The bank accounts book project

1. A brief outline of the application - target view

The book account book is the base application to use structured and well-defined information about bank accounts as a helper. The application could be used by users in a simple way to save, edit and re-use others' bank accounts. That kind of software can be implemented for all types of software (PC's application, mobile or web/cloud-based application). The final milestone for this project could be the software application as an API, which could be used by mobile applications or web applications.

The pros for this kind of software (cloud-based API), is multiplatform usage, which means that frontend applications could be prepared for mobile or web purposes. Accounts and data will be fully compatible because of a single source of software and database.

The high-level architecture of a target project

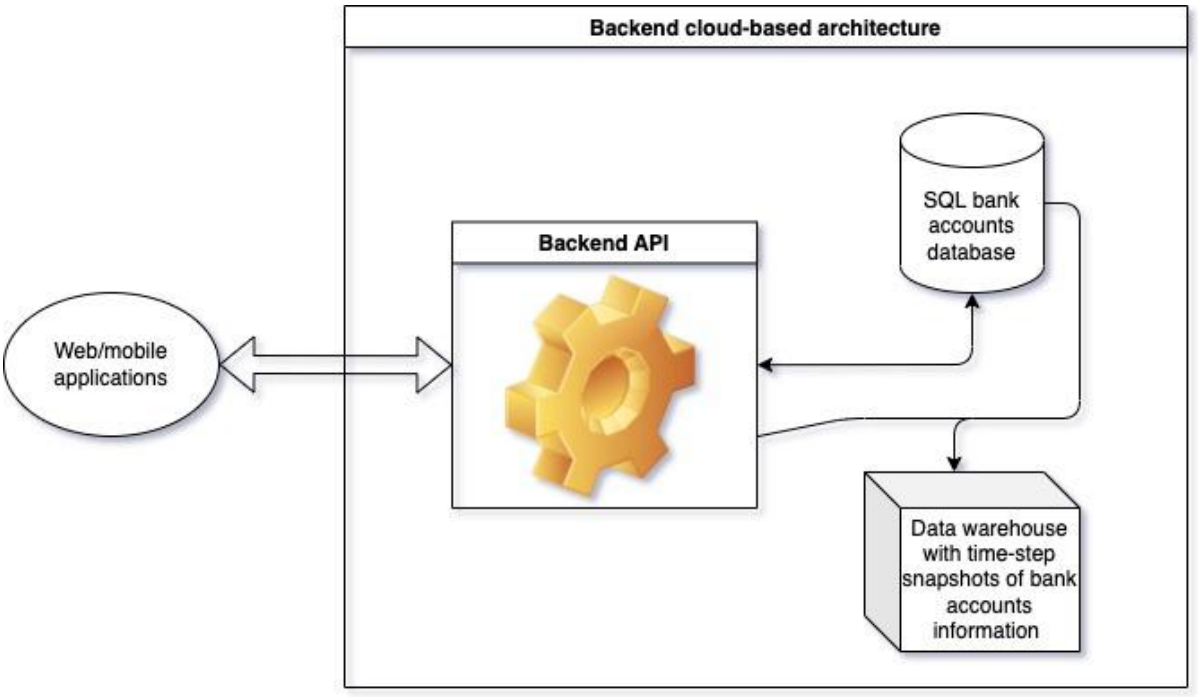

Fig. 1: The high-level architecture of project

Whole architecture is based on 3 sources:

- Backend API - software part of the project, REST API for web/mobile applications
- SQL bank accounts database - a place for storing all information about bank accounts and linked information for all users (authorizing with backend API)

- Data warehouse with time-step snapshots of bank accounts information - a place for storing daily/weekly/monthly snapshot of the whole database for monitoring, analysis or restoring usage.

For the knowledge of data structures and algorithms purposes, It will be described as not web-based software with an internal database system (without using external SQL databases). The project will be based on single application architecture with an in-memory database and processing on this kind of software.

