Problem Solution

Modernising the account management system using Microservice architecture is a great initiative aiming to replace the current monolithic, tightly coupled, single application.

Microservices can bring several benefits, like scalability, fault tolerance, and easier maintenance, to this current application.

According to my current knowledge and experience from my previous jobs, below I will propose a solution for the Account Management system using Microservices.

1.Microservices:

I would suggest having two microservices for the Account domain. Each represents a specific business functionality and runs as an independent process. They can communicate with each other via APIs.

a. Account Service.

- Responsible for managing the core functionalities related to the account itself.

b. Transaction Service.

- Responsible for storing and managing transaction data for each account.

Note: Apart from the above, I have also tried to illustrate some theoretical design suggestions for this microservice, and how I started this task by creating a to-do list, and the time I have spent on the same. Which you will find while reading these documents.

1. Domain modeling:

At a high level, we have mainly two main entities in the Account Management system:

- Account and
- Transaction

Attributes	**Descriptions**
id (Long)	The unique identifier for the account.
accountNumber (String)	The account number associated with the account.
accountType(enum)	Chose Account type while creating a new account i.e., Savings, Fixed, etc.
balance (Double)	The current available balance in the account.

Relationships-: Each account can have multiple transactions (**1-to-Many** relationship).

Attributes	**Descriptions**
id (Long):	The unique identifier for the transaction.
accountNumber (String)	The account number associated with the transaction.
transactionType(enum)	Type of transaction i.e., Deposit, Withdrawal.
amount (double)	Money involved in the transaction.
transactionTime (LocalDateTime)	The timestamp of when the transaction occurred.

Relationships-: Each transaction is associated with a single account (**Many-to-1** relationship).

2. Use cases:

The reason for defining the use cases is to capture the functional requirements of the Account Management system by describing the interactions between external entities, i.e., customers, and what they will achieve after interacting with the Account Management system.

Actor:
The Customer is an external entity that interacts with the Account Management system.

Use Cases	**Descriptions**
1. Create Account	The customer can create a new savings account through the Account Service.
2. Deposit Money	The customer can deposit money into their account using the Account service.
3. Withdraw Money	The Customer can withdraw money from their account using the Account Service.
4. View Balance	The Customer can check the available balance in their account using the Account Service.
5. View Account	The Customer can view their account details, including the balance and the last 10 transactions using the Account Service.

3. API Design:

I have used RESTful API design principles to define the endpoints.

1. API's Endpoint design for Account Service

Rest API's	**Description**

Use case 1: Create Account
Endpoint: 'POST /accounts'
Request Body: (AccountDTO)
{
"accountType": "saving",
"balance": 0.0
}
 	
Response Body: (AccountDTO)
{
"id": 5001,
"accountNumber": "5234567859",
"accountType": "saving",
"balance": 0.0
}
	This endpoint will be responsible for creating a savings account for a customer.
	
	It will receive necessary details like customer information, account type, etc.
	
	It will generate a unique account number and store the account information in the database.

Use case 2: Deposit Money
Endpoint: 'POST /accounts/{accountNumber}/deposit'
Request Body: (AccountDTO)
{
"accountNumber": "5234567859",
"amount": 100
}
Response Body: (AccountDTO)
{
"id": 5001,
"accountNumber": "5234567859",
"balance": 100.0
}
	This endpoint allows customers to deposit money into their savings account.
	
	It will receive the account number and the amount to deposit.
	
	The service will update the account's balance accordingly.

Use case 3: Withdraw Money
Endpoint: 'POST /accounts/{accountNumber}/withdraw'
Request Body: (AccountDTO)
{
"accountNumber": "5234567859",
"amount": 50
}
Response Body: (AccountDTO)
{
	This endpoint allows customers to withdraw money from their savings account.
	
	It will receive the account number and the amount to withdraw.
	
	The service will check if the withdrawal amount is valid based on the account balance.
	
	If the withdrawal is valid, it will update the account balance.

"id": 5001, "accountNumber": "5234567859", "balance": 50.0 }	

Use case 4: View Balance
Endpoint: 'GET /accounts/{accountNumber}/balance'
Request Body: (AccountDTO)
{
"accountNumber": "5234567859"
}
Response Body: (AccountDTO)
{
"amount": 50.0
}
	This endpoint allows customers to retrieve the available balance in their savings account.
	
	It will receive the account number and return the account balance.

Use case 5: View Account
Endpoint: 'GET /accounts/{accountNumber}/ transactions
Request Body: (None required)
{
"accountNumber": "5234567859"
}
Response Body: (List of TransactionDTOs)
[
{
"id": 5001,
"accountNumber": "5234567859",
"amount": 50.0,
"transactionTime": "2023-07-27T08:19:15"
},
{
"id": 5002,
"accountNumber": "5234567859",
"amount": 25.0,
"transactionTime": "2023-07-29T09:11:13"
}
…… up to 8 more…if more record exists
]
	This endpoint allows customers to view their last 10 transactions.
	
