Finance Management System

Coding Task 3

Service Provider Interface/Abstract class Implementation

Implement the Task 2 interface in a class called FinanceRepositoryImpl in package dao.

The FinanceRepositoryImpl class in the dao package provides the concrete implementation for the IFinanceRepository interface. It acts as a bridge between the application and the database, managing all CRUD operations related to users, expenses, and suggestions. This implementation utilizes JDBC for database connectivity via the DBConnUtil class and emphasizes modularity, error handling, and data integrity, especially with transactional operations. The methods are tailored to handle expense tracking efficiently by supporting filtering, categorization, reporting, and summary generation functionalities.

```
package dao;

import entity.Expense;
import entity.Suggestion;
import entity.User;
import util.DBConnUtil;

import java.sql.*;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

public class FinanceRepositoryImpl implements IFinanceRepository {
```

Task 3.1 createUser(User user)

This method is responsible for adding a new user to the system. It takes a User object containing the user's details like username, email, and password. The main purpose of this method is to **register a new user** and persist their details in the database. It executes an INSERT SQL query into the users table, and upon successful insertion, it retrieves the auto-generated user ID using getGeneratedKeys() and sets it back to the user object. This ensures that the user object in memory now has the same ID as the one in the database, enabling further operations like adding expenses under this user.

```java
@Override
public boolean createUser(User user) {
    String query = "INSERT INTO users (username, email, password) VALUES (?, ?, ?)";

    try (Connection con = DBConnUtil.getConnection();
         PreparedStatement pst = con.prepareStatement(query, Statement.RETURN_GENERATED_KEYS)) {

        pst.setString( parameterIndex: 1, user.getUsername());
        pst.setString( parameterIndex: 2, user.getEmail());
        pst.setString( parameterIndex: 3, user.getPassword());

        int rows = pst.executeUpdate();

        if (rows > 0) {
            ResultSet rs = pst.getGeneratedKeys();
            if (rs.next()) {
                int userId = rs.getInt( columnIndex: 1);
                user.setUserId(userId);
            }
            return true;
        }

    } catch (SQLException e) {
        System.out.println("Error in createUser: " + e.getMessage());
    }
    return false;
}
```

Task 3.2 loginUser(String email, String password)

This method is used for **user authentication**. It takes an email and password, validates them against the database, and if a match is found, it returns the corresponding User object. Internally, it performs a SELECT query on the users table where email and password match. If successful, it constructs a User object with data fetched from the database. This method is essential to restrict access to only valid users and maintain a secure login system.

```java
public User loginUser(String email, String password) {
    String query = "SELECT * FROM users WHERE email = ? AND password = ?";

    try (Connection con = DBConnUtil.getConnection();
         PreparedStatement pst = con.prepareStatement(query)) {

        pst.setString( parameterIndex: 1, email);
        pst.setString( parameterIndex: 2, password);

        ResultSet rs = pst.executeQuery();

        if (rs.next()) {
            User user = new User();
            user.setUserId(rs.getInt( columnLabel: "user_id"));
            user.setUsername(rs.getString( columnLabel: "username"));
            user.setEmail(rs.getString( columnLabel: "email"));
            user.setPassword(rs.getString( columnLabel: "password"));
            return user;
        }

    } catch (SQLException e) {
        System.out.println("Error in loginUser: " + e.getMessage());
    }

    return null;
}
```

Task 3.3 addExpense(Expense expense)

The purpose of this method is to **add a new expense entry** into the system. It receives an Expense object which contains fields such as user ID, category ID, amount, description, and date. It performs an INSERT operation into the expenses table, binding each field to the prepared statement. Once inserted, it retrieves the generated expense ID and sets it in the Expense object. This method helps users track their spending and categorize their financial data.

```
@Override
public Expense addExpense(Expense expense) {
    String query = "INSERT INTO expenses (user_id, category_id, amount, date, description) VALUES (?, ?, ?, ?, ?)";

    try (Connection con = DBConnUtil.getConnection();
         PreparedStatement pst = con.prepareStatement(query, Statement.RETURN_GENERATED_KEYS)) {

        pst.setInt( parameterIndex: 1, expense.getUserId());
        pst.setInt( parameterIndex: 2, expense.getCategoryId());
        pst.setDouble( parameterIndex: 3, expense.getAmount());
        pst.setDate( parameterIndex: 4, expense.getDate());
        pst.setString( parameterIndex: 5, expense.getDescription());

        int rows = pst.executeUpdate();

        if (rows > 0) {
            ResultSet rs = pst.getGeneratedKeys();
            if (rs.next()) {
                expense.setExpenseId(rs.getInt( columnIndex: 1));
            }
            return expense;
        }

    } catch (SQLException e) {
        System.out.println("Error in addExpense: " + e.getMessage());
    }

    return null;
}
```

