Utilização de Arquitetura de Microsserviços, CQRS e Event Sourcing em sistemas transacionais: Um Estudo de Caso

Gabriel Isacc Birer Lemes, Giovanni Rodrigues dos Santos, Marcelo Yamashita Zaccaria, Calebe de Paula Bianchini

Faculdade de Computação e Informática (FCI)

Universidade Presbiteriana Mackenzie – São Paulo– SP – Brasil

gabriel.isacc190@gmail.com, giovannirodrigues.2012@gmail.com, marceloyz88@gmail.com, calebe.bianchini@mackenzie.br

Abstract. *This article aims to describe the development of a banking-type transactional system using Microservices Architecture, CQRS approach, Domain Driven Design and Event Sourcing. The Project presents a case study that involves an application based on three domains defined throughout the work, and to meet the proposed development standards, six services were created, two for each domain separated in command and query, linked to this we have a record of all changes made in the system where you can view the entire history.*

Resumo. *Este artigo visa descrever o desenvolvimento de um sistema transacional do tipo bancário utilizando Arquitetura de Microsserviços, abordagem CQRS, Domain Driven Design e Event Sourcing. O Projeto apresenta um estudo de caso que envolve uma aplicação elaborada com base em três domínios definidos ao longo do trabalho, e para atender aos padrões de desenvolvimento propostos, foram criados seis serviços, sendo dois para cada domínio separados em comando e consulta, atrelado a isso temos o registro de todas as alterações realizadas no sistema onde se pode visualizar todo o histórico.*

1. Introdução

Com o intuito de explorar ferramentas as quais auxiliam na mitigação de alguns problemas ocorrentes no cenário de sistemas de informações presentes na atualidade, será trabalhada a implementação de um sistema transacional focando em alguns quesitos tidos como empecilhos no cotidiano de quem utiliza ou mantém uma estrutura como esta funcionando. Sendo assim, o que pode ser feito para atenuar eventuais infortúnios que venham a dificultar a experiência das partes envolvidas como um todo?

Com base em estudos realizados quanto ao crescimento do número de transações, constatou-se que tecnologias como o PIX vem batendo recordes a cada mês, em março

de 2022 já são mais de um bilhão e meio de transações realizadas, dado apresentado no site do Banco Central do Brasil. Visto este cenário, é apresentado na prática, através de um sistema bancário, o necessário para a implementação de um sistema com alto desempenho e escalabilidade, de fácil atualização e implantação, maior flexibilidade da estrutura, trilha de eventos para auditorias, entre outros fatos os quais serão citados nas próximas sessões. Sendo assim, foram selecionadas algumas metodologias e arquiteturas que visam justamente estimular as temáticas citadas anteriormente.

Estruturalmente é associada a utilização da arquitetura de Microsserviços com *Command Query Responsibility Segregation* (CQRS) e *Event Sourcing*, tudo isso atrelado à abordagem *Domain Driven Design*. Ao final será possível visualizar de forma detalhada como o projeto funciona ao ser submetido a estas conjunturas sistêmicas.

2. Referencial Teórico

Para guiar o embasamento teórico na construção deste trabalho, foram selecionados quatro principais temas que norteiam esta pesquisa para a realização da síntese do referencial teórico: Domain Driven Design, Microsserviços, Command Query Responsibility Segregation e Event Sourcing. Este material agregado ao artigo esclarece o motivo pelo qual realizamos determinadas escolhas nessas implementações.

2.1. Domain Driven Design (DDD)

Segundo o autor Eric Evans (2014), *Domain Driven Design* é uma abordagem para o desenvolvimento de softwares complexos, possuindo padrões contidos que mostram as práticas bem-sucedidas de projetos onde os benefícios são originários da modelagem.

Juntos, estes padrões contidos no *Domain Driven Design* estabelecem uma abordagem bem diferente para modelagem e desenvolvimento de software, que vão de detalhes finos até visão de alto nível do sistema.

Alguns termos fundamentais para o desenvolvimento e aplicação do *Domain Driven Design*:

Domínio: Área de assunto para a qual o usuário aplica um programa é o domínio do software.

Modelo: Sistema de abstrações responsável por descrever aspectos selecionados de um domínio além de poder ser utilizado para resolver problemas relacionados ao domínio.

Linguagem Ubíqua: Linguagem estruturada em torno do modelo do domínio e usada por todos os membros da equipe dentro de um contexto delimitado para conectar todas as atividades da equipe ao software.

Contexto: Configuração na qual aparece uma palavra ou declaração que determina seu significado. Declarações sobre um modelo só podem ser entendidas em um contexto.

Contexto Delimitado: Uma descrição de um limite (normalmente um subsistema ou o trabalho de uma equipe específica) dentro do qual um modelo específico é definido e aplicável.

Ainda segundo o autor Evans (2014), é necessário definir explicitamente o contexto no qual um modelo será aplicado. É preciso explicitamente estabelecer limites em termos de organização da equipe, uso em partes específicas do aplicativo e manifestações físicas, como bases de código e esquemas de banco de dados. Aplicando a Integração Contínua para manter os conceitos e os termos do modelo estritamente consistentes dentro desses limites, assim padronizando um único processo de desenvolvimento dentro do contexto estabelecido.

O modelo precisa ser usado como a espinha dorsal de uma linguagem. Dentro de um Contexto Delimitado a utilização da mesma linguagem em diagramas, escrita e especialmente na fala. Sempre deve se reconhecer que uma mudança na linguagem é uma mudança no modelo. Esgotar as dificuldades experimentando expressões alternativas, refletem modelos alternativos. Em seguida, refatorar o código, renomeando classes, métodos e módulos para se adequar ao novo modelo criado. Também é importante solucionar a confusão sobre os termos na conversa, de forma que se chegue a uma concordância com o significado das palavras comuns (Evans, 2014).

É necessário o estabelecimento de um processo para mesclar todo o código desenvolvido e outros artefatos de implementação com frequência. A utilização incansável da Linguagem Ubíqua contribui na elaboração de uma visão compartilhada do modelo à medida que os conceitos evoluem na cabeça das diferentes pessoas envolvidas no projeto (Evans, 2014).