2. Design of data structures and algorithms

In this project the most important thing is well design algorithm for saving, reading and modifying bank accounts information. For this software project, the best option is to use the object-oriented structure of data, which means that a single bank account could be an object of a defined class with all information as property.

Example BankAccount class properties definition:

```
class BankAccount:

    def __init__(self, recipient, recipient_address, account_address, short_name):
        self._recipient = recipient
        self._recipient_address = recipient_address
        self._account_address = account_address
        self._short_name = short_name
```

This single object stores all important information about a single bank account for the project's purposes. (This is a brief definition of class and its properties, without getter and setters method which are not obligatory for project outline).

Book of bank accounts

The in-memory for our project could be separated into two structures:

- List of all bank accounts sorted alphabetically
- Stack of bank accounts - "used recent"

Insertion process

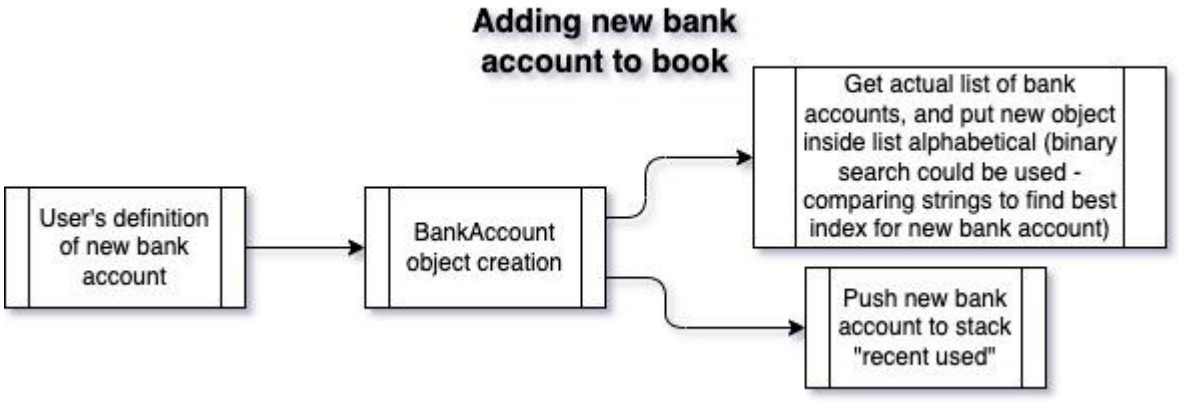

Fig. 2: New bank account insertion process

As in the figure above, the insertion process is separated into two subprocesses after the created instance of BankAccount class:

1. Saving object to Book (list of all bank accounts sorted alphabetically). This process has to be proceed using some kind of search algorithm. The best option is a binary search on strings (short_name property in BankAccount class instance). For searching the best index for our new bank account, binary search is one of the solutions for this problem (Fig. 3). After finding the best place for a new object, the program can put it on the list or create a new list based on the old one (Fig. 4).

