UNIVERSIDADE FEDERAL DE SANTA CATARINA

RECOMENDAÇÕES SOBRE O DESENVOLVIMENTO DE APLICAÇÕES PARA WEB BASEADAS NO PADRÃO *COMMAND QUERY RESPONSABILITY SEGREGATION* E EM *EVENT SOURCING*

FELLIPE BRATTI PASINI

MURILO TEIXEIRA FERNANDES

FLORIANÓPOLIS-SC

2015/2

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE BACHARELADO EM SISTEMA DE INFORMAÇÃO

RECOMENDAÇÕES SOBRE O DESENVOLVIMENTO DE APLICAÇÕES PARA WEB BASEADAS NO PADRÃO *COMMAND QUERY RESPONSABILITY SEGREGATION* E EM *EVENT SOURCING*

FELLIPE BRATTI PASINI
MURILO TEIXEIRA FERNANDES

Trabalho de conclusão de curso apresentado
como parte dos requisitos para obtenção do
grau de Bacharel em Sistemas de Informação

FLORIANÓPOLIS - SC
2015/2

FELLIPE BRATTI PASINI
MURILO TEIXEIRA FERNANDES

RECOMENDAÇÕES SOBRE O DESENVOLVIMENTO DE APLICAÇÕES PARA WEB BASEADAS NO PADRÃO *COMMAND QUERY RESPONSABILITY SEGREGATION* E EM *EVENT SOURCING*

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação

Orientador: Leandro J. Komosinski

Banca examinadora

Frank Siqueira
Carina Friedrich Dorneles

LISTA DE FIGURAS

LISTA DE ABREVIAÇÕES

DDD - Domain-Driven Design

CQRS - Command Query Responsibility Segregation

ES - Event Sourcing

CRUD - create, read, update and delete

JSF - JavaServer Faces

DTO - Data transfer object

JDBC - Java Database Connectivity

SOAP - Simple Object Access Protocol

REST - Representational State Transfer

HTTP - Hypertext Transfer Protoco

HTTPS - Hyper Text Transfer Protocol Secure

FTP - File Transfer Protocol

LDAP - Lightweight Directory Access Protocol

MOM - Message-oriented middleware

JMS - Java Message Service

SMTP - Simple Mail Transfer Protocol

POP3 - Post Office Protocol 3

IMAP - Internet Message Access Protocol

TCP - Transmission Control Protocol

Sumário

Abstract.

To handle with the development of applications that need to meet a high demand for concurrent users are required alternatives to facilitate and increase the scalability of the application. Generally these size applications tend to deal with more complex domains. To address these issues emerged the Command Query Responsibility Segregation design pattern, the storage technique Event Sourcing and development methodology Domain Driven Design. The purpose of this paper is to define a set of recommendations to assist the development of applications that want to use an architecture integrating these concepts. In support, was performed a case study in a sample application format to help identify the most important decision points. Finally, it was established a set of 10 recommendations, separated into five groups, which are: modeling of the database, information retrieval, consistency, concurrency and frameworks.

Keywords*:* *CQRS, Command Query Responsibility Segregation, DDD, Domain Driven Design, Event Sourcing, ES*

Resumo.

Para lidar com o desenvolvimento de aplicações que necessitam atender uma demanda alta de usuários simultâneos são necessárias alternativas para facilitar e aumentar a escalabilidade da aplicação. Geralmente aplicações deste porte tendem a lidar com domínios mais complexos. Para tratar estas questões surgiram o padrão de projeto *Command Query Responsability Segregation*, a técnica de armazenamento *Event Sourcing* e a metodologia de desenvolvimento *Domain Driven Design*. O objetivo desta monografia é definir um conjunto de recomendações para auxiliar o desenvolvimento de aplicações que pretendem utilizar uma arquitetura integrando estes conceitos. Como apoio, foi realizado um estudo de caso no formato de uma aplicação exemplo para ajudar a identificar os pontos de decisão mais importantes. Por fim, foi criado um conjunto de 10 recomendações, separados em cinco grupos, os quais são: modelagem da base de dados, recuperação de informações, consistência, concorrência e *frameworks*.

Palavras-chave: *CQRS, Command Query Responsibility Segregation, DDD, Domain Driven Design, Event Sourcing, ES*

1. Introdução

Aplicações para web, pela sua própria natureza, podem ser utilizadas por um número muito grande de usuários ao mesmo tempo. Esta característica adiciona um requisito a mais para os desenvolvedores deste tipo de aplicação: minimizar a degradação do tempo de resposta em função da quantidade de usuários ativos.

A solução de problemas de escalabilidade depende, em grande parte, de questões relacionadas à arquitetura da aplicação. Portanto, ao iniciar o desenvolvimento de uma aplicação a definição da sua arquitetura é uma das primeiras decisões a serem tomadas.

Tradicionalmente, aplicações para web são organizadas em uma arquitetura em camadas, sendo o modelo de três camadas (modelo - visão - controle, sigla MVC) o mais referenciado (FOWLER, 2006). Cada camada é responsável por uma funcionalidade bem específica. Em particular, as camadas responsáveis pela modelagem do domínio do problema e pela persistência deste domínio são fortemente relacionadas. Isso acontece mesmo quando os modelos de dados são diferentes (modelo orientado a objetos e modelo relacional).

