Table of Contents:

1. [Bank Management System](https://iq.opengenus.org/system-design-of-bank-management-system/#bankmanagementsystem)
2. [System Requirements](https://iq.opengenus.org/system-design-of-bank-management-system/#systemrequirements)
   * [Functional Requirements](https://iq.opengenus.org/system-design-of-bank-management-system/#functionalrequirements)
   * [Non-Functional Requirements](https://iq.opengenus.org/system-design-of-bank-management-system/#nonfunctionalrequirements)
   * [Software Requirements](https://iq.opengenus.org/system-design-of-bank-management-system/#softwarerequirements)
3. [High-Level Design](https://iq.opengenus.org/system-design-of-bank-management-system/#highleveldesign)
   * [Monolithic Architecture](https://iq.opengenus.org/system-design-of-bank-management-system/#monolithicarchitecture)
   * [Event-Driven Architecture](https://iq.opengenus.org/system-design-of-bank-management-system/#eventdrivenarchitecture)
4. [Low-Level Design](https://iq.opengenus.org/system-design-of-bank-management-system/#lowleveldesign)
5. [Database Design](https://iq.opengenus.org/system-design-of-bank-management-system/#databasedesign)
6. [Leading Bank Management Systems](https://iq.opengenus.org/system-design-of-bank-management-system/#leadingbankmanagementsystems)

**Bank Management System**

The bank management system is a set of essential tools and processes that allow banks and their credit institutions to carry out their functions. The components of the bank management system may differ depending on the bank, but generally, the system includes core banking to manage basic transactions, loans, mortgages, and payments accessible via ATM, mobile banking, and branches. Other components that may be included are CRM systems, Risk Management Systems, Human Resource Management Systems, and Business Intelligence systems.

**System Requirements**

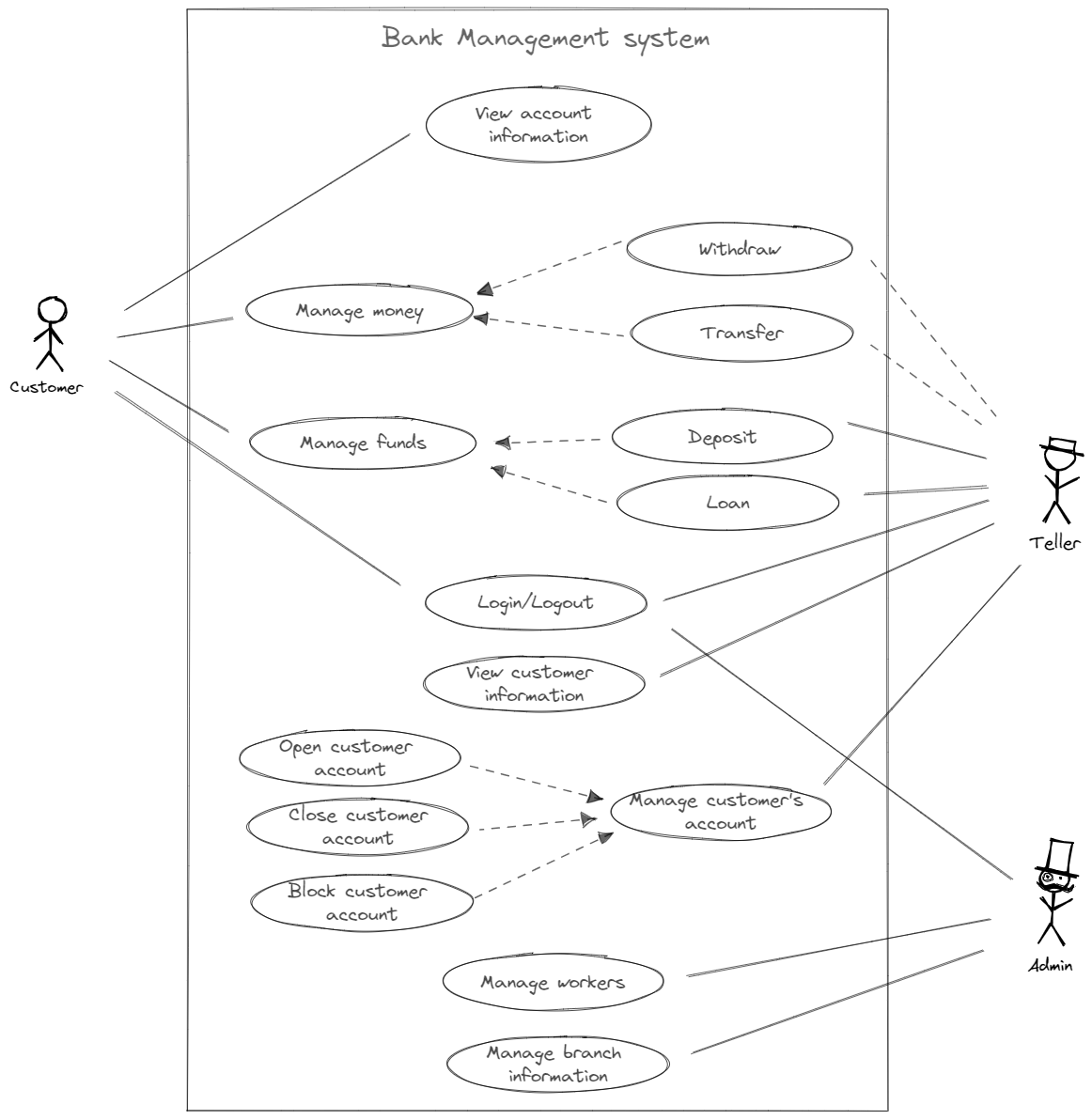
The requirements for a bank management system provide a complete description of the system behavior and are based on the business expectations. The functioning of the system must comply with the laws and regulatory acts of the country. In the United States, for example, Consumer Financial Protection Bureau, Federal Reserve Board, Federal Deposit Insurance Corporation, and Financial Crimes Enforcement Network are the institutions that govern banking activities.

