## Requirement:

To build a microservice which supports creation of a basic loan amount, support adding payments to the account (in any order) and also should be able to return balance amount.

## Assumptions and Decisions:

1. No Batch functionality to automatically add interest to the principal amount. Interest is added to loan amount only when a payment is made.
2. Check balance operation do show the principal amount after adding simple interest upto the date of [check balance] operation.
3. Interest is calculated on Daily reducing balance. i.e. to calculate the interest, the previous principal amount (after the last payment) will be considered.
4. [Add payment] is rejected if the payment amount is more than the principal balance + interest. Loan account doesn’t support negative balance.
5. Performance of the application might degrade after adding multiple payments, as i have not implemented snapshot mechanism, which is essential in Event sourcing applications.
6. Data is eventually consistent between read and write. Although in current application it is almost instant. But the design leverages eventual consistency.
7. The target of this application was mainly on design (lock less implementation) rather than the performance.

## Design consideration:

The application uses following design patterns and concepts:

1. Command Query Request seggregation (CQRS)
2. Event Sourcing
3. Materialized view
4. Eventual consistency
5. Some basic aspects for Domain Driven design

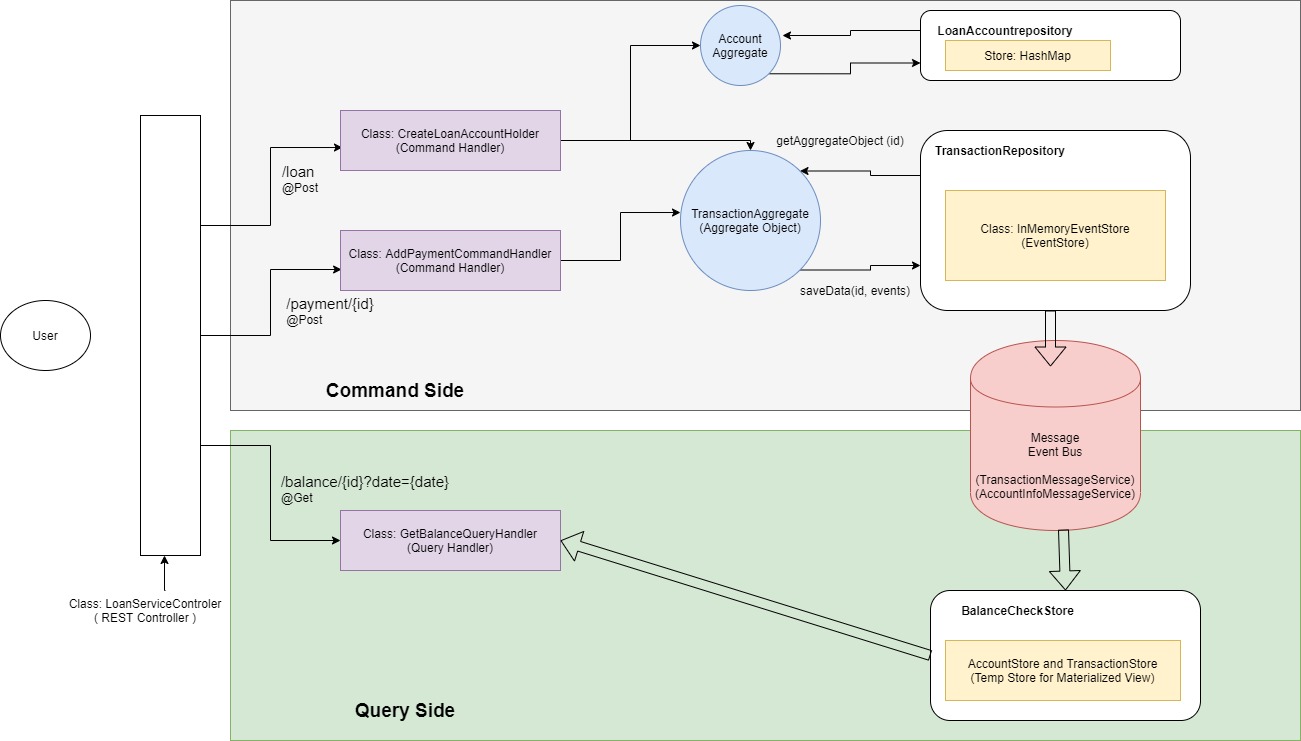
**Rationale** : Loan accounts are better represented as Ledger, where you have only 2 operations DEBIT and CREDIT. *Hence it is natural to store data in a event store rather than RDBMS type store*. The event store i have created is Append-Only store.

