Specification Document

for Banking System

Team 5 Project 2

Made by : Yang Runkang

* Table of Contents
* Introduction
* Purpose
* Scope
* System Overview
* How to Access the System
* Overview of the User Interface
* ATM Operations
* App Operations
* API Specification
* Error handling
* Overview
* Common Issues and Solutions

Introduction

**Purpose**

The purpose of this document is to provide a comprehensive specification for the development of a Python-based banking system. This system aims to integrate both ATM (Automated Teller Machine) and mobile app functionalities to offer users a seamless and efficient banking experience. The document details the system architecture, key features, API specifications, data models, security considerations, and testing and deployment strategies. The primary audience of this document includes software developers, system architects, testers, and system administrators involved in the banking system's development, testing, and deployment processes.

**Scope**

This specification document encompasses the specification phase of the banking system, including its design and implementation process. The document covers the following areas:

**System Architecture:** A detailed overview of the system's architecture, including the interactions between different components.

Data Models: Description of the data models used in the system, including their structure and relationships.

**API Specifications:** Comprehensive details of the available APIs for user operations and system events, covering both ATM and mobile app functionalities.

**Security Considerations:** Measures and best practices implemented to ensure the security of user data and transactions.

**Error Handling:** Strategies and mechanisms to manage and report errors effectively.

This specification encompasses the complete development details of the banking system, including:

* User interface design and interactions, using PyQt5.
* Backend implementation using Python, with data stored not in a traditional SQL database but directly within variables in a Python file named Database.py.
* System functionalities provided through a series of API calls and responses.

Definitions, Acronyms, and Abbreviations

API (Application Programming Interface): A set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other services.

ATM (Automated Teller Machine): An electronic banking outlet that allows customers to complete basic transactions without the aid of a branch representative or teller.

Acount: A unique identifier assigned to each account holder.

APP (Application): In this context, refers to the mobile banking application.

References

PyQt5 Documentation: <https://riverbankcomputing.com/software/pyqt/intro>

Python Documentation: <https://docs.python.org/3/>

System Overview

System Architecture

The banking system is designed to operate seamlessly across two main interfaces: ATM and mobile applications, facilitated by PyQt5 for user interactions. Here is a more detailed breakdown of the architecture:

Frontend

* UI Design: Interfaces are crafted using Qt Designer, where .ui files are generated for various components such as login screens, main transaction windows, and query displays. Specific UI files include applog.ui, appmain.ui, ATMlog.ui, ATMmain.ui, and query.ui.
* UI Conversion: The .ui files are converted into Python modules using pyuic5. These modules, such as applog\_UI.py, appmain\_UI.py, ATMlog\_UI.py, ATMmain\_UI.py, and query\_UI.py, serve as the frontend codebase, interfacing directly with the backend scripts.
* Interaction Logic: Event-driven programming is used to handle user inputs, with PyQt signals and slots managing interactions like button clicks, form submissions, and data entry.

Backend

* Script Organization: Backend functionality is spread across multiple Python scripts in the Code directory. Critical scripts include:
* app.py and ATM.py: Handle login and main interface operations for the app and ATM respectively.
* Database.py: Manages all data regarding accounts, balances, and transaction histories using Python dictionaries instead of a SQL database.
* query.py: Used by both the app and ATM to display transaction histories based on user requests.
* Data Handling: Data is stored in memory with structures such as:
* account\_balance: Dictionary linking accounts to their current balances.
* account\_password: Dictionary storing account-password pairs for authentication.
* detail: Dictionary storing detailed transaction records for each account.
* Communication: The NetClient.py script manages network communications for operations that involve multiple instances of the application, ensuring data consistency and synchronization across sessions.

Key Features

Detailed functionalities provided by the system include:

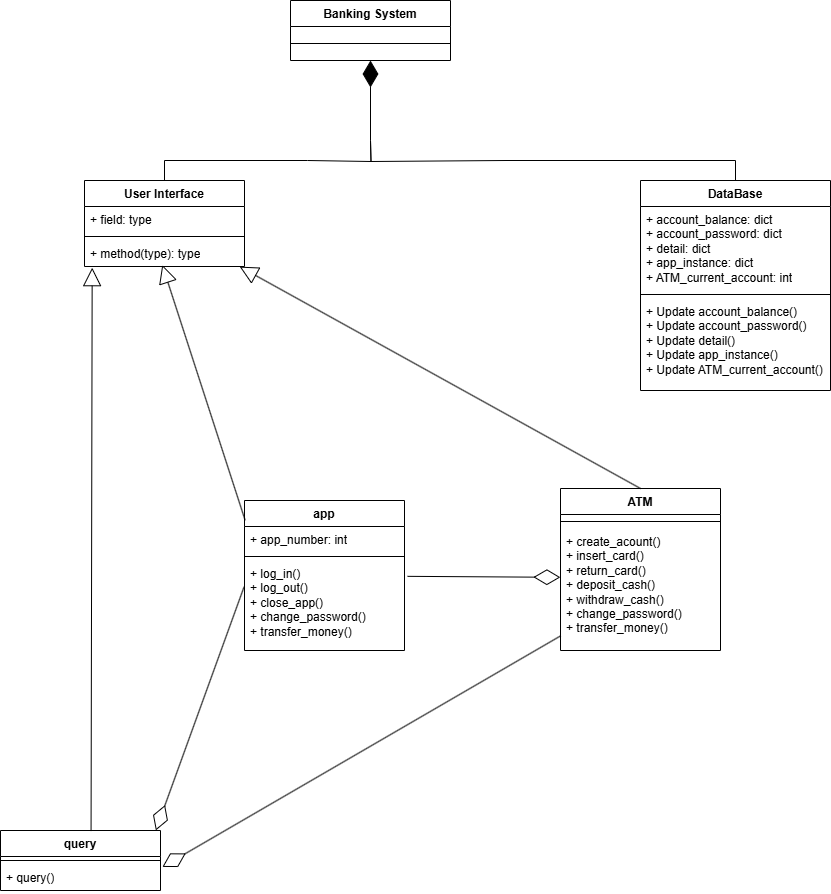
* **Account Operations:** Users can perform operations such as account creation (create\_account@password), account closure, and authentication without traditional IDs and PINs, focusing solely on account numbers and passwords.
* **Financial Transactions:** Includes cash deposits, withdrawals, and inter-account transfers. These are managed through commands like deposit\_cash@num for ATM deposits and transfer\_money@receiver\_id@num for app-based transfers.
* **Security and Authentication:** The system employs basic security measures focusing on password management and validation during the login process to prevent unauthorized access.

