1. Project Overview

The **Seamless Banking System** is an online platform that addresses the challenges of traditional banking by enhancing accessibility, speed, and personalization for users. In today's digital economy, customers expect secure, seamless, and user-friendly banking services that can be accessed anytime, from any device. However, existing banking platforms often suffer from complicated interfaces, slow processing times, and limited customer support, particularly for small businesses and underserved populations.

To solve these problems, the Seamless Banking System will streamline core banking functionalities with an intuitive, secure, and accessible online interface. By implementing features such as real-time transactions, easy account management, and responsive customer support, the platform aims to bridge gaps in accessibility, improve user experience, and provide support through a responsive, scalable system.

Stakeholders and Their Stakes

- **Individual Customers**: Primary users who want a secure, efficient platform for daily banking activities like fund transfers, viewing account information, and applying for loans.
- **Bank Managers**: Oversee platform functionality, ensuring data security, regulatory compliance, and customer satisfaction.
- **Customer Support Team**: Responsible for handling inquiries and addressing issues raised by users, thereby enhancing customer satisfaction.
- **Developers and Administrators**: Maintain and upgrade the platform, monitor security, and manage system performance.
- **Financial Regulatory Bodies**: Interested in ensuring the platform adheres to security and regulatory standards to protect user data and prevent fraud.

How the Software Will Address the Problem

The Seamless Banking System will improve the banking experience by:

- Offering **24/7 access to banking services** from any device, reducing the need for in-person visits.
- Providing a **secure and efficient transaction system**, where users can transfer funds, deposit, withdraw, and check account statements in real-time.
- Implementing **responsive customer service** through a support ticketing system, allowing users to get assistance promptly.
- **Automating notifications** for user actions, account status updates, and loan tracking via Django-OTP, increasing transparency and user trust.

- Ensuring **compliance with financial regulations** to protect user data and maintain a secure banking environment.

---

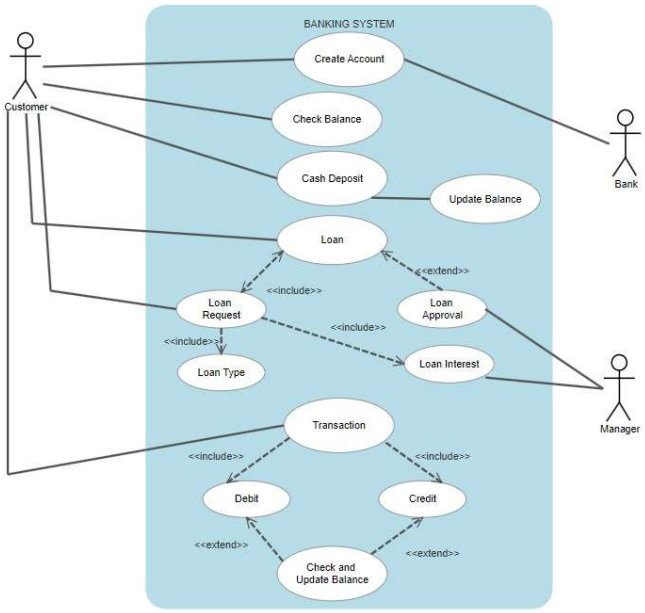
2. General Model

The General Model includes the **Context Diagram** and **Use-case Diagram** for a comprehensive understanding of system interactions and primary functionalities.

Context Diagram

The **Context Diagram** represents high-level interactions between the Seamless Banking System and its external entities.

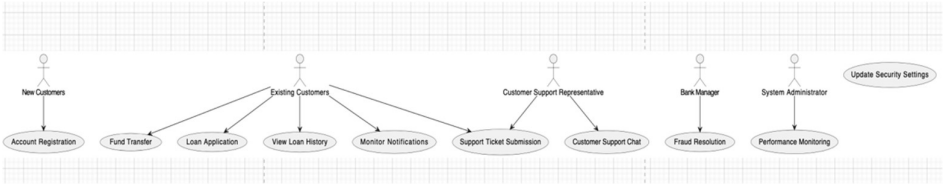
- **Actors**:
  - Customers (New/Existing Customers)
  - Bank Managers
  - Customer Support Team
  - External Banking Networks (for secure fund transfers)
- **Interactions**:
  - Customers interact with the system for account registration, transactions, loan processing, and support requests.
  - Bank Managers access the system for settings, compliance, and monitoring.
  - Customer Support handles user queries and support tickets.



Use-case Diagram

The **Use-case Diagram** shows the main functionalities accessible to users and managers in the Seamless Banking System.

- **Use Cases**:
    - **Create Account**: Register new accounts with the user personal information.
    - **Transactions**: Transfer funds, withdraw, and view bank statements.
    - **Loan**: Apply for loans, track loan status, and make repayments.
    - **Customer Support**: Raise and manage support tickets.
    - **Cash Deposit**: Users can deposit money into their accounts.



---

3. User Stories

1. **As a new user**, I want to register an account so that I can access banking services.

2. **As an existing user**, I want to view my account balance and transaction history to manage my finances.
3. **As a user**, I want to transfer funds to other accounts to pay bills or send money to friends.
4. **As a user**, I want to apply for a loan and track its status to manage my financial needs.
5. **As a user**, I want to submit support tickets to get assistance with issues.

---

4. Product Backlog

ID	User Story	Priority	Status
1	Register new user account	High	Completed
2	Log in as an existing customer	High	Completed
3	View account balance and transactions	High	Completed
4	Transfer funds between accounts	High	Completed
5	Apply for and track loans	Medium	Completed
6	Raise a support ticket	Medium	Completed
7	Receive notifications for transactions	Medium	To-Do
8	Retrieve Fraud transactions	High	Completed

---

5. Main Use Cases

- **Use Case: Account Registration**
  - **Actors**: New Customers
  - **Description**: Allows new Customers to register by providing personal information.
  - **Preconditions**: Access to the registration page.

    - **Postconditions**: New account is created, and login credentials are issued.

- **Use Case: Fund Transfer**
    - **Actors**: Existing Customers
    - **Description**: Enables Customers to transfer funds between accounts.
    - **Preconditions**: Customers is logged in and has sufficient balance.
    - **Postconditions**: Funds are transferred, and both sender and recipient receive confirmation.
- **Use Case: Loan Application**
    - **Actors**: Existing Customers
    - **Description**: Allows Customers to apply for a loan and track its status.
    - **Preconditions**: Customers is logged in and meets eligibility requirements.
    - **Postconditions**: Loan application submitted and status available for tracking.
- **Use Case: Support Ticket Submission**
    - **Actors**: Customers
    - **Description**: Allows users to submit inquiries or issues to customer support.
    - **Preconditions**: Customers is logged in.
    - **Postconditions**: Support ticket is generated and assigned to the support team.
- **Use Case: User Registration and Onboarding**
    - **Actors**: New Customer
    - **Description**: Allows new users to register, create accounts, and complete initial onboarding.
    - **Preconditions**: The user must not have an existing account.
    - **Postconditions**: User is registered, with account credentials set up for login and initial profile information completed.

 Use Case: Internal Fund Transfer (Within Bank)

- **Actors**: Existing Customer
- **Description**: Allows users to transfer funds to another Seamless Bank account.
- **Preconditions**: The user has sufficient balance and details of the recipient.
- **Postconditions**: Transfer completes, and both accounts are updated.

