching the bank connections
    - Different teams can work on different parts at the same time
  - **It's Safe:**
    - The main doorway checks everyone's ID (username, password)
    - If one-part crashes, the rest keep working
  - **It Grows Well:**
    - When lots of people use the zakat calculator during Ramadan, we can just make that part stronger which follows the non-functional scalability measure of being able to handle up to 100,000 users
    - Adding new features (like crypto) won't break existing ones
  - **It's Fast:**
    - The mini apps talk to each other quickly
    - The database handles many users at once
- **To conclude:**
  This way of building the app lets us:
  - Fix or improve parts without stopping everything
  - Keep your money and data safe
  - Handle more users when the app gets popular


Investor
[Person]
A customer who wants to invest his assets and money in halal way and calculate zakat
uses the mobile/web app
[HTTPS]
Invest-wise
[Software System]
Investment management app
Connects to verify account and send OTP
[API]
Bank API
[Software System]
External banking system (OTP, account linking)
System Context
[Online Investing system]
Calls external bank APIs
[HTTPS]
Web application
[Container: Html, css, JavaScript]
Frontend for user interactions
[REST/GraphQL]
BankService
[Container: Node.js]
Links bank accounts
APIGateway
[Container: SpringBoot]
Routes requests to backend services
Makes API calls to
[HTTP]
Makes API calls to
[HTTP
Makes API calls to
[HTTP
PortfolioService
[Container: Django]
Manages asset operations
Fetches asset details for Zakat calculation
[gRPC]
ZakatService
[Container: Python]
Calculates zakat for all assets and investments types.
UserService
[Container: SpringBoot]
Handles user signup/login
[JDBC]
[JDBC]
[JDBC]
MainDB
[Container: MySql]
Stores user, asset, and transaction data
Container
[inside InvestWise [Software System]]
Read asset values
[SQL]
ZakatCalculator
[Component: Python]
Implements nisab thesholds and Islamic rules
Passes asset data for calculation
[JSON]
Generates report
[PDF data]
ZakatController
[Component: Flask]
Calculate-zakat endpoint
Serves PDF to user
[HTTP download]
ReportGenerator
[Component: ReportLab]
Creates downloadable zakat reports.
Elements inside ZakatService
[Component Diagram]

CS251: Architech
Project: Investment

Software Design Specification

II. Class Diagram(s)



Log in & Sign up
Login_Operation
- dbconnector : Database_connector
+ login(username: String, password: String) :
implements
<<interface>>
LoginService
+ login(username: String, password: String)
wrapee
implements
<<abstract>>
LoginVerifier
- wrapee : Log_in
+ login(username: String, password: String) :
extends
extends
PasswordVerifier
- user : User_data
+ login(username: String, password: String) : void
UsernameVerifier
- dbconnector : Database_connector
+ login(username: String, password: String) : void
uses
uses
uses
<<interface>>
SignUpService
+ signup(name: String, username : String, email: String, password: String) : void
implements
Sign_up_operation
- dbconnector : Database_connector
- user: User_data
+ signup(name: String, username : String, email: String, password: String) : void
uses
Database_connector
+ add_user(user: User_data) : void
+ searchForUser(username : String) : User_data
returns
uses
uses
User_data
- name: String
- username: String
- email: String
- password: String
- portfolio : Portfolio
- goals : List<goals>
+ User_data(name : String, username : String, email : String, password : String)
+ get_name() : String
+ get_username() : String
+ get_email() : String
+ get_password():String
+get_portfolio(): Portfolio
validates
<<abstract>>
SignUpValidator
- validate : SignUpValidator
+ validate(user : User_data) : boolean
extends
extends
extends
extends
UsernameValidator
- dbconnector : Database_connector
+ validate(user : User_data) : boolean
PasswordValidator
+ validate(user : User_data) : boolean
NameValidator
+ validate(user : User_data) : boolean
EmailValidator
+ validate(user : User_data) :
Create & manage bank account
BankAccManager
+createAccount(user: User, bank: Bank, cardInfo: CardInfo) : : BankAccoun
+validateAccount(acount: BankAccount) : : boolean
Uses
Uses
Uses
Creates
User
-String name
-String userId
-BankAccount bankAccount
+addAccount(account: BankAccount) : : boolean
+getAccount() : : BankAccount
+removeAccount() : : boolean
BankAccount
-User user
-Bank bank
-CardInfo cardInfo
-LocalDateTime linkedAt
+getUser() : : User
+getBank() : : Bank
+getCardInfo() : : CardInfo
+getLinkedAt() : : LocalDateTime
Bank
-String name
-String bankId
- Map<String, Bank> validBanks
+getName() : : String
+getBankID() : : String
+addValidBank() : : void
+isValidBank() : : boolean
CardInfo
-String cardNumber
-String cardHolderName
-String expiryDate
-String cvv
+isValidCardNumber() : : boolean
+isValidExpiryDate() : : boolean
+isValidCVV() : : boolean
Dashboard
- Portfolio portfolio
- goals : List<Goal>
- ZakatReport zakaReport
- AIAnalyzer analyzer
+ addasset() : void
+ viewperformance() : void
+ editAsset() : void
+ generateZakatReport() : void
+ getAITips() : void
+ displayRiskScore() : void
uses
uses
uses
PriceUpdater
+UpdateStockPrices(List<StockAsset> stocks) : void
+updateCryptoPrices(List<CryptoAsset> crypto) : void
Goal
- String name
- double target
+ track_progress(portfolio : Portfolio) : double
depend on
Has a
AddEditRemove Asset
Portfolio
- Assets: list<Asset>
+ createAsset(): void
+ editAsset(int assetID): void
+ removeAsset(int assetID): void
+ getAssets(): void
contains
Uses
Asset
- assetID : int
- assetType : AssetTypes
- name : String
- quantity : int
- purchaseDate : Date
+ getName() : String
+ setName(name: String): void
+ getQuantity() : double
+ setQuantity(quantity : double): void
+setDate(date : Date): void
+getPrice(): double
+setPrice(price : double): void
<<enum>>
AssetTypes
Stock
Real_Estate
Crypto
Gold
AssetManager
+ addAsset(asset : Asset) : void
+ editAsset(asset : Asset) : void
+ removeAsset(assetID : int) : void
+ validateAsset(asset: Asset) : boolean
+ getAvailableAssetTypes() : List<AssetType>
AssetDatabase
+ storeAsset(asset): void
+ updateAsset(asset): void
+ deleteAsset(asset): void
RiskAnalyzer
+ assesrisk(portfolio : Portfolio) : String
AIAnalyzer
+generateInsights(portfolio: Portfolio): List<String>
+recommendStrategies(): List<Strategy>
ReportGenerator
+ PdfReport()
+ ExcelReport()
Zakat Calculator
ZakatReport
- assets<Asset>: List
- zakatDates<String, LocalDate>: Map
- totalZakatDue: double
+ generatePdf(filename: String) : : void
+ calcTotalZakat(calculator: ZakatCalculator) : : dc
+ addAsset(asset: Asset) : : void
uses
ZakatCalculator
- strategy: ZakatCalcStrategy
+ ZakatCalculator(strategy: ZakatCalcStrategy)
+ calculate(asset: Asset) : : double
ZakatCalculatorFactory
+ getCalculator(assetType: enum) : : ZakatCalcStrategy
returns
uses
CryptoZakatCalc
GoldZakatCalc
StockZakatCalc
EstateZakatCalc
implements
implements
implements
implements
<<interface>>
ZakatCalcStrategy
+ calculate(asset: Asset) : : double
InvalidAssetExp
+ InvalidAssetExp(message: String)
HalalFilter
+ check_halal(asset : Asset) : boolean