A arquitetura em camadas tradicional tem se demonstrado limitadora em termos de escalabilidade. Tipicamente, a camada responsável pela persistência dos dados fica sobrecarregada para atender muitas requisições simultâneas. Além disso, não existe uma distinção entre operações de persistência e operações de leitura, portanto cria-se uma estrutura demasiadamente complexa para atender ambas as operações. Sendo assim, torna-se mais difícil em implementar otimizações, justamente por existir um acoplamento forte entre operações de naturezas completamente distintas (FOWLER, 2005).

Uma das alternativas existentes na literatura baseia-se no padrão de projeto *Command Query Resposability Segregation* (CQRS) e na técnica de *Event Sourcing* (ES). Ocorre, no entanto, que esta alternativa não se limita a ser uma outra forma de organizar os mesmos elementos. É necessária uma mudança radical na forma de modelar o domínio da aplicação e a persistência deste domínio (YOUNG, 2011).

A proposta desta monografia de conclusão de curso é apresentar uma série de recomendações baseado nas dificuldades encontradas ao montar uma arquitetura baseada integrando CQRS e ES, a fim de facilitar o seu aprendizado.

1.1 Objetivo Geral

O objetivo principal deste trabalho é apresentar um conjunto de recomendações sobre o desenvolvimento de aplicações para web que utilizem os conceitos de CQRS e ES.

1.2 Objetivo Específico

Os objetivos específicos do presente trabalho são:

- Desenvolver um conjunto de recomendações para o desenvolvimento de uma arquitetura integrando CQRS e ES;
- Desenvolver um guia prático de implementação da arquitetura no formato de um estudo de caso, contendo um exemplo completo do fluxo de operações de uma funcionalidade de leitura e uma de escrita e fazendo referências as recomendações apresentadas;

1.3 Organização do texto

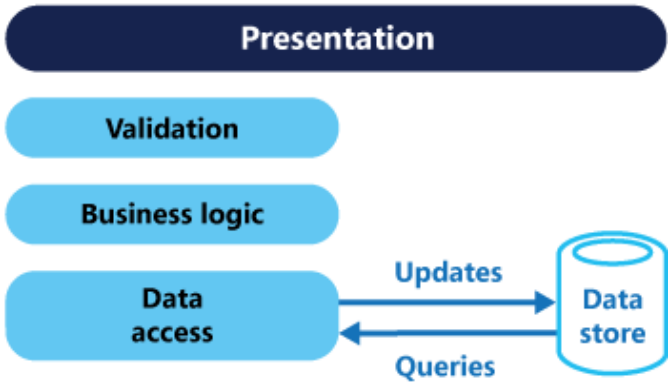
O texto desta monografia é organizado primeiramente com o capítulo de fundamentação teórica sobre os temas envolvidos, com o objetivo de dar ao leitor o conhecimento necessário dos padrões tecnológicos . O segundo capítulo fornece um conjunto de recomendações para a arquitetura proposta, explicando dúvidas que alguém que implemente este padrão possa vir a ter. O terceiro capítulo é a implementação das recomendações do capítulo anterior. É mostrado um exemplo prático e descrito passo a passo. O quarto capítulo descreve os testes realizados e os resultados obtidos. E o último capítulo é referente às conclusões obtidas ao final desta monografia e trabalhos futuros.

2. Fundamentação teórica

2.1 CQRS

Em sistemas tradicionais de gerenciamento de dados, tanto a atualização dos dados (comandos) como as consultas são executados sobre o mesmo conjunto de entidades. As operações CRUD (*Create, Read, Update, Delete*) são aplicadas sobre o mesmo modelo. A Figura 1 mostra um exemplo de arquitetura utilizando CRUD tradicional. O usuário consegue criar, deletar e atualizar informações através da camada de apresentação. Essas operações serão passadas para a camada do domínio da aplicação, que irão executar a validação e processamento de regras de negócios, antes de se conectar com a camada de dados para armazenar ações do usuário em, por exemplo, um banco de dados relacional.

Figura 1 - Utilizando CRUD tradicional



Fonte: Microsoft - CQRS Journey[1]

Em muitos sistemas, esse tipo de arquitetura é suficiente. Porém, conforme eles crescem em complexidade e tamanho, fica cada vez mais difícil ter controle sobre seu nível de performance, manutenção, escalabilidade e conseguir um

[1] Disponível em : https://msdn.microsoft.com/en-us/library/dn568103.aspx**.** Acesso em: maio 2015

desacoplamento entre as camadas. Não é possível realizar modificações em uma camada sem realizar modificações em outra.

CQRS é a sigla para *Command Query Responsibility Segregation* que significa a separação entre o modelo de leitura e o de escrita no desenvolvimento de uma aplicação (YOUNG, 2011). Essa separação de papéis pode levar a uma arquitetura mais eficiente. A ideia por trás desse padrão é deixar bem definido o comportamento do usuário: ou ele deseja realizar uma ação que alterará o estado de um objeto ou ele deseja apenas obter uma informação (sem alterá-la). Cada vez que o lado da escrita recebe, valida e aceita um comando para modificar dados, o sistema irá enviar a mesma atualização para o banco de dados no lado de leitura, mantendo assim, ambos os lados em sincronia. Na arquitetura tradicional, o processamento de ambos ocorre sobre o mesmo modelo de dados.