 Use Case: View Loan History and Statements

- **Actors**: Existing Customer
- **Description**: Enables users to view their loan history, repayment schedule, and loan statements.
- **Preconditions**: The user is logged in with an active or previous loan record.
- **Postconditions**: Loan history and statement are displayed.

 Use Case: Apply for Loan Pre-Approval

- **Actors**: Existing Customer
- **Description**: Allows users to submit a loan pre-approval request based on credit and income profile.
- **Preconditions**: User meets minimum eligibility requirements for pre-approval.
- **Postconditions**: The system analyzes the request and provides initial approval status.

 Use Case: Monitor Real-Time Account Notifications

- **Actors**: Existing Customer, System
- **Description**: Provides real-time alerts for account activities, including deposits, withdrawals, and failed login attempts.
- **Preconditions**: The user has notifications enabled.
- **Postconditions**: Notifications are sent instantly for monitored events.

 Use Case: Fraud Alert Resolution

- **Actors**: Existing Customer, Bank Manager, Support Representative
- **Description**: Enables bank staff to investigate and resolve flagged fraudulent transactions, allowing users to confirm or dispute flagged activities.
- **Preconditions**: A transaction has been flagged as suspicious.
- **Postconditions**: Fraud status is confirmed or dismissed, with corresponding actions.

 Use Case: Customer Support Chat

- **Actors**: Customer, Support Representative
- **Description**: Provides an option for live chat support to resolve account-related issues or answer queries in real time.
- **Preconditions**: The user is logged in and initiates the support chat.
- **Postconditions**: Support interaction is recorded and closed once the issue is resolved.

 Use Case: Transaction Rollback Request

- **Actors**: Customer, Bank Manager
- **Description**: Allows users to request a rollback or reversal on a transaction that was made in error.
- **Preconditions**: The transaction was recently made, and rollback is within allowed timeframe.

- **Postconditions**: Transaction is reviewed and potentially reversed by bank management.

 Use Case: Update Account Security Settings

- **Actors**: Existing Customer
- **Description**: Allows users to update security options like passwords, two-factor authentication, and recovery options.
- **Preconditions**: User is authenticated.
- **Postconditions**: Account security settings are updated to enhance protection.

 Use Case: Generate Tax Statements

- **Actors**: Customer
- **Description**: Provides a summary of account activity to assist users with tax filing, including interest earned, loan interest paid, and account summary.
- **Preconditions**: User has completed a full year of account activity.
- **Postconditions**: Tax statement is generated and available for download.

 Use Case: System Performance Monitoring

- **Actors**: System Administrator
- **Description**: Monitors system performance, uptime, and critical metrics to ensure reliable operation.
- **Preconditions**: System is active, and monitoring is enabled.
- **Postconditions**: Regular reports are generated, and alerts are issued for any performance issues.

 Use Case: Update Loan Repayment Plan

- **Actors**: Customer, Support Representative
- **Description**: Allows users to adjust loan repayment plans, either through refinancing or extending the repayment period.
- **Preconditions**: Customer has a valid loan and meets the conditions for plan change.
- **Postconditions**: Repayment plan is updated, and new terms are documented.

 Use Case: Configure Bank-Wide Notifications

- **Actors**: Bank Manager
- **Description**: Enables configuration of notifications for different account events such as low balance alerts, payment reminders, or policy updates.
- **Preconditions**: Manager is authenticated with required permissions.
- **Postconditions**: Notifications are configured and will be triggered as specified.

Detailed System Design

1. Customer Management Module. (USERID indicates IndividualCustomerID)

IdividualCustomer

- +userID: String
- +username: String
- -password: String
- +email: String
- #profilePicture: String
- #createdAt: Date

NewCustomer

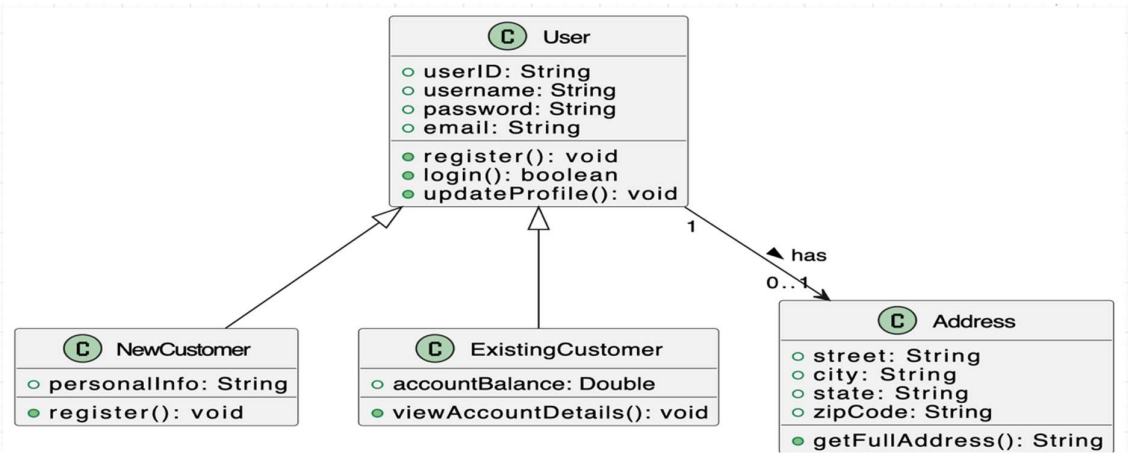
- +termsAccepted: boolean
- +registrationDate: Date

ExistingCustomer

- +accountBalance: float
- +accountType: String
- #loanHistory: List<Loan>

Address

- +street: String
- +city: String
- +state: String
- +zipCode: String
- #country: String



2. Transaction Processing Module

Transaction

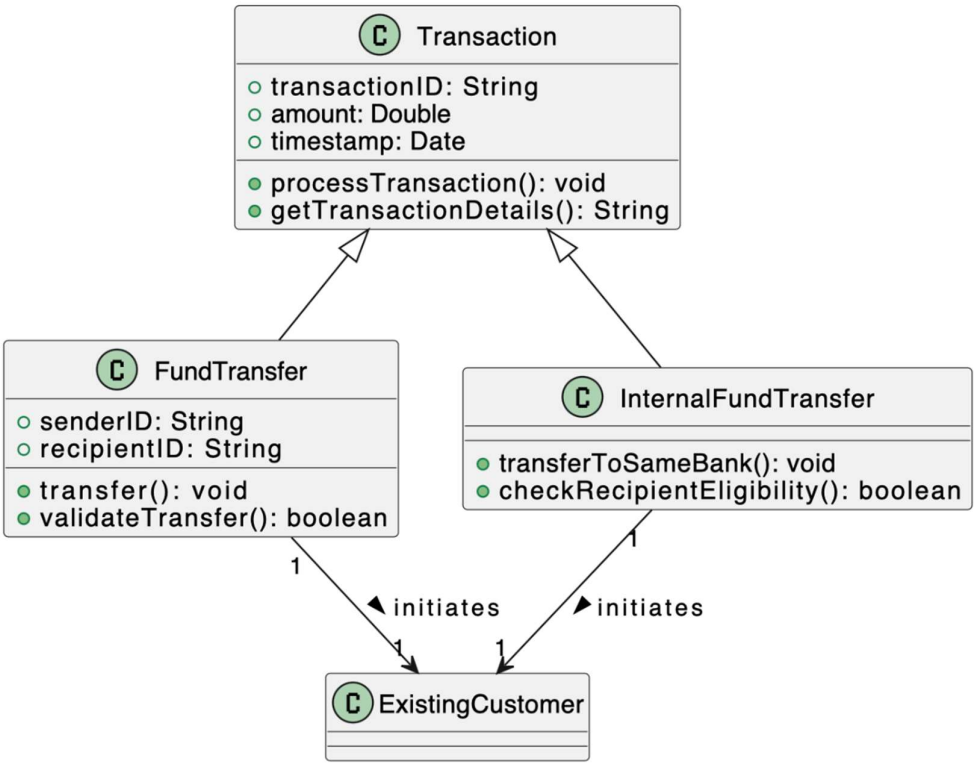
- +transactionID: String
- +amount: float
- +transactionDate: Date
- #transactionType: String
- #status: String