	It will receive the account number and fetch the transaction history from the database.
	
	The service will then return the list of transactions sorted by the timestamp.

2. API's Endpoint design for Transaction Service

Rest API's	**Description**

Endpoint: 'POST / transactions'
Request Body: (Transaction)
 	
Response Body: (Transaction)
{
"id": 5001,
"accountNumber": "5234567859",
"balance": 0.0
}
	This endpoint will be called **internally** by the Account Service whenever a deposit or withdrawal occurs.
	
	It will record the transaction details, such as amount, type (deposit/withdrawal), and timestamp.

3. Error Handling:

For invalid input and business validations, appropriate HTTP status codes would be returned along with meaningful error messages in the response body. For example, when an invalid account number is provided, the response could be:

{
"message": "Invalid input or Invalid account number provided."
}

4. API Documentation:

Swagger can be used to document REST APIs. By adding Swagger annotations to the controller methods, an interactive API documentation page would be generated. This documentation would provide details about each endpoint, request parameters, response types, and possible error responses.

5. Realization

Since the Account service and Transaction service are independent, they will communicate through RESTful endpoints. I would also suggest implementing proper authentication and authorization mechanisms to secure the services and the customer data.

Since the Customer domain is managed by other systems within the bank, as far as I can understand, we can basically interact with those systems through appropriate APIs to fetch customer details when needed. This way, we can maintain a clear separation of concerns and leverage the existing functionality without duplicating efforts.

Finally, with the earlier mentioned use cases, domain modelling, and API design, we can have an understanding of the data entities, their relationships, and the endpoints needed to implement the Account Management system using the Microservice architecture.

2. API Gateway:

To provide unified interface to the clients, I would suggest to have an API gateway that sits in front of above mentioned two microservices which will handle incoming requests from clients. Basically, it routes the request to the appropriate microservice.

3. Database:

Good to have its own database for storing its specific data. For simplicity, we can use separate databases for the Account and Transaction services.
Sometime it might not be possible to have separate database at the initial phase of the project development. It depends on the complexity of data. As I mentioned during our talk. In my previous job where I had a very tough time to separate databases for each microservices due to so many relationships among tables, etc.

4. Event

To ensure loose coupling between microservices, good to implement an event-driven architecture using a message broker (e.g., Apache Kafka or RabbitMQ). For example, when a new transaction is made, the Transaction Service can publish an event, and the Account Service can subscribe to it to update the account balance. It's not that we should have it during a start but nice to keep it open for the future improvement.

5. Authentication and Authorization

To ensure that only authorized users can access the services.

6. Monitoring and logging

Good to Implement logging and monitoring mechanisms to track the performance and health of each microservices. Maybe good to use Grafana for visualization and Prometheus as a data source.

7. Containerization

Recommended to use technologies like Docker for (packing code + dependencies) and Kubernetes (to manage containers) for easy deployment and scalability.

8. Final thought, to-do list, and Time spent

To-Do:

- Understand & analyze given task requirements
- Describes the entities involved (Domain modeling)
- Define functional requirements (maybe non-functional requirements later for System design)
- Identify number of microservices are needed
- Define schema
- Code design
- Design Architecture in terms of layer
- Implementation
- Test
- Document

Time spent:

Total time has been spent approx. 5hrs, for the documentation 2-3 hrs., implementation 2-3 hrs., it was recommended to spend max. 3 hrs., but after starting the task I didn't realize the hours.

Time spends day and time wise:

Saturday: morning 6:00 to 9:00 (including a coffee break) --- 3 hrs.

Sunday: morning 6:00 to 8:00 (No break) --- 2 hrs.

Final thought

I only designed and wrote code at a high level. I did not manage to spend much time on fine-grained details. Even for the implementation, I have written very high-level code to reflect my earlier discussions.

9. Implementation screenshot

The screenshot was not mandatory to attach here since I have already attached the source code along with the email. I just attached them for reference for both services.