UML diagram:

https://drive.google.com/file/d/1eEQOwFJfSIaHXec0GdLlx2YZC-f4V7CN/view?usp=sharing

Software Design Specification

III. Class Descriptions

Class ID	Class Name	Description & Responsibility
1.	NameValidator	– Concrete validator class that ensures that the name is text ,and length of the name < 100 characters – It extends SignUpValidator concrete class)
2.	EmailValidator	– Concrete validator class that checks to make sure that the email is in the correct format – It extends SignUpValidator (concrete class)
3.	UsernameValidator	– Concrete validator class that ensures the username length < 50 and that's unique – It checks if it is unique by checking in the database if it hasn't been used before – It extends SignUpValidator (concrete class)
4.	PasswordValidator	– Concrete validator class that ensures the passwords length < 100, contains uppercase, digit and/or and special character – It extends SignUpValidator (concrete class)
5.	SignUpService	– It is an interface that defines the contract, which is what the Sign_up_operation class must do – Pattern: Strategy Pattern
6.	Sign_up_operation	– It is a concrete class that implements the SignUpService interface which handles new sign-up users – It creates a new user object and adds it to the database list if the username is valid.
7.	SignUpValidator	– It is a base abstract class for all the different validations
8.	Database_connector	– Stores users data in a database and handles the database operations – It is used by Login_operation, Sign_up_operation, log in and sign-up decorators.
9.	User_data	– Class that holds the info of a specific user and contains getters for these fields which the to return the data from the Database_connector – And each user has 1 portfolio associated with his account
10.	LoginService	– Interface that defines the login(username, password) method. – It serves as the base contract for log in operations and log in decorator to implement. – Pattern: Decorator pattern

Class ID	Class Name	Description & Responsibility
11.	LoginVerifier	– Abstract class that implements the LoginService class – Acts as a base class for the Log in decorators. – It holds a reference wrapee for the next component in the chain to be validated (username -> password)
12.	UsernameVerifier	– It is a concrete decorator which extends the abstract class LoginVerifier, and checks if the username exists in the database by using the database connector. If it exists, it passes the User_data wrapee object to pass it to the next decorator PasswordVerifier by calling the wrapee Log_in to continue the chain(using the decorator pattern)
13.	PasswordVerifier	– It is a concrete decorator which extends the abstract class LoginVerifier, and checks to make sure that the password matches with the associated username which was fetched using the UsernameVerifier – It does so by using the User_data object. If it is valid, it forwards the wrapee object to Log_in
14.	Login_operation	– It is a concrete class that implements the LoginService interface which takes the input username and password. It uses the class Database_connector to fetch the user data, and it can be wrapped by other decorators. – Verify username -> verify password -> log in operation
15.	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.
16.	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.
17.	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.