FundTransfer

- +recipientAccountID: String
- +transferDate: Date
- #transferFee: float

InternalFundTransfer

- +transferAmount: float
- +recipientBankID: String



3. Loan Management Module

Loan

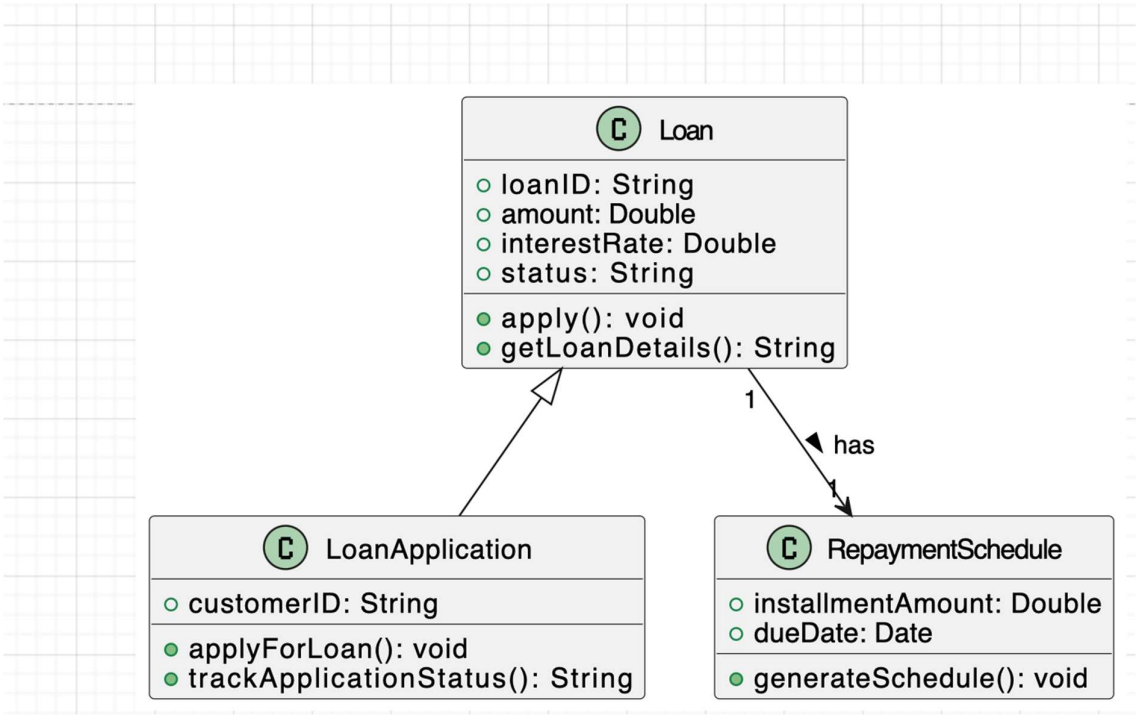
- +loanID: String
- +amount: float
- +interestRate: float
- +startDate: Date
- #endDate: Date
- #status: String

LoanApplication

- +applicationID: String
- +applicantID: String
- +requestedAmount: float
- +submissionDate: Date
- #status: String

RepaymentSchedule

- +repaymentID: String
- +dueDate: Date
- +amountDue: float
- #remainingBalance: float



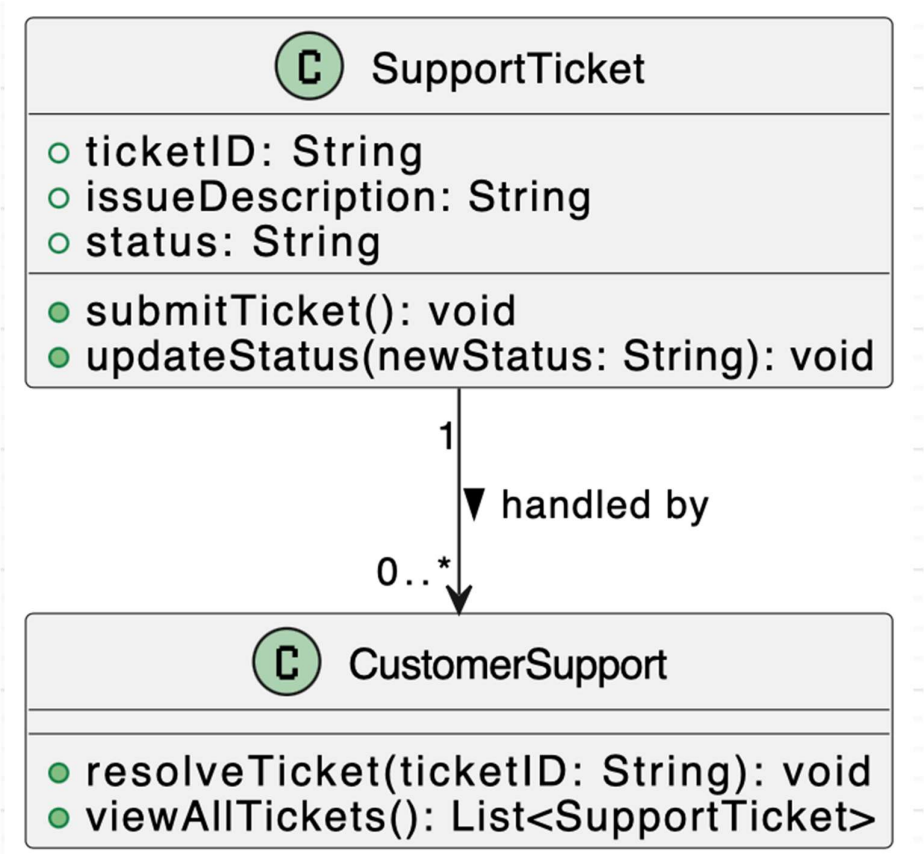
4. Customer Support Module

SupportTicket

- +ticketID: String
- +customerID: String
- +issueDescription: String
- +submissionDate: Date
- #status: String
- #resolutionDate: Date

CustomerSupport

- +supportID: String
- +responseTime: Date
- +resolvedTickets: List<SupportTicket>



