United International University

Assignment 1: Transaction API Implementation Assignment

Project Overview

This is an simple **Pos system** built using Spring Boot 3.5.8 with Java 17. The project follows a layered architecture pattern with clear separation of concerns:

- **Entity Layer**: Database entities (JPA)
- **DTO Layer**: Data Transfer Objects for request/response
- **Repository Layer**: Data access layer (Spring Data JPA)
- **Service Layer**: Business logic
- **Controller Layer**: REST API endpoints

Current Status

The **User API** has been fully implemented and serves as a reference for this assignment. You can find the complete implementation in: -

```
src/main/java/com/example/OrderManagement/entity/UserEntity.java
src/main/java/com/example/OrderManagement/dto/UserDto.java
src/main/java/com/example/OrderManagement/dto/UserResponse.java
src/main/java/com/example/OrderManagement/repository/UserRepository.java
src/main/java/com/example/OrderManagement/service/UserService.java
src/main/java/com/example/OrderManagement/controller/UsersController.java
```

The **Transaction API** is incomplete. Only the `TransactionEntity` class exists. Your task is to implement the complete Transaction API following the same pattern as the User API.

Understanding the Transaction Entity

The `TransactionEntity` class is already defined with the following structure:

```java
@Entity
@Table(name = "transactions")
public class TransactionEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;
    private Double amount;
    @ManyToOne
    @JoinColumn(name = "user_id")
    private UserEntity userEntity;   // Getters and setters
}
```

Key Points: - Each transaction has an `id`, `amount`, and is associated with a `userEntity` - The relationship is `@ManyToOne` - meaning multiple transactions can belong to one user - The foreign key column in the database is `user_id`

Assignment Questions

Question 1: Create TransactionDto Class

Task: Create a `TransactionDto` class in the `dto` package following the same pattern as `UserDto`.

Requirements: - Create the file: `src/main/java/com/example/OrderManagement/dto/TransactionDto.java`

Include the following fields:

- `amount` (Double) - represents the transaction amount

- `userId` (Integer) - represents the ID of the user who owns this transaction

- Include a constructor that takes both parameters

- Include getter and setter methods for both fields

- Follow the same coding style as `UserDto` (including the fluent setter pattern if used)

Hint: Look at `UserDto.java` to understand the pattern. Your DTO should be similar but adapted for transaction data.

Question 2: Create TransactionResponse Class

Task: Create a `TransactionResponse` class in the `dto` package following the same pattern as `UserResponse`.

Requirements: - Create the file: `src/main/java/com/example/OrderManagement/dto/TransactionResponse.java` - Include the following fields:

- `id` (Integer)

- transaction ID

- `amount` (Double)

- `userInfo`(UserResponse) – User info of the associated user

- Include getter and setter methods for all fields

- Follow the same coding style as `UserResponse`

Hint: This class is used to return transaction data in API responses. It should include the ID (which is generated by the database) unlike the DTO.

Question 3: Create TransactionRepository Interface

Task: Create a `TransactionRepository` interface in the `repository` package following the same pattern as `UserRepository`.

Requirements: - Create the file: `src/main/java/com/example/OrderManagement/repository/TransactionRepository.java` - Extend `JpaRepository<TransactionEntity, Integer>`

- Add the following custom query methods:

- `List<TransactionEntity> findAllByUserEntityId(Integer userId)`

- Find all transactions for a specific user

- Optionally, you can add more query methods like:

- `List<TransactionEntity> findAllByAmountGreaterThan(Double amount)`

- Find transactions with amount greater than a value

Hint: - Look at `UserRepository.java` to see how custom queries are defined - Spring Data JPA can automatically generate query methods based on method names - You can also use `@Query` annotation for custom JPQL queries if needed

---

Question 4: Implement TransactionService Class

Task: Create a `TransactionService` class in the `service` package following the same pattern as `UserService`.

