BFCD test-code

Introduction

This document serves as the README for the test-code, which can be found at: `test-code`.

This project is developed in .NET 8, utilizing OpenAPI (Swagger-UI) for testing and development purposes.

Upon running the project, Swagger-UI will appear in the browser, accessible via port 7071 (https://localhost:7071).

Answer to extra questions

How will you design/organize the micro services for your API product?

Designing microservices for a retail banking API product entails decomposing the functionality into smaller, cohesive, and loosely coupled services. These services should be capable of independent development, deployment, and scaling.

I opted to implement the project using a layered architecture design, also known as onion architecture. This choice was driven by the retail banking sector's emphasis on customer-centric services, which are integral to customers' daily lives. By organizing the system into layers, the implementation and maintenance of these services become more manageable.

How will you break down the business requirements into user stories?

Breaking down business requirements into user stories starts with identifying and defining specific features or functionalities from the perspective of end-

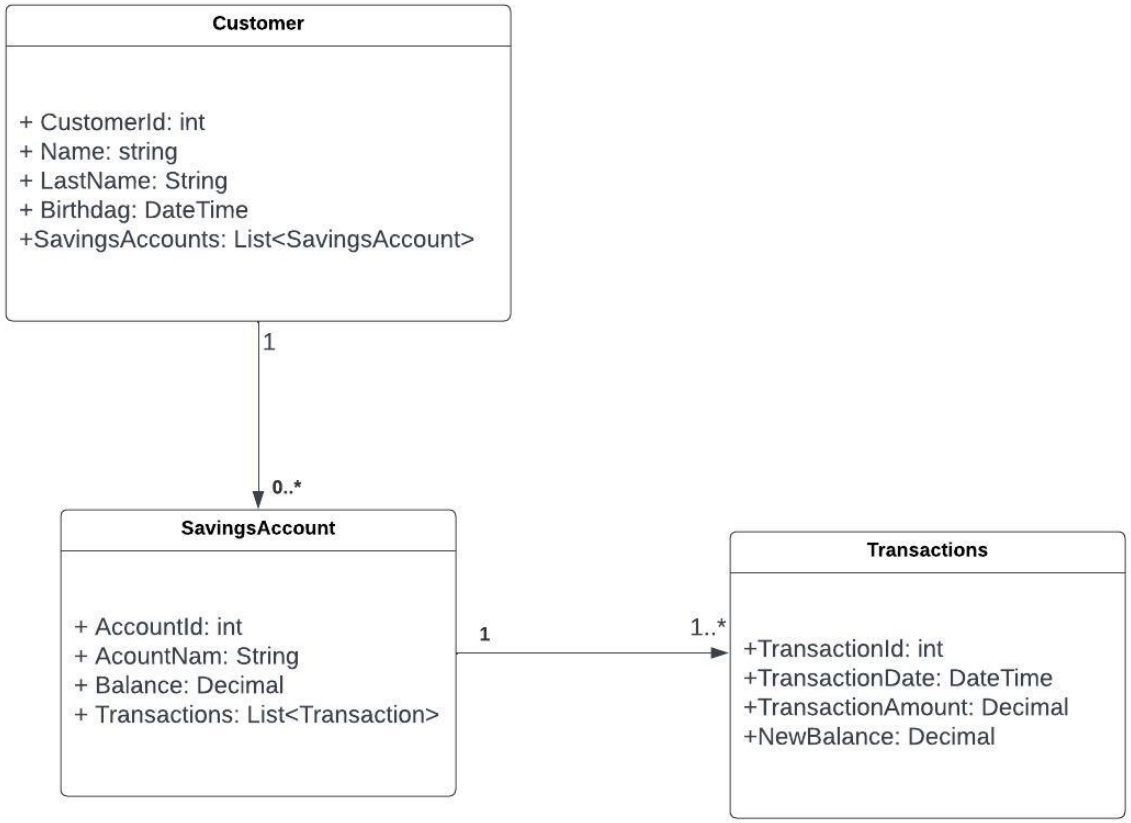
users. These functionalities are determined based on user input, output, and the main actions expected to be performed.

Each functionality or a group of functionalities that share the same domain can be defined as a microservice. These microservices encapsulate specific business capabilities and can be developed, deployed, and scaled independently.

Different services can integrate to form a cohesive system, allowing for the combination of various functionalities to deliver a comprehensive solution. This modular approach facilitates easier maintenance, scalability, and flexibility in the overall system architecture.

Data model

There are 3 entities in designed domain as can be seen in the following diagram:



Repository

For this project, InMemory-repository approach has been utilized. Due to the absence of an actual database, various services have been implemented alongside repository implementations.

API integration

The solution can be tested on Swagger-UI, where various requests are implemented in the respective controllers. The decision on whether to use a GET or POST API-requests was made based on changes expected in the database(active repository). Security was not prioritized in this decision-making process.