Data module

Use Cases

Use cases for system functionality:

Class diagram of the system architecture

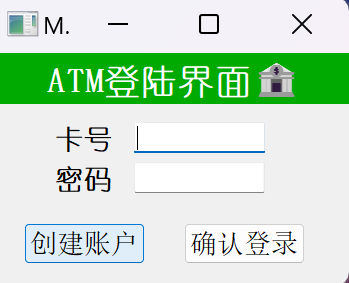


API Specifications

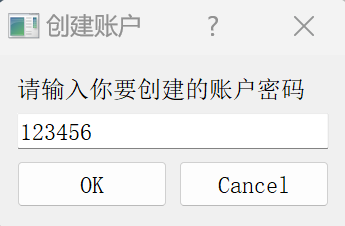
This section outlines the specific APIs available for user operations and system events, detailing how each API interacts with the system's backend and frontend components.

User Operations

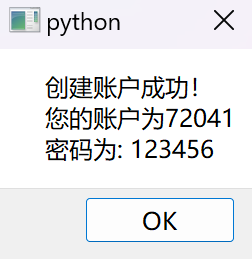
* ON ATM
* **S1.1 Create Account (create\_account@password)**

****

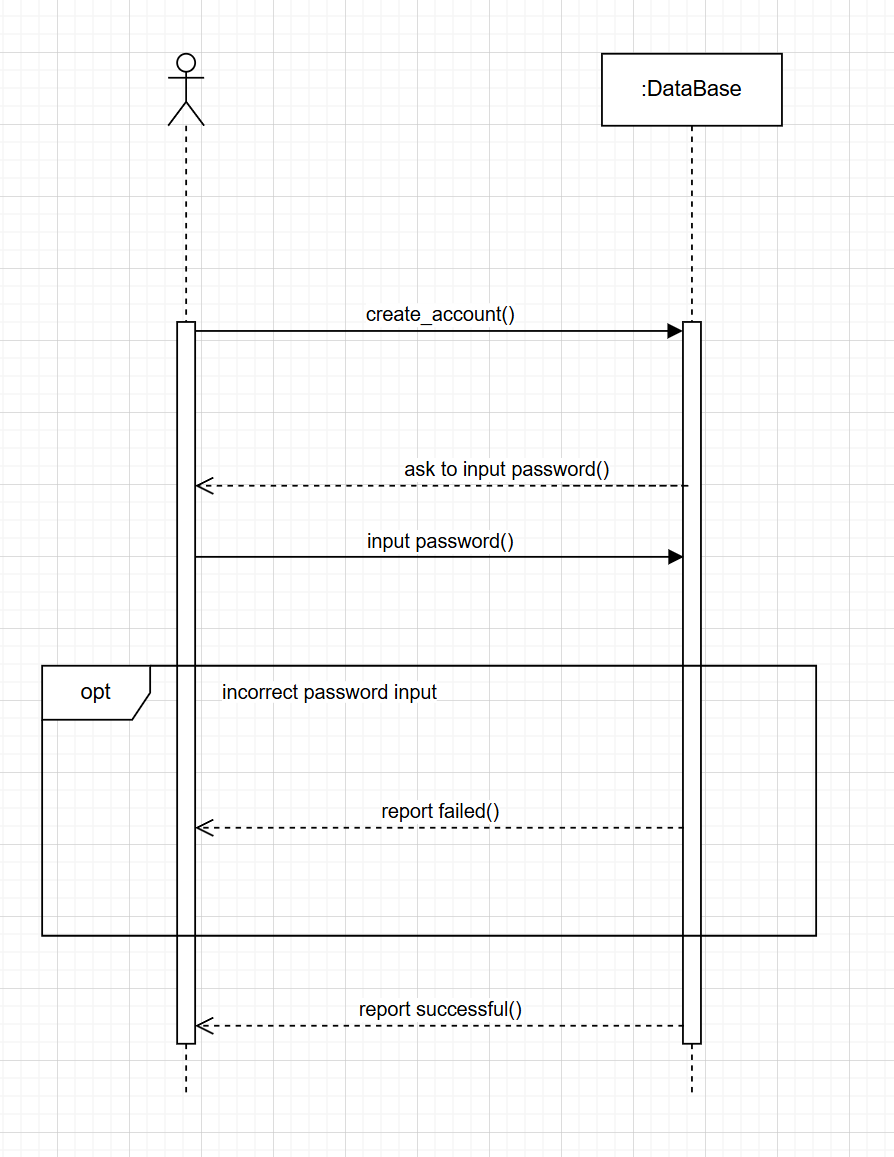
* **Frontend:** The user selects the "Create Account" option and inputs a password.



* **Backend:**
* Calls generate\_random\_number from Database.py to generate a unique account number.
* Stores the new account number and password in account\_password.
* Initializes the account balance in account\_balance with $0.
* Triggers the event account\_created@id.
* **Data Flow:** Input -> Generate Account -> Store Data -> Trigger Event.
* **System Response:** The ATM will display account\_created@id, this means you have been assigned a new card, and the ATM will display the new card number and password.



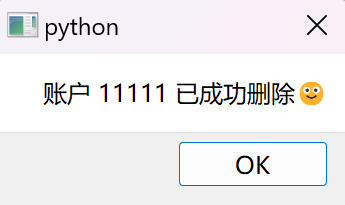
* **Failure Case:** If account generation fails (e.g., random number collision), triggers failed@create\_account.



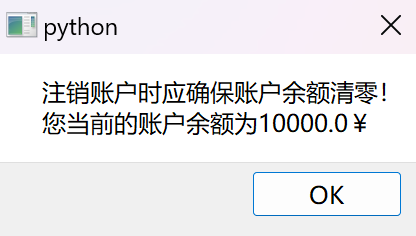
* **S1.2 Close Account (close\_account)**

