# Corexfin Banking Solutions: Data Modeling Design

This document outlines a comprehensive data modeling design for Corexfin Banking Solutions, focusing on the persistence layer and data flow. It details the key database tables, corresponding Java entity classes, and Data Transfer Objects (DTOs) essential for the application's core services. This design aims to provide a clear, concise, and technical foundation for developing robust and scalable banking software.

# Database Tables: Simplified Schema

A well-defined database schema is the backbone of any robust banking application. The following tables represent the core entities within Corexfin Banking Solutions, designed for efficiency, integrity, and scalability. Each table includes critical columns, data types, and notes on their purpose and relationships.

| Table | Column | Type | Notes |
|---|---|---|---|
| Bank | bank_id | VARCHAR (PK) | Unique bank identifier |
| | name | VARCHAR | Bank name |
| | domain | VARCHAR | Bank domain (e.g., hexbank.com) |
| | owner_user_id | VARCHAR (FK) | Reference to bank owner user |
| Branch | branch_id | VARCHAR (PK) | Unique branch identifier |
| | bank_id | VARCHAR (FK) | Linked bank |
| | name | VARCHAR | Branch name |
| | address | TEXT | Branch address |

UUID strings are recommended for all primary keys to ensure global uniqueness and prevent conflicts in distributed environments. This approach simplifies data merging and replication across various systems within the Corexfin ecosystem.

# User and Account Tables

The User and Account tables are central to managing customer data and financial records. The schema accounts for various user roles, account types, and critical status flags to ensure comprehensive data management.

| Table | Column | Type | Notes |
|---|---|---|---|
| User | user_id | VARCHAR (PK) | Unique user id |
| | username | VARCHAR | Login username/email |
| | password | VARCHAR | Hashed password |
| | roles | VARCHAR ARRAY | Roles (BANK_OWNER, MANAGER, CUSTOMER) |
| | bank_id | VARCHAR (FK) | Associated bank |
| | branch_id | VARCHAR (FK) | Branch (nullable for bank owner) |
| | profile | JSONB | User profile info (name, phone, email, etc.) |
| | active | BOOLEAN | Active status |
| Account | account_id | VARCHAR (PK) | Unique account id |
| | bank_id | VARCHAR (FK) | Associated bank |
| | branch_id | VARCHAR (FK) | Branch |
| | customer_id | VARCHAR (FK) | User who owns the account |
| | account_type | VARCHAR | SAVINGS, CURRENT, etc. |
| | balance | DECIMAL | Current balance |
| | status | VARCHAR | ACTIVE, SUSPENDED, CLOSED |
| | internet_banking_enabled | BOOLEAN | IB enabled flag |

The profile column in the User table, defined as JSONB, provides flexibility for storing diverse user attributes without modifying the schema for every new field. This approach aligns with agile development practices, allowing for rapid iteration on user profiles.

# Transaction and Internet Banking Request Tables

These tables manage critical financial operations and user requests. The Transaction table captures all financial movements, while the InternetBankingRequest table handles the lifecycle of online banking access requests. Proper indexing on foreign keys and timestamp columns will be essential for query performance and auditing.

| Table | Column | Type | Notes |
|---|---|---|---|
| InternetBankingRequest | request_id | VARCHAR (PK) | Unique request id |
| | account_id | VARCHAR (FK) | Account linked |
| | customer_id | VARCHAR (FK) | Customer requesting IB |
| | status | VARCHAR | PENDING, APPROVED, REJECTED |
| | requested_at | TIMESTAMP | Request timestamp |
| | approved_by | VARCHAR (FK) | User who approved |
| | approved_at | TIMESTAMP | Approval timestamp |
| Transaction | transaction_id | VARCHAR (PK) | Unique transaction id |
| | from_account | VARCHAR (FK) | Source account (nullable for deposits) |
| | to_account | VARCHAR (FK) | Destination account (nullable for withdrawals) |
| | amount | DECIMAL | Transaction amount |
| | type | VARCHAR | TRANSFER, DEPOSIT, WITHDRAWAL |
| | status | VARCHAR | PENDING, POSTED, FAILED |
| | created_at | TIMESTAMP | Creation time |

The `from_account` and `to_account` fields in the Transaction table are designed to support various transaction types, including deposits (where `from_account` is null) and withdrawals (where `to_account` is null).

# Entity Classes: Java Representation

Entity classes serve as the direct mapping of database tables to Java objects, forming the foundation of the persistence layer. These classes are typically annotated for use with Object-Relational Mapping (ORM) frameworks like Hibernate in a Spring Boot environment, simplifying database interactions.

### Bank Entity

```
@Entity
public class Bank {
  @Id private String bankId;
  private String name;
  private String domain;
  private String
ownerUserId;
}
```

### Branch Entity

```
@Entity
public class Branch {
  @Id private String
branchId;
  private String bankId;
  private String name;
  private String address;
}
```

### User Entity

```
@Entity
public class User {
  @Id private String userId;
  private String username;
  private String password;
  @ElementCollection(fetch
= FetchType.EAGER)
  private Set<String> roles;
  private String bankId;
  private String branchId;
  @Type(type = "jsonb")
  private String profile;
  private boolean active;
}
```

For the User entity, the `roles` are defined as a `Set<String>` with `FetchType.EAGER`, ensuring that user roles are loaded immediately when a User object is retrieved. This design choice provides flexibility for role management and access control.