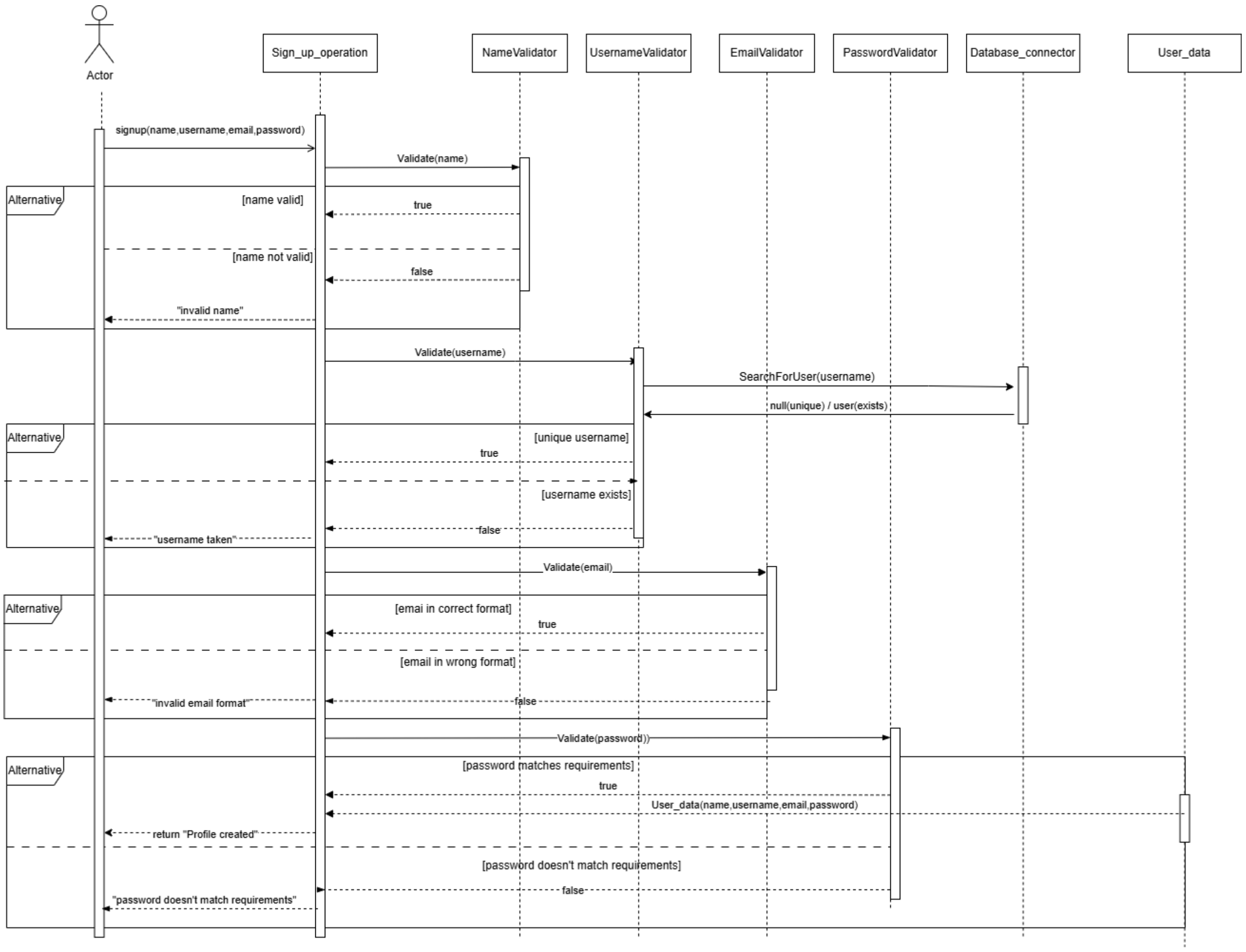
Class ID	Class Name	Description & Responsibility
18.	Asset	Represents a single investment asset (e.g., stock, crypto, real estate). It holds all the data needed to describe the asset like asset name, quantity of this asset, date the investor purchased the asset and the price he bought it with.
19.	AssetType (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
20.	EstateZakatCalc	Calculates zakat for real estate assets. (concrete strategy)
21.	ZakatCalculator	It uses a ZakatCalcStrategy instance to perform the actual calculation. (Context class)
22.	ZakatCalculatorFactory	Returns the appropriate ZakatCalcStrategy based on asset type. • Pattern: factory
23.	InvalidAssetExp	Raised when asset data is incomplete.
24.	ZakatReport	Holds calculated zakat data and generates a PDF report.
25.	Bank	Represents a bank and stores data like bank name, bank ID and checks if the bank exists in the list of supported banks

Class ID	Class Name	Description & Responsibility
26.	BankAccManager	Acts as the main controller class that interacts with all the components: card verification, OTP, and account linking. Pattern: Facade
27.	CardInfo	Holds and validates the user's card data.
28.	BankAccount	Represents a successfully linked account.
29.	User	Represents a user with one linked BankAccount. Stores user information and associated account.
30.	RiskAnalyzer	Evaluates risk levels associated with a financial portfolio; it takes the portfolio object as input then returns a string describing risk level
31.	AIAnalyzer	Uses AI to analyze portfolios and suggest actions.
32.	ReportGenerator	Generates excel and pdf reports
33.	DashBoard	It acts as the central interface for users to interact with their financial data. It provides access to the user's investment portfolio, financial goals, and tools to update prices or manage assets. It functions as a high-level controller that organizes core functionalities for visualizing and managing user investments.
34.	PriceUpdater	It is responsible for updating real-time price data for investment assets such as stocks, crypto, real estate, and gold. It retrieves the latest values and updates relevant asset objects to reflect current market conditions.
35.	Goal	Represents a user's financial objective, such as saving for retirement or reaching a certain net worth. It helps track progress toward the goal and may be used for generating progress reports and visualizations.