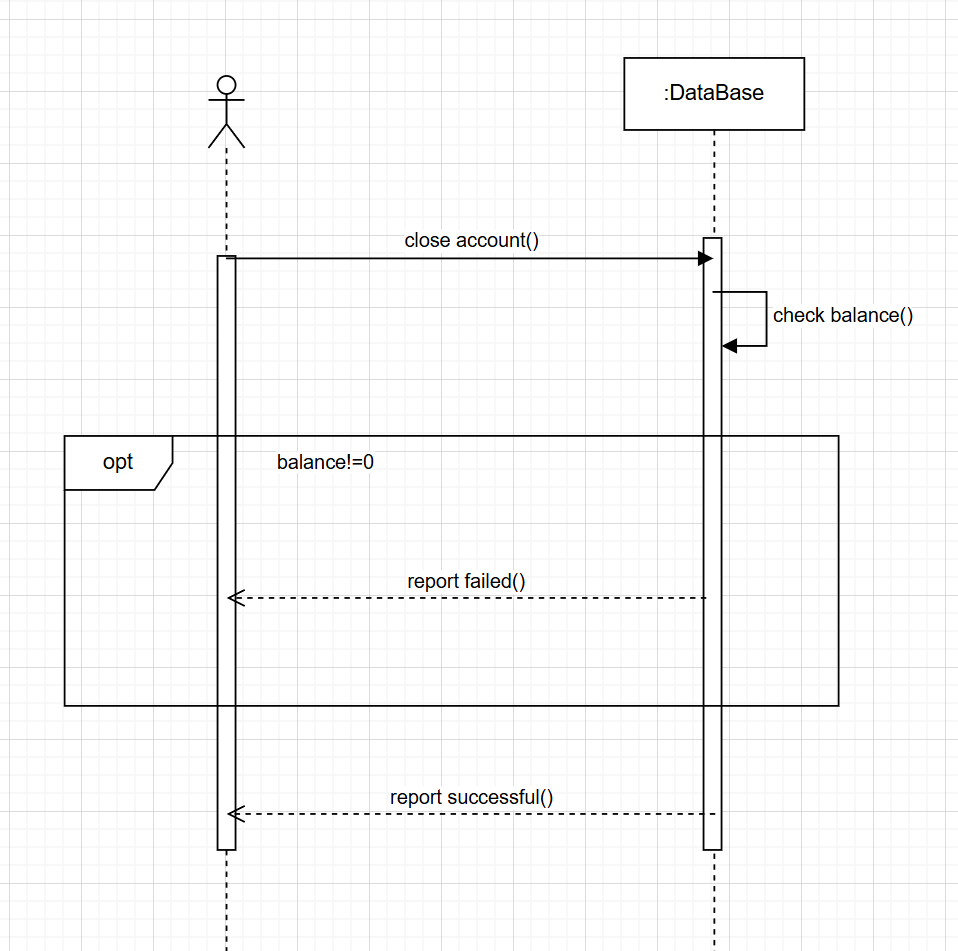
****

* **Frontend:** User selects the "Close Account" option and confirms their decision.
* **Backend:**
* Validates the user session or requires re-authentication.
* Marks the account as closed and removes it from active accounts.
* Triggers the event account\_closed@id.

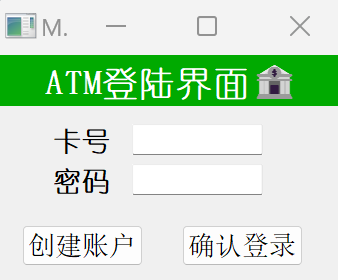


* **Data Flow:** Authentication -> Close Account -> Trigger Event.
* **Failure Case:** If the account is already closed or does not exist, triggers failed@close\_account.
* Only when the account balance is zero can it be safely closed. If this requirement is not met, the system will display a failure to close the account and exit the account closure operation.

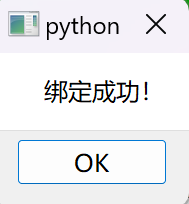


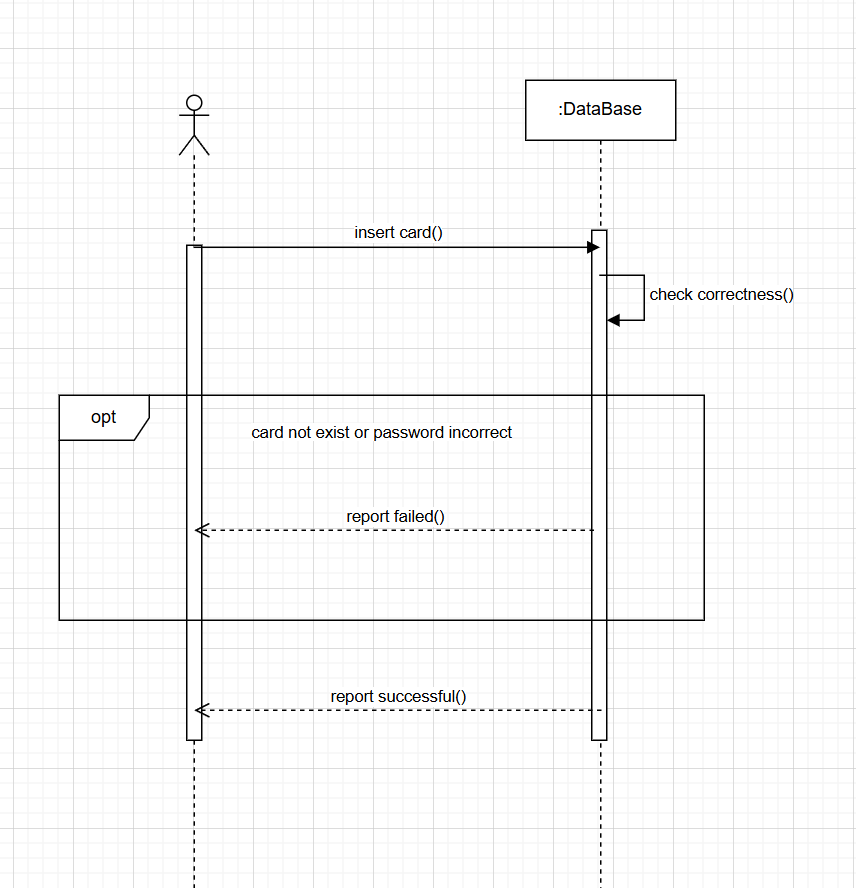


* **S1.3 Insert Card (insert\_card@id)**

****

* **Frontend:** User selects "Insert Card" and enters their account number.
* **Backend:**
* Validates if the account number exists.
* Simulates card insertion by setting ATM\_current\_account to the account number.
* Triggers card\_inserted@id.
* **Data Flow:** Validate Account -> Simulate Insertion -> Trigger Event.
* **Failure Case:** If the account number does not exist, triggers failed@insert\_card.
* The ATM will display a confirmation message, card\_inserted@id, indicating that the card insertion has been recognized and you can proceed with transactions.