Estabelecer um processo para mesclar todo o código e outros artefatos de implementação com frequência é importante, usando de testes para sinalizar rapidamente a fragmentação do sistema. É preciso também exercer a Linguagem Ubíqua para elaborar uma visão compartilhada do modelo à medida que os conceitos evoluem na cabeça de diferentes pessoas (Evans, 2014).

2.2 Microsserviços

Microsserviço é uma abordagem arquitetural que consiste no funcionamento em conjunto de pequenos serviços independentes podendo ser utilizadas diferentes linguagens, formando um sistema e comunicando-se geralmente por meio de REST APIs (FOWLER & JAMES, 2014).

Como toda a arquitetura, a abordagem de Microsserviços também tem seus prós e contras, e segundo Fowler (2015) uma estrutura com fortes limites de módulo, como é o caso desta, beneficia as equipes responsáveis por cada unidade, visto que a comunicação entre elas acaba se tornando mais independente, formal e organizada, devido a existência de uma forte delimitação entre os serviços. Uma das características positivas é no momento de implantação em ambientes produtivos, no qual é possível realizar independentemente, e caso ocorra algum erro, não impedirá o funcionamento do sistema como um todo, e sim somente do serviço em questão, isso facilita a realização de entregas contínuas e redução dos ciclos de melhorias. No artigo de Fowler (2015) é citado a diversidade tecnológica, na qual permite que cada equipe responsável pelo seu serviço escolha a ferramenta mais adequada para resolução de seu problema, libertando a equipe de realizar escolhas precipitadas e de difícil reversão, uma vez que renovar as tecnologias e frameworks utilizados muitas vezes se vê necessário depois de um tempo.

A distribuição sistêmica que a arquitetura de Microsserviços causa pode ser um problema caso não sejam tomadas algumas providências para mitigação dessa questão com relação à velocidade e à resiliência. Outro problema proveniente desta abordagem é a inconsistência eventual devido ao gerenciamento descentralizado de dados, no qual atrapalha o desempenho do sistema como um todo. A complexidade operacional também é um ponto de atenção, pois pode ocasionar na sobrecarga das operações, visto que se trata de diversos microsserviços, exigindo a introdução de uma cultura devops. Estes custos citados não são exclusivamente desta arquitetura em específico, estando presente também em uma estrutura monolítica por exemplo, mas os cuidados os quais devem ser tomados são mais acentuados nos casos de Microsserviços (FOWLER, 2015).

A título de comparação entre uma arquitetura de serviços e monolítica, evidencia-se o motivo do uso de uma em contrapartida a outra. Em um sistema de monolito tem uma única base de código para toda a aplicação, é confuso e difícil de manter, alta complexidade na implementação, necessidade de janelas de manutenção, necessidade de inatividade do sistema, geralmente baseado em uma única linguagem de programação e requer reestruturação de todas as aplicações, mesmo identificado onde está o gargalo do sistema. Já na arquitetura de Microsserviços tem múltiplos códigos base, mais legível e compreensível, implementação mais fácil, cada microsserviço pode ser desenvolvido na linguagem melhor adaptada à sua necessidade, possível escalar somente o serviço identificado como obstáculo (BAKSHI, 2017).

2.3 Command Query Responsibility Segregation (CQRS)

Conceito de *Command Query Responsibility Segregation* (CQRS) foi baseado no *Command Query Separation* (CQS) de 1988 escrito por Bertrand Meyer, e é definido

por Young (2010) como a separação das responsabilidades de escrita (comando) e leitura (query), segregadas em dois objetos. Segundo Young (2010, p. 17), é citado a seguinte frase: “Se você tem um valor retornado você não pode alterar o estado. Se você alterar o estado seu tipo de retorno deve ser vazio”.[1]

Segundo Fowler (2011) esta arquitetura é separada em dois processos lógicos, ou até mesmo em dois hardwares diferentes, podendo compartilhar do mesmo banco de dados, conectando os dois processos entre si ou pode também utilizar dois bancos de dados distintos, sendo necessário uma comunicação adicional entre eles. Um benefício interessante sobre esta arquitetura é referente ao desempenho, na qual é possível dimensionar a leitura e a gravação de forma independente, podendo até aplicar técnicas de otimização conforme sua necessidade, como por exemplo uma determinada forma de acesso ao banco para leitura e outra para gravação. Suas características o tornam favoráveis na utilização com o modelo de *Event Sourcing* e também com o *Domain Driven Design*.

Em um trabalho coordenado por Rajković, Janković e Milenković (2013) levantou alguns dados interessantes referente à performance na utilização da arquitetura de CQRS em um sistema de informações médicas em utilização por mais de 20 instalações de cuidado primário na Sérvia, na qual foi relatado que após dois anos de uso por estas instituições identificaram uma resposta lenta do sistema devido ao tráfego desnecessário de dados, na qual não havia uma otimização das queries utilizadas. Ao utilizar o CQRS e passar os dados de utilização mais frequentes e de rara alteração para o banco de dados de leitura observou-se que no geral o tempo de resposta do sistema melhorou em 40% e o tráfego médio de dados foi reduzido em um terço.

2.4 Event Sourcing

Evento Sourcing é citado por Richardson (2018) em seu livro como uma maneira de estruturar a lógica do negócio em entidades, na qual é atrelado a ele uma sequência de eventos e cada evento é um estado de alteração que esta entidade sofreu. Isso beneficia em momentos de auditoria e regulamentação pelo fato de ser preservado todo seu histórico e beneficia a arquitetura de microsserviços, a qual está sendo utilizada no sistema, pois publica os eventos dos domínios.