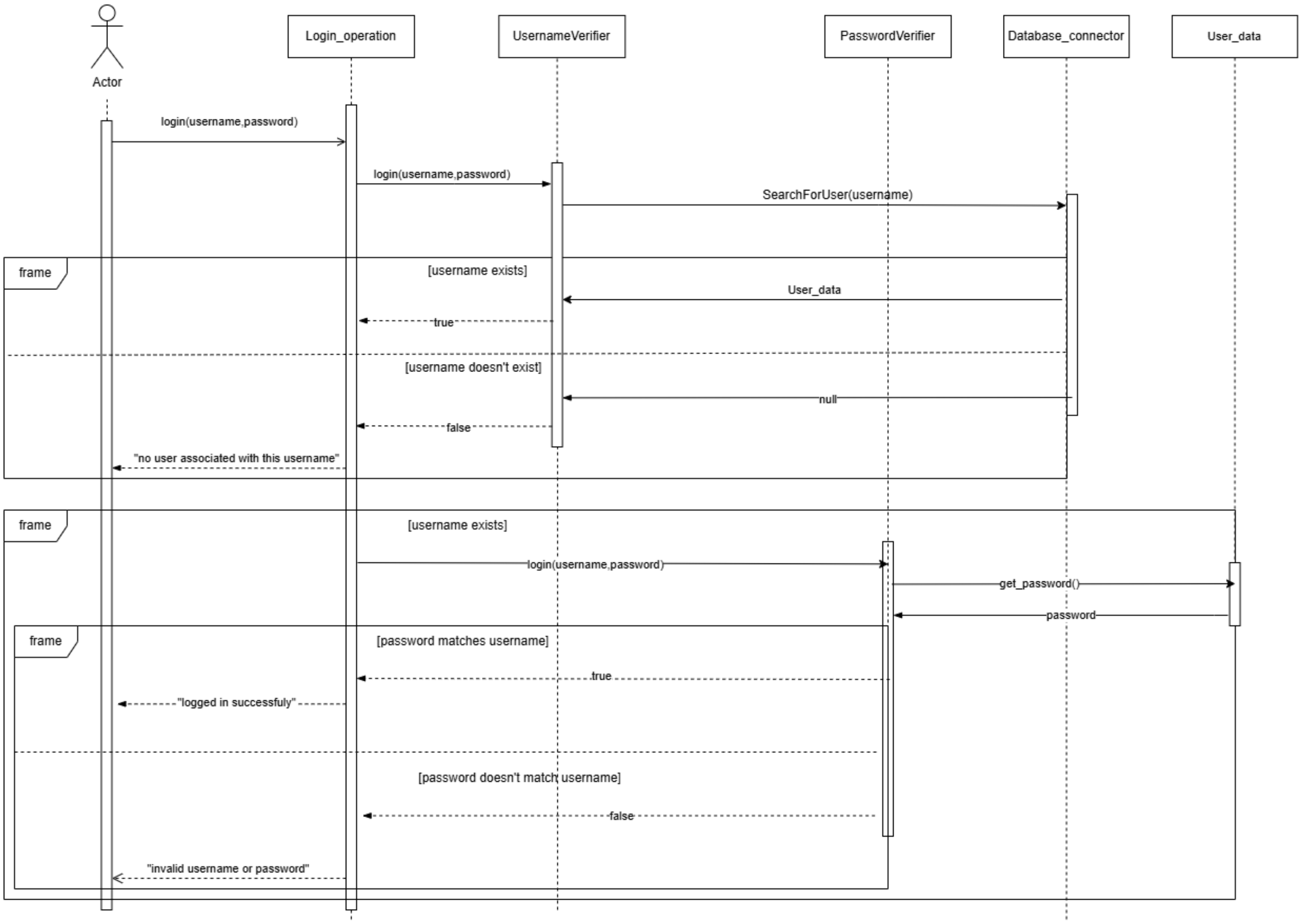
Software Design Specification

IV. Sequence diagrams

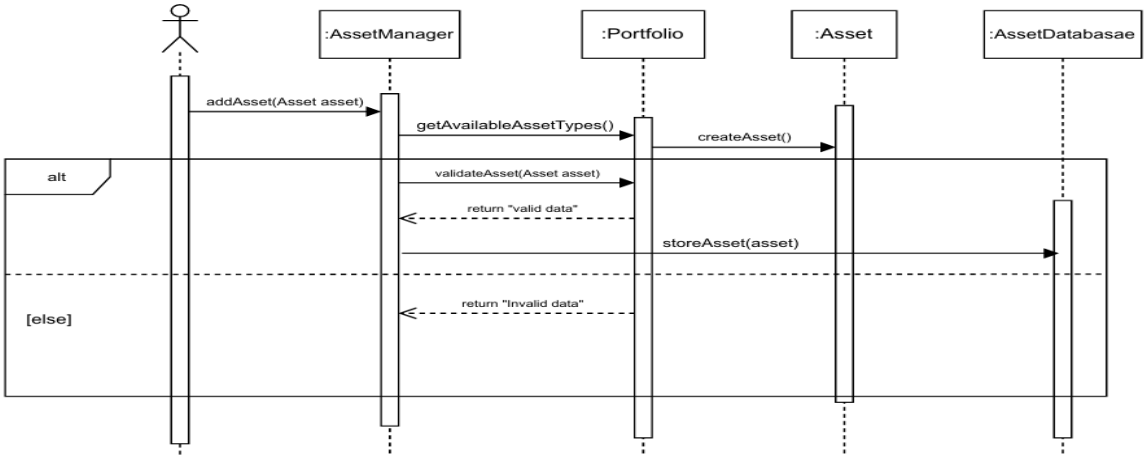
1. Sign Up (user story 1)



