Banking API Documentation

Joaquin Gonzalez Alvarez

November 20, 2024

1 Introduction

This document serves as a detailed description of a simple banking application developed as part of a technical test for the company Valesa. The goal of the application is to showcase the skills in backend development, particularly in creating a functional system using the Go programming language.

2 Application Mechanisms

The application is a basic banking system developed in Go using the `Gin` framework to create a REST API. The system allows users to perform operations on bank accounts, carry out transactions, and transfer money between accounts. Data is stored in memory using Go data structures, without any persistent database.

2.1 Data Structures

The application uses two main structures:

- **Account**: Holds information about each bank account, including the account ID, the owner's name, and the balance.
- **Transaction**: Represents a financial transaction (either a deposit or withdrawal), associated with an account and a specific amount.

2.2 Controllers and Features

The API controllers handle client requests. There are controllers for creating accounts, retrieving account information, performing transactions, and transferring money between accounts.

The data access logic (in-memory storage) is separated into an independent module to ensure better organization and code reuse.

2.3 Error Handling

The application manages common errors such as:

- Insufficient balance when attempting to withdraw funds.
- Transactions with negative or zero amounts.
- Transfer attempts between the same account.
- Access attempts to non-existent accounts.

3 Installing the Application

To install and run this application on your local machine, follow the steps below:

3.1 Prerequisites

- **Go** version 1.18 or higher. If not installed, follow the instructions at Go Downloads.
- An appropriate development environment for Go.
- `Git` installed on your system to clone the repository.

3.2 Installation Steps

1. Clone the repository to your local machine:

```
git clone https://github.com/iamximo/ValesaChallange.git
```

2. Navigate to the project directory:

```
cd ValesaChallange
```

3. Install the required dependencies:

```
go mod tidy
```

4. Build and run the server:

```
go run main.go
```

The server will be available at `http://localhost:8080`.

3.3 Running the Application

Once the application is running, it will be accessible on port 8080 by default. You can send HTTP requests to the API using tools such as `Postman`, `curl`, or integrate it into your frontend.

4 Curl Commands for API Requests

The following are the `curl` commands corresponding to the Postman collection for interacting with the Banking API.

4.1 (POST /accounts)

```
curl -X POST http://localhost:8080/accounts \
     -H "Content-Type: application/json" \
     -d '{
           "owner": "JOQUIN",
           "initial_balance": 0
         }'
```

4.2 (GET /accounts/id)

```
curl -X GET http://localhost:8080/accounts/990c804f-cb50-4994-8df3-2b1787bf87ae
```

4.3 (GET /accounts)

```
curl -X GET http://localhost:8080/accounts
```

4.4 (POST /accounts/id/transactions)

```
curl -X POST http://localhost:8080/accounts/990c804f-cb50-
4994-8df3-2b1787bf87ae/transactions \
     -H "Content-Type: application/json" \
     -d '{
           "type": "deposit",
           "amount": 6
         }'
```

4.5 (GET /accounts/id/transactions)

```
curl -X GET http://localhost:8080/accounts/8e26b8b1-59e1-4
ef7-8dc1-764f8eea55da/transactions
```

4.6 (POST /transfer)

```
curl -X POST http://localhost:8080/transfer \
     -H "Content-Type: application/json" \
     -d '{
           "from_account_id": "8e26b8b1-59e1-4ef7-8dc1-764f8eea55da",
           "to_account_id": "67f67506-55d7-4d58-9e64-8280d8904a7d",
           "amount": 0.1
         }'
```

5 Using the REST API

The application exposes several endpoints (URLs) to interact with the API. Below is a description of these endpoints along with usage examples.

5.1 Request and Response Formats

All requests and responses are handled in `JSON` format.

5.2 Usage Examples

5.2.1 Create an Account

Request:

```
POST /accounts
Content-Type: application/json

{
  "owner": "Joaquin",
  "initial_balance": 1000.0
}
```

Response:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "id": "f06b22bb-d9f0-42a3-9b9b-b234f9f329d9",
  "owner": "Joaquin",
  "balance": 1000.0
}
```

5.2.2 Get an Account by ID

Request:

```
GET /accounts/{id}
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": "f06b22bb-d9f0-42a3-9b9b-b234f9f329d9",
  "owner": "Joaquin",
  "balance": 1000.0
}
```

5.2.3 Perform a Transaction

Request:

```
POST /accounts/{id}/transactions
Content-Type: application/json

{
  "type": "deposit",
  "amount": 500.0
}
```

Response:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "id": "acfe2d8f-fb83-455b-bce3-f93e2f4c02db",
  "accountId": "f06b22bb-d9f0-42a3-9b9b-b234f9f329d9",
  "type": "deposit",
  "amount": 500.0,
  "timestamp": "2024-11-20T10:32:00Z"
}
```

5.2.4 Transfer Between Accounts

Request:

```
POST /transfer
Content-Type: application/json

{
  "from_account_id": "f06b22bb-d9f0-42a3-9b9b-b234f9f329d9",
  "to_account_id": "8ab21f7f-476e-4873-b032-9d183d0fb2c1",
  "amount": 200.0
}
```

Response:

```
HTTP/1.1 201 Created
Content-Type: application/json

[
  {
    "id": "df82307c-d1bc-4bdb-99f3-e28f3b7f0638",
    "accountId": "f06b22bb-d9f0-42a3-9b9b-b234f9f329d9",
    "type": "withdrawal",
    "amount": 200.0,
    "timestamp": "2024-11-20T10:35:00Z"
  },
  {
    "id": "e11e9f7d-e5ab-4f8a-b60e-6744e334c121",
    "accountId": "8ab21f7f-476e-4873-b032-9d183d0fb2c1",
    "type": "deposit",
    "amount": 200.0,
    "timestamp": "2024-11-20T10:35:00Z"
  }
]
```

6 Running Tests

To run the tests for the Banking API, follow these steps:

1. Navigate to the project directory where the Go module is located:

```
cd ValesaChallange
```

2. Run the tests with the following command:

```
go test ./...
```

6.1 Resetting the Storage Between Tests

To ensure that tests do not affect each other, it reset the in-memory storage before each test. The storage can be reset using the `Reset()` function in the `storage` package.

7 Future Improvements

To ensure data persistence using a database, I would add a new abstraction layer to separate business logic from the data storage (currently unified in ‘storage.go‘). This approach would improve the maintainability and scalability.

Additionally, this change would address potential delays caused by the use of mutexes, as database systems or ORM packages often handle concurrency more efficiently.

Finally, I would implement pagination system to avoid fetching all the data at once, which is especially important when dealing with large datasets. This would enhance performance andreduce memory usage.