Segundo Greg Young (YOUNG, 2011), um dos criadores do CQRS, este padrão pode ser definido como a simples criação de duas classes onde anteriormente havia apenas uma. Uma classe conterá apenas métodos que alteram o estado (valor dos atributos) do objeto. Outra classe conterá apenas métodos que fazem a leitura dos valores dos atributos do objeto. Portanto, utiliza-se dois modelos de dados diferentes, um para a atualização dos dados(comando) e outro para consulta dos mesmos, ambos agora com necessidades diferentes**.** Greg Young descreve algumas propriedades decorrentes desta separação (YOUNG, 2011):

- **Consistência:**
    - Comando - É mais fácil processar transações com dados consistentes do que lidar com dados falhos que uma eventual consistência destes na base de dados possa trazer.
    - Consulta - Para muitos sistemas, não tem problema o lado de leitura ficar inconsistente por um breve período de tempo (em inglês, *eventual consistence*).

- **Armazenamento dos dados**:
    - Comando - No lado da escrita, o processamento de transações se estivesse em uma estrutura relacional, iria armazenar os dados de uma maneira normalizada, provavelmente na terceira forma normal. Se utilizado em conjunto com *Event Sourcing*, seria apenas a inserção de um evento.
    - Consulta - O lado da leitura armazena os dados desnormalizados, minimizando o número de relações. Em uma estrutura relacional, ficaria na primeira forma normal.

- **Escalabilidade**:
    - Comando - Na maioria dos sistemas, principalmente web, os comandos executam um número bem baixo de transações. Escalabilidade então não é crítico.
    - Consulta - O lado da leitura, principalmente web, executam a maioria das transações. Escalabilidade normalmente é necessária no lado de leitura.

2.1.1 Conceitos

2.1.1.1 Lado Leitura

Neste lado os métodos só serão capazes de recuperar dados. Udi Dahan, outro pesquisador que junto a Greg Young fez grandes pesquisas no que o padrão CQRS é capaz de fornecer, questiona que se os dados que estão sendo mostrados ao usuário são antigos, então é realmente necessário ir para a base de dados mestre? Por que transformar essas estruturas da terceira forma normal para objetos do domínio se nós queremos apenas dados? (DAHAN, 2015).

Como não é necessário recuperar um objeto inteiro, você pode fazer uso de DTOs (*Data Transfer Objects*). Um DTO é um padrão para o transporte de dados entre diferentes componentes de um sistema. Em outras palavras, seria uma classe que possui exatamente o necessário para um determinado processo, evitando passar uma entidade complexa ou cada atributo individualmente. Sendo assim, é possível recuperar DTOs diretamente da base de dados, sem a necessidade de realizar nenhuma transformação.

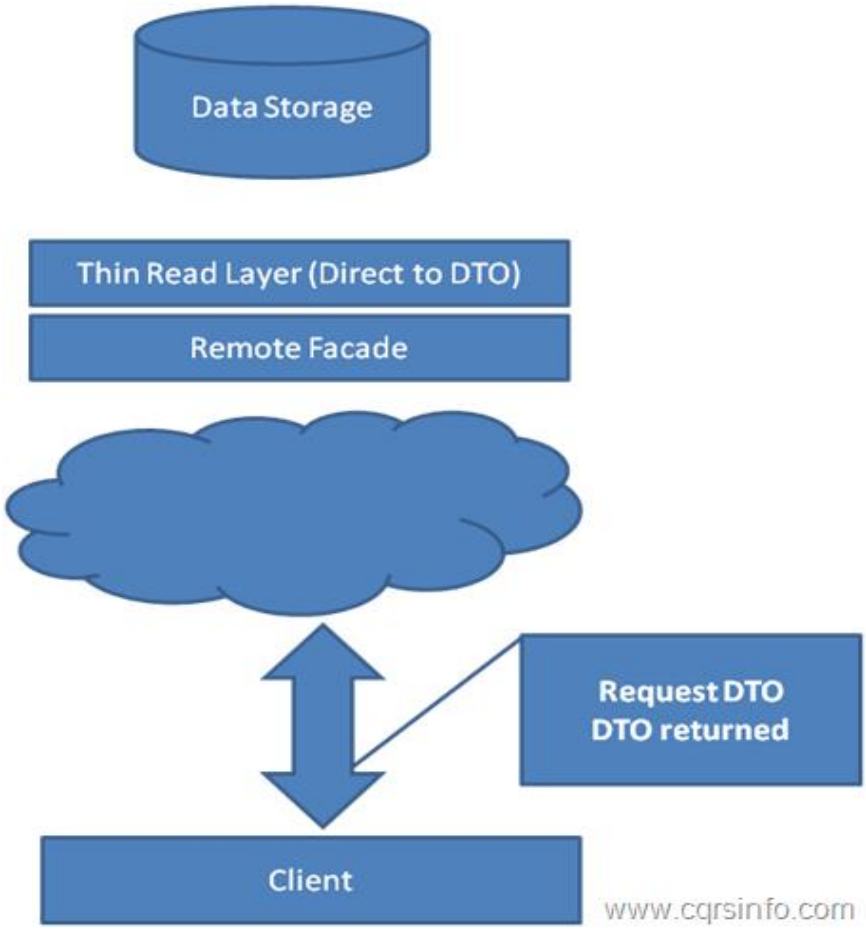
Udi Dahan recomenda uma estrutura para o armazenamento dos dados em que nela os dados estarão fora de sincronia com a base de dados mestre, por que realmente não importa se eles estão um pouco desatualizados. É possível criar uma tabela para cada visão, fazendo então apenas um "`select * from minhavisao`" e ligar o resultado com a tela. Essas tabelas estão em um formato não normalizado, sem relacionamentos.

A ideia através do lado de leitura é dar suporte a vários usuários ao mesmo tempo, tirando todo o processamento pesado e efetuando consultas simples realizadas através de uma recuperação direta dos dados com DTO's, a fim de

otimizar as operações de consulta que normalmente são as mais frequentes nos sistemas utilizados.