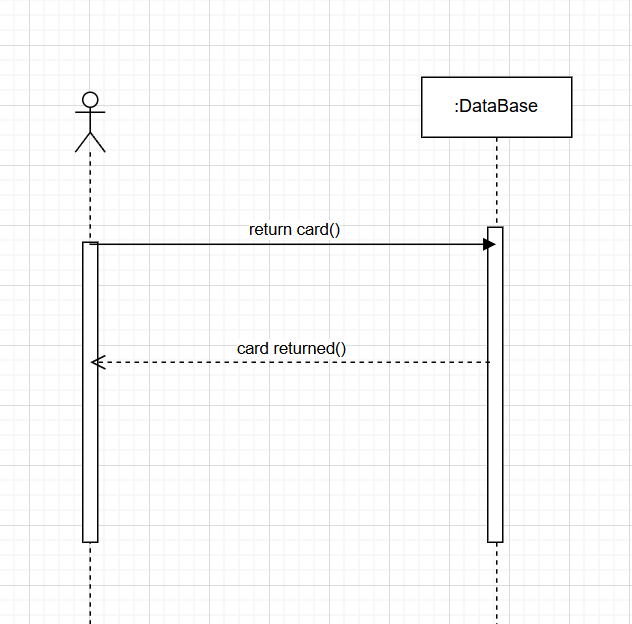




* **S1.4 Return Card (return\_card)**

****

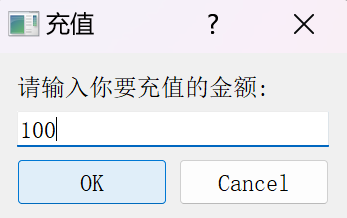
* **Frontend:** User selects "Return Card".
* **Backend:**
* Resets ATM\_current\_account to zero, simulating card return.
* Triggers card\_returned@id using the last used account number.
* **Data Flow:** Reset Account -> Trigger Event.
* **Failure Case:** If no card was inserted prior, triggers failed@return\_card.



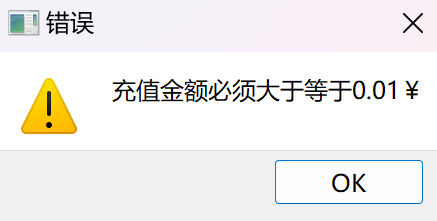
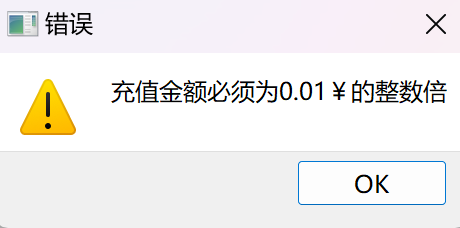
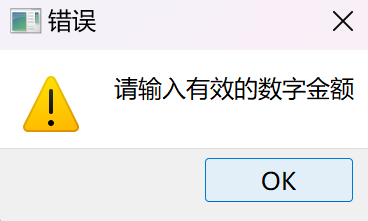
* **S1.5 Deposit Cash (deposit\_cash@num)**

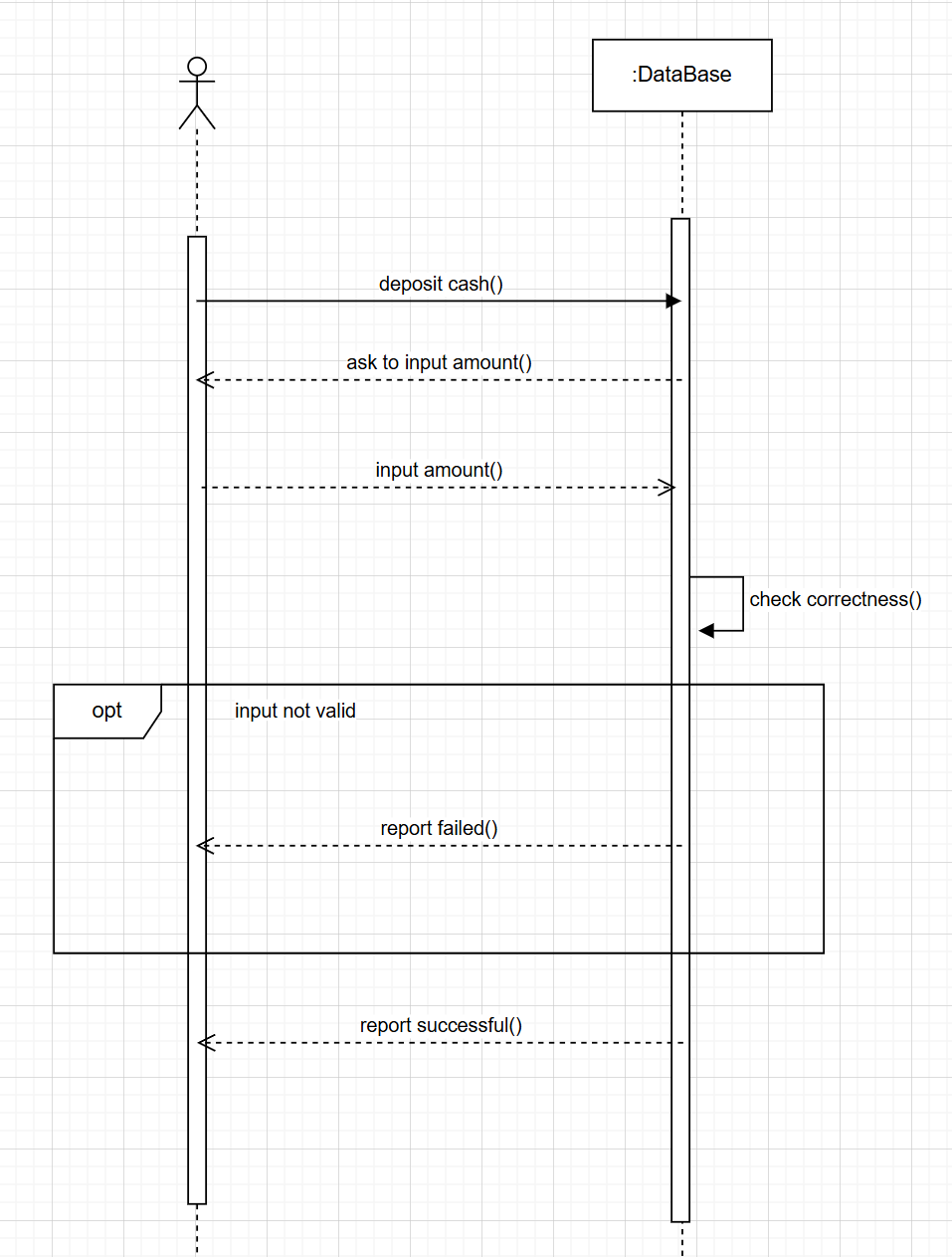
****

* **Frontend:** User enters the amount of cash to deposit.