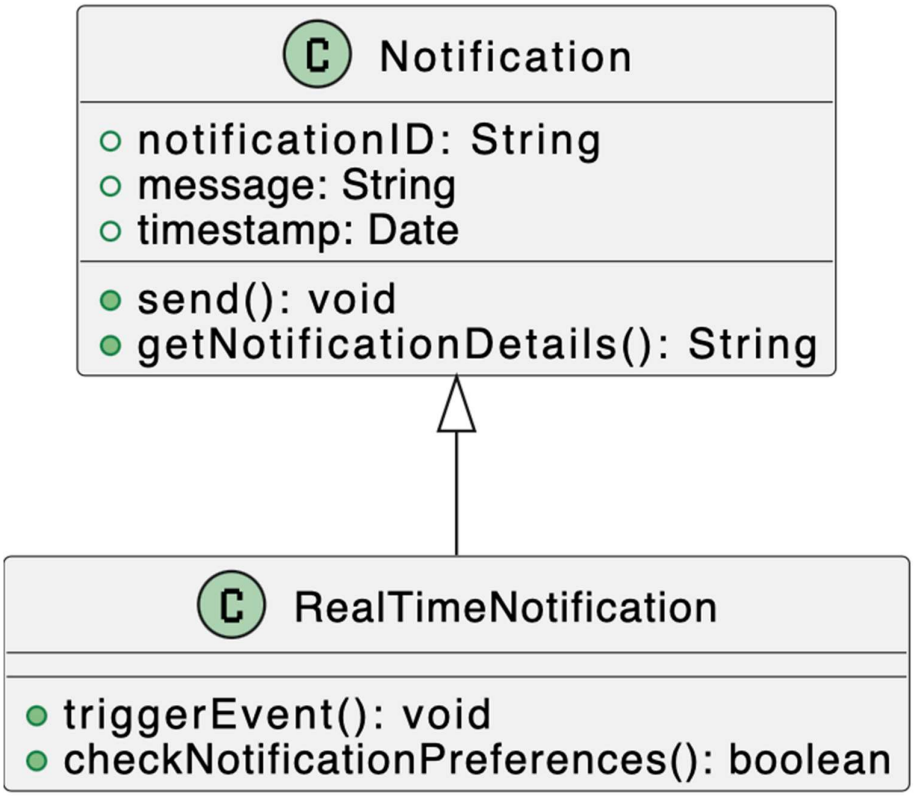
5. Notification System

Notification

- +notificationID: String
- +userID: String
- +message: String
- +notificationType: String
- #createdAt: Date

RealTimeNotification

- +eventType: String
- +timestamp: Date
- #isEnabled: boolean



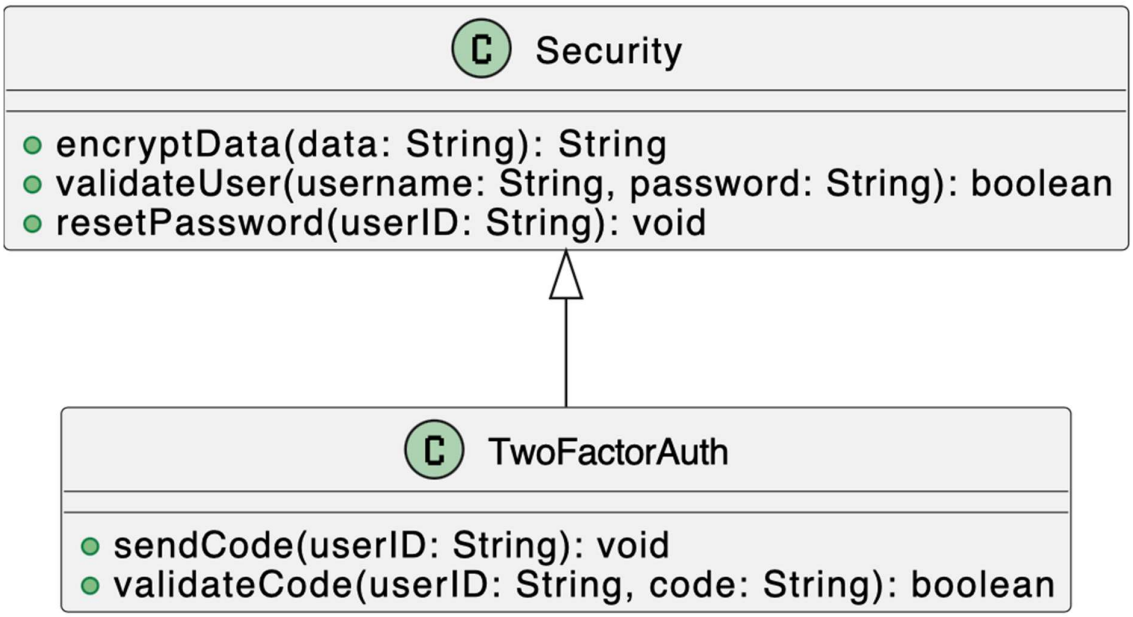
6. Security Module

Security

- +securityID: String
- +encryptionMethod: String
- #lastUpdated: Date

TwoFactorAuth

- +authID: String
- +userID: String
- +verificationCode: String
- #expirationTime: Date


Security
encryptData(data: String): String
validateUser(username: String, password: String): boolean
resetPassword(userID: String): void
TwoFactorAuth
sendCode(userID: String): void
validateCode(userID: String, code: String): boolean

2.1 Subsystem Architecture

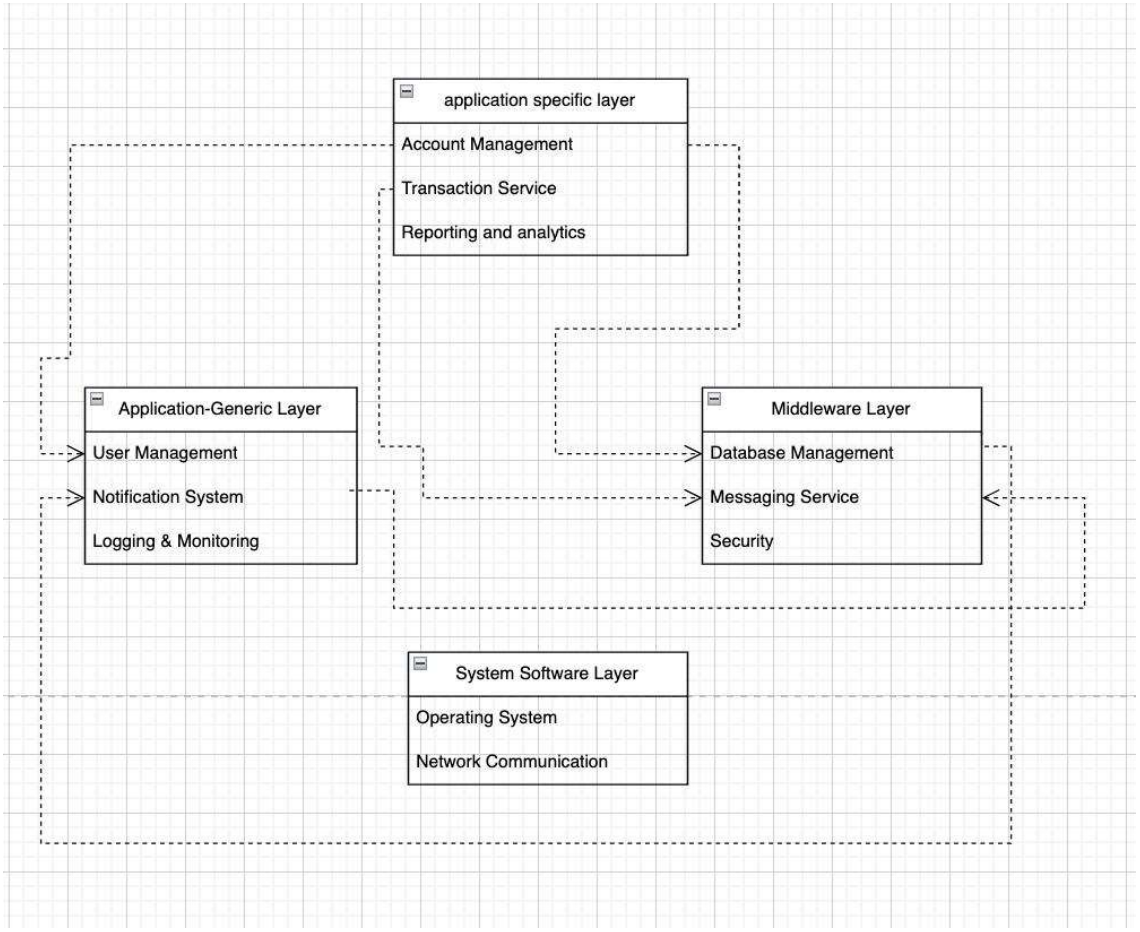
Application-Specific Layer:
This layer is responsible for the essential banking functions. The Account Management Package maintains customer accounts (creation, changes, and balance management) with Domain-Driven Design (DDD), which aligns software with banking business logic, ensuring flexibility and maintainability. The Transaction Service Package uses an Event-Driven Architecture (EDA) to manage asynchronous events such as transactions and balance updates, which improves system responsiveness. The Reporting & Analytics Package creates client reports and analytics using a Layered Architecture, which separates business logic from data processing for modularity and maintainability.Application-Specific Layer