1. Account Service Implementation screenshots

Entity: Account

```
14 @Entity
15 @Table(name = "account")
16 public class Account {
17
18     @Id
19     @GeneratedValue(strategy = GenerationType.AUTO)
20     private Long id;
21
22     @Column(name = "account_type")
23     @Enumerated(EnumType.STRING)
24     private AccountType accountType;
25
26     @Column(name = "account_number", unique = true, nullable = false)
27     private String accountNumber;
28
29     private Double balance;
30
```

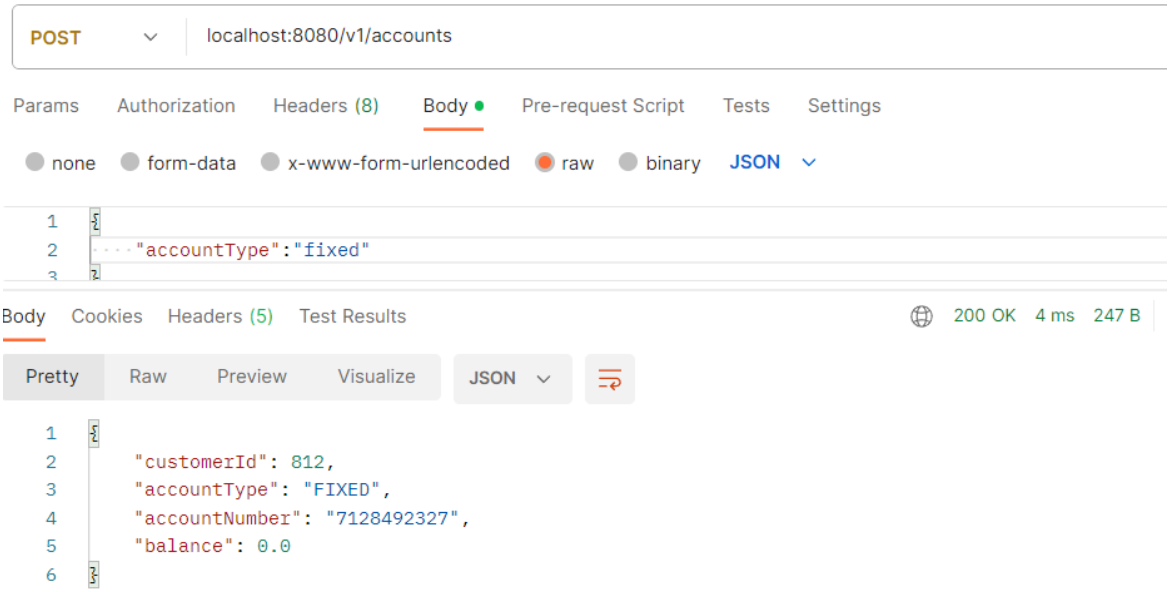
Service: Account Service and Their Implementation

```java
public interface AccountService {

    /**
     * Creates a new savings account for the customer.
     *
     * @param accountDTO for which account needs to be created
     * @return The created AccountDTO.
     */
    AccountDTO createAccount(AccountDTO accountDTO);

    /**
     * Deposits money to the specified account.
     *
     * @param accountNumber for which money needs to be deposited.
     * @param amount(money) to be deposited.
     * @return updated AccountDTO.
     */
    AccountDTO depositMoney(String accountNumber, double amount);

    /**
     * Withdraws money from the provided account.
     *
     * @param accountNumber from which money needs to be withdrawn.
     * @param amount(money) to be withdrawn.
     * @return updated AccountDTO.
     */
    AccountDTO withdrawMoney(String accountNumber, double amount);

    /**
     * Retrieves the available balance for the provided account.
     *
     * @param accountNumber for which available balance needs to be fetched.
     * @return available balance as a double value.
     */
    double getAvailableBalance(String accountNumber);

    /**
     * Retrieves the last 10 transactions for the provided account.
     *
     * @param accountNumber for which transactions need to be fetched.
     * @return A list of TransactionDTO representing the last 10 transactions.
     */
    List<TransactionDTO> getLast10Transactions(String accountNumber);
}
```

```java
@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountRepository accountRepository;

    // temporary storage, replace with accountRepository for real db in future
    private Map<String, Account> accounts = new HashMap<>();
    private Map<String, List<Transaction>> transactions = new HashMap<>();

    @Override
    @Transactional
    public AccountDTO createAccount(AccountDTO dto) {

        try {
            Account account = new Account();

            String accountNumber = AccountUtils.generateRandomAccountNumber();
            double initialBalance = 0.0;
            account.setId(100 + (long) (Math.random() * 899));
            account.setAccountNumber(accountNumber);
            account.setBalance(initialBalance);
            AccountType accountType = AccountType.valueOf(dto.getAccountType().toUpperCase());
            account.setAccountType(accountType);

            // Validate allowed values for accountType: SAVINGS, FIXED, CURRENT
            switch (accountType) {
            case SAVINGS, FIXED, CURRENT -> accounts.put(accountNumber, account);

            default -> throw new AccountServiceException("Invalid account type: " + dto.getAccountType());
            }

            return AccountMapper.mapAccountToDTO(account);
        } catch (Exception e) {
            throw new AccountServiceException("Invalid input: " + e.getMessage());
        }

    }
```

```java
@RestController
@RequestMapping("/v1/accounts")
public class AccountResource {

    @Autowired
    private AccountService accountService;

    @PostMapping
    public ResponseEntity<AccountDTO> createAccount(@Valid @RequestBody AccountDTO accountDto) {
        AccountDTO createdAccount = accountService.createAccount(accountDto);
        return ResponseEntity.ok(createdAccount);
    }

    @PostMapping("/{accountNumber}/deposit")
    public ResponseEntity<AccountDTO> depositMoney(@PathVariable String accountNumber,
            @RequestBody TransactionDTO transactionDTO) {
        AccountDTO updatedAccount = accountService.depositMoney(accountNumber, transactionDTO.getAmount());
        return ResponseEntity.ok(updatedAccount);
    }

    @PostMapping("/{accountNumber}/withdraw")
    public ResponseEntity<AccountDTO> withdrawMoney(@PathVariable String accountNumber,
            @RequestBody TransactionDTO transactionDTO) {
        AccountDTO updatedAccount = accountService.withdrawMoney(accountNumber, transactionDTO.getAmount());
        return ResponseEntity.ok(updatedAccount);
    }

    @GetMapping("/{accountNumber}/balance")
    public ResponseEntity<Double> getAvailableBalance(@PathVariable String accountNumber) {
        double balance = accountService.getAvailableBalance(accountNumber);
        return ResponseEntity.ok(balance);
    }

    @GetMapping("/{accountNumber}/transactions")
    public ResponseEntity<List<TransactionDTO>> getLast10Transactions(@PathVariable String accountNumber) {
        List<TransactionDTO> transactions = accountService.getLast10Transactions(accountNumber);
        return ResponseEntity.ok(transactions);
    }
```