* **Backend:**
* Validates if a card is "inserted" and session is active.
* Adds the specified amount to account\_balance.
* Triggers cash\_deposited@num.
* **Data Flow:** Validate Session -> Deposit Cash -> Update Balance -> Trigger Event.
* **Failure Case:** If no card is inserted or session times out, triggers failed@deposit\_cash.
* Each recharge amount should be no less than ¥0.01, with a precision up to ¥0.01. If these requirements are not met, the recharge amount will be deemed invalid, and the recharge operation will be automatically terminated.

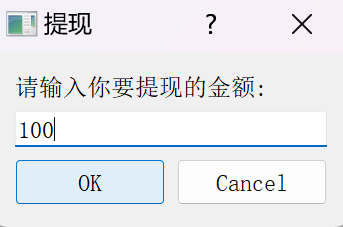
**** **** 

****

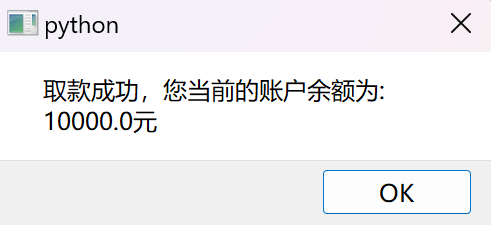
* **S1.6 Withdraw Cash (withdraw\_cash@num@password)**

****

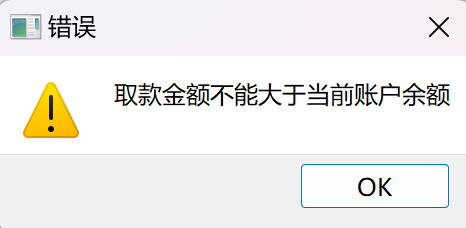
* **Frontend:** User inputs the amount and their password.

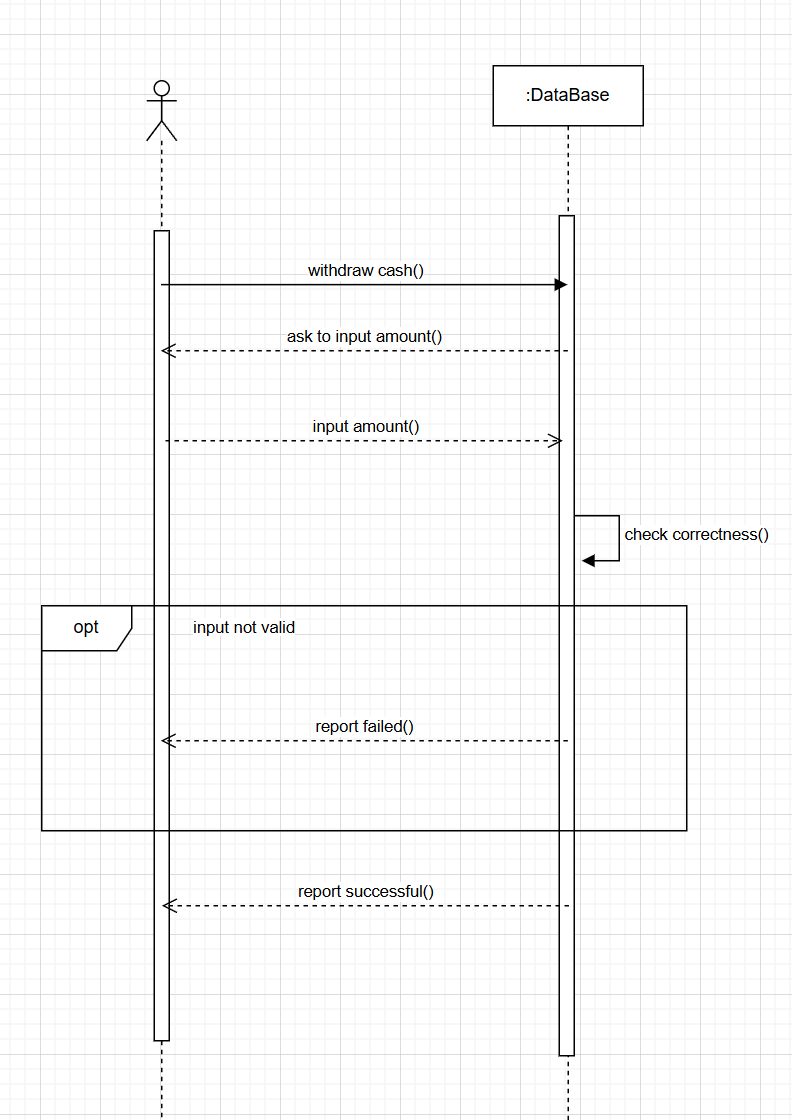


* **Backend:**
* Validates the password and checks for sufficient balance.
* Deducts the amount from account\_balance if successful.
* Triggers cash\_withdrawn@num.

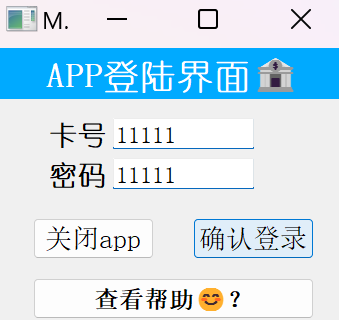


* **Data Flow:** Authenticate -> Check Balance -> Withdraw Cash -> Trigger Event.
* **Failure Case:** If password is incorrect, balance insufficient, or no card inserted, triggers failed@withdraw\_cash.
* Each withdrawal amount should be no less than ¥0.01, with a precision up to ¥0.01. Additionally, the withdrawal amount must not exceed the current account balance. If these requirements are not met, the withdrawal amount will be deemed invalid, and the withdrawal operation will be automatically terminated.