Application-Generic Layer:
This layer offers reusable services. The User Management Package uses a Service-Oriented Architecture (SOA) to manage user authentication, registration, and access control, ensuring scalability and ease of connection with third-party services. The Notification System Package, which is in charge of sending user notifications, also employs EDA to trigger notifications in reaction to system events, which improves responsiveness. The Logging & Monitoring Package monitors system performance and logs actions, leveraging SOA to enable reusable, centralised monitoring of all system components.

Middleware Layer :
The middleware layer connects the application and system layers, handling data storage and communication. The Database Management Package uses a Layered Architecture to segregate database interactions from business logic, resulting in improved scalability and maintainability. The Messaging Service Package facilitates asynchronous communication between system components by decoupling services through SOA, allowing for scalability and fault tolerance. The Security Package manages system-wide authentication, encryption, and data protection, providing secure operations at all levels.

System Software Layer :
The System Software Layer provides critical low-level functions for the Seamless Banking System, assuring interface with hardware and network infrastructure. The Operating System Package handles resources such as memory, CPU, and file operations through a Layered Architecture that separates system management from business logic, assuring stability and security. The Network Communication Package facilitates safe data transfer while also providing features such as messaging and database access. This layer serves as the system's foundation, ensuring that higher-level software components run smoothly and consistently.

Each of these packages and architectural styles assures that the system is adaptable, scalable, safe, and maintained, meeting the essential needs of a unified banking environment.



2.2 Deployment Architecture

The seamless banking system uses a distributed, multi-layer architecture across different machines, enabling scalability, security, and fault tolerance. Here's an overview of each layer and its deployment:

User Layer: Users access the system via web or mobile apps, which communicate with backend servers using HTTPS for secure, encrypted connections over the Internet.

Network Security Layer: A firewall between the Internet and the internal network filters traffic, allowing only secure HTTPS connections. It connects to the internal load balancer over Ethernet.

Application Layer: A load balancer distributes incoming traffic across API Gateways and microservices, ensuring high availability and low latency. This layer uses HTTP/2 for efficient internal communication.

Backend Services Layer: API Gateway processes external requests, directing them to microservices like Account Service, Transaction Service, Notification Service, and Authentication Server. Each microservice is deployed separately and communicates over HTTP or gRPC for speed and efficiency.

Enterprise Service Layer: The Enterprise Service Bus (ESB) manages asynchronous messaging between microservices, using AMQP for reliable, event-driven communications.

Data Layer: Consists of a Primary Database (RDBMS), NoSQL Database, and Data Warehouse. They store structured and unstructured data, respectively, and communicate with backend services using JDBC or RESTful APIs.

External Systems Layer: Integrates third-party APIs (e.g., payment gateways, credit scoring services) over HTTPS, using JSON/XML for secure, standardized communication.

Communication Protocols: HTTPS secures external communications, gRPC provides fast microservice interactions, AMQP ensures reliable messaging through the ESB, and JDBC maintains secure database access. This layered deployment supports scalability and secure, seamless user experiences across devices.

2.3 Persistent Data Storage

1. Information to the database

- User information: Personal data such as name, date of birth, address, contact number.
- Account data: Account number, account type, balance.
- Authentication data: Credentials hash values, two-factor authentication.
- Transaction history: Unique transaction number, transaction time and other data.
- Login session data: Login attempts, time, IP addresses, session keys.
- Other miscellaneous data: Investments.

2. Database Schema

User table:

- User_id (INT)
- Name (VARCHAR)
- Date of birth (DATE)
- Email (VARCHAR)
- Phone (INT)
- Address (TEXT)
- Encrypted_pass (VARCHAR)
- Account_created_at (TIMESTAMP)
- Updated_at (TIMESTAMP)

Transaction table:

- Transaction_id (INT)
- Account_id (INT)
- Date_time (TIMESTAMP)
- Amount (FLOAT)
- Transaction_type (VARCHAR)
- Transaction_status (VARCHAR)
- After_balance (FLOAT)

Account table:

- Account_id (INT)
- User_id (INT)

- Account_number (VARCHAR)
- Account_type (VARCHAR)
- Balance (FLOAT)
- Status (TEXT)
- Created_at (TIMESTAMP)

Authentication table:

- Authentication_id (INT)
- User_id (INT)
- Login_attempts (INT)
- Last_login (TIMESTAMP)
- Two_factor_code (VARCHAR)

Session table:

- Session_id (INT)
- User_id (INT)
- Authentication_id (INT)
- Ip_user (VARCHAR)
- Device_info (VARCHAR)
- Session_start (TIMESTAMP)
- Session_end (TIMESTAMP)

2.4 Global Control Flow

It can be further explained that each related feature is according to the assumption of the way in which the system execution is controlled can be explained as follows:

1. Procedural or Event-Driven

* Assumption: The given system in nature is procedural or event-driven. Therefore, in event-driven systems, a system basically operates by waiting in a loop for events - such as the user or messages from the outside - and lets each user or event trigger activities that are unique in some order. This opens a flexible model where the interactions from each user or set of events create a different sequence of activities because of the order and context in which they are presented.

• Procedural: The system goes through some sort of set sequence in operations or steps for each of the users of the system. Each user has to go through the same sort

of processing every time this occurs, thereby allowing consistency but perhaps rigidity in flexibility.
* APPLICATION EXAMPLE Web-based application: the user interaction-application through things such as button clicks, form fill-ups, navigation-will be event-driven. Everything on the contrary, when there is a data processing pipeline that stage by stage is dealing with data, then, in essence, the procedural model will guide the fixed evolution through stages of data transformation.