The key requirements that need to be offered by the Bank Management System can be classified into functional and non-functional requirements.

**Functional Requirements**

Functional Requirements describe the service that the banking management system must offer, they are subdivided into three access levels: Admin Mode, Teller Mode, and Customer Mode:

* Customer:
  1. Sign in with login and password.
  2. Update personal details.
  3. Change password.
  4. View balance.
  5. View personal history of transactions.
  6. Transfer money.
  7. Withdraw.
  8. Submit Cash.
* Teller:
  1. Sign in with login and password.
  2. Change password.
  3. Register new bank customers.
  4. View customer information.
  5. Manage customer accounts.
* Admin:
  1. Sign in with login and password.
  2. View manager and customer details.
  3. Add or update bank branch details.
  4. Add or update manager details.

The use case diagram provides an overview of the basic services:  


**Non-Functional Requirements**

Non-functional requirements specify criteria that can be used to judge the operation of a system as a whole rather than specific behaviors. They describe emergent properties like security, performance, and availability and, unlike the functional requirements that can be worked around, are essential to fulfill for a usable system. The estimation of whether the product fulfills the non-functional requirement or not usually reduces to a boolean answer: yes or no.

For a bank management system, the most important non-functional requirements include security, performance, usability, and availability.

**Security**  
Bank management systems are notorious for being subject to malicious attacks, so security is the major requirement for the system. Unauthorized access to the data is not permissible. The data must be backed up daily and stored in a secured location, at a distance from different facilities of the system.

Online transactions and stored digital files must be encrypted according to 128-bit or 256-bit AES encryption standards. The system also must employ firewall software as a defense against network attacks.

From the client-side, the system must provide an automatic log-out after an inactivity period, accept only secure passwords that have sufficient length and non-alphabetic characters, and block login attempts after several unsuccessful trials.

**Performance**  
The bank management system is a multi-client system that must reach response time targets for each of the clients during simultaneous calls and must be able to run a target number of transactions per second without failure. The system must effectively utilize the hardware and energy resources to minimize operational costs.