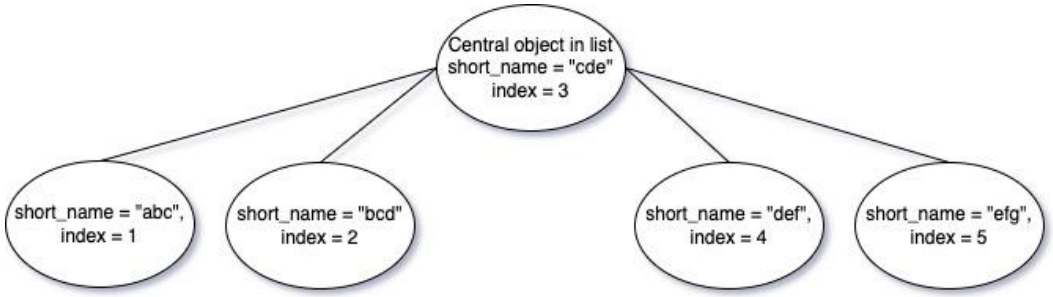

Fig.3. List of bank accounts in binary tree representation

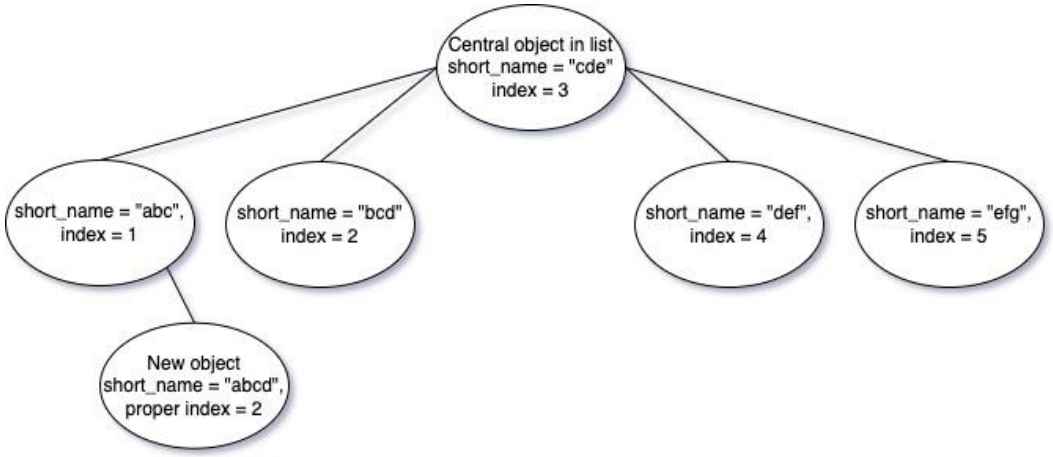

Fig. 4 List of bank accounts in binary tree representation with the founded index for a new account

2. Pushing new object to the stack of “recent” BankAccount objects.

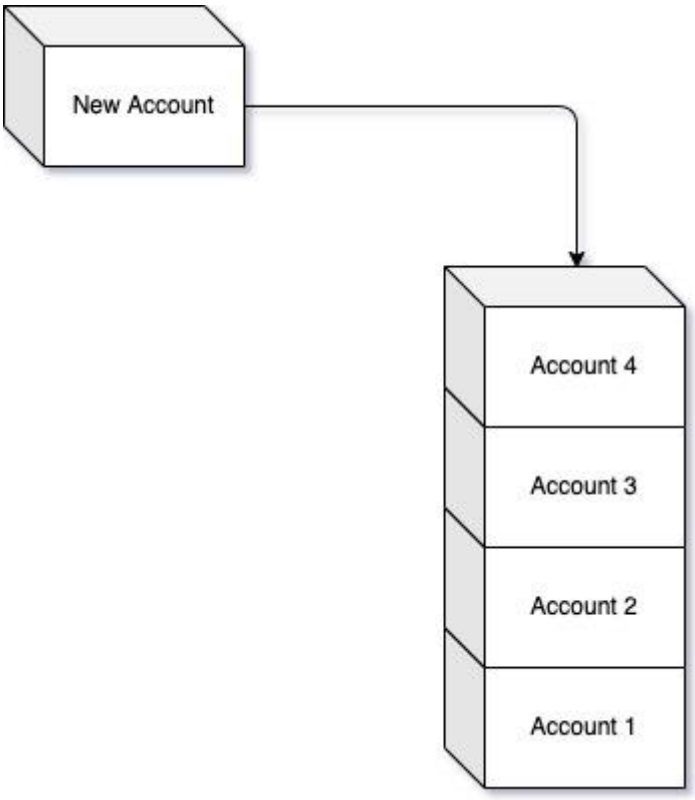

Fig. 5 A stack of recent BankAccount objects (numbers are presented to graphical separation - items are not sorted with id or name)

Searching accounts

All searching methods excepting searching by short-name for this project, is designed as loop iteration searching. It means that searching by recipient_address or account_address has to be done in iteration search.

```
search_address = "..." # search phrase provided by user
target_account = None
for account in accounts:
    if account._account_address == search_address:
        target_account = account
        break

if target_account:
    print("There is no account with address: " + search_address)
else:
    # Returning information about target_account
```

Searching account by short_name could be done in the way presented on Fig.3 and Fig. 4, because application stores alphabetically sorted by short_name database of all accounts filled in the insertion process.

