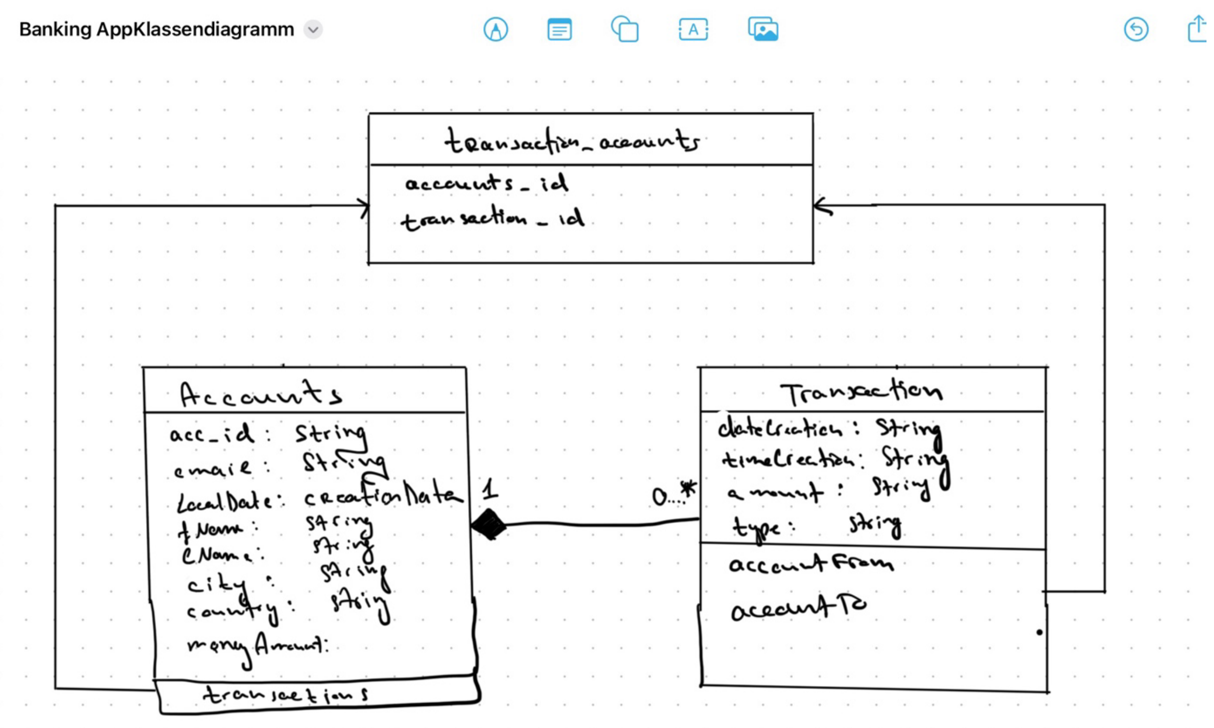
Class-Diagram Banking-App

"Einfache Bankanwendung mit der Möglichkeit, Konten zu verwalten, Geld zwischen Konten zu überweisen und zugehörige Transaktionen zu verwalten."
Mit Hilfe Klassen Diagramm erstelle ich Klasse und deren Beziehungen zwischen Klasse und Objekten, das ermöglicht präzise und mit weniger Fehler bei Code Schreiben machen, außerdem Klassen Diagramm kann man schnell erstellen und Klassen einfach ändern.
Benutze Komposition, die Beziehungen „1" für eine Eins-zu-Eins-Beziehung und "0..*" für eine Eins-zu-Viele-Beziehung darzustellen.



# BankApp :

Bank application
Simple bank application with ability to manage accounts, transfer money between accounts and manage related transactions.

## Environment:
* Java version: 11
* Maven
* Spring Boot version: 3.2.2 RELEASE

## General requirements:

* SQL database should be used for storing accounts and transactions. Please use in-memory MySQL and add it as dependency.
* Application should be able to work in multithreading environment.
* All responses with errors should have the same format.
* New transaction should be created every time when amount of money is changed in account.
* All layers of the application should be covered by unit tests.