Em um trabalho realizado por Bianchini, Bezerra e Vasconcellos (2018), um sistema ERP foi refatorado utilizando *Event Sourcing* e neste cenário puderam observar diversos benefícios. Pelo fato de todas as ações realizadas no sistema estarem registradas, foi possível depurar eventuais erros de forma mais fácil e rápida, bastou apenas realizar uma cópia da base de dados, identificar a ação desencadeadora do erro, apagar tudo

[1] If you have a return value you cannot mutate state. If you mutate state your return type must be void.

deste ponto em diante e realizar o gatilho disparado pelo cliente, assim a análise e o entendimento do problema foi mais efetivo. Outro ponto beneficiado foi a análise de operações realizadas, como se tem um histórico detalhado, é possível analisar por que em dado momento houve problema em uma determinada compra, sendo possível até mesmo identificar por meio de relatórios novas necessidades do negócio.

3. Experimento

A forma como a execução do projeto foi feita e como a teoria foi colocada na prática será descrita nesta seção do documento, assim, de forma mais detalhada e palpável será possível entender toda a trajetória das etapas do sistema até o resultado final.

Para abranger as tecnologias citadas, o desenvolvimento do projeto foi realizado na linguagem de programação Java e baseado na implementação de um sistema bancário, devido ao fato desse tipo de sistema ter acesso simultâneo de diversos usuários, pela alta taxa de transações ocorrentes na aplicação, conforme citado anteriormente com base nos dados divulgados pelo Banco Central do Brasil, e também por se tratar de sistema que necessita ter consistência de informações.

3.1 Aplicação do Domain Driven Design para definir os domínios do sistema e arquitetura do sistema

Para iniciar a criação do projeto, foram discutidas algumas formas para obter a visão total do escopo do mesmo e a definição de todas as partes as quais seriam utilizadas para o desenvolvimento do sistema. Depois da análise de todas as opções, foi escolhida a abordagem Domain Driven Design (DDD) criada por Eric Evans.

A escolha do DDD se deu pelo fato de que com essa abordagem foi possível realizar, de maneira estratégica e de fácil entendimento, a definição dos domínios de negócio os quais foram utilizados e de seus respectivos contextos delimitados dentro do sistema.

No começo da aplicação desta abordagem, foi decidido o domínio do projeto através da análise das principais características que um sistema transacional de contexto bancário possui. Essa análise foi gerada por meio de um estudo dos principais bancos digitais e seus respectivos aplicativos, como o Nubank e o Mercado Pago, verificando quais métodos eram vitais para ter como base no desenvolvimento da aplicação.

Com o domínio consolidado entre os integrantes do grupo, foi desenvolvida a modelagem, em UML, de um diagrama de domínio contendo todas as classes utilizadas na estruturação do sistema, como: Conta, Cliente, Transações, juntas de seus respectivos atributos.

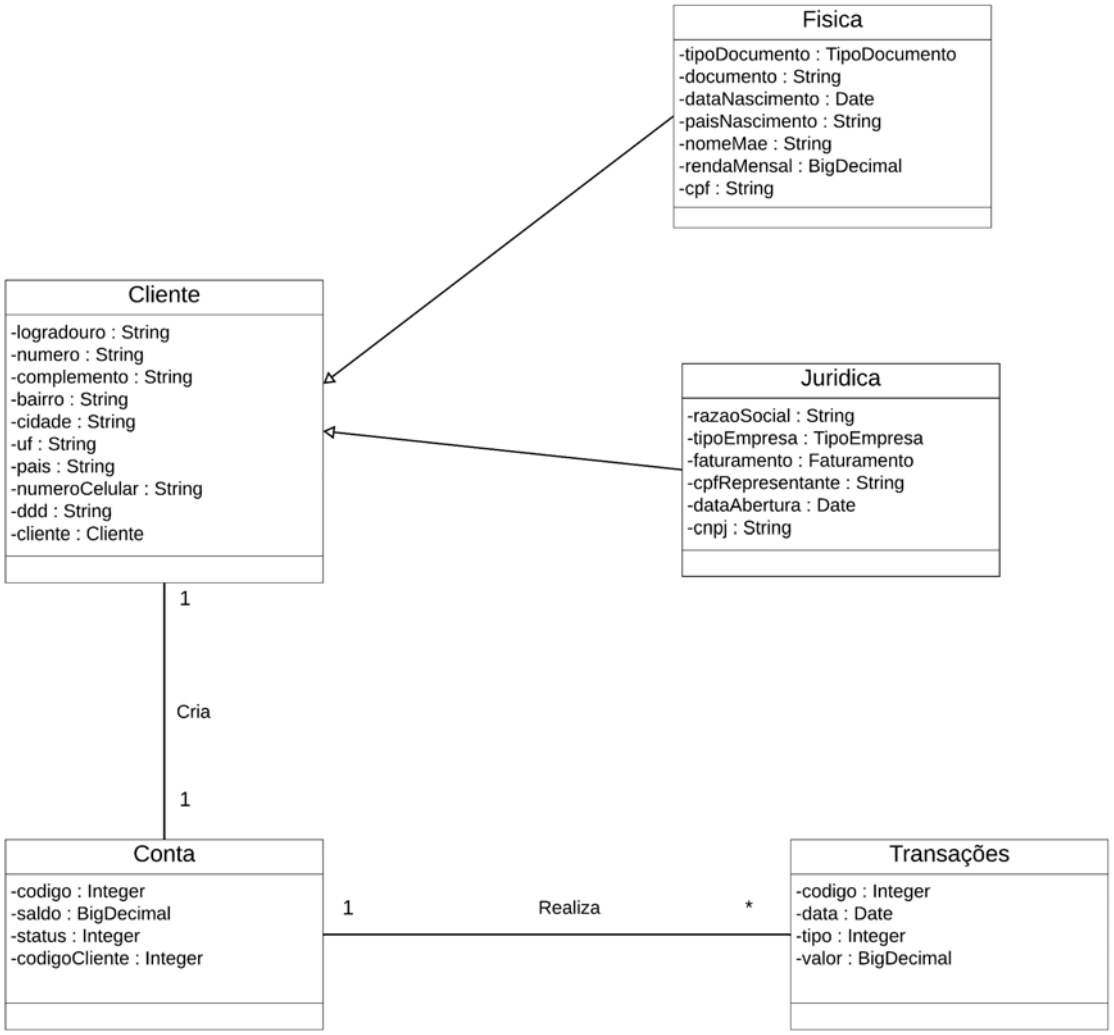

Figura 1. Diagrama de Domínio do projeto.