Recent bank accounts

This is part of the application that could be used most often by users. The stack saved in memory perfectly fits the idea of recently used bank accounts by the user. At the start order of poped accounts are in order of their insertion. In the situation of using a single account (getting its information, copying etc.) stack pops an element to the used one and fill itself again in the same order without a selected item, and pushes this used item on the top of the stack. This whole process is presented below.

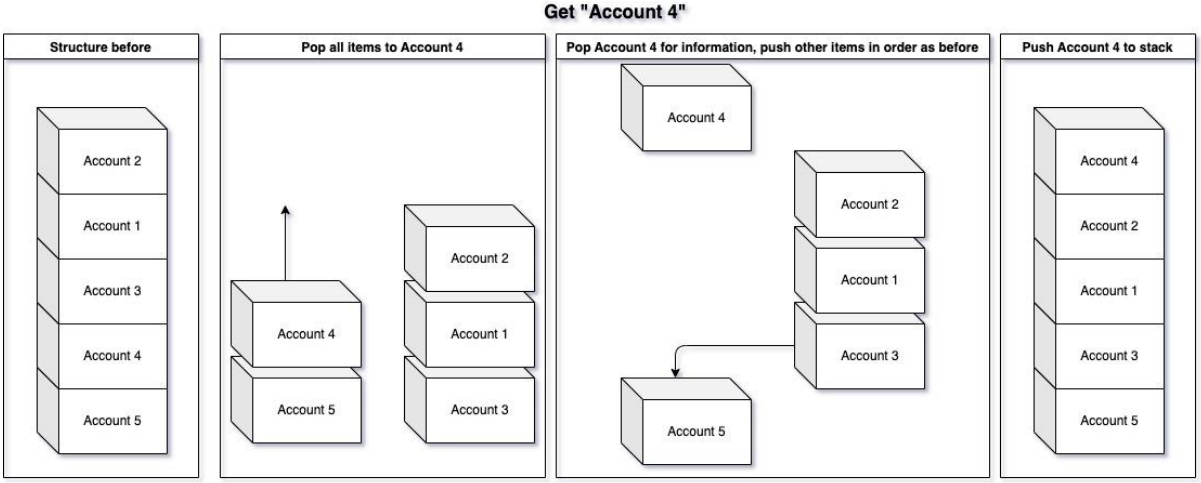

Fig. 6 Process of getting “recent” account from the stack

3. Test plan

Test plan should be separated into memory-based structures testing and correctness of data returned by the application.

1. **Memory-based structure testing**
   These tests have to contain integration tests with the data. The result of the tests should show, that mostly insertion and “recent” bank account searching does not cause data loss. In these test validation have to check these kind of cases:
   a. Checking status of list of data before and after insertion of new bank account (assertions for new length of the list - N + 1; and correct index of inserted object - alphabetical order)
   b. For Python language should be tested types of objects inside list - trying to insert new bank account with different object type than BankAccount should throw the exception or break the process **before** integrating with memory. Languages as Java or Kotlin won’t let to this.

    c. Searching for recently used accounts should not lose any data in the process shown on Fig. 6.

2. **Correctness of data**
   These part of e.g. unit tests should improve the correctness of data returned by the application after searching.
    a. Binary search by short_name should return the proper object for a given phrase.
    b. Part of the project, which is responsible for returning bank account's data in a structured way, should also be tested with unit tests, e.g. saved account address should be formatted before returning etc.

Future work

In this project is a lot of work to improve better usability and user's experience with usage of algorithm and data structures. The examples:

- history of usage (when, how many times)
- add connection about accounts (e.g. 2 accounts can be owned by one person)