Task 3.4 getAllExpensesByUserId(int userId)

This method is designed to **fetch all expenses of a specific user**. It runs a SELECT query with a JOIN on the categories table to fetch not just the IDs but also the names of the categories. This enhances user experience by showing meaningful category names along with expenses. It loops through the result set and constructs a list of Expense objects, which is then returned. This helps users to get a complete overview of their expenses in one go.

```
@Override
public List<Expense> getAllExpensesByUserId(int userId) {
    List<Expense> expenses = new ArrayList<>();
    String query = "SELECT e.*, c.category_name FROM expenses e " +
            "JOIN expenseCategories c ON e.category_id = c.category_id " +
            "WHERE e.user_id = ?";

    try (Connection con = DBConnUtil.getConnection();
         PreparedStatement pst = con.prepareStatement(query)) {

        pst.setInt( parameterIndex: 1, userId);
        ResultSet rs = pst.executeQuery();
```

```
        while (rs.next()) {
            Expense expense = new Expense();
            expense.setExpenseId(rs.getInt( columnLabel: "expense_id"));
            expense.setUserId(rs.getInt( columnLabel: "user_id"));
            expense.setCategoryId(rs.getInt( columnLabel: "category_id"));
            expense.setAmount(rs.getDouble( columnLabel: "amount"));
            expense.setDate(rs.getDate( columnLabel: "date"));
            expense.setDescription(rs.getString( columnLabel: "description"));
            expense.setCategoryName(rs.getString( columnLabel: "category_name"));

            expenses.add(expense);
        }

    } catch (SQLException e) {
        System.out.println("Error in getAllExpensesByUserId: " + e.getMessage());
    }

    return expenses;
}
```

Task 3.5 getExpensesByDateRange(int userId, Date fromDate, Date toDate)

This method allows users to **filter their expenses based on a date range**, such as viewing monthly or weekly spending. It takes a user ID and two dates (start and end), and runs a SQL query using BETWEEN on the date field. This is helpful for time-based analysis and budget tracking. The method constructs Expense objects for each result and returns them as a list for further processing or display.

```
public List<Expense> getExpensesByDateRange(int userId, Date fromDate, Date toDate) {
    List<Expense> expenses = new ArrayList<>();
    String query = "SELECT e.*, c.category_name FROM expenses e " +
            "JOIN expenseCategories c ON e.category_id = c.category_id " +
            "WHERE e.user_id = ? AND e.date BETWEEN ? AND ?";

    try (Connection con = DBConnUtil.getConnection();
         PreparedStatement pst = con.prepareStatement(query)) {

        pst.setInt( parameterIndex: 1, userId);
        pst.setDate( parameterIndex: 2, fromDate);
        pst.setDate( parameterIndex: 3, toDate);

        ResultSet rs = pst.executeQuery();

        while (rs.next()) {
            Expense expense = new Expense();
            expense.setExpenseId(rs.getInt( columnLabel: "expense_id"));
            expense.setUserId(rs.getInt( columnLabel: "user_id"));
            expense.setCategoryId(rs.getInt( columnLabel: "category_id"));
            expense.setAmount(rs.getDouble( columnLabel: "amount"));
            expense.setDate(rs.getDate( columnLabel: "date"));
            expense.setDescription(rs.getString( columnLabel: "description"));
            expense.setCategoryName(rs.getString( columnLabel: "category_name"));
            expenses.add(expense);
        }

    } catch (SQLException e) {
        System.out.println("Error in getExpensesByDateRange: " + e.getMessage());
    }
```

Task 3.6 getExpensesByCategory(int userId, int categoryId)