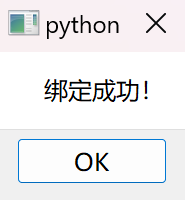
****

****

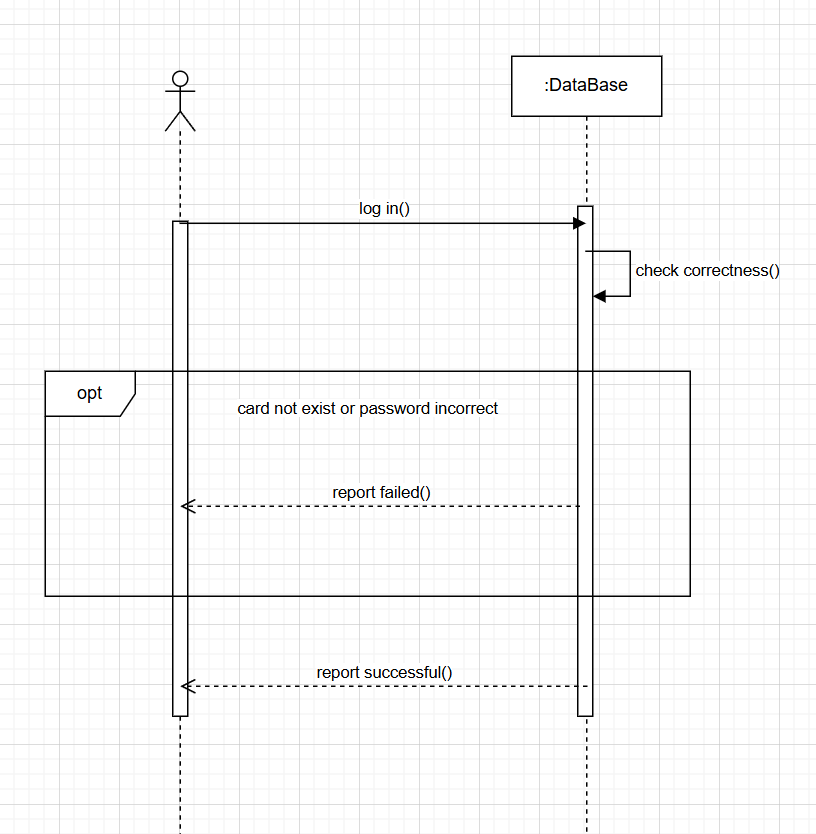
* On APP
* **S2.1 Log In (log\_in@id@password#app\_id)**



* **Frontend:** User enters their account number and password to log in.
* **Backend:**
* Validates the credentials against account\_password.
* If successful, associates the session with a unique app\_id and sets the user as logged in.
* Triggers logged\_in@id#app\_id.



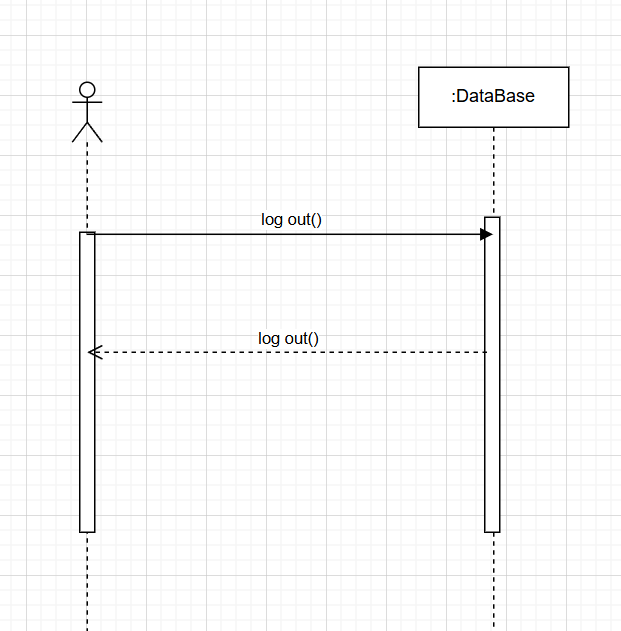
* **Data Flow:** Credential Input -> Validation -> Session Setup -> Trigger Event.
* **Failure Case:** If credentials are incorrect or the account does not exist, triggers failed@log\_in.



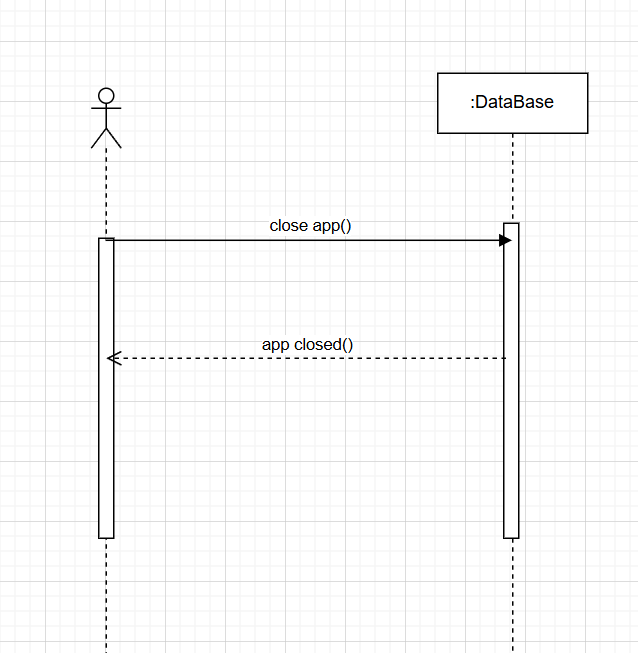
* **S2.2 Log Out (log\_out#app\_id)**

****

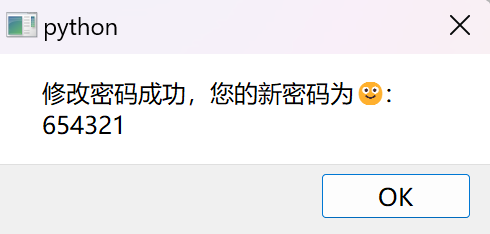
* **Frontend:** User selects to log out.
* **Backend:**
* Terminates the session associated with the app\_id.
* Triggers logged\_out@id#app\_id using the account associated with the app\_id.
* Data Flow: Session Termination -> Trigger Event.
* **Failure Case:** If the app\_id does not exist or no user is logged in, triggers failed@log\_out.



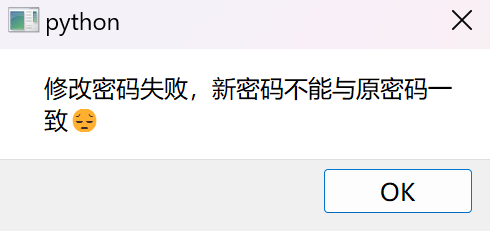
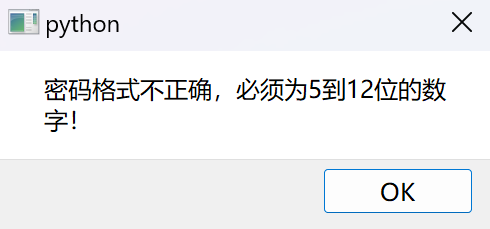
* **S2.3 Close App (close\_app#app\_id)**
* **Frontend:** User selects to close the app.
* **Backend:**
* Closes the application instance identified by app\_id.
* Triggers app\_closed#app\_id.
* **Data Flow:** Close App -> Trigger Event.
* **Failure Case:** If the app\_id is invalid or the app is not running, triggers failed@close\_app.