2. TIME DEPEND

* Assumption: It may be real-time or non-real-time. If it is non-real time, it reacts to the events only without giving any strict timing consideration. Delays in processing just do not affect its functionality.
Real-time systems impose critical timing constraints on tasks' execution. In a real-time application, periodic operations may be there; it means the task would take place after an interval, and for each interval, it provides particular time.
* Applicative Example: Sensors in a control system may need to poll data in real time on a periodic basis at fixed intervals for the precision of data updates. On the opposite hand, a non-real-time system-a background job of data processing, for instance-does not have rigid timing and executes data when system resources are available.

3. Concurrency

* Assumption: Perhaps it is utilizing the system underlying this system, employing multiple threads to handle simultaneous operations; therefore, assume that some elements utilize underlying threads either for responsiveness or efficiency.
* Thread Synchronization: Threads are designed to synchronize for concurrency through mechanisms such as locks, semaphores, or message queues so that no conflict arises and data is shared safely. The common web server may run threads of many clients requesting service. These need to be synchronized to avoid data collisions.
Application Example: The request of every client for a multi-threading web server is actually served in a different thread, and it accepts multiple simultaneous connections. Synchronization mechanisms make sure that no thread accesses shared resources in parallel; therefore, data consistency and system stability can be achieved.

While defining those aspects of the flow of control, one gets a clear idea whether the system is designed for flexible interaction-event-driven-operates under strict time constraints-real time-or benefits from multithreading to improve parallel handling of tasks.

Static View Semantics

The static view of the Seamless Banking System is structured to provide a clear and coherent representation of the system's architecture, capturing the key components, modules, and their relationships. Here's an assessment of the static view in terms of correctness, completeness, and logical coherence:

1. Correctness of the Static Structure

- **Class Representation**: Each major component of the system is represented by appropriate classes, such as User, Transaction, Loan, and SupportTicket. Each class encapsulates attributes and methods relevant to its responsibility, which aligns with the object-oriented principles of encapsulation and modularity.
- **Attribute and Method Specification**: The attributes are defined with types and visibility, and methods are specified with parameters and return types, which accurately reflects the intended functionality of each class. For example, the Transaction class has methods for processing transactions, which logically follow from its attributes related to transaction details.

2. Level of Detail

- **Decomposition**: The decomposition of major modules into classes is appropriate. Each class is broken down into key attributes and operations that reflect a cohesive set of responsibilities. For instance, the User class includes attributes such as userID, name, and email, along with operations like register() and updateProfile(), which encapsulate the user management functionalities.
- **Granularity**: The level of detail is sufficient to understand the roles of each class without overwhelming complexity. For example, the Loan class has specific attributes for loan amount, interest rate, and repayment terms, which are crucial for the loan management functionalities.

3. Cohesion of Responsibilities

- **Responsibilities**: Each class exhibits high cohesion, meaning that the attributes and methods are closely related to the primary purpose of the class. For instance, the SupportTicket class is solely focused on functionalities related to customer support inquiries, such as creating, updating, and resolving support tickets. This ensures that the system adheres to the Single Responsibility Principle, which enhances maintainability.
- **Logical Grouping**: Classes are logically grouped based on their functionalities. For instance, all classes related to financial transactions (Transaction, Loan, FundTransfer) are organized together, making it easy to understand their interrelationships.

4. Relationships Between Classes

- **Associations and Multiplicity**: The relationships between classes are well-defined and reflect realistic interactions within the system. For example, the association between User and Transaction classes indicates that a user can initiate multiple transactions (1..* multiplicity), which is a logical representation of user behavior.
- **Aggregation and Composition**: Where necessary, aggregation and composition relationships are utilized to represent part-whole hierarchies. For instance, if a Loan is composed of multiple Transaction records (such as repayments), this relationship can be clearly articulated in the static view.

5. Overall Completeness and Reasonableness

- **Completeness**: The static view covers all major components of the system, providing a comprehensive understanding of the structure. All necessary classes have been included, and their attributes and methods are described adequately.
- **Reasonableness**: The design is reasonable as it provides a logical framework that supports the system's functionalities. The choices made in class relationships, responsibilities, and attributes are consistent with the requirements of a banking system.

Static View Quality Assessment

The static view of the Seamless Banking System is evaluated based on several quality metrics: identification and application of design patterns, loose coupling, high cohesion, and overall structural integrity. Here's an in-depth analysis:

1. Identification and Application of Design Patterns

- **Design Patterns Utilized**: The design leverages several well-established design patterns that enhance flexibility and maintainability:
    - **Factory Pattern**: This pattern can be applied in the creation of Transaction and Loan objects, allowing the system to handle different types of transactions or loans without altering the client code. This encapsulates the instantiation logic and promotes adherence to the Open/Closed Principle.
    - **Observer Pattern**: For real-time account notifications, the observer pattern can be employed, where Userinstances subscribe to account activity updates. This promotes a decoupled design, as the notification system can operate independently of the user interface.

    - **Singleton Pattern**: The DatabaseConnection class can use the singleton pattern to ensure that only one instance of the database connection exists, controlling access to shared resources effectively.
- **Justification of Patterns**: The use of these patterns is justified as they facilitate scalability and modifications in the future. For instance, if a new type of transaction needs to be added, the factory pattern allows for easy integration without affecting existing transaction processing code.

2. Loose Coupling

- **Decoupled Modules**: The design promotes loose coupling by ensuring that classes interact through well-defined interfaces rather than direct dependencies. For example, the User class does not directly manipulate the Transaction class but instead interacts through service methods that handle the business logic.
- **Dependency Inversion**: The static view adheres to the Dependency Inversion Principle, where high-level modules (like user services) are not dependent on low-level modules (like transaction processing). Instead, both depend on abstractions (interfaces). This makes the system more adaptable to changes, such as modifying the underlying transaction logic or swapping out database implementations.

3. High Cohesion

- **Cohesive Classes**: Each class within the design exhibits high cohesion. For example:
    - The User class contains methods specifically related to user management, such as register(), login(), and updateProfile(). All attributes and operations are closely related to the concept of a user.
    - The Loan class focuses solely on loan-related functionalities, including attributes like amount, interestRate, and methods for calculating payments and handling applications.
- **Single Responsibility Principle**: Each class adheres to the Single Responsibility Principle (SRP), which states that a class should have only one reason to change. This is evident in how classes are organized around specific functionalities, such as transaction processing, user management, and loan handling.

4. Overall Structural Integrity

- **Well-Defined Relationships**: The associations between classes are clear and logical. For example, a User can initiate multiple Transaction objects, and a Transaction may belong to a specific Loan. The multiplicity constraints are

accurately represented, which reflects real-world relationships in a banking context.

- **Maintainability and Scalability**: The quality design supports future enhancements. For instance, if additional functionalities (such as cryptocurrency transactions) need to be added, they can be integrated with minimal disruption to the existing architecture.