**Usability**  
The system must provide different graphical interfaces for customers, tellers, and admins. All system interfaces must be user-friendly and simple to learn, including helping hints and messages and intuitive workflow, especially in a client interface: the client must be able to fast learn and use the interface without prior knowledge of banking terminology or rules.

The interfaces must automatically adjust to devices with different screen sizes, and allow to change typeface size and color scheme to improve readability.

**Availability**  
The system must be available during bank working hours. The mobile banking and ATM must be available round-the-clock with minimal maintenance times, reaching 99.999% availability time per year.

**Software Requirements**

Software requirements specification (SRS) is the description of the software system that is going to be developed, it is made at the latest phase of analysis, after the functional and non-functional requirements. The set of programming tools and technologies that can be applied to the bank management system depends on whether the on-premise, cloud, or hybrid computing model is used.

Most large-scale financial institutes have their core banking system run on-premise, which may be enforced by the legal system requirement to facilitate servers that store personal data on the territory of the country.

The development of the system is based on the following technologies:

1. Servers running Windows Server/Linux OS, ATM running Windows 10.
2. For the backend part, a scalable programming language supporting multi-threading like Java is required, and Python is required for the data analysis and fraud detection engine.
3. Modern front-end frameworks like React/AngularVue/jQuery for user-interface.
4. Relational DBMS with an engine that supports ACID transactions like Microsoft SQL Server or Oracle RDBMS.

**High-Level Design**

After analysis and agreement on the requirements, the next stage is to describe system architecture on a high level.

**Monolithic Architecture**

The traditional way to implement bank management systems is with a monolithic architecture, where different tasks are managed with a single unified unit. As shown in the diagram below, Java handles a variety of jobs that come from the branch office, from electronic card processing and from the requests to API open to other banks and clients.

The load balancer distributes evenly jobs between application servers that run multiple copies of the application, and on the backside, the application manages database requests. The load balancer also plays the role of the inner firewall, in addition to the outer firewall that provides first-level protection from network attacks.

The architecture is simple and thus easy to develop and deploy, scaling is also relatively easy with the load balancers - just add another application/database server when needed.

The monolithic architecture is the easiest to implement, but it is hard to maintain in the long run, and it is very hard to add new features or update the old ones. Moreover, with a monolithic architecture, it is hard to introduce new tools and frameworks to the developing stack, as there are no points inside the monolithic structure to connect new technologies.

**Event-Driven Architecture**

Event-driven architecture is one of the alternatives to the traditional Monolithic architecture that centers around "events" rather than entities. For example, events when an office teller submits a new client, when a customer presses an ATM button to withdraw money, or when the account balance is getting lower than the threshold. All incoming events are registered within the API layer and added to the Kafka stream.

The Apache Kafka, an open-source distributed event streaming platform, distributes events between the consumers: Application Jobs, Data storage, Statistics collectors, Notification routines, and the special Fraud Check engine. Fraud Check is written in Python to automatically detect and block suspicious transactions to prevent money laundering and other violations.

With the event-driven architecture, implementing new services is easier, as there is no need to constantly synchronize states between them, for example, to synchronize the current balance state with the upcoming transactions from credit cards and open API. Instead, the program can introduce an even that controls and fixes the balance changes, so the transaction service only needs information about balance changes and does not care about other transactions and processes.

As event-driven architecture requires a careful design in the initial stages and thus needs more time and resources spend on development, this is a good option for large financial institutions that aim for a large client auditory and provide a row of different services.

**Low-Level Design**

The Bank Management System application defines all available operations in a Bank interface, according to the Data Access Object Pattern (DAO). The actual implementations of this interface are interchangeable and do not require changes on the front-end:

public interface Bank {

// Create

boolean add(Customer customer);

boolean add(Employee employee);

boolean add(Account account);

boolean add(Loan loan);

boolean add(Branch branch);

boolean add(Transaction transaction);

// Read

List<Customer>findAllCustomers();

Customer findCustomer(String param);

List<Employee>findAllEmployees();

Employee findEmployee(String param);

List<Account>findAccountsOfCustomer();

List<Loan>findLoansOfCustomer();

