High-Level Architecture & Design

Contents

[Introduction 3](#_Toc191914620)

[High-Level Design Representation 3](#_Toc191914621)

[System Components 3](#_Toc191914622)

[Core Banking Modules: 3](#_Toc191914623)

[API Gateway: 4](#_Toc191914624)

[Communication Pattern: 5](#_Toc191914625)

[Databases and Data Management 5](#_Toc191914626)

[Database Selection: 5](#_Toc191914627)

[Data Consistency: 8](#_Toc191914628)

[Data Encryption and Access Control: 8](#_Toc191914629)

[Cloud Services and Deployment 8](#_Toc191914630)

[Cloud Provider: 8](#_Toc191914631)

[Cloud-Native Services: AWS Cloud services that can be used for this application are listed below: 8](#_Toc191914632)

[Auto-scaling: 9](#_Toc191914633)

[Security Considerations 9](#_Toc191914634)

[Authentication and Authorization: 9](#_Toc191914635)

[Data Encryption: 9](#_Toc191914636)

[Compliance: 9](#_Toc191914637)

[Scalability and High Availability 10](#_Toc191914638)

[Load Balancing: 10](#_Toc191914639)

[Auto-Scaling: 10](#_Toc191914640)

[High Availability & Disaster Recovery: 10](#_Toc191914641)

[Monitoring, Logging, and Auditing 10](#_Toc191914642)

[Testing Strategy 11](#_Toc191914643)

# Introduction

This document describes the high-level architecture & design of the new Banking application to be built using NodeJS microservices.

# High-Level Design Representation

|  |
| --- |
|  |

# System Components

## Core Banking Modules:

The core banking platform will have the following key modules, each functioning as a separate NodeJS microservice:

* **Account Management Service**: Microservice that has functions for managing user accounts such as account creation, update and balance checks.

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **URL path** | **Usage** |
| GET | /accounts | Retrieves a list of accounts associated with the customerId parameter. |
| GET | /accounts/{accountId} | Retrieves the details of specific account. |
| GET | /accounts/{ accountId }/balance | Retrieves the balance of the specific account. |
| POST | /accounts | Creates a new account for the customer |
| PUT | /accounts/{ accountId } | Updates details of an existing account. |
| DELETE | /accounts/{ accountId } | Deletes/Closes an existing account. |

* **Transaction Service**: Microservice that has functions to handle financial transactions like deposits, withdrawals, transfers and history.

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **URL path** | **Usage** |
| GET | /accounts/{accountId}/transactions | Retrieves a list of transactions for an account. |
| POST | /accounts/{accountId}/transactions | Perform a transaction on an existing account. |

* **Customer Management Service**: Microservice that has functions for managing customer data like KYC (Know Your Customer) and registration.

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **URL path** | **Usage** |
| GET | /customers/{customerId} | Retrieves the details of the customer. |
| POST | /customers | Register a new customer. |

* **Loan Service:** Microservice that has functions for managing loan applications, repayments and history.

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **URL path** | **Usage** |
| GET | /loans | Retrieves a list of loans. |
| GET | /loans/{loanId} | Retrieves the details of specific loan. |
| GET | /loans/{loanId }/schedule | Retrieves the loan repayment schedule. |
| POST | /loans | Apply for a new loan. |
| PUT | /loans/{loanId } | Updates status of an existing loan. |

* **Notification Service**: Microservice that sends notifications for transaction updates, alerts, etc.

## API Gateway:

The **API Gateway** acts as the single-entry point for all incoming requests. It routes the requests to the relevant microservices and performs tasks like:

* Authentication and Authorization (using OAuth 2.0, JWT).
* Rate-limiting.
* Load balancing and request routing.
* Aggregating responses from multiple microservices. (optional)

## Communication Pattern:

* The communication pattern used would be **RESTful APIs** (HTTP/HTTPS) with JSON as the data format for inter-service communication.

# Databases and Data Management

## Database Selection:

Each microservice will have its own isolated database to ensure loose coupling. The choice of database depends on the nature of the service:

* **Account Management**: **Relational Database** (**PostgreSQL**) to maintain structured data such as account details, customer ID, balances, etc.
  + **Data Model**:

|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Data Type | Constraints | Description |
| account\_id | UUID | Primary Key | Unique Identifier for account |
| customer\_id | UUID | Foreign Key | Identifier for customer |
| account\_number | VARCHAR(20) | Unique , Not Null | Account Number |
| account\_type | ENUM | Not Null | Checking, Savings, etc. |
| balance | Decimal(15,2) | Not Null, Default 0.00 | Current balance |
| currency | VARCHAR(3) | Not Null | ISO Currency Code |
| status | ENUM | Not Nul, Default ‘Active’ | Active, Suspended, Closed |
| created\_at | Timestamp | Not Null, Default current timestamp | Account creation date |
| updated\_at | Timestamp | Not Null, Default current timestamp | Last updated timestamp |