Requirements: - Create the file: `src/main/java/com/example/OrderManagement/service/TransactionService.java` - Annotate the class with `@Service` - Inject `TransactionRepository` and `UserRepository` using constructor injection - Implement the following methods:

4.1: createTransaction(TransactionDto transactionDto)

- **Purpose:** Create a new transaction
- **Validation:**
  - Throw `InvalidClassException` if `amount` is null
  - Throw `InvalidClassException` if `amount` is negative or zero
  - Verify that the user with the given `userId` exists (use `UserRepository.findById()`)

    - If user doesn't exist, throw `FileNotFoundException` with message "User not found"
- **Logic:**
    - Create a new `TransactionEntity`
    - Set the amount from DTO
    - Fetch the `UserEntity` using `userId` from the repository
    - Set the `userEntity` in the transaction
    - Save the transaction using `TransactionRepository.save()`

4.2: getAllTransactions()

- **Purpose:** Retrieve all transactions
- **Return:** `List<TransactionResponse>`
- **Logic:**
    - Fetch all transactions from repository
    - Convert each `TransactionEntity` to `TransactionResponse`
    - Return the list

4.3: getTransactionById(Integer id)

- **Purpose:** Retrieve a transaction by its ID
- **Return:** `TransactionResponse`
- **Exception:** Throw `FileNotFoundException` with message "Transaction not found" if transaction doesn't exist
- **Logic:**
    - Find transaction by ID
    - Convert to `TransactionResponse`
    - Return the response

4.4: getTransactionsByUserId(Integer userId)

- **Purpose:** Retrieve all transactions for a specific user
- **Return:** `List<TransactionResponse>`
- **Logic:**
    - Use the custom repository method to find transactions by user ID
    - Convert each entity to response
    - Return the list (empty list if no transactions found)

4.5: updateTransaction(Integer id, TransactionDto transactionDto)

- **Purpose:** Update an existing transaction
- **Exception:** Throw `ChangeSetPersister.NotFoundException` if transaction doesn't exist
- **Validation:** Same validation as `createTransaction` (amount validation, user existence)
- **Logic:**
    - Find the transaction by ID

- Update the amount
- If userId is provided and different, update the userEntity (fetch new user, verify existence)
- Save the updated transaction

4.6: deleteTransaction(Integer id)

- **Purpose:** Delete a transaction by ID
- **Exception:** Throw `ChangeSetPersister.NotFoundException` if transaction doesn't exist
- **Logic:**
  - Find the transaction by ID
  - Delete it using repository

Hint: - Study `UserService.java` carefully to understand the pattern - Pay attention to how exceptions are thrown and handled - Remember to convert between Entity and DTO/Response objects - Always validate input data before processing

---

Question 5: Implement TransactionsController Class

Task: Create a `TransactionsController` class in the `controller` package following the same pattern as `UsersController`.

Requirements: - Create the file: `src/main/java/com/example/OrderManagement/controller/TransactionsController.java` - Annotate the class with `@RestController` - Use `@RequestMapping("transactions")` for the base path - Inject `TransactionService` using constructor injection - Implement the following REST endpoints:

5.1: POST /transactions

- **Method:** `createTransaction(@RequestBody TransactionDto transactionDto)`
- **Return:** `ResponseEntity<Void>`
- **Exception:** Handle `InvalidClassException` and `FileNotFoundException`
- **Status Code:** 200 OK on success

5.2: GET /transactions/list

- **Method:** `getAllTransactions()`
- **Return:** `ResponseEntity<List<TransactionResponse>>`
- **Status Code:** 200 OK

5.3: GET /transactions/{id}

- **Method:** `getTransactionById(@PathVariable Integer id)`
- **Return:** `ResponseEntity<TransactionResponse>`
- **Exception:** Handle `FileNotFoundException`
- **Status Code:** 200 OK

5.4: GET /transactions/user/{userId}

- **Method:** `getTransactionsByUserId(@PathVariable Integer userId)`
- **Return:** `ResponseEntity<List<TransactionResponse>>`
- **Status Code:** 200 OK

5.5: PUT /transactions/update/{id}

- **Method:** `updateTransaction(@PathVariable Integer id, @RequestBody TransactionDto transactionDto)`
- **Return:** `ResponseEntity<Void>`
- **Exception:** Handle `ChangeSetPersister.NotFoundException`, `InvalidClassException`, `FileNotFoundException`
- **Status Code:** 200 OK on success