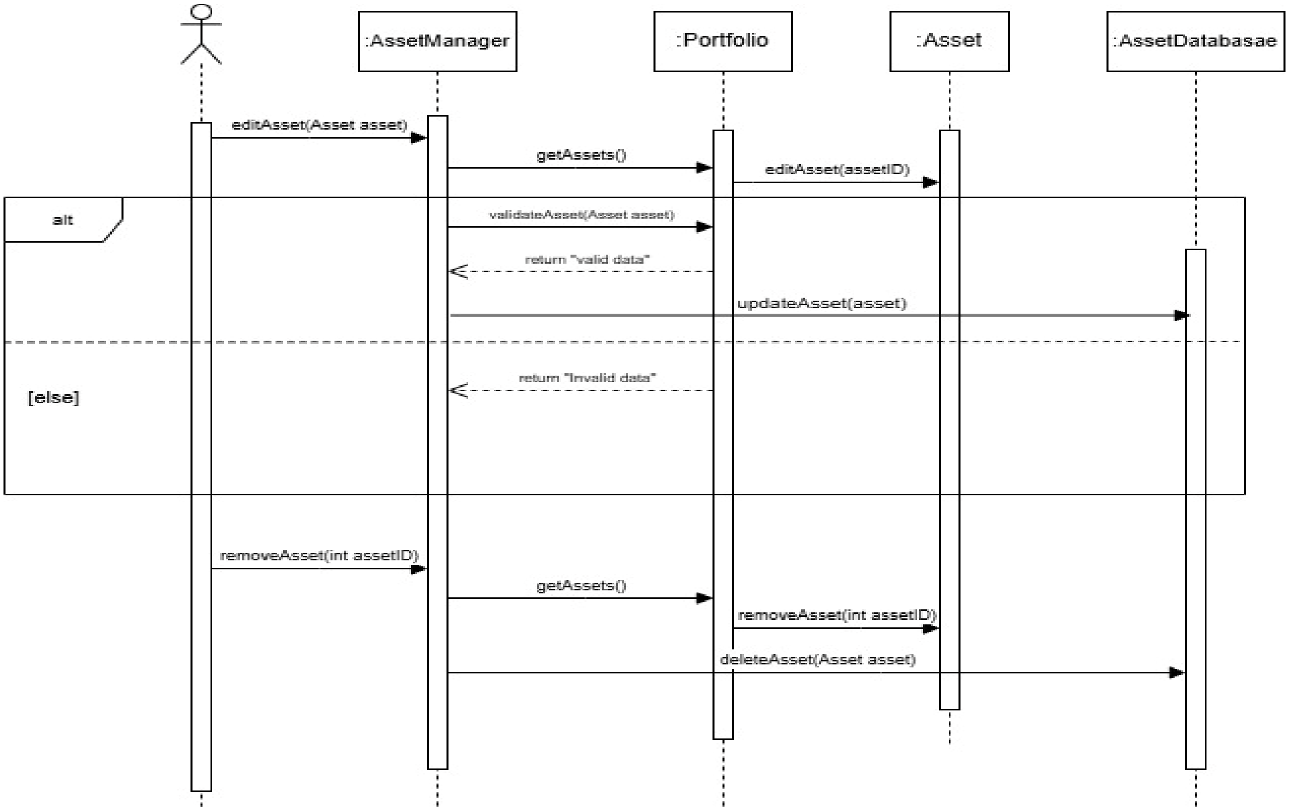
2. Log in (user story 2)