Test via the Postman client:



Test by code:

```
@SpringBootTest
public class AccountResourceTest {

    @Autowired
    private AccountResource accountResource;

    @Test
    public void testCreateAccount() {
        AccountDTO dto = new AccountDTO();

        // Test with valid account type: SAVINGS, expected to pass
        dto.setAccountType("savings");
        ResponseEntity<AccountDTO> accountDTO = accountResource.createAccount(dto);

        assertNotNull(accountDTO);
        assertEquals(10, accountDTO.getBody().getAccountNumber().length());
        assertEquals(0.0, accountDTO.getBody().getBalance());

    }
```

Transaction Service Implementation Screenshots

```
@Entity
@Table(name = "transaction")
public class Transaction {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String accountNumber;

    @Positive(message = "Amount must be positive")
    private Double amount;

    @Enumerated(EnumType.STRING)
    private TransactionType type;

    private LocalDateTime transactionTime;
```

```java
public interface TransactionService {
    /**
     * Creates a new transaction for the provided account.
     *
     * @param transactionDTO contains transaction details.
     * @return The created TransactionDTO.
     */
    TransactionDTO createTransaction(TransactionDTO transactionDTO);

    /**
     * Retrieves the last 10 transactions for the specified account.
     *
     * @param accountNumber for which transactions need to be fetched.
     * @return A list of TransactionDTO representing the last 10 transactions.
     */
    List<TransactionDTO> getLast10Transactions(String accountNumber);
}
```

```java
@Service
public class TransactionServiceImpl implements TransactionService {

    @Autowired
    private TransactionRepository transactionRepository;

    @Override
    public TransactionDTO createTransaction(TransactionDTO transactionDTO) {
        // Implement the logic to create a new transaction and save it to the repository
        // Map the TransactionDTO to the entity and vice versa
        // Return the created TransactionDTO
        return transactionDTO;
    }

    @Override
    public List<TransactionDTO> getLast10Transactions(String accountNumber) {
        // Implement the logic to retrieve the last 10 transactions for the provided
        // Map the entities to TransactionDTOs
        // Return the list of TransactionDTOs
        return null;
    }
```

```java
@RestController
@RequestMapping("/v1/transactions")
public class TransactionResource {

    @Autowired
    private TransactionService transactionService;

    @PostMapping
    public ResponseEntity<TransactionDTO> createTransaction(@RequestBody TransactionDTO transactionDTO) {
        TransactionDTO createdTransaction = transactionService.createTransaction(transactionDTO);
        return ResponseEntity.ok(createdTransaction);
    }

    @GetMapping("/{accountNumber}/last10")
    public ResponseEntity<List<TransactionDTO>> getLast10Transactions(@PathVariable String accountNumber) {
        List<TransactionDTO> transactions = transactionService.getLast10Transactions(accountNumber);
        return ResponseEntity.ok(transactions);
    }

}
```