Also using CQRS and Event Sourcing improves the performance and partition failure.

No RDBMS type locks.

## System Diagram:

For bigger image: <https://github.com/prabal77/loan-microservice/blob/master/resources/System-Diagram.jpg>



The microservice is broken down into two logical parts

1. Command side
2. Query side

### Command Side:

Handles all the command type operations e.g. Create Loan account, Add Payment and payment reversal (internal operation).

The central REST controller (spring controller com.prabal.loanservice.controller.LoanServiceController.java), exposes the end points and invokes appropriate CommandHandler (singleton) based on input command.

1. com.prabal.loanservice.command.create.CreateLoanAccountHolder.java
2. com.prabal.loanservice.command.payment.AddPaymentCommandHandler.java

Each command handler then fetches the appropriate record from the Record repository com.prabal.loanservice.store.repo.TransactionRepository.java, which in turn contains an in-memory event store (custom developed com.prabal.loanservice.store.InMemoryEventStore.java). The repository returns an Aggregate object, representing individual record/event chain.

**All the business logic for this particular application is added to Aggregate object for simplicity** com.prabal.loanservice.domain.model.TransactionAggregate.java.

Everytime a new aggregate is requested from Event store. The aggregate will fetch all the previous events for the particular record and replay them to generate the updated state of the record. Performance of this can be improved by enabling snap shotting. I choose not to implement it as it was making the application too big for a sample test code.

The updated state as replayed form the historic events are stored inside Aggregate object as com.prabal.loanservice.domain.model.TransactionDataWrapper.java

Every action/command is represented as sequence of Events. Command handler applies events to the instance of aggregate object. Which after modifying the inner state, saves the event to event store; if everything is valid or else throw error.

Storing data to event store also emits changes to a message service bus, for interested parties (read side here) to listen and act upon.

This way the application’s state is maintained as a result of Events applied to each other in same sequence.

### Query Side

For querying the data, instead of reading the actual data directly from the Write store. It depends on a temp/query store which has been populated from all the Events emitted by Write side. This temp store is a materialized view of all the Write events, tweeked as per the query requirements. So that the queries perform faster.

### Data Exchange bewteen Command and Query Side

In the current system data exchange happens via a Message Pub-sub service. Which i have implemented using RxJava’s publish Observable. In standard system this should be replaced by a proper Pub-Sub system.

## Handling back dated payment

A payment is considered back dated, if it is executed after a payment with higher date. A backdated payment needs to revert all the payment with transaction date before it. Apply the backdated payment, then apply all payments which were reverted.

I have used event store which is append-only database implementation. This ensures lock-less updates and is natural to Loan transaction operations.

Before adding addPayment event, the application checks for any transaction with transaction date greater than current event. If found, it generates a PaymentReversal event and process it. Also adding the original event to PendingValidEvents list. Once all higher payments are processed, it applies the original back dated payment. Followed by all the pendingValidEvents (which were reverted).

## Add Payment operation flow

Figure (b): Shows a typical Add Payment operation flow. It also captures how backdated transaction is handled. Which is via reversal event generations.

For bigger image: <https://github.com/prabal77/loan-microservice/blob/master/resources/Add-Payment-Flow.jpg>

A screenshot of a cell phone

Description automatically generated

## Sample Application

I have create a sample application to test the service, Which performs following operation and then logs to the console. Figure (c).

1. Create 2 loan account
2. Add payments to the account in following order
   1. Create loan P = $1000, R = 10% , Date = D1 (current date) (Transaction – T1)
   2. Add payment A = $200, payment date = D1 + 1 year (Transaction – T2)
   3. Add payment A = $250, payment date = D1 + 2 year (Transaction – T3)
   4. Add payment A = $300, payment date = D1 + 3 year (Transaction – T4)
   5. Add Back dated payment A = $150, payment date = D1 + 6 months (Transaction – T5)
   6. Add back dated payment A = $ 50, payment date = D1 + 1 year and 6 months (Transaction – T6)
3. Print balance latest (i.e. D1 + 3 years)
4. Print audit log - of all event stored in the TransactionStore object.