3. Add Asset (user story 3)

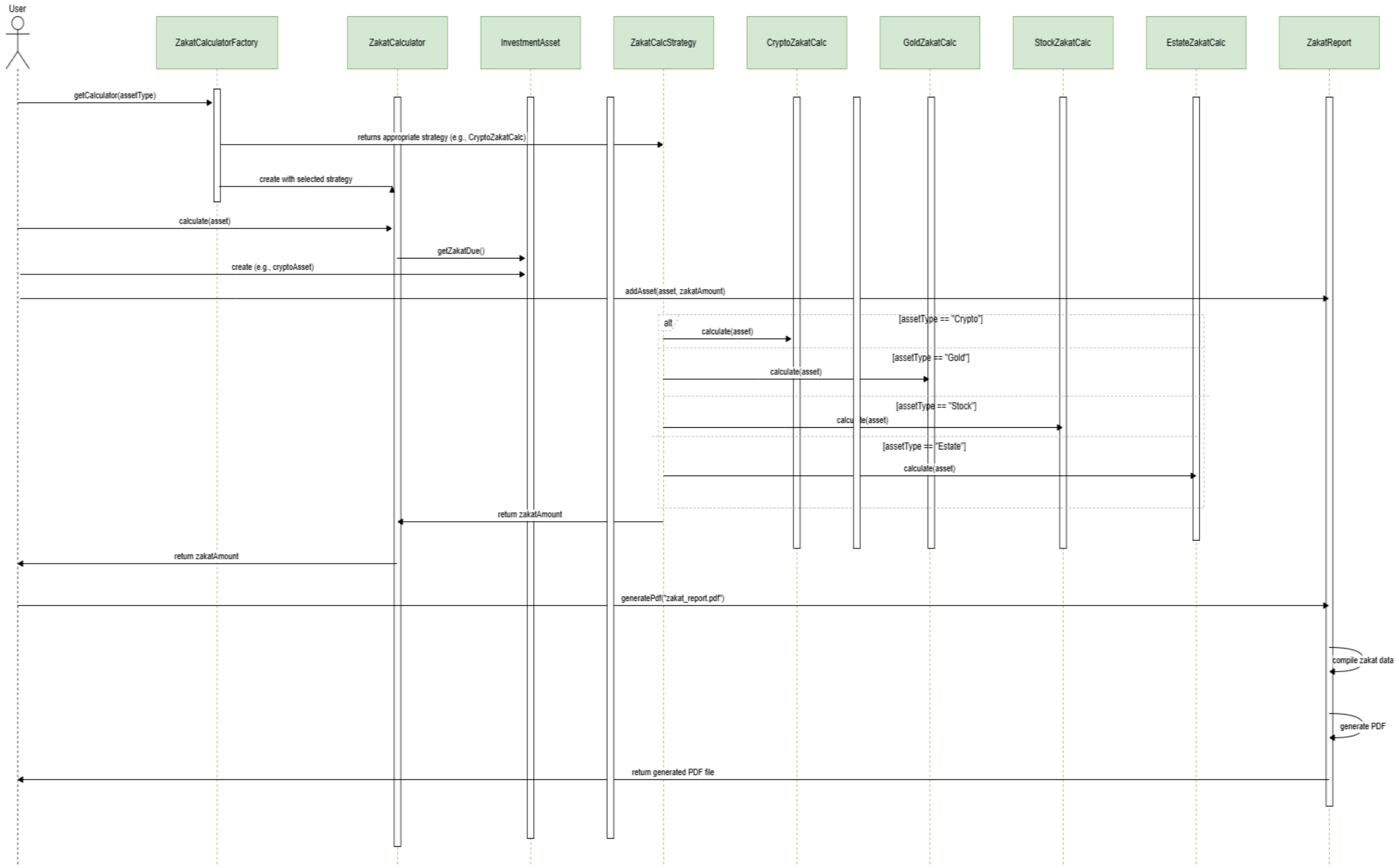


4. Edit/remove Asset (user story 4)

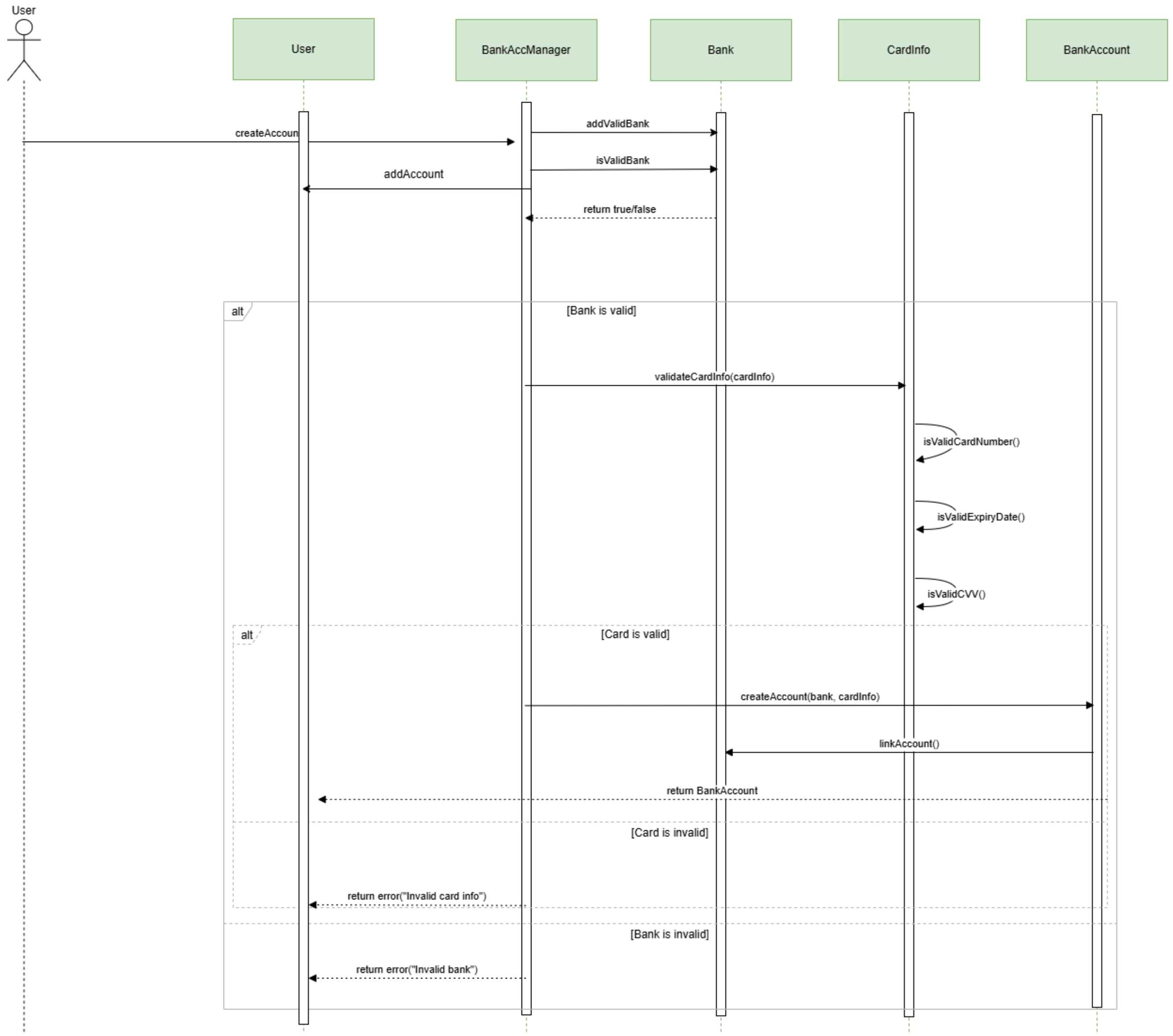


Software Design Specification

5. Zakat Calculator (user story 8)



6. Connect and Manage Bank account (user story 10)



Class - Sequence Usage Table

Sequence Diagram	Classes Used	All Methods Used
1. Log-In	Login_Operation	login(username, password)
	UsernameVerifier	login(username, password)
	PasswordVerifier	login(username, password)
	Database_connector	SearchForUser(username)
	User_data	get_password() → Returns stored password
2. Sign-up	Sign_up_operation	signup(name,username,email,password)
	NameValidator	Validate(user : User_data) -> return true if the name is in the correct format
	UsernameValidator	Validate(user : User_data) -> return true if the username is unique
	Database_connector	SearchForUser(username) ->return user_dta object if a user is found with the passed username, else null add_user(name, username, email, password
	EmailValidator	Validate(user : User_data) -> return true if the email is in the correct format
	PasswordValidator	Validate(user : User_data) -> return true if the password contains all the required characters
	User_data	User_data(name,username, email,password)
3. Add Asset	Portfolio	createAsset()