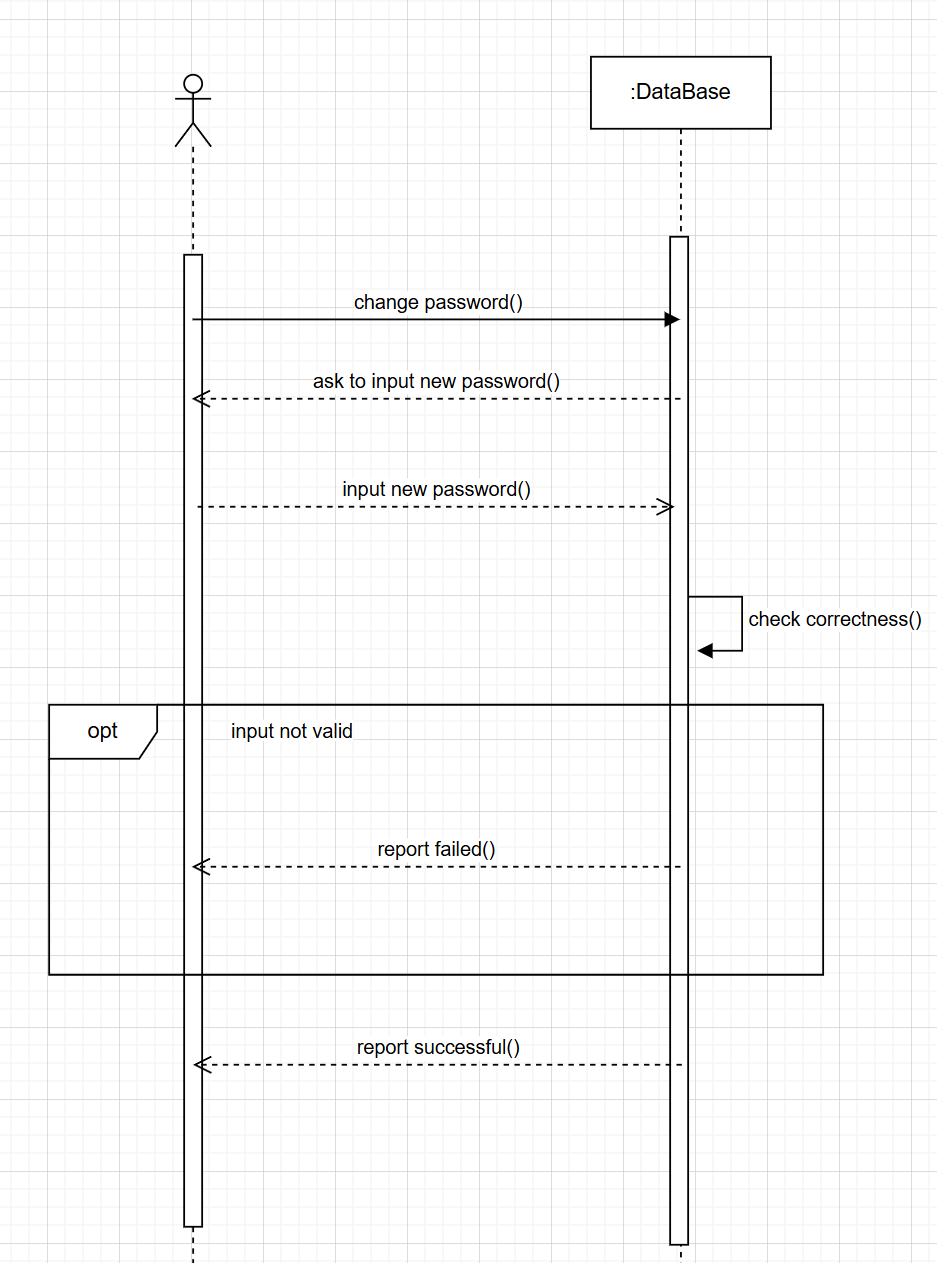


* Both ATM and App
* **S1.7 & S2.4 Change Password (change\_password@new\_password(#app\_id))**
* **Frontend:** User chooses to change their password, entering the new password.
* **Backend:**
* Validates the user's current session or re-authenticates if necessary.
* Updates the account\_password with the new password.
* Optionally logs the change with an app\_id if done via the app.
* Triggers password\_changed(#app\_id).



* **Data Flow:** Validate Session -> Update Password -> Log Event.
* **Failure Case:** If authentication fails or the session is invalid, triggers failed@change\_password.
* New password should consist of 5 to 12 digits. Additionally, ensure that the new password is not the same as the original one. If these requirements are not met, the system will display a failure to modify the password and exit the password modification operation.

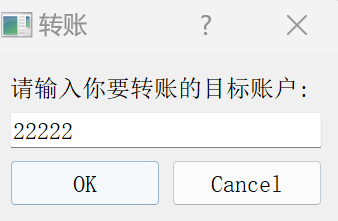
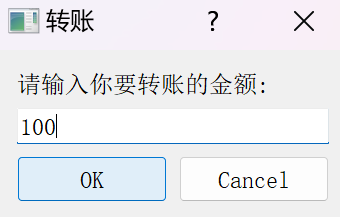
**** ****



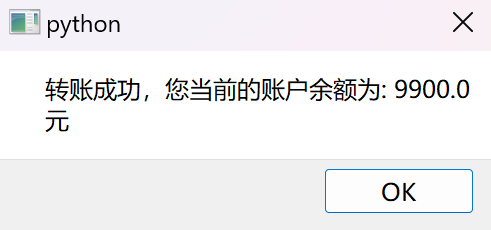
* **S1.8 & S2.5 Transfer Money (transfer\_money@receiver\_id@num(#app\_id))**

** **

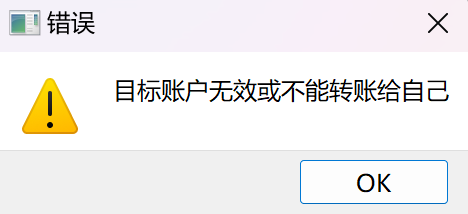
* **Frontend:** User inputs the receiver's account number and the amount to transfer.