A figura 2 mostra como seria o modelo do lado da leitura.

Figura 2 - Lado da leitura



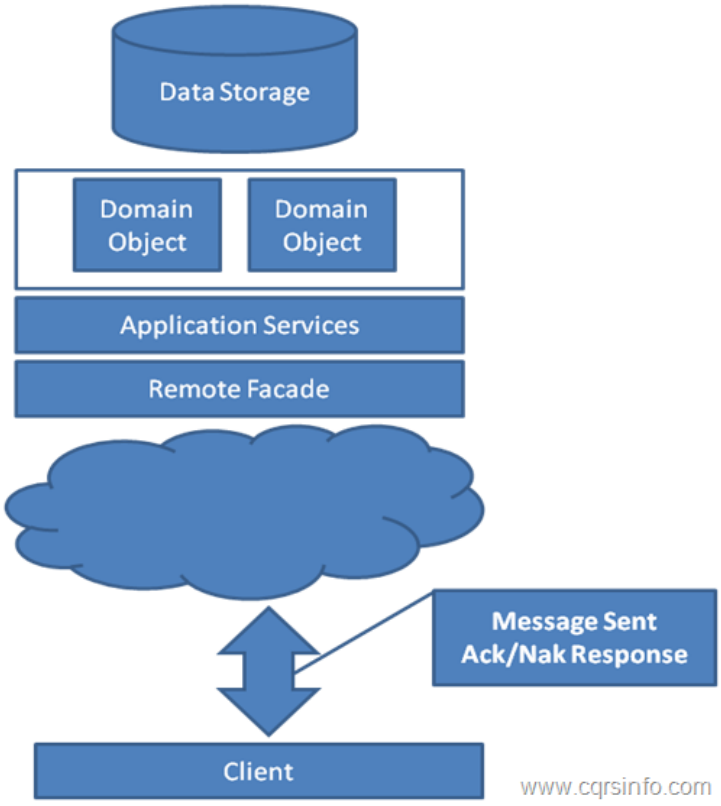
Fonte: CQRS Introduction[2]

[2] Disponível em: https://cqrs.wordpress.com/documents/cqrs-introduction/. Acesso em: maio 2015

2.1.1.2 Lado Escrita

Neste lado da aplicação é possível alterar o estado de um objeto através de um comando. O domínio agora foca somente no processamento dos comandos. O cliente envia comandos que fornecem informações às entidades sobre o que deve ser feito. Comandos são imperativos, por exemplo: “mudeNome, criePedido”. Esses comandos acionam um ‘manipulador de comandos’. Este deve ser responsável em publicar uma mensagem, avisando, em algum momento futuro, o lado da leitura que houve uma mudança e que ele deve atualizar-se.

Figura 3 - Lado da escrita



Fonte: CQRS Introduction[3]

[3] Disponível em: https://cqrs.wordpress.com/documents/cqrs-introduction/. Acesso em: maio 2015

Um comando representa o desejo do usuário. Ele sempre contém o identificador afetado pelo comando. A resposta indicará se o comando foi aceito ou não (se não foi violada nenhuma regra de negócio).

2.2 *Event Sourcing*

No desenvolvimento de um sistema é necessário decidir de que forma serão armazenadas as informações. Em certos contextos de sistemas armazenar apenas o estado atual da aplicação pode ser suficiente, em outras talvez seja necessário algo a mais. A técnica de *Event Sourcing* representa esse “algo a mais”, pois é possível reconstruir o estado da nossa aplicação em qualquer instante do tempo, sendo esse tipo de informação muito importante para diversas finalidades, as quais serão apresentadas em tópicos subsequentes.

2.2.1 Conceitos

Uma abordagem utilizando *Event Sourcing* está preocupada com as mudanças/eventos que ocorrem durante a execução do sistema, persistindo tais informações de forma que posteriormente possa ser construído, em qualquer instante do tempo, o estado da aplicação. *Event Sourcing* fornece a garantia que todas as mudanças que a aplicação venha a sofrer seja gravado em uma sequência de eventos. (FOWLER, 2005)

Os eventos representam ações que já aconteceram na aplicação, ou seja, eles só podem e devem ser gerados quando não existem mais validações de negócio a serem feitas.

Eventos possuem um tamanho pequeno, e podem ser facilmente armazenados em qualquer lugar, talvez em um bloco de texto ou em um banco de dados relacional. Mas é muito importante ressaltar a preservação destes eventos que serão salvos, afinal eles possuem valores importantes do negócio. Portanto, é importante considerar salvar de forma permanente os eventos, e assim nunca os deletar.(FOWLER, 2005).

Segundo artigo publicado pela Microsoft (CQRS Journey, 2012), os benefícios para aplicação decorrente do uso da persistência de eventos são:

Desempenho: Como os eventos são imutáveis, você pode apenas realizar operações de adição (*append*), quando for salvá-los. Os eventos são também objetos simples, autônomos. Ambos fatores podem levar a um melhor desempenho e escalabilidade para o sistema do que as abordagens que usam modelos de armazenamento relacional muito complexos.

Simplificação: Eventos são simples objetos que descrevem o que aconteceu no sistema. Por simplesmente salvar eventos, você está evitando as complicações associadas à persistência de objetos do domínio complexos numa base relacional.

Trilha de auditoria: Eventos são imutáveis e armazenam a história completa do estado do sistema. Como tal, eles podem prover uma trilha de auditoria detalhada do que aconteceu dentro do sistema.