In real-world scenarios, for sensitive information such as a 'customer ID', which could be a CPR-number, only POST requests should be used to protect the secrecy of the information in URLs and logs.

Deploy and UI guide

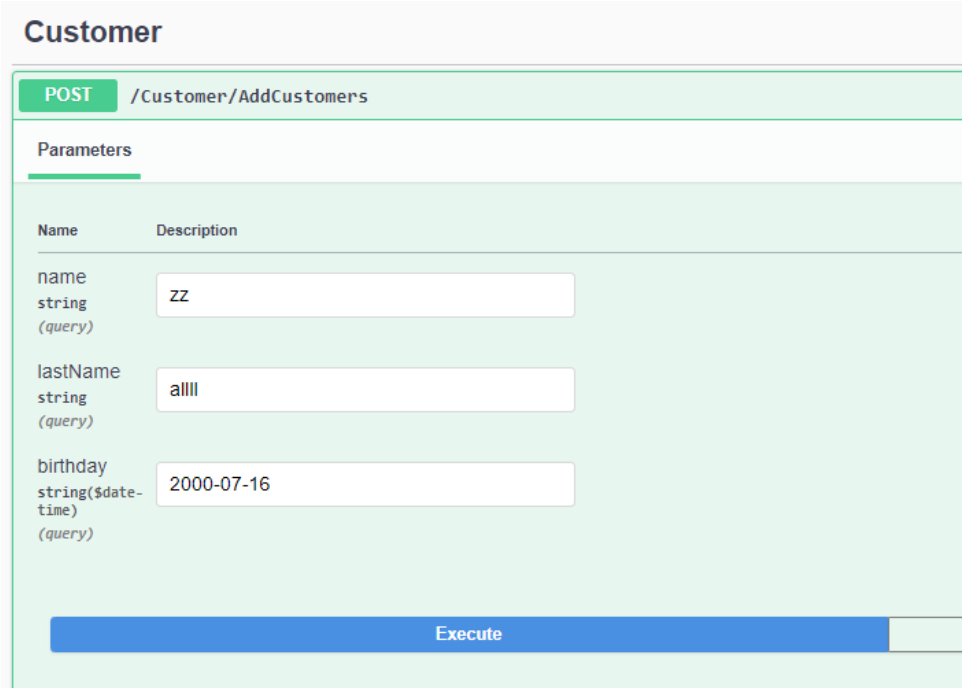
In this section, a guide through various functionalities is provided.

Different functionalities are provided for customer and savings account. The important API’s are as follow:

Add Customers

First you can create customer to have some test data in the repository.

▼ Swagger-UI

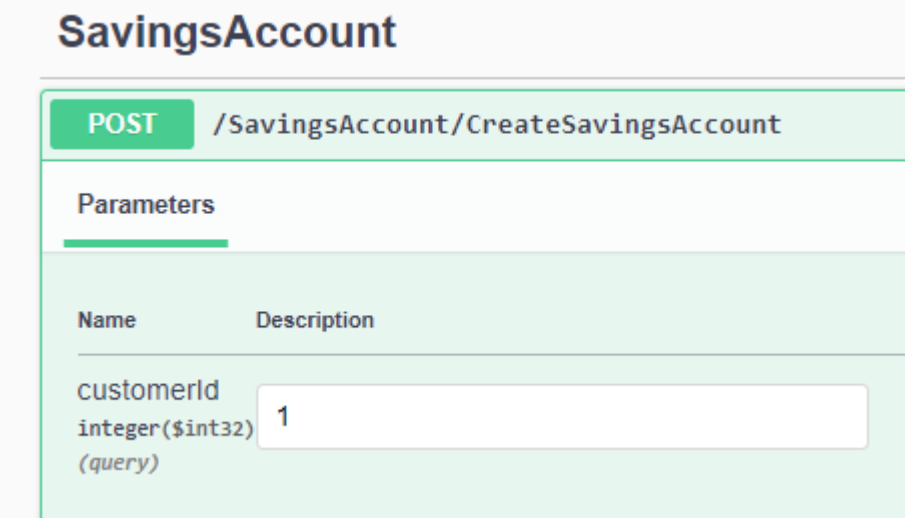


Create savings account

Creating savings account in obtained of 2 parts of input:

- The first one is the customer ID, which is the owner of savings Account.
- The next part is the name and the amount of savings account. You do not need to fill transactions info. Transaction is going to be created and saved automatically.

