Banking Transactions API Task

You are tasked with creating a simple banking transactions API. The API should handle user accounts and transactions between these accounts. Your goal is to design and implement a RESTful API using your choice of programming language and framework. The API should perform the following operations:

1. **Create a new user account with an initial balance.**
2. **Transfer funds from one account to another.**
3. **Retrieve the transaction history for a given account.**

Requirements

1. **Design an appropriate endpoint path for each operation.**
2. **Implement dependency injection (or equivalent) to manage your service layers.**
3. **Use DTOs (Data Transfer Objects) to pass data between layers (Controller, Service, Repository).**
4. **Ensure that all endpoints validate the input data.**
5. **Handle errors gracefully (e.g., insufficient funds, invalid account IDs).**
6. **Store data in memory (a simple data structure will suffice; no need for a database).**

Implementation Guidelines

1. **Three-Layer Design: Separate the application into Controller, Service, and Repository layers (or their equivalents in your chosen framework).**
2. **Dependency Injection: Use dependency injection (or equivalent) to manage your service and repository layers.**
3. **Error Handling: Implement proper error handling for cases such as insufficient funds or invalid account IDs.**
-

Submission Guidelines

- **Provide the source code in a GitHub repository.**
- **Include a README file with instructions on how to build and run the application.**
- **Document any assumptions made during the implementation.**

Additional Information

Suggested Frameworks by Language

- **Java: Spring Boot**
- **Python: Flask or Django**
- **JavaScript/TypeScript: Express.js (Node.js)**
- **C#: ASP.NET Core**
- **Ruby: Ruby on Rails**
- **PHP: Laravel**
- **Go: Gin**
- **Kotlin: Ktor**
- **Rust: Actix or Rocket**

Choose the language and framework you are most comfortable with or best suits the task, and follow the guidelines to implement the API.

dependency injection (DI) frameworks or mechanisms for each suggested language and framework:

Java: Spring Boot

- **Dependency Injection**: Spring Framework's built-in DI mechanism.

Python

- **Flask**: Flask-Injector
- **Django**: Django does not have built-in DI, but you can use third-party libraries like django-injector.

JavaScript/TypeScript: Express.js (Node.js)

- **Dependency Injection**: InversifyJS

C#: ASP.NET Core

- **Dependency Injection**: Built-in with ASP.NET Core.

Ruby: Ruby on Rails

- **Dependency Injection**: Dry-rb's dry-container and dry-auto_inject.

PHP: Laravel

- **Dependency Injection**: Built-in with Laravel.

Go: Gin

- **Dependency Injection**: Google Wire or Fx.

Kotlin: Ktor

- **Dependency Injection**: Koin or Dagger (via Kotlin).

Rust: Actix or Rocket

- **Dependency Injection**: There's no standard DI framework, but you can use service locators or manual DI patterns.

Three Layer Design:

Java: Spring Boot

- **Presentation Layer**: Controllers (REST Controllers).
- **Business Logic Layer**: Services.
- **Data Access Layer**: Repositories (using Spring Data JPA or similar).

Python: Flask or Django

- **Presentation Layer**: Routes or Views.
- **Business Logic Layer**: Services (can be implemented as separate modules or classes).
- **Data Access Layer**: Repositories or Models.

JavaScript/TypeScript: Express.js (Node.js)

- **Presentation Layer**: Route Handlers.
- **Business Logic Layer**: Services (implemented as separate modules or classes).
- **Data Access Layer**: Repositories or Data Access Objects (DAOs).

C#: ASP.NET Core

- **Presentation Layer**: Controllers.
- **Business Logic Layer**: Services.
- **Data Access Layer**: Repositories (using Entity Framework or similar).

Ruby: Ruby on Rails

- **Presentation Layer**: Controllers.
- **Business Logic Layer**: Services (can be implemented as service objects).
- **Data Access Layer**: Models.

PHP: Laravel

- **Presentation Layer**: Controllers.
- **Business Logic Layer**: Services (can be implemented as service classes).
- **Data Access Layer**: Repositories or Models (using Eloquent ORM).

Go: Gin

- **Presentation Layer**: Handlers.
- **Business Logic Layer**: Services.
- **Data Access Layer**: Repositories.

Kotlin: Ktor

- **Presentation Layer**: Controllers or Route Handlers.
- **Business Logic Layer**: Services.
- **Data Access Layer**: Repositories (using Exposed or similar).

Rust: Actix or Rocket

- **Presentation Layer**: Handlers.
- **Business Logic Layer**: Services.
- **Data Access Layer**: Repositories.