Justification of Class Decomposition and Module Design

The proposed design for the Seamless Banking System is based on a modular approach, allowing for a clear separation of responsibilities, promoting maintainability, scalability, and ease of understanding. The decomposition into specific classes under each module is justified as follows:

1. Customer Management Module

User Class: This class serves as a base for both new and existing customers. It encapsulates shared attributes such as userID, username, and password, ensuring that common functionalities are managed centrally.

NewCustomer Class: This class extends the User class and includes attributes specific to new customers, such as termsAccepted and registrationDate. This separation allows for distinct handling of new customers during the onboarding process.

ExistingCustomer Class: Similarly, this class extends the User class, focusing on attributes pertinent to existing customers, including accountBalance and loanHistory. This separation ensures that functionalities related to account management and transaction histories are distinct and organized.

Address Class: This class manages the address details of users. Keeping it separate enhances reusability and allows for easy modifications to address-related functionalities.

2. Transaction Processing Module

Transaction Class: This serves as the core class for managing all transaction-related attributes and methods. By encapsulating attributes like transactionID and amount, it provides a centralized place for handling all transaction operations.

FundTransfer Class: This class is specialized for fund transfers, extending the Transaction class. It includes additional details specific to fund transfers, such as recipientAccountID and transferFee, allowing for easy management of transfer-specific logic.

InternalFundTransfer Class: Similar to FundTransfer, this class handles transfers specifically within the bank. Its focus on internal transactions helps maintain a clear distinction between different transaction types.

3. Loan Management Module

Loan Class: This class encapsulates attributes related to loan details, such as loanID, amount, and interestRate. It acts as the foundation for loan management functionalities.
LoanApplication Class: It focuses on handling loan applications, separating the process of loan requests from loan management. This encapsulation allows for streamlined application processes and tracking.
RepaymentSchedule Class: This class is dedicated to managing loan repayment schedules, enhancing the organization of repayment-related logic and history.

4. Customer Support Module
SupportTicket Class: It encapsulates the details related to customer support inquiries. By having this dedicated class, we ensure that support-related functionalities are managed independently and efficiently.
CustomerSupport Class: This class manages customer support operations, including tracking resolved tickets and response times, thus centralizing customer service management.

5. Notification System
Notification Class: This class centralizes the management of user notifications, ensuring that all notifications are handled in one place, facilitating easy modifications and enhancements.
RealTimeNotification Class: This class specializes in real-time notifications, allowing for a clear distinction between standard notifications and those requiring immediate user attention.

6. Security Module
Security Class: This class focuses on managing security attributes like encryptionMethod and lastUpdated, ensuring that security considerations are encapsulated.
TwoFactorAuth Class: It specializes in handling two-factor authentication, providing a dedicated structure for verification processes.

Alternative Designs
One alternative design could have been to combine related classes into fewer modules, for example, merging the Loanand Transaction classes into a single financial operations module. However, this would increase complexity and reduce clarity in the system, making it harder to maintain and scale.
By keeping modules and classes distinct, we enhance clarity and maintainability. Each class has a single responsibility, adhering to the Single Responsibility Principle of object-oriented design, making it easier to test, debug, and extend.
Design Patterns Used

1. **Factory Pattern**: This pattern could be employed in the User Management Module to create instances of NewCustomer or ExistingCustomer. The factory pattern encapsulates the instantiation logic and abstracts the creation process, making it easier to manage user types without modifying existing code.

2. **Singleton Pattern**: This pattern might be applied to the Notification system to ensure that only one instance of notification management exists, preventing redundant notification instances and ensuring consistent behavior across the application.
3. **Observer Pattern**: This could be used in the Notification System to notify users of changes in their account status or transactions. By having observers (users) listen to events from the subject (transaction or account), we can decouple the components and allow for easy extensibility of notification types.

Conclusion

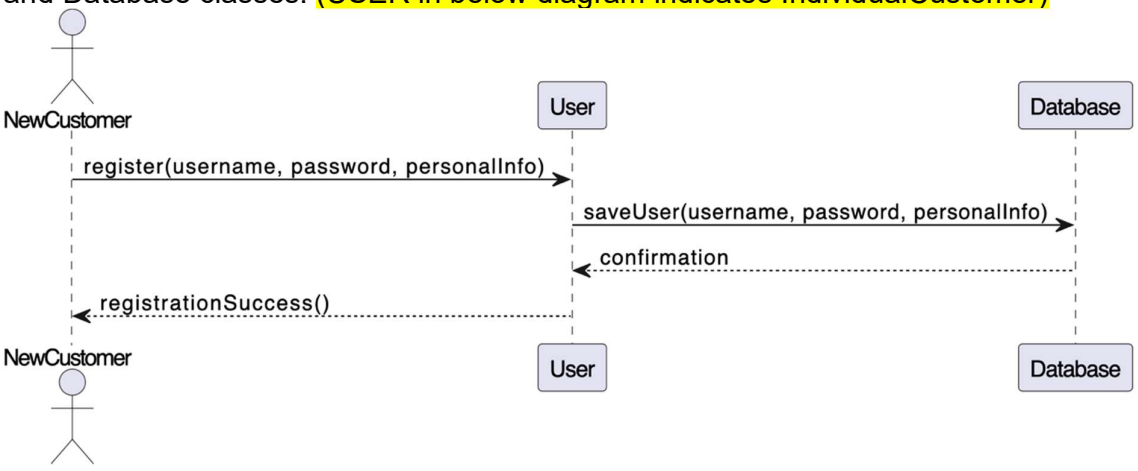
The proposed design emphasizes modularity, clarity, and maintainability. By decomposing the system into well-defined classes, we can manage responsibilities effectively, making future modifications easier and enhancing overall system robustness. The selected design patterns further optimize system functionality and scalability, providing a solid foundation for future enhancements and development.

Dynamic View

The dynamic view of the Seamless Banking System focuses on the system's behavior through UML sequence diagrams. These diagrams illustrate the interactions between classes over time to support specific functions of the system. Below, I present several key sequence diagrams that correspond to critical use cases from the system.

1. Sequence Diagram for Account Registration

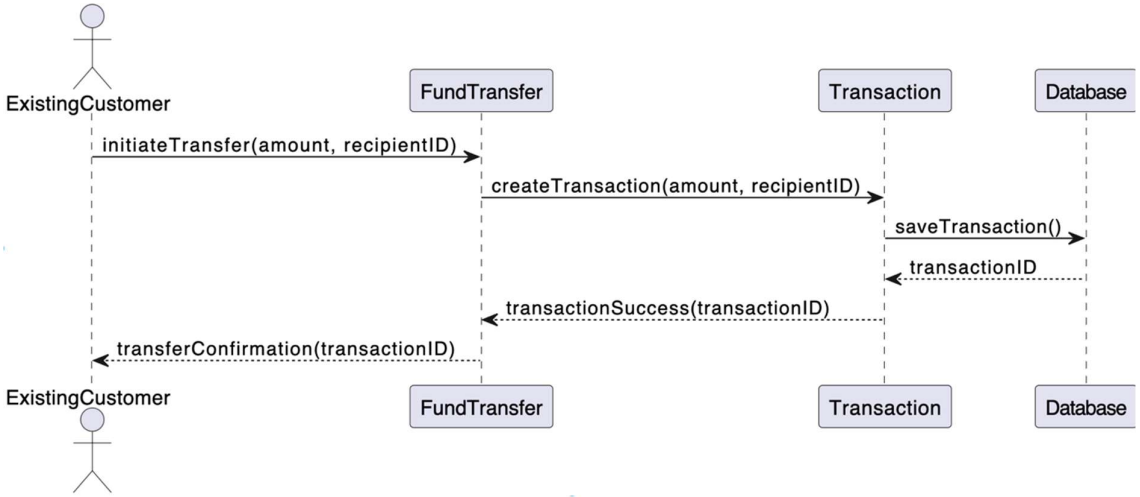
This sequence diagram illustrates the process of a new customer registering for an account. The interaction involves the NewCustomer, IndividualCustomer , and Database classes. (USER in below diagram indicates IndividualCustomer)



Explanation:

- The NewCustomer initiates the registration by invoking the register method on the User.
- The IndividualCustomer then interacts with the Database to save the user information.
- Upon successful saving, the database sends a confirmation back to the User, which then communicates a success message to the NewCustomer.