* **Transaction Service**: **Distributed NoSQL Database** (**MongoDB**) for handling high write throughput and high availability for tranctional data.
  + **Data Model**:

|  |  |  |
| --- | --- | --- |
| Field Name | Data Type | Description |
| \_id | ObjectId | Unique Identifier |
| account\_id | ObjectId | Account Identifier |
| customer\_id | ObjectId | Customer Identifier |
| transaction\_type | String | Credit, Debit |
| amount | Decimal128 | Transaction amount |
| currency | String | ISO Code |
| status | String | Pending, Completed, Failed |
| payment\_method | String | Card, Bank\_Transfer, UPI, etc. |
| merchant\_details | Object | Details of Merchant |
| description | String | Transaction description |
| created\_at | Timestamp | Timestamp of transaction created |
| updated\_at | Timestamp | Timestamp of last update |

* **Loan Service**: **Relational Database** (**PostgreSQL**) for structured loan data and approvals.
  + **Data Model**:

*Loan Table*

|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Data Type | Constraints | Description |
| loan\_id | UUID | Primary Key | Unique Identifier for loan |
| customer\_id | UUID | Foreign key | Customer Identifier |
| account\_id | UUID | Foreign Key | Account Identifier of the account for payment |
| loan\_type | ENUM | Not Null | Personal, Home, etc. |
| principal\_amount | DECIMAL(15,2) | Not Null | Original loan amount |
| interest\_rate | DECIMAL(5,2) | Not Null | Interest rate in percentage |
| tenure\_months | INT | Not Null | Loan Tenure |
| emi\_amount | DECIMAL(15,2) | Not Null | Monthly EMI |
| status | ENUM | Not Null, Default ‘Pending’ | Pending, Approved, Rejected, etc. |
| disbursed\_date | DATE | Null | Date when loan disbursed |
| due\_date | DATE | Not Null | Next EMI due date |
| created\_at | TIMESTAMP | Not Null, Default current timestamp | Loan creation timestamp |
| updated\_at | TIMESTAMP | Not Null, Default current timestamp | Loan updated timestamp |

*Loan Repayment Table*:

|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Data Type | Constraints | Description |
| repayment\_id | UUID | Primary Key | Unique Identifier for repayment |
| loan\_id | UUID | Foreign key | Loan Identifier |
| payment\_date | TIMESTAMP | Not Null | EMI payment timestamp |
| amount\_paid | DECIMAL(15,2) | Not Null | Amount paid in repayment. |
| payment\_status | ENUM | Not Null, Default ‘Pending’ | Pending, Completed, failed |
| payment\_method | ENUM | Not Null | Card, UPI, Bank\_Transfer |
| referece\_id | VARCHAR(50) | Unique, Not Null | Transaction reference Id |
| created\_at | TIMESTAMP | Not Null, Default current timestamp | Loan creation timestamp |
| updated\_at | TIMESTAMP | Not Null, Default current timestamp | Loan updated timestamp |

*Loan Application Table*:

|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Data Type | Constraints | Description |
| application\_id | UUID | Primary Key | Unique Id for application |
| customer\_id | UUID | Foreign key | Customer Identifier |
| requested\_amount | DECIMAL(15,2) | Not Null | Amount requested |
| requested\_tenure | INT | Not Null | Requested Tenure |
| loan\_type | ENUM | Not Null | Personal, Home, etc. |
| application\_status | ENUM | Default ‘Pending’ | Pending, Approved, Rejected |
| remarks | TEXT | Null | Remarks from loan officer |
| created\_at | TIMESTAMP | Not Null, Default current timestamp | Loan creation timestamp |
| updated\_at | TIMESTAMP | Not Null, Default current timestamp | Loan updated timestamp |

* **Customer Management**: **Relational Database** (**PostgreSQL**) to store sensitive information such as KYC data, customer profiles, etc.
  + **Data Model**:

|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Data Type | Constraints | Description |
| customer\_id | UUID | Primary Key | Unique Identifier for customer |
| first\_name | VARCHAR(50) | Not Null | First Name of Customer |
| last\_name | VARCHAR(50) | Not Null | Last Name of Customer |
| email | VARCHAR(100) | Unique, Not Null | Customer email. |
| phone\_number | VARCHAR(20) | Unique, Not Null | Contact Number |
| date\_of\_birth | DATE | Not Null | Customer’s birth date |
| address | TEXT | Null | Residential Address |
| notify\_pref | VARCHAR(50) | Not Null | Email, Phone |
| created\_at | Timestamp | Not Null, Default current timestamp | Account creation date |
| updated\_at | Timestamp | Not Null, Default current timestamp | Last updated timestamp |

* **Notification Service**: **NoSQL Database** (**MongoDB**) for notifying the customer based on his preference.