Integração com outros sub-sistemas: Seu armazenador de eventos pode publicar os eventos para notificar outros sub-sistemas interessados nas mudanças ocorridas no estado da aplicação. Mais uma vez, a base de eventos fornece um registro completo de todos os eventos que foram publicados para outros sistemas.

Derivando valores de negócio adicional a partir do histórico de eventos: Ao armazenar eventos, adiciona-se a capacidade de determinar o estado do sistema em qualquer instante do tempo através da consulta de eventos de um objeto do domínio neste mesmo instante. Isto lhe permite responder perguntas históricas sobre o negócio do sistema. Além disso, não se sabe que tipos de perguntas o usuário pode querer fazer sobre informações armazenadas em um sistema. Se forem armazenados eventos, você não está descartando informações que possam revelar-se valiosa no futuro.

Solução de problemas de produção: Você pode utilizar o armazenamento de eventos para solucionar problemas em um sistema de produção, levando uma cópia da base de eventos de produção e reproduzi-la em um ambiente de testes. É possível saber o tempo em que um problema ocorreu no sistema de produção, então é possível repetir o fluxo de eventos até o ponto em que aconteceu este erro.

Corrigindo erros: Quando encontrado um erro de codificação que faz com que o sistema calcule um valor incorreto. Ao invés de ser corrigido o erro de codificação e ajustado o valor do dado armazenado, é corrigido o erro de codificação e reproduzido o fluxo de eventos para que o sistema calcule corretamente o valor com base nesta nova versão do código.

Testando: Todas as mudanças de estado em seus agregados são registradas como eventos. Portanto, é possível testar se um comando teve o efeito esperado apenas verificando o(s) evento(s).

Uma questão que precisa ser levantada é em relação à nomenclatura dada aos eventos. Como dito anteriormente, os eventos representam ações já concretizadas na aplicação, portanto se deve nomeá-las de um modo que expresse,

com precisão, o que aconteceu, além de facilitar a comunicação entre os membros da equipe. Para a maioria dos autores a sugestão é colocar o verbo no passado juntando com o sufixo *Event* (Evento), por exemplo, `UsuarioLogadoEvento` ou `MusicaAdicionadaEvento`.

Além da questão da nomenclatura dos eventos, é necessário definir a estrutura de persistência dos mesmos. Nesta questão, pode ser definida a utilização de apenas duas tabelas, uma referente aos agregados e a outra referente aos eventos em si. Em relação à tabela de agregados, cada registro de sua composição representa unicamente um agregado da aplicação, possuindo um coluna referente ao `aggregateID` e uma para a última versão de evento relacionado a este agregado, esta, especificamente, serve para o controle de concorrência dos eventos. Além disso, é possível ter uma coluna, opcional, para persistir o nome qualificado dos agregados, podendo esta informação ser utilizada para diversos fins.(BUILDING an event storage,2010[4]). No formato de imagem, esta tabela poderia ser representada como mostra a Figura 4:

Figura 4 - Tabela de agregados

	aggregate_id	type	version
▶	e68e4410-012c-421c-9b22-f52c72e4db7c	projeto.tcc.dominio.entidades.usuario.Usuario	1444770802373

Fonte: elaborado pelo autor(2015)

A outra tabela refere-se unicamente aos eventos. É nesta tabela que os dados dos eventos são efetivamente persistidos. Pode-se definir a estrutura dessa tabela contendo as seguintes colunas: um `aggregateID` (faz referência à tabela de agregados), uma coluna no formato de *blob* (representa os dados em si) e uma

[4] Disponível em: https://cqrs.wordpress.com/documents/building-event-storage/.

coluna de versão (representa a versão do agregado no momento da geração do evento). Esta tabela pode ter um ou múltiplos registros para um agregado, uma vez que ela armazena todos os eventos gerados pelo mesmo. Esta tabela poderia ser visualizada como mostrado na Figura 5. (Observação: A última coluna apresentada na tabela abaixo foi uma decisão específica necessária para o estudo de caso realizado. Portanto, não existe nenhuma referência por parte dos autores mencionados sobre o uso dessa coluna.)

Figura 5 - Tabela de eventos

ideventstore	aggregate_id	events	version	groupVersion
486	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444397535174	1444397535174
487	3411275c-123f-4724-a53b-19806ffde99d	BLOB	1444397667990	1444397667990
488	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444418864869	1444418864869
489	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444419383601	1444419383601
490	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444420880052	1444420880052
491	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444420889403	1444420889403
492	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444420891573	1444420891573
493	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444420892208	1444420892208
494	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444420892368	1444420892368
495	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444420896439	1444420896439
496	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444420898890	1444420898890
497	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444420989977	1444420989977
498	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444420994253	1444420994253
499	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444420995483	1444420995483
500	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444422457817	1444422457817
501	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444422512875	1444422512875
502	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444422514603	1444422514603
503	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444422515993	1444422515993
504	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444423037375	1444423037375
505	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444423039948	1444423039948
506	e1c3e517-c912-4b35-bb00-f82733736c45	BLOB	1444423152054	1444423152054

Fonte: elaborado pelo autor(2015)

2.2.2 SNAPSHOTS

Snapshots representam o estado de um agregado em um instante do tempo, ou seja, se tem propriamente todos os valores dos atributos de um agregado na forma de um registro da tabela de eventos.

O uso de *Snapshots* neste tipo de arquitetura pode ser definido como um “extra” ou “opcional”. A sua utilização pode ser recomendada quando se identifica perda de desempenho na reconstrução do(s) agregado(s), visto que se pode, em algum momento da utilização do sistema, ter uma quantidade consideravelmente grande de eventos e que a reconstrução de todos os eventos associados a um agregado pode se tornar um processo custoso e demorado para aplicação.