▼ Swagger-UI

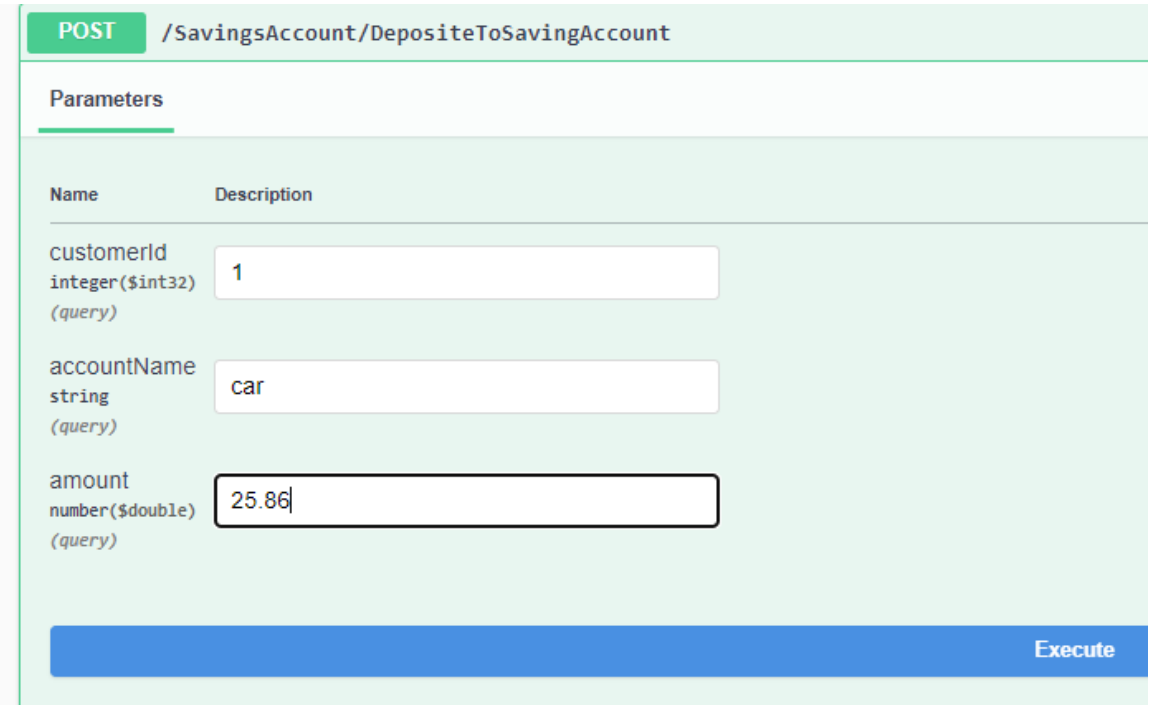


```
{
  "accountId": 0,
  "acountName": "Home",
  "balance": 100,
  "transactions": [
    {
      "transactionId": 0,
      "transactionDate": "2024-05-12T09:54:31.889Z",
      "transactionAmount": 0,
      "newBalance": 0
    }
  ]
}
```


Deposit/withdraw to savings account

By providing the customer ID, savings account name, and amount, you can deposit or withdraw from the savings account. In transactions, positive/negative amounts indicate deposits/withdrawals, respectively.

▼ Swagger-UI



Get all customers

In the end to see all customers, their savings account and all of transaction for each account, the response can be extracted from ‘GetAllCustomers’ end-point.

▼ Response example


```
[
  {
    "customerId": 1,
    "name": "sara",
    "lastName": "aa",
    "birthdag": "1991-07-16T00:00:00",
    "savingsAccounts": [
      {
        "accountId": 1,
        "acountName": "car",
        "balance": 77488.9864,
        "transactions": [
          {
            "transactionId": 1,
            "transactionDate": "2024-05-12T12:26:01.2985
32+02:00",
            "transactionAmount": 78255.86,
            "newBalance": 78255.86
          },
          {
            "transactionId": 2,
            "transactionDate": "2024-05-12T12:27:04.8142
752+02:00",
            "transactionAmount": 6.65,
            "newBalance": 78262.51
          },
          {
            "transactionId": 3,
            "transactionDate": "2024-05-12T12:27:28.4001
817+02:00",
            "transactionAmount": -7.65,
            "newBalance": 78254.86
          },
```

```
          {
            "transactionId": 4,
            "transactionDate": "2024-05-12T12:27:32.9572
513+02:00",
            "transactionAmount": -765.8736,
            "newBalance": 77488.9864
          }
        ]
      }
    ]
  },
  {
    "customerId": 2,
    "name": "s",
    "lastName": "aaa",
    "birthdag": "2000-07-16T00:00:00",
    "savingsAccounts": [
      {
        "accountId": 2,
        "acountName": "home",
        "balance": 78255.86,
        "transactions": [
          {
            "transactionId": 1,
            "transactionDate": "2024-05-12T12:26:38.3154
276+02:00",
            "transactionAmount": 78255.86,
            "newBalance": 78255.86
          }
        ]
      }
    ]
  }
]
```