Após a obtenção de todo o domínio já definido, o grupo aplicou o conceito de Linguagem Ubíqua desenvolvido por Eric Evans para criar uma espécie de linguagem com termos e denominações próprias do projeto, para assim alcançar uma fácil comunicação entre os integrantes e uma melhor forma de desenvolvimento das próximas etapas sem o risco de perda de significado ao longo do processo.

Glossário da Linguagem Ubíqua do projeto:

- Cliente: Usuário final do sistema, independente do seu tipo.
- Conta: Conta, serviço, que armazena e movimenta crédito.
- Transação: Movimentação de crédito entre contas, independente do seu tipo.

Possuindo uma linguagem definida e fluente entre os integrantes, iniciou-se a delimitação dos contextos da aplicação, definindo a responsabilidade de cada parte do

sistema bem estabelecidas de acordo com o que foi determinado nas etapas anteriores. Assim, possibilitou uma maior facilidade em entender qual função cada contexto do sistema irá ter e um melhor entendimento na hora da programação do mesmo.

Alguns exemplos de histórias contendo o contexto delimitado definido do sistema:

- Cliente escolhe qual tipo de transação deseja realizar.
- Cliente altera dados pessoais.
- Extrato exibe todos os tipos de transação realizada pelo cliente.

Após realizar todas as etapas, citadas anteriormente, propostas pelo DDD, o grupo também desenvolveu, de forma mais facilitada e clara, uma arquitetura de software para o sistema a ser desenvolvido, visto na imagem a seguir:

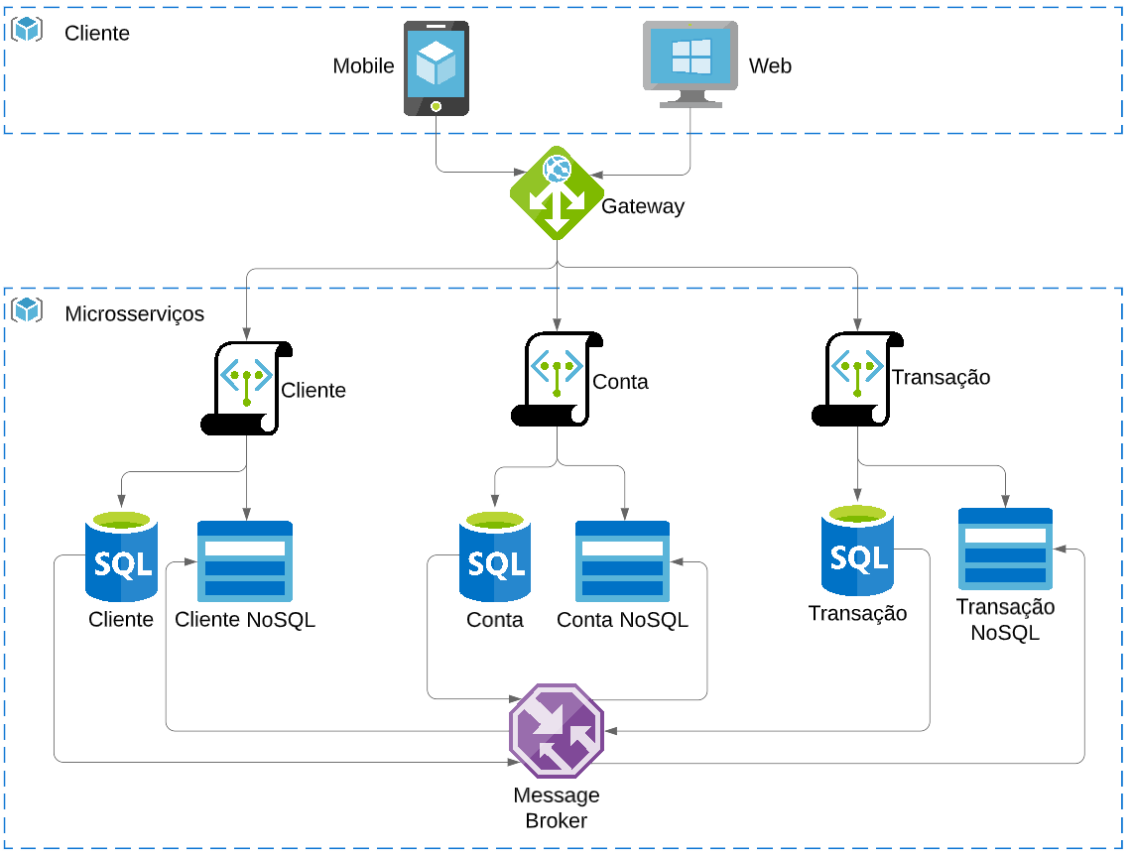

Figura 2. Arquitetura inicial do projeto.

Nesta arquitetura, é possível localizar duas camadas, sendo a primeira a camada de Microsserviços e a segunda a camada de Cliente. Na primeira mencionada se encontram os microsserviços desenvolvidos no projeto, sendo eles compostos por Cliente, Conta e

Transação que estão conectados aos seus respectivos bancos de dados relacionais e não relacionais como mostrado na imagem acima. Além disso, nesta mesma camada possui o Message Broker, responsável por realizar a troca de mensagens entre os serviços, criando uma fila com essas mensagens as quais são registradas em forma de eventos no sistema.

Já na segunda camada é onde fica a interface que o cliente final irá utilizar, sendo composta pelas chamadas das requisições dos serviços através de um front-end.

3.2 Implementação da arquitetura de Microsserviços

Com a visão total do escopo do projeto definida de forma clara e objetiva, foi dada sequência para a próxima etapa para o desenvolvimento dos serviços que compõem a arquitetura de Microsserviços.

Para garantir a comunicação de todas as partes englobadas no sistema, foi necessário a implementação de um Service Discovery com roteamento dinâmico, ficando responsável por registrar as instâncias de cada serviço toda vez que eles são iniciados. Assim, quando for realizada qualquer requisição, o Service Discovery direcionará para o serviço correto.