This method is used when a user wants to **analyze expenses within a specific category** (e.g., Food, Travel, Bills). It filters data from the expenses table by both userId and categoryId and returns matching records. This is useful for tracking how much is being spent in each category and aids in better financial planning.

```
public List<Expense> getExpensesByCategory(int userId, int categoryId) {
    List<Expense> expenses = new ArrayList<>();
    String query = "SELECT e.*, c.category_name FROM expenses e " +
            "JOIN expenseCategories c ON e.category_id = c.category_id " +
            "WHERE e.user_id = ? AND e.category_id = ?";

    try (Connection con = DBConnUtil.getConnection();
         PreparedStatement pst = con.prepareStatement(query)) {

        pst.setInt( parameterIndex: 1, userId);
        pst.setInt( parameterIndex: 2, categoryId);

        ResultSet rs = pst.executeQuery();

        while (rs.next()) {
            Expense expense = new Expense();
            expense.setExpenseId(rs.getInt( columnLabel: "expense_id"));
            expense.setUserId(rs.getInt( columnLabel: "user_id"));
            expense.setCategoryId(rs.getInt( columnLabel: "category_id"));
            expense.setAmount(rs.getDouble( columnLabel: "amount"));
            expense.setDate(rs.getDate( columnLabel: "date"));
            expense.setDescription(rs.getString( columnLabel: "description"));
            expense.setCategoryName(rs.getString( columnLabel: "category_name"));
            expenses.add(expense);
        }

    } catch (SQLException e) {
        System.out.println("Error in getExpensesByCategory: " + e.getMessage());
    }
    return expenses;
}
```

Task 3.7 getExpenseById(int expenseId, int userId)

This method retrieves a **specific expense record** based on its ID and the user ID. It is used when a user wants to **view or edit a particular expense entry**. It ensures that only the owner of the expense (matching user ID) can access it, thus maintaining data privacy. It runs a SELECT query and returns the matched Expense object.

```
public Expense getExpenseById(int expenseId, int userId) {
    String query = "SELECT * FROM expenses WHERE expense_id = ? AND user_id = ?";
    try (Connection con = DBConnUtil.getConnection();
         PreparedStatement pst = con.prepareStatement(query)) {

        pst.setInt( parameterIndex: 1, expenseId);
        pst.setInt( parameterIndex: 2, userId);
        ResultSet rs = pst.executeQuery();

        if (rs.next()) {
            Expense e = new Expense();
            e.setExpenseId(rs.getInt( columnLabel: "expense_id"));
            e.setUserId(rs.getInt( columnLabel: "user_id"));
            e.setCategoryId(rs.getInt( columnLabel: "category_id"));
            e.setAmount(rs.getDouble( columnLabel: "amount"));
            e.setDate(rs.getDate( columnLabel: "date"));
            e.setDescription(rs.getString( columnLabel: "description"));
            return e;
        }

    } catch (SQLException e) {
        System.out.println("Error in getExpenseById: " + e.getMessage());
    }
    return null;
}
```

Task 3.8 updateExpense(Expense expense)

This method allows users to **edit or update an existing expense**. It takes an Expense object that includes updated values along with the expense ID and user ID. The method runs an UPDATE SQL query, changing fields like amount, category, date, and description. It's crucial for maintaining accurate records, especially if the user needs to correct or adjust an entry.

```
@Override
public boolean updateExpense(Expense expense) {
    String query = "UPDATE expenses SET category_id = ?, amount = ?, date = ?, description = ? WHERE expense_id = ? AND user_id = ?";

    try (Connection con = DBConnUtil.getConnection();
         PreparedStatement pst = con.prepareStatement(query)) {

        pst.setInt( parameterIndex: 1, expense.getCategoryId());
        pst.setDouble( parameterIndex: 2, expense.getAmount());
        pst.setDate( parameterIndex: 3, expense.getDate());
        pst.setString( parameterIndex: 4, expense.getDescription());
        pst.setInt( parameterIndex: 5, expense.getExpenseId());
        pst.setInt( parameterIndex: 6, expense.getUserId());

        int rows = pst.executeUpdate();
        return rows > 0;

    } catch (SQLException e) {
        System.out.println("Error in updateExpense: " + e.getMessage());
    }

    return false;
}
```

Task 3.9 deleteExpense(int expenseId, int userId)

This method enables users to **delete a specific expense entry**. It checks both the expense ID and user ID before performing the deletion to ensure that users cannot delete others' data. It executes a DELETE SQL query, and if successful, the expense is removed permanently. This keeps the database clean and secure.

```
@Override
public boolean deleteExpense(int expenseId, int userId) {
    String query = "DELETE FROM expenses WHERE expense_id = ? AND user_id = ?";

    try (Connection con = DBConnUtil.getConnection();
         PreparedStatement pst = con.prepareStatement(query)) {

        pst.setInt( parameterIndex: 1, expenseId);
        pst.setInt( parameterIndex: 2, userId);

        int rows = pst.executeUpdate();
        return rows > 0;

    } catch (SQLException e) {
        System.out.println("Error in deleteExpense: " + e.getMessage());
    }

    return false;
}
```

