**Project Overview: POS Banking System with Microservices**

**Introduction**

This document outlines the architecture and implementation details for a POS (Point of Sale) system developed using .NET 9.0 with Clean Architecture principles. The system will leverage microservices to ensure modularity, scalability, and maintainability. The core modules include:

1. **User Management**
2. **Account Management**
3. **Transaction Management**

**Technology Stack**

* **Framework:** .NET 9.0
* **Database:** SQL Server
* **ORM:** Dapper
* **Communication:** Azure Service Bus
* **Validation:** FluentValidation
* **Mediator Pattern:** MediatR
* **Error Handling:** ErrorOr
* **Unit of Work:** Custom implementation
* **API Gateway:** Ocelot

**Modules and Features**

**1. User Management**

* Create and manage users.
* Associate accounts with users.
* Validate user inputs using FluentValidation.

**2. Account Management**

* Add accounts for users.
* Manage account balances.
* Maintain a record of account details.

**3. Transaction Management**

* Transfer funds between accounts.
* Record debit and credit transactions.
* Ensure consistency with UnitOfWork.

**Workflow**

1. **Create User**
   * Input: User details (Name, Email, etc.).
   * Process: Validate input, create user record in the User Management microservice.
   * Output: User ID.
2. **Add Account to User**
   * Input: User ID, account details (e.g., account type, initial balance).
   * Process: Validate input, create an account linked to the User ID in the Account Management microservice.
   * Output: Account ID.
3. **Add Balance to User Account**
   * Input: Account ID, amount to be added.
   * Process: Validate input, update the balance in the Account Management microservice.
   * Output: Updated balance.
4. **Transfer Funds**
   * Input: Source Account ID, Target Account ID, transfer amount.
   * Process:
     + Validate input.
     + Deduct amount from the source account.
     + Add amount to the target account.
     + Log the transaction details (debit for the source, credit for the target) in the Transaction Management microservice.
   * Output: Success or failure message with transaction details.
5. **Register Transaction Details**
   * Input: Transaction details (source account, target account, amount, timestamp).
   * Process: Store transaction details in the Transaction Management microservice.
   * Output: Transaction ID.

**Architecture**

**Clean Architecture Layers**

1. **Presentation Layer**
   * API controllers exposed through the Ocelot API Gateway.
   * Handles HTTP requests and responses.
2. **Application Layer**
   * Contains MediatR commands and queries.
   * Implements business logic using services and validators.
3. **Domain Layer**
   * Contains core business entities and domain logic.
   * Ensures high cohesion and separation of concerns.
4. **Infrastructure Layer**
   * Handles database interactions using Dapper.
   * Implements UnitOfWork for transaction management.
   * Integrates with Azure Service Bus for inter-service communication.

**Microservices**

1. **User Management Service**
   * Responsible for creating and managing users.
   * Database: User table with details such as UserID, Name, Email, etc.
2. **Account Management Service**
   * Responsible for managing user accounts and balances.
   * Database: Account table with details such as AccountID, UserID, Balance, etc.
3. **Transaction Management Service**
   * Responsible for recording and managing transactions.
   * Database: Transaction table with details such as TransactionID, SourceAccountID, TargetAccountID, Amount, Timestamp, Type (Debit/Credit).

**API Gateway**

* Ocelot will route and aggregate requests to the respective microservices.
* Centralized authentication and authorization.

**Communication**

* Services will communicate asynchronously using Azure Service Bus for events like fund transfers and transaction logging.

**Implementation Details**

**1. Dapper for Data Access**

* Lightweight ORM for high-performance database interactions.
* Stored procedures for complex queries.

**2. MediatR for CQRS**

* Commands for creating users, accounts, and transactions.
* Queries for retrieving user, account, and transaction details.

**3. FluentValidation**

* Centralized validation for all incoming requests.
* Ensures data consistency and integrity.

**4. ErrorOr for Error Handling**

* Unified approach to error handling.
* Provides a consistent error response structure.

**5. UnitOfWork Pattern**

* Ensures atomicity in database operations.
* Manages transaction scope across multiple repositories.

**Endpoints**

**User Management Service**

* POST /users: Create a new user.
* GET /users/{id}: Get user details.

**Account Management Service**

* POST /accounts: Add an account to a user.
* PATCH /accounts/{id}/balance: Add balance to an account.
* GET /accounts/{id}: Get account details.

**Transaction Management Service**

* POST /transactions: Transfer funds between accounts.
* GET /transactions/{id}: Get transaction details.

**Deployment**

* Dockerized microservices.
* Deployed on Azure Kubernetes Service (AKS).
* API Gateway deployed as a separate service.

**Testing**

* Unit tests for individual components.
* Integration tests for inter-service communication.
* Load testing for scalability.

**Conclusion**

This POS system architecture ensures modularity, scalability, and maintainability. Leveraging modern technologies, Clean Architecture principles, and best practices, it meets the functional and non-functional requirements efficiently.