Para realizar a abordagem do Service Discovery foi utilizado o Eureka Server, podendo ser encontrado no projeto Spring Cloud Netflix. Com isso foi adicionado na arquitetura desenvolvida inicialmente um novo componente chamado Discovery que representa o Eureka Server, como se pode observar na figura abaixo.

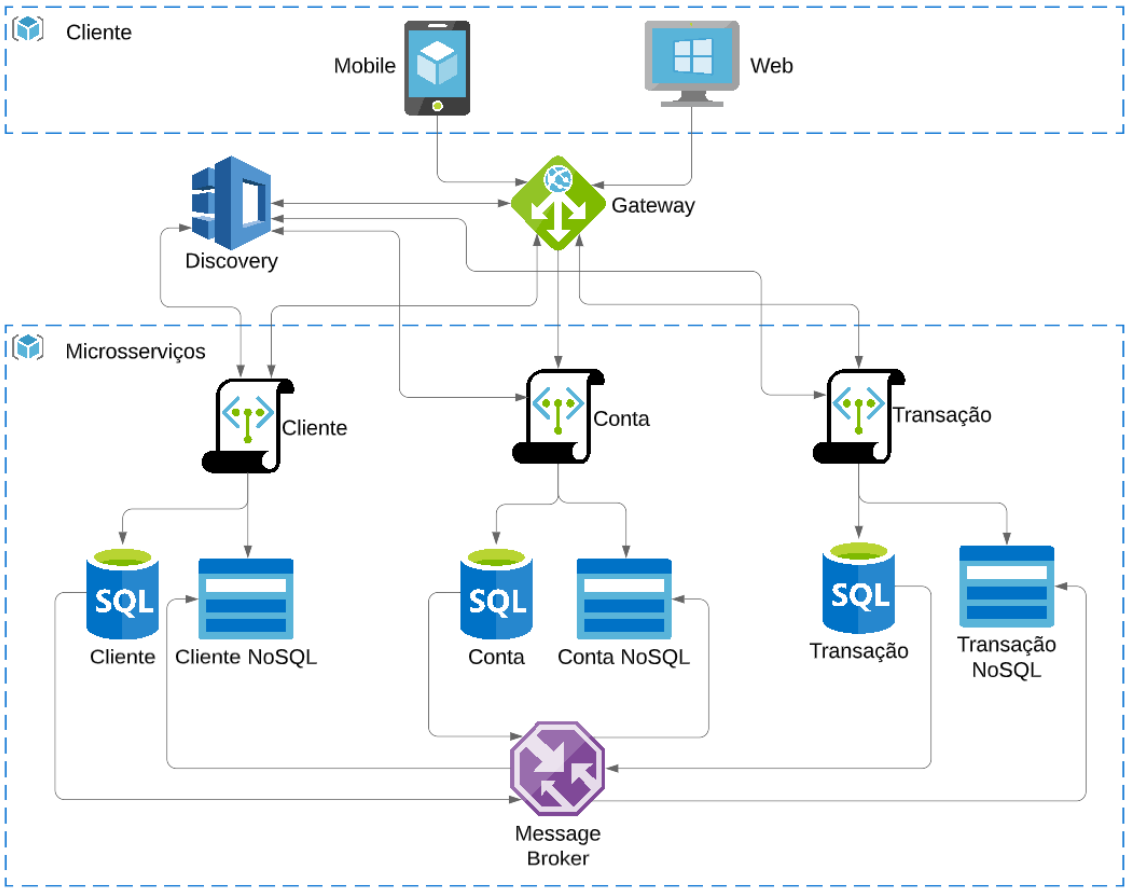

Figura 3. Nova arquitetura do projeto incluindo o Discovery.

Com a criação do Service Discovery criado, é necessário adicionar Eureka ao código. Para isso foi realizado a modificação no arquivo "DiscoveryApplication.java", importando a dependência e habilitando com a anotação "EnableEurekaServer", como demonstrado na figura abaixo:

```
@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class,
HibernateJpaAutoConfiguration.class})
@EnableEurekaServer
public class DiscoveryApplication {
    public static void main(String[] args) {
        SpringApplication.run(DiscoveryApplication.class, args);
    }
}
```

Figura 4. DiscoveryApplication.

Foi necessário criar uma API Gateway, responsável por receber todas as requisições do lado do cliente e realizar o roteamento para comunicação com os microsserviços

internos utilizando o sistema do Zuul, no qual também faz parte do Spring Cloud Netflix, sendo este representado na arquitetura como Gateway.

Da mesma forma que foi implementado o Eureka é necessário adicionar o Zuul no código, sendo assim foi modificado o arquivo "GatewayApplication.java" com suas dependências e anotação "EnableZuulProxy". Segue a representação abaixo:

```
@SpringBootApplication
@EnableZuulProxy
@EnableEurekaClient
public class GatewayApplication{
   public static void main(String[] args) {
      SpringApplication.run(GatewayApplication.class, arg);
   }
}
```

Figura 5. GatewayApplication.

Foram criados três serviços para estruturar toda a jornada do usuário dentro do sistema, que são:

- Cliente: Contendo as dependências do Eureka e do Zuul e com as informações de codigo, nome, email, CEP, logradouro, numero, complemento, bairro, cidade, uf, pais, ddd, numero (telefone) e dependendo do tipo de pessoa temos informações específicas, no caso da pessoa física temos cpf, TipoDocumento, documento, dataNascimento, paisNascimento, nomeMae e rendaMensal. No caso de pessoa jurídica temos cnpj, razaoSocial, tipoEmpresa, faturamento, dataAbertura e cpfRepresentante.
- Conta: Necessário também as dependências do Eureka e do Zuul e informações de código, saldo, status e codigoCliente.
- Transacao: Dependências do Eureka e do Zuul e codigo, data, tipo e valor.

Com isso foi possível garantir o pleno funcionamento da arquitetura de microsserviços elaborada para o sistema de forma independente.