O processo de reconstrução de um objeto quando não utilizado a heurística de *Snapshots* é começar a ler os eventos desde o início (primeira versão associada) até o fim dos eventos. Como dito anteriormente, pode haver milhares de eventos associados, o que poderia tornar o processo de reconstrução demorado. Quando se faz uso de *Snapshots* se começa a ler do final, utilizando o último evento associado, até encontrar o *Snapshot* desejado, aplicando, ou não, os eventos já lidos.

2.3 *Domain Driven-Design (DDD)*

A parte mais difícil no desenvolvimento de softwares robustos não é sua implementação e sim a compreensão do domínio que o engloba. *Domain Driven Design* é uma metodologia para abordar domínios com alta complexidade. Tem como seu principal foco o domínio do projeto. O domínio é o ponto central de qualquer aplicação.

Segundo Evans, criador do DDD, o coração do software está na sua capacidade de resolver problemas relacionados ao domínio para o seu usuário. Todas as outras características, por mais vitais que possam ser se apoiam nessa finalidade básica (EVANS, 2003, p4).

Dependendo do projeto, o nível de complexidade do negócio pode ser muito alto. Sendo assim, são necessárias técnicas para tentar minimizar as dificuldades e simplificar o problema. O objetivo do *Domain Driven Design* é dividir um negócio de alta complexidade em componentes que sejam gerenciáveis, levando em conta escalabilidade e consistência.

2.3.1 Conceitos

Ter uma linguagem onipresente entre todas as pessoas envolvidas no desenvolvimento de um software é importante, pois facilita a comunicação e evita à perda de informações valiosas. Evans acredita que é extremamente importante todos os membros falarem a mesma linguagem durante o dia-a-dia. Segundo ele, a tradução enfraquece a comunicação e torna anêmica a assimilação do conhecimento.

Quando os desenvolvedores entendem a fundo o modelo, então é mais simples e fácil de melhorar/refatorar o código de modo que o mesmo reflita de maneira natural o negócio. Além disso, segundo Evans, a UML nem sempre é tão boa para representar regras de negócio muito complexas. Então os técnicos perdem muito tempo tentando adequar o negócio com as representações disponíveis de UML e no fim se gasta tempo e esforço e não se consegue representar de uma maneira satisfatória e de fácil compreensão por outros membros da equipe. Para Evans, diagramas UML são muito bons em comunicar as relações entre os objetos, e são fiéis em mostrar as interações, mas não transmitem as definições conceituais desses objetos (EVANS, 2003, p 32).

Entidades devem sempre refletir o negócio, ou seja, elas devem ser puras e refletir somente o que elas foram criadas para refletir. Quando o desenvolvedor sentir que determinada coisa (atributo, método, etc) não faz sentido em uma entidade talvez seja melhor parar, pensar e achar um lugar melhor para isto, pois, caso seja adicionado isto a uma determinada entidade, talvez a mesma perca o seu real significado e posteriormente novos colaboradores não consigam entender o que aquela entidade deveria representar do negócio.

Após apresentar os principais motivos que alavancaram o surgimento da metodologia, podem ser definidos seus elementos básicos. São eles:

- **Entidades**: Objetos só fazem parte deste tipo de classificação se estiver nitidamente claro que necessitam de uma identidade no sistema. São os principais elementos que refletem o domínio.
- **Objetos de Valor**: Ao contrário das entidades, os Objetos de Valor são elementos que não possuem identidade e são imutáveis. Em Java, um exemplo deste tipo de classificação é a classe String, pois, não

possui uma identidade definida e, além disso, suas instâncias são imutáveis. Quando se concatena duas strings, por exemplo, é criado uma nova instância com os valores combinados.

- **Agregados**: Elementos deste tipo englobam Entidades e Objetos de Valor em uma única classe, servindo para preservar a integridade do modelo. Além disso, existirá uma classe que servirá como raiz do agregado, filtrando todas as chamadas para classes internas do agregado.
- **Repositórios**: São elementos (classes) que gerenciam o ciclo de vida de outros objetos no que diz respeito às operações de criação, remoção e alteração (persistência) de objetos.
- **Serviços**: Os serviços são classes criadas única e exclusivamente para implementar as funcionalidades do negócio ou também como é chamado, a lógica do negócio. Não guardam estado.
- **Fábricas**: Certas classes são demasiadamente complexas de se criar. Neste caso se abstrai toda lógica de criação e transfere essas responsabilidades a classes deste tipo.
- **Módulos**: São formas de estruturar nosso projeto. Para Evans, a escolha de módulos deve levantar em conta um conjunto coeso de conceitos, pois isso normalmente gera um baixo acoplamento entre os mesmos. Além disso, os módulos e seus nomes devem refletir uma visão aprofundada do domínio (EVANS, 2003, p 105).

Além dos elementos básicos da metodologia é preciso definir a arquitetura geral da aplicação. Para Evans, a arquitetura de um sistema baseado em DDD teria as seguintes camadas: uma camada de interface com usuário (UI), uma camada de

aplicação, uma camada de domínio e uma camada de infraestrutura. Assim, diante desta arquitetura ele define o que cada camada deve fazer, da seguinte forma:

- **Interface com usuário**: Representa os resultados ao usuário e captura comandos do mesmo, podendo o usuário ser humano ou outro sistema de computador;
- **Aplicação**: Apenas coordena as entradas do usuário e ativa os modelos responsáveis para tratar estas entradas, não deve conter regras de negócio e nenhuma informação do estado de processamento do negócio;
- **Domínio**: Representa os conceitos e regras do negócio, segundo o autor esta camada seria o coração do software. Aqui é onde deverá ser feito a implementação das regras/conceitos do domínio da aplicação. Além disso, invoca chamadas da camada de infra-estrutura.
- **Infraestrutura**: Fornece os recursos técnicos para as camadas superiores como envio de mensagens e persistência de dados.

