Banking API

Summary

Create an API for managing banking accounts.

Exercise 1 - Create a bank account

In this exercise you will implement an API to create a bank account.

Requirements

- You can create a bank account.
- Each bank account has a unique account number.
- Account numbers are generated automatically.
- New bank accounts start with a balance of 0.

API

Endpoint to create a bank account:

```
POST /account
```

Request body is a JSON describing the new account, e.g.:

For example:

```
{
    "name": "Mr. Black"
}
```

Response body is a JSON describing the new account, e.g.:

```
{
    "account-number": 1,
    "name": "Mr. Black",
```

```
    "balance": 0
}
```

Exercise 2 - View a bank account

In this exercise you will extend the API with functionality to view a bank account.

Requirements

- You can retrieve an existing bank account.

API

Endpoint

```
GET /account/:id
```

The response is a JSON describing the account, e.g.:

```
{
    "account-number": 1,
    "name": "Mr. Black",
    "balance": 0
}
```

Exercise 3 - Deposit money to an account

In this exercise you will extend the API with functionality to add money to a bank account.

Requirements

- You can deposit money to an existing bank account.
- You can only deposit a positive amount of money.

API

Endpoint

```
POST /account/:id/deposit
```

Request body is a JSON describing the amount of money to deposit, e.g.:

```
{
    "amount": 100
}
```

Response body is a JSON describing the new situation, e.g.:

```
{
    "account-number": 1,
    "name": "Mr. Black",
    "balance": 100
}
```

Exercise 4 - Withdraw money from an account

In this exercise you will extend the API with functionality to withdraw money from a bank account.

Requirements

- You can withdraw money from an existing bank account.
- You can only withdraw a positive amount of money.
- The resulting balance should not fall below zero.

API

Endpoint

```
POST /account/:id/withdraw
```

Request body is a JSON describing the amount of money to withdraw, e.g.:

```
{
    "amount": 5
}
```

Response body will be a JSON describing the new situation, e.g.:

```
{
    "account-number": 1,
    "name": "Mr. Black",
    "balance": 95
}
```

Exercise 5 - Transfer money between accounts

In this exercise you are will extend the API with functionality to move money between accounts.

Requirements

- You can send money from one existing account to another existing account.
- You cannot send money from account to itself.
- The resulting balance of the sending account should not fall below zero.

API

Endpoint

```
POST /account/:id/send
```

Request body is a JSON describing the amount of money to send and the destination, e.g.:

```
{
    "amount": 50,
    "account-number": 800
}
```

Response body is a JSON describing the situation of the sending account, e.g.:

```
{
    "account-number": 1,
    "name": "Mr. Black",
    "balance": 95
}
```

Exercise 6 - Retrieve account audit log (advanced)

In this exercise you will implement functionality to retrieve the audit log of an account.

Requirements

- You can retrieve the audit log of an account.
- The audit log consists of records describing the events on the account.
- An audit record has the following fields:
  - `sequence`: incrementing transaction sequence number
  - `debit`: amount of money that was removed
  - `credit`: amount of money that was added
  - `description`: describes the action that took place. Example values:
  - "withdraw" (a money withdraw)
  - "deposit" (a money deposit)
  - "send to #900" (a money transfer to account 900)
  - "receive from #800" (a money transfer from account 800)

API

Endpoint to retrieve the audit log:

```
GET /account/:id/audit
```

The endpoint returns the audit log records in reverse chronological order.

Assuming the following sequence of transactions:

- deposit $100 to account #1
- transfer $5 from account #1 to account #900
- transfer $10 from account #800 to account #1
- withdraw $20 from account #1

The endpoint would return the following JSON:

```
[
    {
        "sequence": 3,
        "debit": 20,
        "description": "withdraw"
    },
    {
        "sequence": 2,
        "credit": 10,
        "description": "receive from #800"
    },
    {
        "sequence": 1,
        "debit": 5,
        "description": "send to #900"
    },
    {
        "sequence": 0,
        "credit": 100,
        "description": "deposit"
    }
]
```

Exercise 7 - Thread Safety (advanced)

You probably did not think too much about thread safety. But what would happen in your API if you send 1000 concurrent request to deposit $1 to account A? What should happen is that in the end the balance of the account is exactly $1000 higher than before but does your API work like that?

In this exercise you will make sure that all your endpoints are thread safe. Think about the following:

- Are your account numbers really unique?
- Does deposit work correctly with 1000 concurrent calls?
- Does withdraw work correctly with 1000 concurrent calls?
- Does send work correctly with 1000 concurrent calls?

- Is the audit log correctly maintained with concurrent calls?

Requirements

- The API works correctly in the face of multiple concurrent calls.

Exercise 8 - Persistence (advanced)

Maybe you kept all accounts in memory which is fine as long as the API is running. You can make the API more robust by persisting the account a file or a database.

In this exercise you will implement persistence of the accounts so that stopping and starting the API will no longer be a problem. When implementing the persistence you have to think again how to make sure that your API keeps on working correctly in the face of many concurrent calls. E.g. does the API work correctly with 1000 concurrent withdraw calls?

Requirements

- The API maintains its state after a restart.