List<Transaction> findTransactionsOfAccount();

List<Transaction> findTransactionsOfCustomer();

String getTransactionDetails();

// Update

boolean update(Customer customer);

boolean update(Employee employee);

boolean update(Account account);

boolean update(Loan loan);

boolean update(Branch branch);

// Delete

boolean delete(Customer customer);

boolean delete(Employee employee);

boolean delete(Account account);

boolean delete(Loan loan);

boolean delete(Branch branch);

}

public class BankImp implements Bank{

// Implementation of Bank methods

}

The application has six persistence entities that correspond to the database tables: Account, Branch, Customer, Employee, Loan, and Transaction. Abstract class Person reuses repeating code for Customer and Employee classes and is not present in the database.

@Getter

@Setter

public abstract class Person {

private int id;

private String login;

private int passhash;

private String name;

private String phone;

private String email;

private Date registrationDate;

}

@Getter

@Setter

public class Customer extends Person{

private Set<Account> accounts;

private Set<Loan> loans;

}

@Getter

@Setter

public class Employee extends Person{

private String position;

private Employee manager;

private Branch branch;

}

@Getter

@Setter

public class Account {

private int id;

private Customer customer;

private Branch branch;

private Date openingDate;

private BigDecimal currentBalance;

private BigDecimal interestRate;

}

@Getter

@Setter

public class Loan {

private int id;

private Customer customer;

private Branch branch;

private Date startingDate;

private Date dueDate;

private BigDecimal amount;

}

@Getter

@Setter

public class Transaction {

private int id;

private Date date;

private BigDecimal amount;

private Account account;

private Employee teller;

}

@Getter

@Setter

public class Branch {

private String id;

private String address;

private String phone;

}

Java

Copy

The actions over the persistence entities are controlled via controller classes, one controller for each of the persistence entities. These controllers are to be called in the implementation of the Bank interface.

public class CustomerController {

public Customer createCustomer();

public boolean removeCustomer(){};

public Customer getCustomerOfAccount(Account account){};

public Customer findCustomerByID(int id) {};

public List<Customer> findCustomersByName(String name) {};

}

public class EmployeeController {

public Employee createEmployee(){};

public boolean updateEmployee(){};

public boolean removeEmployee(){};

public boolean setManager(){};

}

public class AccountController {

public Account createAccount(){};

public boolean updateAccount(){};

public boolean removeAccount(){};

public List<Accounts> getAccountsOfCustomer();

public String getAccountDetails();

}

public class LoanController {

public Loan createLoan(){};

public boolean updateLoan(){};

public boolean removeLoan(){};

public List<Loan> getLoansOfCustomer();

public String getLoanDetails();

}

public class TransactionController {

public List<Transaction> getTransactionsOfAccount(Account account){};

public String getTransactionDetails(Transaction transaction);

@Transactional

public boolean withdraw(Account account, BigDecimal amount) {}

@Transactional

public boolean deposit(Account account, BigDecimal amount) {}

@Transactional

public boolean transfer(Account from, Account to, BigDecimal amount) {}

}

public class BranchController {

public Branch createBranch(){};

public boolean updateBranch(){};

public boolean removeBranch(){};