3. Recomendações

Este capítulo apresenta um conjunto de recomendações relacionadas a pontos importantes de decisão no desenvolvimento de aplicações que utilizam ou utilizarão uma arquitetura integrando *Command Query Responsability Segregatio*n e *Event Sourcing.* Portanto, foi desenvolvido um conjunto de 10 recomendações separadas em cinco grupos distintos, as quais estão relacionadas à: modelagem da base de dados, recuperação de informações, consistência, concorrência e *frameworks.*

3.1 Grupo 1 – modelagem da base de dados:

3.1.1 Modelagem das tabelas da base de escrita.

Seguiu-se a convenção dos principais autores, os quais recomendam a utilização de apenas duas tabelas para o armazenamento de eventos.

A primeira delas representa os agregados da aplicação, ou seja, cada agregado da aplicação terá apenas um único registro correspondente nesta tabela. Sua estrutura é composta de um atributo `aggregateID`, para identificar unicamente um agregado, um atributo *type* que representa o nome qualificado da classe do agregado e um atributo *version*, o qual representa em qual versão o agregado está.

A segunda tabela representa os eventos em si. Sua estrutura é composta de identificador auto incremental gerado pelo banco, um atributo `aggregateID` para criar a ligação entre os eventos e seu agregado, o terceiro atributo é *events*, o qual representa os dados serializados presentes no evento, no formato de *BLOB*, o próxima atributo é *version*, o qual representa a versão do evento e o último atributo é o *groupVersion*, o qual foi criado por ter sido identificado a necessidade de criar uma ligação entre eventos gerados pelo mesmo comando. Sendo assim, este

atributo armazenará a versão do primeiro evento de uma lista de eventos produzida por um mesmo comando.

3.1.2 Motivos para persistência de eventos em base de dados relacional e não em arquivos.

Os motivos principais estão relacionados a questão do tamanho máximo que um arquivo pode ter, questões de segurança (controle de acesso, restrições de integridade) e tolerância a erros e falhas (quedas de energia, por exemplo). Além disso, pelas próprias definições do padrão CQRS, os principais gargalos em aplicações estão relacionados com a leitura de dados e não na sua persistência, então, ainda que para inserir um novo registro num arquivo de texto fosse necessário apenas um "*append*", a consulta seria muito mais rápida utilizando bancos de dados relacionais, principalmente pelo benefício de indexação de colunas.

3.2 Grupo 2 – recuperação de informações:

3.2.1 Situações para a realização de busca de valores no lado de leitura e no lado de escrita.

De um lado pode-se acessar diretamente DTOs, com os quais o acesso é praticamente instantâneo, e de outro se tem registrado todos os eventos. Mas quando se deve usar um e quando usar outro? Utiliza-se o *login* como exemplo. Na execução de um *login* é necessária uma busca dos valores do usuário para efeitos de validação. Mas em qual modelo de dados é feito a busca? Uma DTO no lado de leitura com *login* e senha? Ou se busca os eventos correspondentes para a

reconstrução do estado mais atual de um objeto `Usuario`? Bom, a conclusão feita é que depende da funcionalidade alvo. No caso do *login*, foi optado por utilizar uma mescla dos dois artifícios. Buscou-se no DTO o identificador do agregado correspondente ao *login* digitado. Então, foi feita uma busca pelos eventos correspondentes a esse identificador para transformar esses eventos em um objeto `Usuario` e validar a senha correspondente. Por isso, é importante a precisão dos dados neste tipo de validação, sendo necessário buscar os eventos associados ao usuário.

De modo geral, foi utilizada a base de eventos para validações, a base de leitura para resumos de informações para o usuário, e, em alguns casos específicos, como o citado acima, se fez necessário uma mescla de ambos.

3.2.2 Convertendo eventos em objetos.

Como a aplicação foi desenvolvida através da linguagem Java (característica apresentada no capítulo 4), é feito o uso do artifício da reflexão para implementar essa conversão. Outras linguagens devem possuir artifícios semelhantes para resolver esta situação. A questão é a necessidade de, dinamicamente, invocar o método (no estudo de caso chamado de "`aplicaMudanca`") com diferentes argumentos (neste caso os diferentes argumentos são diferentes implementações de eventos. Ex: `UsuarioCadastradoEvento`, `UsuarioLogadoEvento`, etc).

Descrevendo mais especificamente, o algoritmo ireta sobre a lista de eventos (histórico do agregado) e para cada evento dessa lista é utilizado o artifício `method.invoke()` da API de reflexão, passando o evento atual. Assim, a linguagem identificará qual método "`aplicaMudanca`" da classe `RestauradorAtributos(`padrão de classe responsável em implementar todos os

métodos que aplicam mudanças a uma entidade do sistema) de algum agregado específico que deve ser chamado.