3.3 Implementação do Event Sourcing e Command Query Responsibility Segregation (CQRS)

O Event Sourcing e o CQRS foram adicionados simultaneamente, assim é possível minimizar o retrabalho na criação e configuração das classes. Nesta seção é descrito o que é necessário para a implementação destas abordagens.

Primeiramente é necessário subir o servidor de mensageria, RabbitMQ representado pelo componente *Message Broker* no diagrama de arquitetura, na figura três, para

garantir que se tenha as filas de eventos na qual cada serviço de command e query utiliza. Com isso pronto, é possível segregar cada serviço (Cliente, Conta e transacao) em suas respectivas responsabilidades de comando e de consulta.

Nos serviços de comando foram adicionados as dependências do JPA, responsável pela criação das tabelas de banco de dados e seus relacionamentos, adição da biblioteca do MysqlConnector, dependência do Starter-amqp para conectar facilmente a uma instância do RabbitMQ e dependência do axon-framework para configuração da fila de eventos. No arquivo "application.yml" é configurado as conexões com o banco de dados mysql, nome da aplicação que será registrada no Eureka, o nome da fila de eventos no RabbitMQ e porta a qual será executada a aplicação. Segue exemplo da configuração:

```
datasource:
   url:
jdbc:mysql://localhost:3306/transaction?createDatabaseIfNotExist=true&allowPublic
KeyRetrieval=true&sslMode=DISABLED&useLegacyDatetimeCode=false&serverTi
mezone=GMT-3
   username: root
   password: root

axon:
  amqp:
   exchange: bank.events
```

Figura 6. Configuração do banco de dados e nome da fila de eventos.

Foi necessária a criação dos aggregates responsáveis por tratar os comandos, tornando possível o envio dos eventos para a fila.

```
@Aggregate
public class TransactionAggregate {

   @AggregateIdentifier
   private Integer id;
   private Date date;
   private Integer type;
   private BigDecimal value;
   private Integer accountId;
   public TransactionAggregate() { }
```

```
    @CommandHandler
    public TransactionAggregate(CreateTransactionCommand cmd) {
        Assert.hasLength(cmd.getType().toString(), "Tipo da transacao nao pode ser
nulo.");
        Assert.hasLength(cmd.getValue().toString(), "Valor da transacao nao pode ser
nulo.");
        Assert.hasLength(cmd.getAccountId().toString(), "Codigo da conta da transacao
nao pode ser nulo.");
        apply(new CreateTransactionEvent(cmd.getId(), cmd.getType(), cmd.getValue(),
cmd.getAccountId()));
    }

    @EventSourcingHandler
    public void on(CreateTransactionEvent event) {
        this.id = event.getId();
        this.type = event.getType();
        this.value = event.getValue();
        this.accountId = event.getAccountId();
    }
}
```

Figura 7. Exemplo de aggregate.

```
public class CreateTransactionCommand {

    @TargetAggregateIdentifier
    private Integer id;

    private Integer type;

    private BigDecimal value;

    private Integer accountId;

    public CreateTransactionCommand(Integer id, Integer type, BigDecimal value,
Integer accountId) {
        this.id = id;
        this.type = type;
        this.value = value;
        this.accountId = accountId;
    }
}
```

Figura 8. Exemplo de comando.

```
public class CreateTransactionEvent {
   private Integer id;

   private Integer type;

   private BigDecimal value;

   private Integer accountId;

   public CreateTransactionEvent(Integer id, Integer type, BigDecimal value, Integer
accountId) {
       this.id = id;
       this.type = type;
       this.value = value;
       this.accountId = accountId;
   }
}
```

Figura 9. Exemplo de evento.

Com a interação do usuário são gerados comandos, que por sua vez disparam eventos para a fila de eventos, possibilitando a captura do mesmo por diversos controladores de eventos na aplicação e persistido em banco de dados, sendo que cada serviço do sistema possui o seu próprio banco de dados para persistência e outro para consulta, atendendo assim aos padrões do event sourcing.


```
[
   {
      "type": "TransactionAggregate",
      "aggregateIdentifier": "1",
      "sequenceNumber": 0,
      "identifier": "f0c9b6ca-ae6c-4279-959e-1b033f13399d",
      "timestamp": "2022-05-16T05:00:21.943685200Z",
      "metaData": {
         "traceId": "a29ed13b-80c1-4092-87b0-b312e1d4f14b",
         "correlationId": "a29ed13b-80c1-4092-87b0-b312e1d4f14b"
      },
      "payload": {
         "type": 0,
         "value": "100",
         "accountId": 603163
      },
      "payloadType": "tcc.bank.account.Event.CreateTransactionEvent"
   }
```

```
]
```

Figura 10. Exemplo de evento gerado na criação de conta.

Referente aos serviços de query, nos arquivos “pom” também foram adicionadas as dependências Starter-amqp e Axon-framework, além disso a dependência do MongoDB. O “application.yml” segue o padrão de configuração do serviço de comandos, com a adição do handling que é responsável por manipular os eventos os quais foram disparados para a fila.

```
axon:
  amqp:
    exchange: bank.events
  eventhandling:
    processors:
      amqpEvents:
        source: complaintEventsMethod
```

Figura 11. Exemplo de configuração do eventhandling.

O eventhandling realiza o processamento do evento capturado da fila, e armazena no banco de dados não relacional utilizado para consulta.

```
@Component
@ProcessingGroup("amqpEvents")
public class TransactionEventProcessor {

    private final TransactionRepository transactionRepository;

    public TransactionEventProcessor(TransactionRepository transactionRepository) {
        this.transactionRepository = transactionRepository;
    }

    @EventHandler
    public void on(CreateTransactionEvent event) {
        transactionRepository.save(new TransactionModel(event.getId(),
Date.from(Instant.now()), event.getType(), event.getValue(), event.getAccountId()));
    }
}
```

Figura 12. Exemplo de eventhandling.

Assim conclui-se a implementação do CQRS e do Event Sourcing.