## How to Run the application

In ./bin folder there are 2 jars and 2 bat files. You need java >= 1.8 to run the application

##### Run the microservice application:

$~> loan-server.bat or $~> java -jar loanservice-server.jar

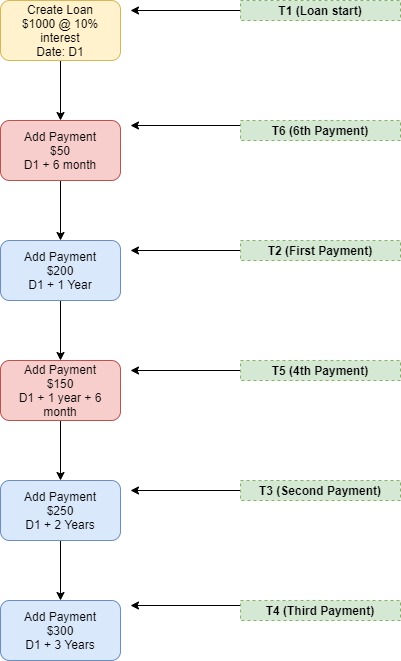
##### Run the sample client application:

$~> client.bar or $~> java -jar loanclient.jar

Running sample client application will generate a file in **client-output.txt** in the same directory with output.

You can also test using any HTTP client like CURL, POSTMAN or so.

Figure (c) <https://github.com/prabal77/loan-microservice/blob/master/resources/Example-transaction-sequence.jpg>



## REST End point details

1. Create new loan:

* url: <http://localhost:8080/loan>
* Method: POST
* Request Body:

{

    "amount": "1000",

    "interest": "20",

    "startDate": "2020-06-23"

}

* ResponseBody:

{

    "accountId": "3fc4ea2e-d5ad-40ad-a491-5a1e0f3f35d1",

    "loanAmount": "1000",

    "interestRate": "20",

    "startDate": "2020-06-23"

}

Validation: Valid loanAmount ( positive real numbers), interest amount (positive real number). Date format strictly: java.time.LocalDate

1. Add new payment:

* url: [http://localhost:8080/payment/{{account\_id}}](http://localhost:8080/payment/%7b%7baccount_id%7d%7d)
* Method: POST
* Request Body:

{

    "amount": 1000,

    "transactionDate": "2021-06-29"

}

* Response Body:

{

    "transactionId": "3fc4ea2e-d5ad-40ad-a491-5a1e0f3f35d1"

}

Throw error if adding the payment, exceeds the total principal balance. Applicable to even in case of back dated payment.

1. Check balance:

* url: [http://localhost:8080/balance/{{account\_id}}?date=2025-06-23](http://localhost:8080/balance/%7b%7baccount_id%7d%7d?date=2025-06-23)
* Method: GET
* Date field is optional, if not passed it will return the principal balance of the last payment date.

1. Get audit logs:

* url: <http://localhost:8080/audit>
* Method: GET
* Response Body: String

# Question 2 SQL Queries

1. **Which person has the greatest total expense amount?**

**Query**:

SELECT

\*

FROM

persons

WHERE

id = (SELECT

person\_id

FROM

(SELECT

person\_id, SUM(amount) AS sum

FROM

expenses

GROUP BY person\_id) temp2

WHERE

temp2.sum = (SELECT

MAX(temp.sum)

FROM

(SELECT

person\_id, SUM(amount) AS sum

FROM

expenses

GROUP BY person\_id) AS temp));

1. **Which person has the greatest total end balance considering all incomes and expenses?**

**Query**:

select \* from persons where id = (

select person\_id from

(select tempIncomes.person\_id, (tempIncomes.amount - tempExpenses.amount) as amount from

(select person\_id,sum(amount) as amount from expenses group by person\_id) tempExpenses,

(select person\_id, sum(amount) as amount from incomes group by person\_id) tempIncomes

where tempExpenses.person\_id = tempIncomes.person\_id) totalBalance

where amount = (

select max((tempIncomes.amount - tempExpenses.amount)) as balance from

(select person\_id,sum(amount) as amount from expenses group by person\_id) tempExpenses,

(select person\_id, sum(amount) as amount from incomes group by person\_id) tempIncomes

where tempExpenses.person\_id = tempIncomes.person\_id))