Task 3.10 getCategoryWiseReport(int userId, Date start, Date end)

This method generates a **summary report of expenses grouped by category** within a specified time range. It runs a SELECT query with GROUP BY category_id, summing the amounts to show how much was spent in each category. It's a powerful tool for financial analysis and helps users identify their major areas of spending.

```
@Override
public Map<String, Double> getCategoryWiseReport(int userId, Date start, Date end) {
    Map<String, Double> report = new LinkedHashMap<>();
    String query = "SELECT c.category_name, SUM(e.amount) AS total_amount " +
            "FROM expenses e JOIN expensecategories c ON e.category_id = c.category_id " +
            "WHERE e.user_id = ? AND e.date BETWEEN ? AND ? " +
            "GROUP BY c.category_name ORDER BY total_amount DESC";

    try (Connection con = DBConnUtil.getConnection();
         PreparedStatement pst = con.prepareStatement(query)) {

        pst.setInt( parameterIndex: 1, userId);
        pst.setDate( parameterIndex: 2, start);
        pst.setDate( parameterIndex: 3, end);

        ResultSet rs = pst.executeQuery();

        while (rs.next()) {
            String categoryName = rs.getString( columnLabel: "category_name");
            double total = rs.getDouble( columnLabel: "total_amount");
            report.put(categoryName, total);
        }

    } catch (SQLException e) {
        System.out.println("Error in getCategoryWiseReport: " + e.getMessage());
    }

    return report;
```

Task 3.11 getMonthlySummary(int userId, Date start, Date end)

This method provides a **monthly breakdown of expenses** for a user between two dates. It uses SQL date functions like MONTH() and YEAR() to group expenses by month and year. The result is a list showing total amounts spent per month, which helps users spot patterns and track spending trends over time.

```
@Override
public Map<String, Double> getMonthlySummary(int userId, Date start, Date end) {
    Map<String, Double> summary = new LinkedHashMap<>();
    String query = "SELECT DATE_FORMAT(date, '%Y-%m') AS month, SUM(amount) AS total " +
            "FROM expenses WHERE user_id = ? AND date BETWEEN ? AND ? " +
            "GROUP BY month ORDER BY month";

    try (Connection con = DBConnUtil.getConnection();
         PreparedStatement pst = con.prepareStatement(query)) {...} catch (SQLException e) {
        System.out.println("Error in getMonthlySummary: " + e.getMessage());
    }

    return summary;
}
```

Task 3.12 addSuggestion(Suggestion suggestion)

This method allows users to **submit feedback or suggestions** to the application. It takes a Suggestion object containing the user ID and suggestion text, and inserts it into a suggestions table. This is important for improving the application based on real user input, making it more user-centric and reliable.

```
@Override
public boolean addSuggestion(Suggestion suggestion) {
    String query = "INSERT INTO suggestions (user_id, suggestion_text) VALUES (?, ?)";

    try (Connection con = DBConnUtil.getConnection();
         PreparedStatement pst = con.prepareStatement(query)) {...} catch (SQLException e) {
        System.out.println("Error in addSuggestion: " + e.getMessage());
    }
    return false;
}
```

Task 3.13 deleteUserAndExpenses(int userId)

This method ensures **safe and complete deletion of a user's data**. It starts a transaction, first deleting all expenses associated with the user and then deleting the user entry. Using transaction management, it guarantees that both deletions occur together (or neither at all if an error occurs). This preserves **data integrity** and ensures the system is left in a consistent state.

```
@Override
public boolean deleteUserAndExpenses(int userId) {
    String deleteExpenses = "DELETE FROM expenses WHERE user_id = ?";
    String deleteUser = "DELETE FROM users WHERE user_id = ?";

    try (Connection con = DBConnUtil.getConnection()) {...} catch (SQLException e) {
        System.out.println("Connection error in deleteUserAndExpenses: " + e.getMessage());
    }

    return false;
}
```

Conclusion

The FinanceRepositoryImpl class provides a comprehensive and efficient data access layer for a finance management system. By handling various operations like user authentication, expense tracking, and generating insightful reports, it ensures seamless interaction with the database while maintaining data consistency and integrity. The inclusion of filtering, categorization, and summary features makes this implementation robust and user-centric.