	Asset	getName()
		setName(String name)
		getQuantity()
		setQuantity(int quantity)
		getDate()
		setDate(Date date)
		getPrice()
		setPrice(double price)
		to get the Asset details

Sequence Diagram	Classes Used	All Methods Used
	AssetManager	getAvailableAssetTypes() -> make the user choose asset type from enum AssetTypes. addAsset(Asset asset) -> creates a new Asset object with given details. validateAsset(Asset asset) -> return true if asset info all valid and false if not (eg. : date is in the future)
	AssetDatabase	storeAsset(asset)
4. Edit Asset	Portfoilio	editAsset(int assetID) getAsset(int assetID) -> will get the Asset chosen by user

	Asset	setName(String name)
		setQuantity(int quantity)
		setDate(Date date)
		setPrice(double price)
		to get the new data

	AssetManager	editAsset(asset) validateAsset(Asset asset) -> validate the new updates
	AssetDatabase	updateAsset(asset) -> update the database with new data
5. Remove Asset	Portfolio	removeAsset(int assetID) getAsset(int assetID) -> will get the Asset chosen by user
	AssetManager	removeAsset(int assetID) -> Deletes the asset from the portfolio
	AssetDatabase	deleteAsset(asset) -> remove the selected asset from the database

6. Zakat-Calculation	ZakatCalculatorFactory	getCalculator(assetType)
		create(strategy)

	ZakatCalculator	calculate(asset)
		return zakat amount

Software Design Specification

Sequence Diagram	Classes Used	All Methods Used
	InvestmentAsset	getZakatDue()
	ZakatCalcStrategy	calculate(asset)
	CryptoZakatCalc	calculate(asset)
	GoldZakatCalc	calculate(asset)
	StockZakatCalc	calculate(asset)
	EstateZakatCalc	calculate(asset)
	ZakatReport	compile zakat data generate PDF generatePdf("zakat_report.pdf")

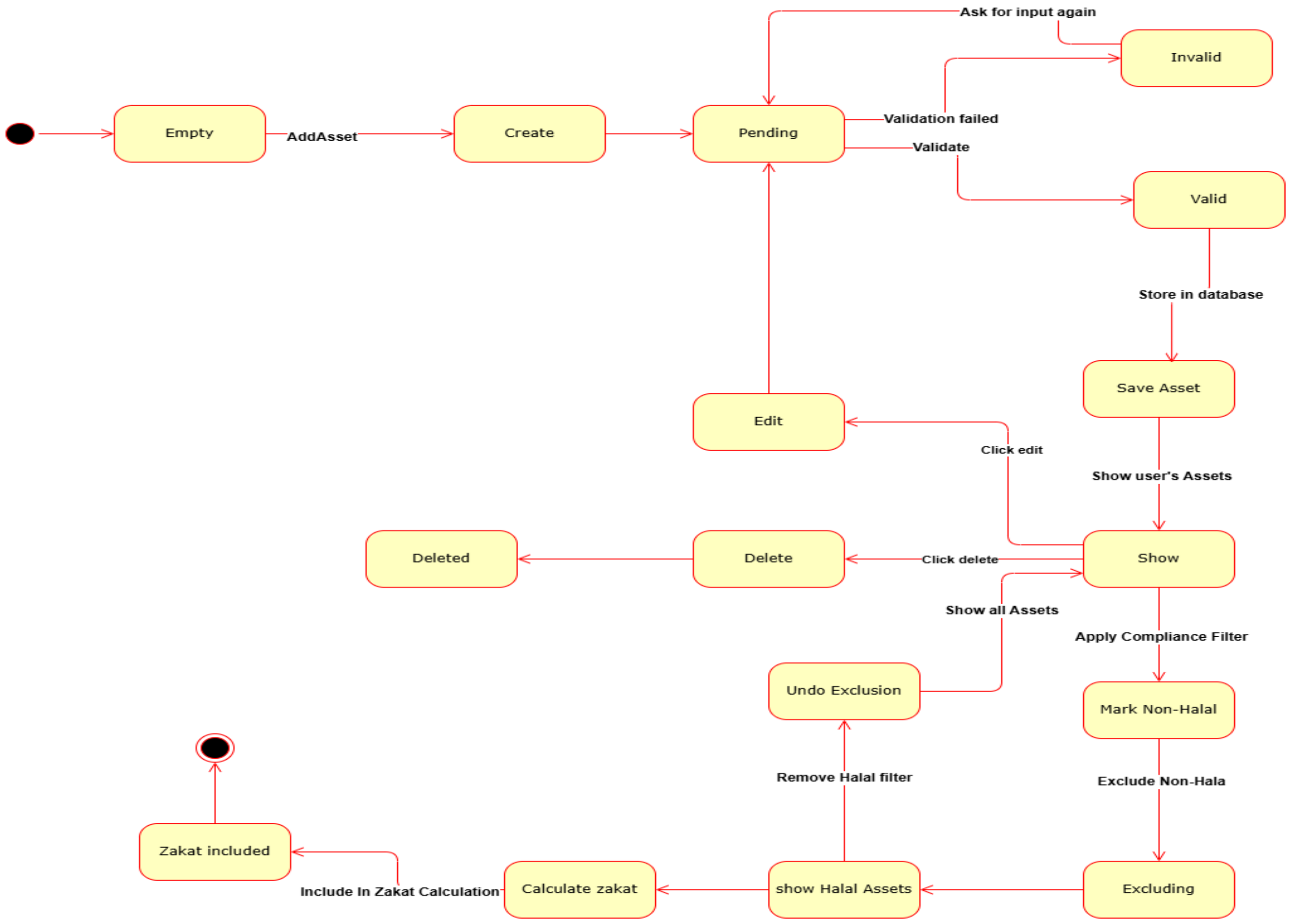
7. Connect-bank-account	BankAccManager	createAccount() -> return BankAccount
		validateCardInfo(cardInfo)
		isValidBank()