4. Resultados Obtidos

Em relação aos resultados obtidos com a implementação do sistema atrelado a todas as abordagens e arquiteturas agregadas, será destacado alguns pontos que foram possíveis observar durante sua implementação. O projeto está disponível no GitHub (https://github.com/TCC-Mackenzie) para melhor visualização do que está sendo apresentado.

No sistema pode-se executar as principais funções no que se diz respeito a sistemas transacionais desde a abertura de uma conta até realizar transações bancárias, a atualização do saldo da conta após uma transação se faz de forma manual porém caso seja implementado futuramente um eventhandling no serviço de conta-command será possível realizar esta atualização automaticamente, nas figuras abaixo estão representadas algumas das funções implementadas.

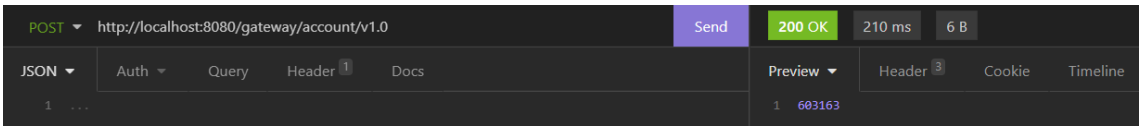

Figura 13. Criação de conta.

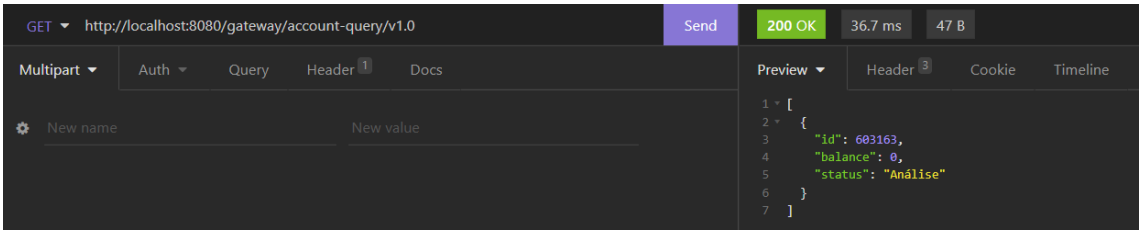

Figura 14. Consulta das informações da conta criada.

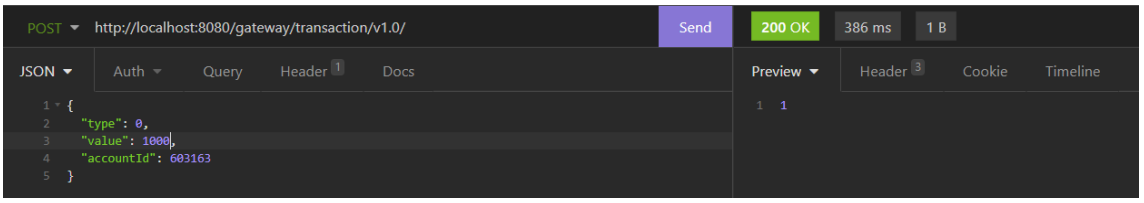

Figura 15. Criação de transação.

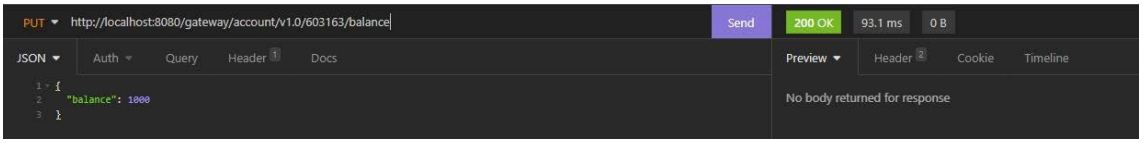

Figura 16. Atualização manual do saldo.

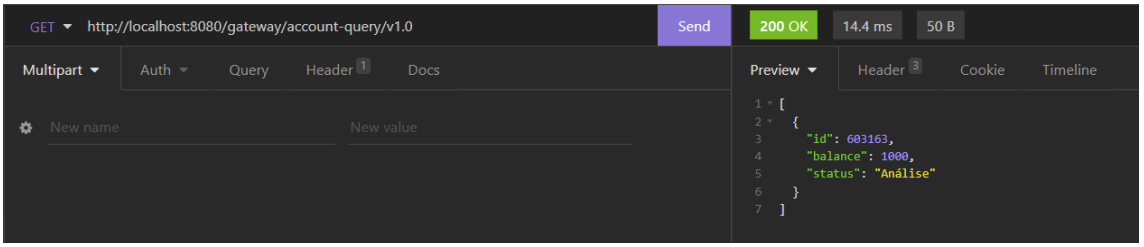

Figura 17. Consulta do saldo da conta.