## Data Consistency:

* Use **Eventual Consistency** between services to ensure data is synchronized. Event-driven architecture with **Apache Kafka** or **Amazon SQS** can handle the communication of events between services.

## Data Encryption and Access Control:

* **Data Encryption**: Use **AES-256** encryption for sensitive financial data both **at rest** and **in transit**. For TLS/SSL, use the latest protocols to ensure encryption of data in transit.
* **Access Control**: Implement role-based access control (RBAC) using **OAuth 2.0** for authentication and **JWT tokens** for authorization. Each microservice verifies the token to ensure the caller has the required permissions.

# Cloud Services and Deployment

## Cloud Provider:

* **Amazon WebServices (AWS)** to be chosen for its scalability, reliability, and comprehensive set of managed services for running Node.js applications, microservices, and databases.

## Cloud-Native Services: AWS Cloud services that can be used for this application are listed below:

* **AWS EC2 or AWS Elastic Beanstalk**: To host the NodeJS services.
* **Amazon RDS (Relational Database Service)**: Host relational databases (PostgreSQL) for Account and Loan services.
* **MongoDB Atlas on AWS (NoSQL Database Service)**: Document database for transaction services.

Some of the other cloud native services that can be used to enhance the application are:

* **Amazon S3**: Store logs, backups, and other static resources.
* **Amazon SQS (Simple Queue Service)**: For asynchronous communication between services, particularly useful for decoupling heavy workflows like transaction processing and loan approval.
* **Amazon SNS (Simple Notification Service)**: For sending out notifications such as transaction alerts and email confirmations.

## Auto-scaling:

* Use **AWS Elastic Load Balancer (ELB)** to distribute incoming traffic across multiple instances of services and to scale automatically based on traffic loads.
* Set up **auto-scaling groups** to ensure the system can dynamically scale up or down based on demand.

# Security Considerations

## Authentication and Authorization:

* **OAuth 2.0** with **JWT tokens** for handling authentication and authorization.
* **Multi-Factor Authentication (MFA)** for additional security, especially for high-risk operations like logging in, performing financial transactions, or updating sensitive account data.

## Data Encryption:

* **AES-256** encryption for storing sensitive data such as account details and transaction records in databases.
* **TLS 1.2+** encryption for securing data in transit between microservices and between the client and the server.

## Compliance:

* **PCI-DSS**: For transaction-related data, ensure compliance by encrypting card information and adhering to audit logging requirements.
* **GDPR**: Ensure proper data retention policies, provide customers with data access rights, and implement data anonymization where needed.
* **PSD2**: Integrate APIs with third-party providers using **Open Banking standards**, ensuring strong customer authentication (SCA) and secure access.

# Scalability and High Availability

## Load Balancing:

* Use **AWS Elastic Load Balancer (ELB)** for distributing incoming traffic to multiple service instances, ensuring high availability and fault tolerance.
* Implement **horizontal scaling** by adding or removing instances based on demand (via **AWS Auto Scaling**).

## Auto-Scaling:

* Enable **auto-scaling** for each microservice, ensuring the platform can handle increased load (e.g., during peak banking hours) without manual intervention.

## High Availability & Disaster Recovery:

* **Data Replication**: Set up **multi-AZ (Availability Zone) replication** for databases to ensure high availability.
* **Failover Mechanisms**: Use **AWS Route 53** for DNS failover between regions, ensuring that services remain available in case of regional failure.
* **Geo-Redundant Backups**: Regularly back up data to multiple geographic regions to ensure disaster recovery.

# Monitoring, Logging, and Auditing

* **Monitoring**: Implement monitoring with **AWS CloudWatch** for metrics like system health, error rates, and response times.
* **Logging**: Use centralized logging solutions like **AWS CloudWatch Logs** or **ELK stack (Elasticsearch, Logstash, Kibana)** to aggregate logs from all microservices for auditing, debugging, and troubleshooting.
  + **Application logging**: Winston library to log at different levels (INFO, DEBUG, etc.) as well as define those levels and log formats.
  + **Audit logging**: User interactions like account creation, etc. should be logged in to the PostgresSQL database table.
* **Auditing**: Use **AWS CloudTrail** to track all API calls made to AWS services, ensuring compliance with financial regulations.

# Testing Strategy

To ensure robust code quality in the microservices development, we do the following:

* **Unit Testing**: Unit tests to validate individual functions or modules in isolation to verify correctness. **Mocha (testing framework)** and **Chai (assertion library)** testing tools with **Sinon** for mocking dependencies and/or database/3rd party calls can be used to achieve this. A target code coverage of 90% to ensure the modules achieve the individual requirements.
* **Integration Testing**: Integration tests to verify the interactions between microservices and functions. **SuperTest** testing library can be used to test the API endpoints which internally would test the integration between the services.
* **End-to-end testing**: E2E testing of the API endpoints using tools like **Postman** that simulates real world user interactions with close to actual data inputs.