	Bank	isValidBank() -> return true if the bank is valid else return false

	CardInfo	isValidCardNumber()
		isValidExpiryDate()
		isValidCVV()

	User	addAccount()
	BankAccount	createAccount(bank, cardInfo)

V. State Diagram



VI. SOLID Principles

1. Single Responsibility Principle (SRP)

The Single Responsibility Principle (SRP) is applied in classes like User, BankAccount, and LoginService, where each class handles one clear responsibility. For the user class, the only responsibility is storing the users data. For the BankAccount class, it only does account management. And for the log in class, it only does login verification, where each verification is set apart into different classes for each class to have only one field to verify.

2. Open/Closed Principle (OCP)

The Open/Closed Principle (OCP) is evident in the ZakatCalculatorFactory and its strategy implementations (CryptoZakatCalc, GoldZakatCalc, etc.), which allow new zakat calculation types to be added without modifying existing code.

3. Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP) is applied where high-level modules like ZakatReport depend on abstractions (ZakatCalcStrategy interface), not concrete implementations, making the system flexible and easier to maintain.

VII. Design Patterns

1. Decorator Pattern

The Decorator Pattern is applied in the login verification process. The LoginVerifier class serves as the base decorator, while UsernameVerifier and PasswordVerifier act as concrete decorators that add specific checks. These decorators wrap the core Login_operation logic and execute their validation sequentially—first validating the username, then the password. . 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.

2. Strategy Pattern

The Strategy Pattern is used in two places: first, in the SignUpService interface to support the different sign-up validation parts (validating name, email, password, and username). By defining a common signup() method, the system can easily switch between sign-up validation algorithms without changing existing code , where Sign_up_operation selects and applies the appropriate validation dynamically. This makes it flexible and easy to add new sign-up requirement later. Second, the pattern is applied in zakat calculations, where the ZakatCalcStrategy interface defines a calculate() method implemented by concrete strategies like CryptoZakatCalc and GoldZakatCalc. This makes it easy to support new asset types and customize behavior without modifying existing logic.

3.Factory pattern

The Factory Design Pattern is applied through the ZakatCalculatorFactory class, which is responsible for selecting and returning the appropriate zakat calculation strategy based on the asset type provided by the user. This encapsulates the instantiation logic for the different strategy classes and abstracts it away from the rest of the system. Instead of directly instantiating strategy objects, the client code requests a calculator or strategy from the factory, which decides which concrete implementation to return. This approach reduces coupling between components, simplifies object creation, and makes it easier to manage changes or add new asset types without modifying existing client code.

4.Facade pattern

The Facade Pattern is used in this design through the BankAccManager class, which acts as a simplified interface to a more complex subsystem involving Bank, CardInfo, and BankAccount. Instead of requiring external classes (like User) to handle the creation and linking of bank accounts by directly interacting with multiple components, BankAccManager encapsulates this complexity in its linkAccount() method. This hides the internal details of how a BankAccount is created—such as combining a Bank object, a CardInfo instance, and setting the linking timestamp—and presents a clean, unified interface to clients, making it a clear example of the Facade Pattern.

5. Data Access Object Pattern

The DAO Pattern provides an abstraction layer between the business logic and the database. It encapsulates all the operations for accessing and manipulating data. It is used in the AssetDatabase class to handle all persistence-related operations such as saving, updating, and deleting assets from the database. This helps if the database technology changes, only AssetDatabase needs to be modified. Keeping the business logic in AssetManager clean and separated from low-level database details which make the program more dependable and stable.

6. Entity Pattern

The Entity Pattern is used in this design through the Asset and Portfolio classes, which represent core domain objects with a unique identity, well-defined attributes, and minimal internal behavior. These classes act as persistent data models that directly map to real-world concepts — such as an investment item (Asset) and an investor's collection (Portfolio).The Asset class encapsulates investment-related properties like name, quantity, purchase price, date, and asset type, and may include simple validation logic. Similarly, the Portfolio class maintains a list of assets and provides basic operations to add, update, or remove them. These entity classes are distinct from service or logic components like AssetManager, which performs higher-level operations. By isolating data and identity in well-structured objects, the system adheres to the Entity Pattern, facilitating clean separation between business logic, persistence, and domain modeling. This pattern ensures consistent state representation.

Tools

- Architecture diagram, UML diagram, Sequence diagrams, state diagram
  draw.io: https://app.diagrams.net/

Ownership Report

Item	Owners
Nourhan Mohammed Ahmed	• Description and audience • Sign up (user story 1) – Sequence diagram and sequence usage table – UML – Class description • Log in (user story 2) – Sequence diagram and sequence usage table – UML – Class description • Combined the work in the report • Solid principles • Part of design patterns
Nada Amin Fawzy	• Architecture diagram • Zakat calculation (user story 8) – Sequence diagram and sequence usage table – UML – Class description • Connect and manage bank account (user story 10) – Sequence diagram and sequence usage table – UML – Class description • Part of design patterns
Safia Mohammed Saeid	• Add Assets (user story 3) - Sequence diagram and sequence usage table - UML

	- Class description • Edit/remove Assets (user story 4) - Sequence diagram and sequence usage table - UML - Class description • State diagram • Part of design patterns