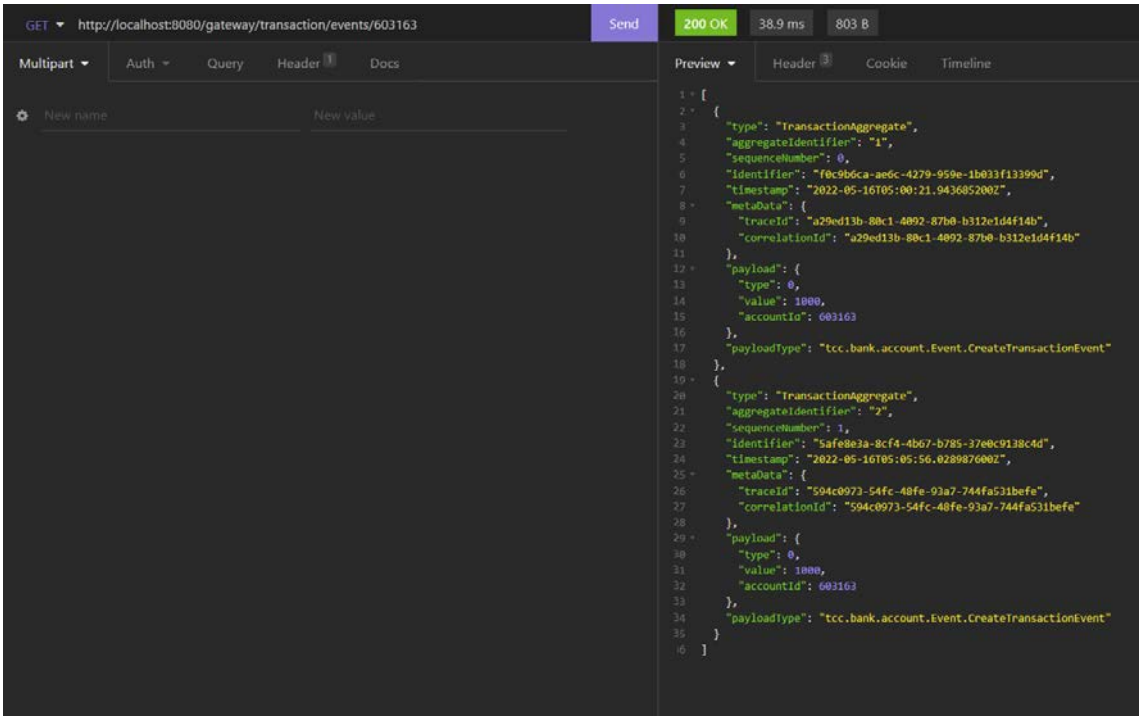

Figura 18. Consulta da lista de eventos de transação.

O CQRS foi devidamente implementado, resultando na diluição do esforço exigido na execução de cada serviço pelo fato das responsabilidades estarem divididas entre comando e leitura conforme demonstra as figuras 13 e 14 respectivamente, por exemplo. Como pode-se observar nas figuras

O Event Sourcing possibilitou com que fosse possível consultar, caso necessário, a trilha de eventos, ou seja, todo o histórico de alterações é persistido dentro de uma base de dados para poder ser consultada e auditada quando necessário, assim como demonstra a figura 18.

Com relação ao Domain Driven Design foi mais fácil visualizar o domínio do projeto em relação ao sistema transacional, entender quais são as classes de domínio e através da linguagem ubíqua criamos termos que beneficiaram a comunicação do grupo.

5. Considerações finais

Este trabalho foi responsável por realizar um estudo sobre os principais conceitos de DDD, Microsserviços, CQRS e Event Sourcing e aplicar os mesmos em um sistema transacional funcional.

Utilizando a abordagem DDD, foi possível criar uma maior compreensão do domínio do sistema, colocando cada um destes domínios em um contexto delimitado, cada um com sua respectiva responsabilidade, podendo criar orientações de fácil entendimento para os integrantes na hora de desenvolver o código.

Ao longo do desenvolvimento deste projeto, foi possível perceber que a arquitetura de Microsserviços possibilitou um desenvolvimento mais fluido e de fácil entendimento. Este tipo de arquitetura permitiu uma grande facilidade na hora de realizar manutenções no código pelo fato de os serviços estarem separados individualmente. Além disso, esta arquitetura possibilitou a criação de um sistema com escalabilidade.

Em relação ao CQRS e Event Sourcing, pode-se concluir que foram abordagens mais complicadas para serem aplicadas no projeto pelo fato de ser o primeiro contato do grupo com estas abordagens e que também adicionou um alto nível de complexidade ao código, exigindo mantimento da aplicação como um todo muito bem organizada. Apesar das dificuldades na aplicação, ambas as abordagens foram desenvolvidas e aplicadas no projeto.

Como o foco do projeto foi direcionado para o desenvolvimento e aplicação das abordagens citadas acima, o grupo percebeu que ainda existem pontos a serem trabalhados como possíveis melhorias para o sistema implementado, como a atualização de saldo de conta automático ao realizar uma transação e a criação de uma interface para o usuário realizar as interações com o sistema. Além destes pontos, também é notável a necessidade futura de realizar teste de escalabilidade para medir o quão bem o projeto escala em resposta a variados níveis de carga, simulando o acesso de diversos usuários simultaneamente realizando transações.

Com este artigo, espera-se que o projeto contenha informações e experiências para pessoas as quais possuem como finalidade a busca de conhecimento a fim de utilizar essas abordagens do ramo da tecnologia em futuros trabalhos.

Referências

BAKSHI, Kapil. (2017) "Microservices-Based Software Architecture and Approaches". In IEE.

BIANCHINI, Calebe; BEZERRA, Vinicius; VASCONCELLOS, Pedro. (2018) "Applying Event sourcing in a ERP system: a case study" In. IEE.

Evans, E. (2014) "Domain-Driven Design Reference: Definitions and Pattern Summaries". Dog Ear Publishing.

Estatísticas do PIX, Banco Central do Brasil. Disponível em: https://www.bcb.gov.br/estabilidadefinanceira/estatisticaspix. Acesso em 04 Jun. 2022.

FOWLER, Martin. (2011) "CQRS". Disponível em: https://martinfowler.com/bliki/CQRS.html. Acesso em 30 Abr. 2022.

FOWLER, Martin. (2015) "Microservice Trade-Offs". Disponível em: https://martinfowler.com/articles/microservice-trade-offs.html#boundaries. Acesso em 28 Abr. 2022.

FOWLER, Martin; LEWIS, James. (2014) "Microservices, a definition of this new architectural term". Disponível em: https://martinfowler.com/articles/microservices.html. Acesso em 28 Abr. 2022.

MILENKOVIĆ, Aleksandar; JANKOVIĆ, Dragan; RAJKOVIĆ, Petar. (2013). "Using CQRS Pattern for Improving Performances in Medical Information Systems". Disponível em: http://ceur-ws.org/Vol-1036/p86-Rajkovic.pdf. Acesso em 09 Maio 2022.

RICHARDSON, Chris. (2018). "Microservices Patterns: With Examples in Java". 1.ed. Nova Iorque: Manning Publications CO, cap 6, p. 183-219.

Spring Cloud Netflix, [s.d.]. Disponível em: https://spring.io/projects/spring-cloud-netflix. Acesso em: 28 de Abr. 2022.

YOUNG Greg. (2010) "CQRS Documents". Disponível em: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf. Acesso em: 30 Abr. 2022.