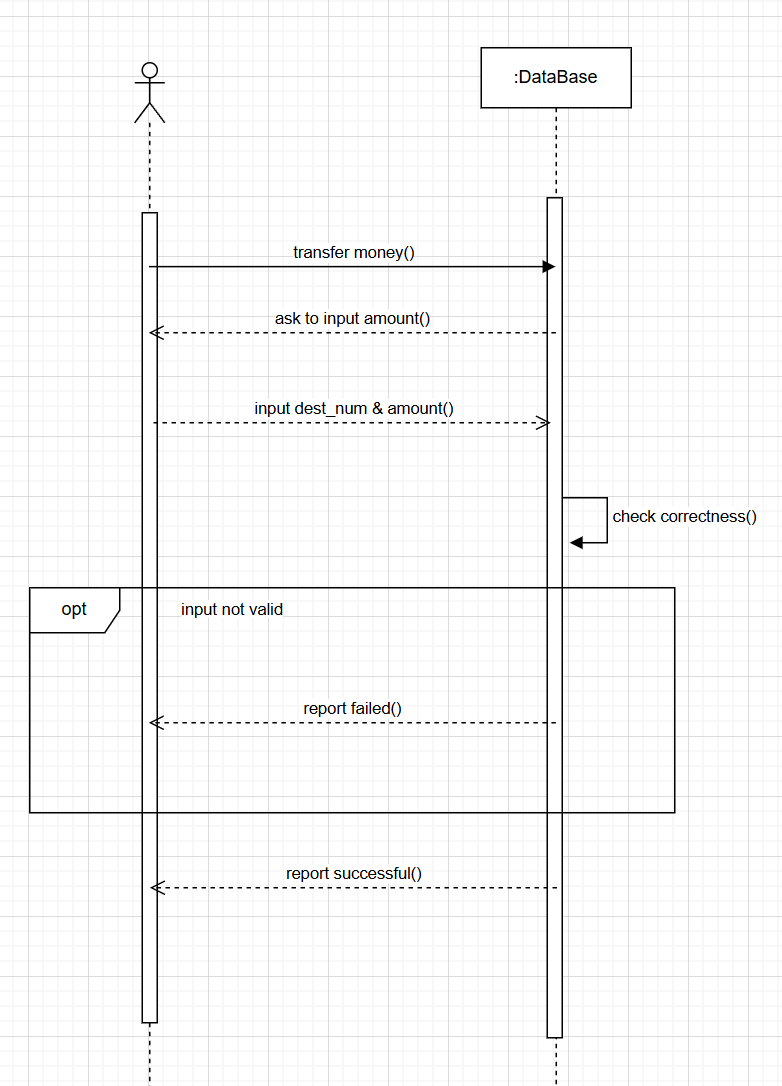
 

* **Backend:**
* Checks for the existence of both sender and receiver accounts.
* Validates that the sender has sufficient funds.
* Deducts the amount from the sender's balance and adds it to the receiver's balance.
* Optionally logs the transaction with an app\_id if done via the app.
* Triggers money\_transfered@num(#app\_id).



* **Data Flow:** Validate Accounts -> Check Funds -> Transfer Money -> Log Event.
* **Failure Case:** If any validation fails, such as non-existent accounts or insufficient funds, triggers failed@transfer\_money.
* . The amount requirements are similar to those for deposits and withdrawals. Additionally, the transfer should ensure that the target account exists and should not transfer to yourself. If the target account does not exist in the database or you transfer to yourself, the transfer will also fail. If these requirements are not met, the system will display the transfer as invalid and automatically terminate the transfer operation.

****

****

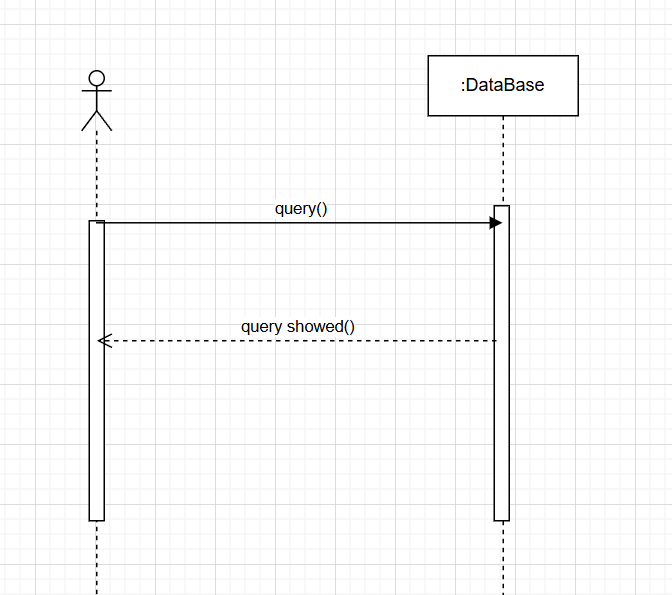
* **S1.9 & S2.6 Query Transactions (query(#app\_id))**
* **Frontend:** User selects to view their transaction history.

**** ****

* **Backend:**
* Retrieves transaction details from detail dictionary where transactions are logged.
* Optionally identifies the session with an app\_id if done via the app.
* Displays the transaction history to the user.
* Triggers query\_showed(#app\_id).



* **Data Flow:** Fetch Data -> Display History -> Log Event.
* **Failure Case:** If there is an error retrieving the data or if no transactions exist, triggers failed@query.



Error Handling

Overview

Even with a robust and user-friendly banking system, users may occasionally encounter issues. The previous section has described the possible failure operations for each scenario. This section will classify the aforementioned failure operations for user awareness. This section provides detailed troubleshooting steps to resolve common problems that may arise while using the ATM or mobile app.

Common Issues and Solutions

* Logging problem
* **Problem Description:** Error message to indicate that the user may not enter the main page of the ATM or app.
* Please check if the account and password input is incorrect, or if the account has been deactivated.
* Transaction problem
* **Problem Description:** Error message to indicate that the user may not complete transactions.
* **Basic requirement:**

The amount of money of each transaction must be a number.

The amount of money of each transaction should be no less than ¥0.01.

The amount of money of each transaction should be consist with a precision up to ¥0.01.

* **Additional requirement:**
* **For withdraw:**
* the withdrawal amount must not exceed the current account balance
* **For transfer:**
* the transfer should ensure that the target account exists
* Account setting problem
* **Problem Description:** Error message to indicate that the user can not create account due to the incorrect password input.
* Password requirements: must be 5 to 12 digits.
* **Problem Description:** Error message to indicate that the user can not change the password.
* New password requirements: must be 5 to 12 digits and should not be same with the previous one.
* **Problem Description:** Error message to indicate that the user can not close the account on ATM.
* Only when the account balance is zero can it be safely closed.
* App Crashes Unexpectedly
* **Problem Description:** The app closes suddenly.
* Restart the app.