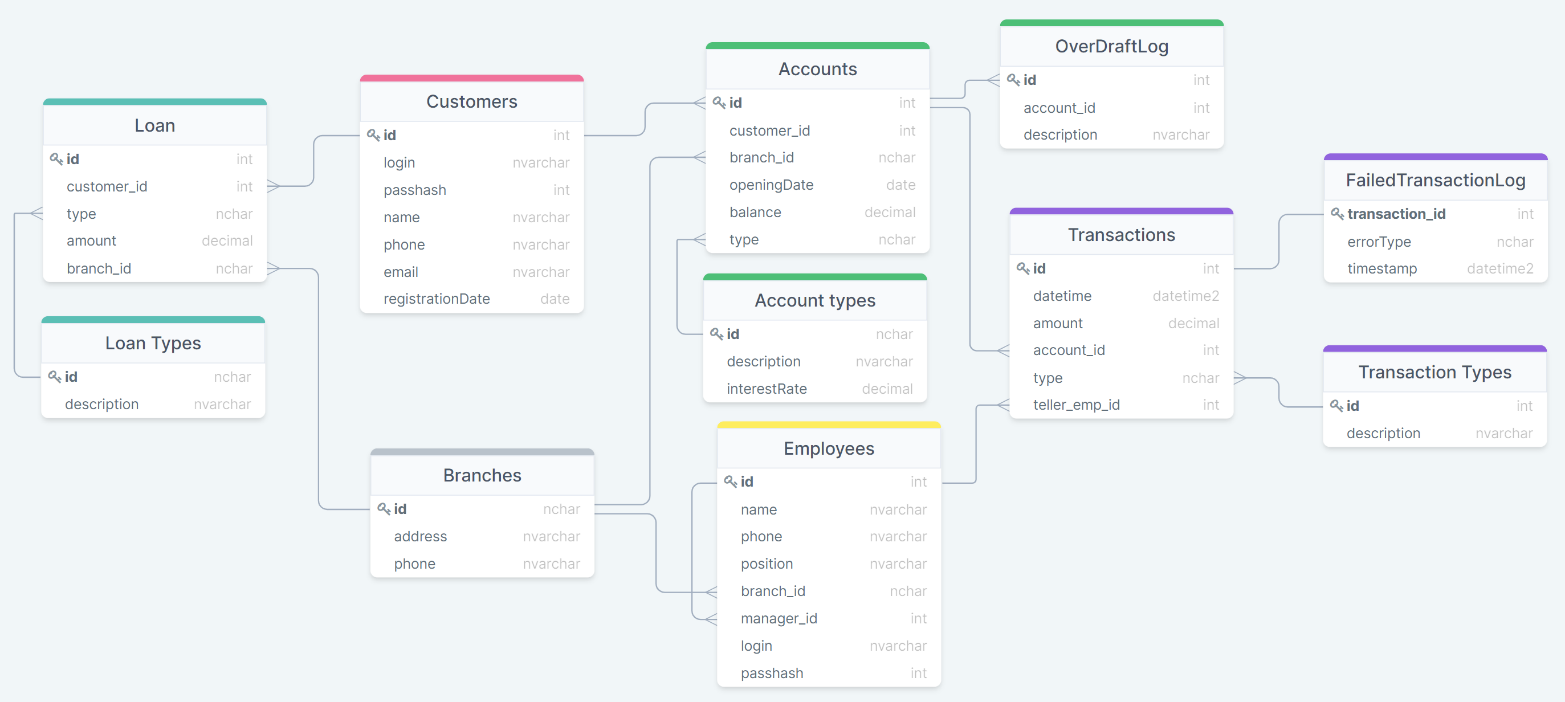
public String getBranchDetails(){};

public List<Employee> getBranchEmployees(){};

}

The most important class to pay attention to is the TransactionController with withdraw(), deposit() and transfer() methods, all of which must provide absolute accuracy as neither bank nor client is willing to lose money. Such methods should be managed as a single unit of work (database transaction) with the ability to roll back all changes, and include multiple checks of sending and receiving accounts, checks for fraud, and others before committing the transaction.

**Database Design**

The Business Management System uses an SQL-based database like Microsoft SQL Server or Oracle RDBMS and includes tables for main business entities and the link tables between them.  


**Customers** relation stores information about bank clients, each of the customers can have multiple balance **Accounts** and **Loans**.

**Branches** relation store information about bank branches, with bank **Employees** each assigned to one of the bank branches. Client accounts and Loans are also assigned to one of the bank branches for easier management. Each employee can have another employee assigned as the manager.

When a client withdraws, submits, or transfers money either by credit card, mobile banking, or at the bank branch office, the information about the action is stored in the **Transactions** table. Not all transactions are accepted, as the bank has multiple checks to perform, the history of failed transactions is stored in the **FailedTransactionsLog** table to collect data and provide the basis for bank security systems improvement.

The database has some tables not reflected in the domain model: **Loan Types**, **Account Types**, and **Transaction Types**, as they represent static information