# Account, Internet Banking Request, and Transaction Entities

These entity classes represent the core operational aspects of Corexfin Banking Solutions, enabling the management of financial accounts, online banking requests, and all transaction records. Each class includes specific fields that mirror the database schema, ensuring seamless data persistence.

## Account Entity

```
@Entity
public class Account {
  @Id private String accountId;
  private String bankId;
  private String branchId;
  private String customerId;
  private String accountType;
  private BigDecimal balance;
  private String status;
  private boolean internetBankingEnabled;
}
```

## InternetBankingRequest Entity

```
@Entity
public class InternetBankingRequest {
  @Id private String requestId;
  private String accountId;
  private String customerId;
  private String status;
  private Instant requestedAt;
  private String approvedBy;
  private Instant approvedAt;
}
```

## Transaction Entity

```
@Entity
public class Transaction {
  @Id private String transactionId;
  private String fromAccount;
  private String toAccount;
  private BigDecimal amount;
  private String type;
  private String status;
  private Instant createdAt;
}
```

The use of `Instant` for timestamp fields (e.g., `requestedAt`, `approvedAt`, `createdAt`) is crucial for precise, time-zone-agnostic datetime management, which is vital for financial applications that operate globally.

# Data Transfer Objects (DTOs)

DTOs are lightweight objects used to transfer data between different layers of the application, typically between the service layer and the presentation layer. They encapsulate data for specific operations, often exposing a subset of an entity's fields to prevent over-exposure and simplify data payloads. This approach enhances security and performance by sending only necessary data.

### BankDTO

```
public class BankDTO {
  private String bankId;
  private String name;
  private String domain;
  private String ownerUserId;
}
```

### UserDTO

```
public class UserDTO {
  private String userId;
  private String username;
  private Set<String> roles;
  private String bankId;
  private String branchId;
  private Map<String, String> profile;
  private boolean active;
}
```

For UserDTO, the `profile` is represented as a `Map<String, String>`, providing a more convenient format for client-side consumption compared to a raw JSON string. This requires a mapping utility to convert between the `JSONB` in the database/entity and the `Map` in the DTO.

# Account, Internet Banking Request, and Transaction DTOs

Continuing with the DTO definitions, these classes are tailored for specific use cases involving accounts, online banking requests, and transaction details. They facilitate clear communication between services and external interfaces, ensuring that data is consistently formatted and validated before processing.

### AccountDTO

```
public class AccountDTO {
  private String accountId;
  private String bankId;
  private String branchId;
  private String customerId;
  private String accountType;
  private BigDecimal balance;
  private String status;
  private boolean internetBankingEnabled;
}
```

### InternetBankingRequestDTO

```
public class InternetBankingRequestDTO {
  private String requestId;
  private String accountId;
  private String customerId;
  private String status;
  private Instant requestedAt;
  private String approvedBy;
  private Instant approvedAt;
}
```

### TransactionDTO

```
public class TransactionDTO {
  private String transactionId;
  private String fromAccount;
  private String toAccount;
  private BigDecimal amount;
  private String type;
  private String status;
  private Instant createdAt;
}
```

The `BigDecimal` type is used for monetary values (`balance`, `amount`) in both entities and DTOs to ensure precise arithmetic, avoiding floating-point inaccuracies common with `double` or `float` types in financial calculations.

## Key Design Considerations

Several critical considerations underpin this data modeling design to ensure the Corexfin Banking Solutions application is performant, secure, and maintainable.

### Unique Identifiers

Utilise UUID strings (Universally Unique Identifiers) for all primary keys. This approach guarantees uniqueness across distributed systems, simplifies data merging, and eliminates the need for sequence generators, which can be a bottleneck in high-throughput environments. UUIDs also enhance security by making it harder to guess or enumerate records.

### JSONB for Flexible Profiles

Storing user profile information as JSONB in the database offers schema flexibility. This allows for dynamic addition of user attributes without requiring schema migrations, supporting agile development and evolving business requirements. Proper indexing on JSONB fields will be essential for query performance.

### Role Management

Representing user roles as a `Set<String>` within the User entity provides a robust and extensible mechanism for managing access permissions. This facilitates fine-grained access control and integration with authentication and authorization frameworks, ensuring that users only access resources commensurate with their privileges.

These considerations collectively contribute to a data model that is not only functional but also adaptable to future demands and scalable for a growing user base and transaction volume.

# Conclusion and Recommendations

This data modeling design provides a robust foundation for Corexfin Banking Solutions. The defined database schema, entity classes, and DTOs establish a clear and efficient data flow, crucial for building a secure and high-performance banking application.

1

### Implement Mapping Libraries

Utilise mapping libraries such as MapStruct or ModelMapper to automate the conversion between entity classes and DTOs. This reduces boilerplate code, minimises errors, and significantly improves development efficiency.

2

### Generate DDL Scripts

Develop Data Definition Language (DDL) scripts for all tables. This ensures consistent database schema deployment across different environments (development, testing, production) and facilitates version control of the database structure.

3

### Full Stack Snippets

Consider generating full Spring Boot entity, repository, and service code snippets based on this data model. This will accelerate the development of the persistence and service layers, allowing the team to focus on business logic and user interface development.

By adhering to these recommendations, Corexfin Banking Solutions can achieve a highly organised, maintainable, and scalable application architecture, ready to meet the demands of modern digital banking.