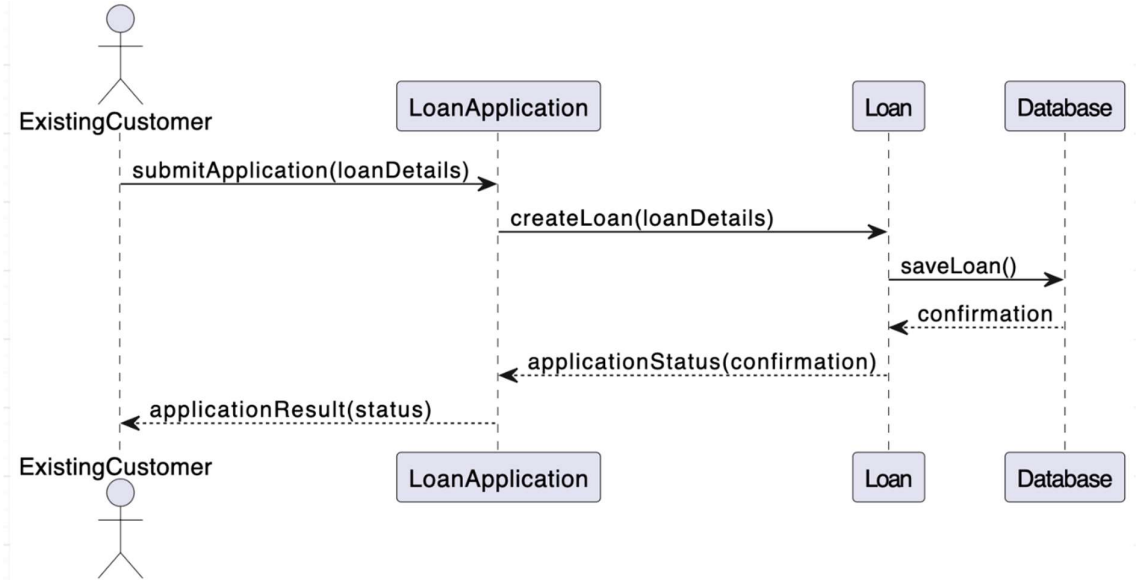
2. Sequence Diagram for Fund Transfer
This sequence diagram depicts the steps involved when an existing customer performs a fund transfer.



Explanation:

- The ExistingCustomer requests a fund transfer through FundTransfer.
- A Transaction instance is created with the transfer details, which is then saved to the Database.
- Once the transaction is saved, a confirmation is sent back through the chain, ultimately reaching the ExistingCustomer.

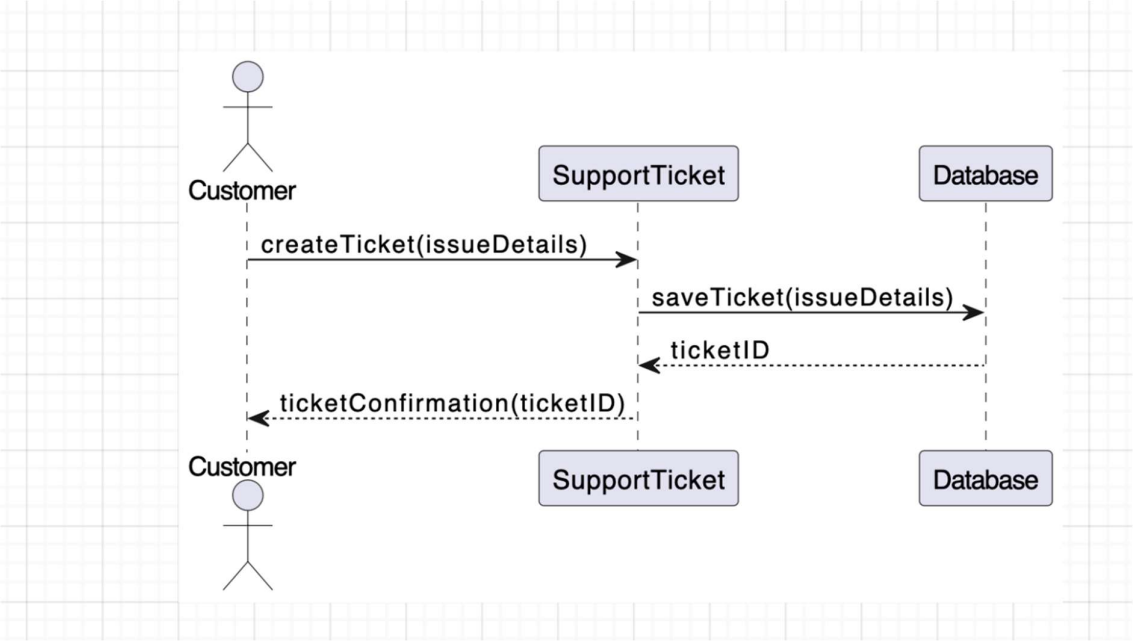
3. Sequence Diagram for Loan Application
This diagram shows the interactions when an existing customer applies for a loan.



Explanation:

- The ExistingCustomer submits a loan application through LoanApplication.
- The LoanApplication creates a new loan instance and saves it in the Database.
- After saving, the loan confirmation is sent back to the customer via LoanApplication.

4. Sequence Diagram for Support Ticket Submission
This sequence diagram outlines the steps involved when a customer submits a support ticket.



Test Plans and Sprint Reviews

Test Plans
- **Objective**: Validate the core functionalities of the Seamless Banking System, including user registration, fund transfers, loan applications, and support ticket submissions.
- **Testing Methods**: Unit testing for individual components, integration testing for interactions between components, and user acceptance testing for overall system functionality.

Previous Sprints
- **Sprint 1**: Focused on setting up the initial architecture and class diagrams. Achievements included completing the design of core classes related to user management and transaction processing.

- **Sprint 2**: Implemented basic functionalities for account registration and fund transfer. Initial testing was conducted to ensure that these functionalities worked as intended.

Current Sprint

- **Sprint 3**: Aiming to finalize the loan management functionalities and customer support system. Testing will involve user scenarios to validate the registration, loan application, and support ticket processes.
- **Sprint 4**: Deployment

Brief Sprint Review

- **Sprint Review for Sprint 2**: The team successfully implemented account registration and fund transfer features. TA feedback was positive, indicating ease of use. Minor bugs were identified in the fund transfer process, which will be addressed in the upcoming sprint. And also additional features like notifications and retive fraud transactions