## Data:

#### Example of Account data JSON object:
```{
  "id": 1,
  "email": "user@host.com",
  "creationDate": "2024-01-31",
"firstName": "Anna",
  "lastName": "Angel",
  "country": "Germany",
  "city": "Berlin",
  "amountOfMoney": 17000,99,
  "transactions": [5, 8, 32, 6]
  }
```
#### Examples of a Transaction data JSON object:
```

  {
  "id": 1,
  "dateTime": "2024-01-31 23:59:59",
  "type": "Transfer between accounts",
  "accountFrom": "2",
  "accountTo": "4",
  "amount": 17000.99
  }

  {
  "id": 2,
  "dateTime": "2024-01-21 23:59:59",
  "type": "Depositing own money to account",
  "accountFrom": "5",
  "accountTo": "5",
  "amount": 50
  }
```

#### Example of a error response JSON object:
```
 {
   "timestamp": "2024-01-31 23:59:59",
   "problems": ["problem1", "problem2"]
  }
```
## Requirements:

The ```REST``` service must expose the ```/accounts``` and ```/transactions``` endpoints, which allows for managing the collection of account and transaction records in the following way:

```POST``` request to ```/accounts:```

* creates a new account data record
* expects a valid account data object as its body payload, except that it does not have an id property
* adds the given object to the collection and assigns a unique long id to it
* the response code is 201 and the response body is the created record, including its unique id

```GET``` request to ```/accounts:```
* the response code is 200
* the response body is an array of matching records, ordered by their ids in increasing order
* accepts an optional query string parameter, date, in the format ```YYYY-MM-DD,``` for example ```/account?date=2022-11-25.``` When this parameter is present, only the records with the matching date are returned.
* accepts an optional query string parameter, city, and when this parameter is present, only the records with the matching city are returned. The value of this parameter is case insensitive, so "London" and "london" are equivalent. Moreover, it might contain several values, separated by commas (e.g. city=london,Munich), meaning that records with the city matching any of these values must be returned.

* accepts an optional query string parameter, sort, that can take one of two values: either "creationDate" or "-creationDate". If the value is "creationDate", then the ordering is by date in ascending order. If it is "-creationDate", then the ordering is by creationDate in descending order. If there are two records with the same creationDate, the one with the smaller id must come first.

```GET``` request to ```/accounts/<id>:```

* returns a record with the given id
* if the matching record exists, the response code is 200 and the response body is the matching object
* if there is no record in the collection with the given id, the response code is 404

```PATCH``` request to ```/accounts/<id>:```
* updates an account with the given id
* if the matching record exists, the response code is 200 and the response body
* is the updated object if there is no record in the collection with the given id, the response code is 404.

```PUT``` request to ```/accounts?from=<fromId>&to=<toId>&amount=<moneyAmount>:```
* transfers money between accounts with given ids
* if the matching records exists and account has enough money to transfer, the response code is 200 and the empty response body
* if there is no record(s) in the collection with the given id, the response code is 404 with error(s) explanation in response
* if account has not enough money to transfer, the response code is 400 with error explanation in response

```GET``` request to ```/transactions:```

* the response code is ```200```
* the response body is an array of matching records, ordered by their ids in increasing order
* accepts an optional query string parameter, date, in the format ```YYYY-MM-DD,``` for example ```/transaction?date=2022-11-25.``` When this parameter is present, only the records with the matching date are returned.
* accepts an optional query string parameter, type, and when this parameter is present, only the records with the matching type are returned. It might contain several values, separated by commas, meaning that records with the type matching any of these values must be returned.
* accepts an optional query string parameter, sort, that can take one of two values: either "dateTime" or "-dateTime". If the value is "dateTime", then the ordering is by dateTime in ascending order. If it is "-dateTime", then the ordering is by dateTime in descending order. If there are two records with the same dateTime, the one with the smaller id must come first.

```GET``` request to ```/transactions/<id>:```
* returns a record with the given id

* if the matching record exists, the response code is ```200``` and the response body is the matching object
* if there is no record in the collection with the given id, the response code is ```404```