5.6: DELETE /transactions/{id}

- **Method:** `deleteTransaction(@PathVariable Integer id)`
- **Return:** `ResponseEntity<Void>`
- **Exception:** Handle `ChangeSetPersister.NotFoundException`
- **Status Code:** 200 OK on success

Hint: - Study `UsersController.java` to understand the exact pattern - Use the same exception handling approach - Follow the same naming conventions for methods - Use appropriate annotations: `@PostMapping`, `@GetMapping`, `@PutMapping`, `@DeleteMapping` - Use `@PathVariable` for path parameters and `@RequestBody` for request body

---

Deliverables Checklist

Before submitting, ensure you have completed all of the following:

- ☐ **TransactionDto.java** created in `dto` package with correct fields and methods
- ☐ **TransactionResponse.java** created in `dto` package with correct fields and methods
- ☐ **TransactionRepository.java** created in `repository` package extending JpaRepository with custom query methods
- ☐ **TransactionService.java** created in `service` package with:
  - ☐ `createTransaction()` method with proper validation
  - ☐ `getAllTransactions()` method
  - ☐ `getTransactionById()` method with exception handling
  - ☐ `getTransactionsByUserId()` method
  - ☐ `updateTransaction()` method with validation
  - ☐ `deleteTransaction()` method with exception handling

- ☐ **TransactionsController.java** created in `controller` package with all 6 endpoints
- ☐ Code compiles without errors
- ☐ Code follows the same pattern and style as User API
- ☐ Proper exception handling implemented
- ☐ Validation logic implemented for transaction creation and updates

---

Testing Requirements

After implementing the API, test your endpoints using Postman, cURL, or any REST client:

Test Cases:

1. **Create Transaction:**
   - Test with valid data (amount > 0, valid userId)
   - Test with null amount (should throw exception)
   - Test with negative amount (should throw exception)
   - Test with invalid userId (should throw exception)
2. **Get All Transactions:**
   - Should return list of all transactions (may be empty)
3. **Get Transaction by ID:**
   - Test with valid ID
   - Test with invalid ID (should throw exception)
4. **Get Transactions by User ID:**
   - Test with valid userId
   - Test with userId that has no transactions (should return empty list)
5. **Update Transaction:**
   - Test with valid data
   - Test with invalid transaction ID (should throw exception)
   - Test with invalid userId (should throw exception)
6. **Delete Transaction:**
   - Test with valid ID
   - Test with invalid ID (should throw exception)

Sample Test Data:

First, create a user using the User API:

```
POST http://localhost:7320/users
{
  "name": "John Doe",
  "phoneNumber": "1234567890"
}
```

Then create transactions:

```
POST http://localhost:7320/transactions
{
  "amount": 100.50,
  "userId": 1
}
```

Code Style Guidelines

1. **Follow existing patterns:** Your code should look like it was written by the same developer who wrote the User API
2. **Naming conventions:**
   - Classes: PascalCase (e.g., `TransactionService`)
   - Methods: camelCase (e.g., `getTransactionById`)
   - Variables: camelCase (e.g., `transactionDto`)
3. **Package structure:** Maintain the same package structure as User API
4. **Annotations:** Use the same annotation style and placement
5. **Exception handling:** Follow the same exception types and messages pattern
6. **Comments:** Add comments only where necessary for clarity

---

Submission Guidelines

1. **Code Quality:**
   - Code must compile without errors
   - Code must follow Java naming conventions
   - Code must be consistent with existing codebase style
2. **Functionality:**
   - All endpoints must work correctly
   - All validations must be implemented
   - Exception handling must be proper
3. **Testing:**
   - Test all endpoints before submission
   - Document any issues or limitations
4. **File Organization:**
   - All files should be in the correct packages
   - File names should match class names exactly

Additional Notes

- The application runs on port **7320** (as configured in `application.yaml`)
- Database: MySQL database named `possystem`

- The relationship between Transaction and User is already established in the entity layer
- Make sure to handle the relationship properly when creating/updating transactions
- When fetching a user for a transaction, always verify the user exists before proceeding

---

Questions?

If you have any questions about the assignment:

1. Review the User API implementation carefully - it contains all the patterns you need

2. Check Spring Boot and Spring Data JPA documentation

3. Consult with your instructor if needed

Good luck with your implementation!