3.2.3 Situações para reconstrução de objetos utilizando eventos.

A necessidade para a reconstrução de objetos utilizando eventos e não apenas *views* (explicado no capítulo 2.1.2.1), parte da exigência de reconstituir, com exatidão, todos os valores do objeto. Isso significa que geralmente um objeto é reconstruído quando se quer validar alguma informação. Entretanto, vale ressaltar que neste caso pode ser utilizado do bom senso, baseado no seu nível de criticidade, pois pode-se utilizar as "*views*" disponíveis no sistema para ganhar uma melhoria no desempenho.

3.3 Grupo 3 - consistência:

3.3.1 Todos os comandos devem possuir um `aggregateID`.

O `aggregateID` é um atributo utilizado como chave para identificar um agregado no sistema. Quando comandos são gerados e aceitos, estes irão ser transformados em eventos. Os eventos representam ações que foram efetuadas em entidades do sistema. Portanto, quando for necessário reconstituir um agregado, é necessário recuperar todos e apenas os eventos relacionados ao agregado, de modo que este fique em um estado consistente e final.

Sendo assim, como os comandos representam o primeiro passo no fluxo de operações do sistema, é importante que estes possuam o `aggregateID`, pois

posteriormente esta informação será repassada quando forem criados os eventos correspondentes.

3.3.2 Sincronização da base de escrita com a base de leitura.

Como o padrão *Command Query Responsability Segregation* separa a arquitetura em “lados” diferentes, faz-se necessário também o uso de bases de dados distintas. Uma, relacionado a escrita, a qual apenas armazena novos eventos, e a outra, relacionada a leitura, a qual apenas disponibiliza informações no formato de tabelas não normalizadas (*views*). Porém, para esta arquitetura funcionar é necessário que exista uma sincronização entre estas diferentes bases de dados, ou seja, em um determinado intervalo de tempo, novos eventos que foram persistidos na base de escrita devem ser transformados e publicados em informações para a base de leitura.

Como apresentado no parágrafo anterior, é necessário realizar uma sincronização entre as bases de dados e também decidir a frequência de tempo em que essa sincronização deve ser realizada. Em aplicações que possuem domínios não críticos é possível ter uma sincronização com tempos mais esparsos e/ou filas de eventos maiores, justamente por se tratarem de dados não críticos, pois, não existe a necessidade de sobrecarga do sistema para manter os dados atualizados em *“real-time”.*

Em aplicações com domínios mais críticos é provável que esta sincronização aconteça mais vezes, justamente por ser necessário garantir a fidelidade dos dados informados pelos usuários com os dados que eles estão querendo consultar/atualizar.

Nesta aplicação, foi feito o uso da tecnologia Future da linguagem Java para realizar a publicação dos eventos na base de leitura. Esta tecnologia fica com a responsabilidade de decidir o momento da publicação, baseado nos recursos disponíveis do sistema. Foi feito o uso desta abordagem pelo domínio da aplicação estar relacionado a um aplicativo de música, o qual, de modo geral, não trabalha com dados críticos, diferentemente de sistemas bancários, de saúde e de transporte.

3.3.3 Usar o atributo *version* dos eventos para garantir a sincronização entre a base de escrita e a base de leitura.

Caso fosse identificada alguma instabilidade em algum período específico, poderiam ser republicados eventos a partir deste mesmo período, fazendo uso do atributo *version* da tabela de eventos, o qual representa o *timestamp* da data em que o evento ocorreu.

3.4 Grupo 4 - concorrência:

3.4.1 Usar a estratégia otimista para tratar o acesso concorrente na validação de comandos.

Portanto, o primeiro comando a chegar terá prioridade na validação. Neste caso pode ser implementado uma fila (*queue*) de comandos.

Na aplicação, foi criado um método "*synchronized*" na classe `PosProcessadorComandos` (classe responsável por executar a validação da versão do comando). Por definição, métodos sincronizados impedem a interferência de outras *threads* que tentam acessar o mesmo recurso, neste caso o método de

validação. Sendo assim, comandos que chegarem posteriormente para executar o método são enfileirados até o termino da validação do comando atual.

3.5 Grupo 5 – frameworks:

3.5.1 Frameworks que ajudam na produção de um sistema CQRS:

Lokad.CQRS: Framework desenvolvido pela Lokad que compartilha sua experiência com CQRS, DDD e ES. Possui componentes e ferramentas que salvam tempo do desenvolvedor. Ele mostra também como construir sistemas distribuídos e escaláveis que possam rodas localmente ou pela nuvem[5].

NCQRS: Framework para .NET, que ajuda a construir aplicações escaláveis, extensíveis e de fácil manutenção utilizando o padrão de arquitetura CQRS. Ele realiza isso disponibilizando uma infraestrutura e uma implementação que facilita a construção da aplicação ao lidar com comandos, modelagem de domínio, event sourcing, etc. Isso ajuda o desenvolvedor abstrair toda a complexidade que há por trás de uma arquitetura que envolva CQRS, DDD e ES, através de anotações, convenções e suporte para configuração[6].

AXON: Desenvolvido para implementadores que querem criar um aplicativo Java com base nos princípios da arquitetura CQRS. É focado em tornar a vida mais fácil, através de anotações, e cenários de testes[7].

[5] Disponível em: http://lokad.github.io/lokad-cqrs/

[6] Disponível em: https://github.com/pjvds/ncqrs

[7] Disponível em: http://www.axonframework.org/

4. Aplicando as recomendações: um estudo de caso

4.1 Preparação

Para este projeto, será desenvolvida uma aplicação exemplo abordando os conceitos/metodologias apresentados ao longo deste documento. O domínio desta aplicação será um serviço de música digital. Este exemplo não visa construir um software comercial nem tampouco ficar preso a certas tecnologias. O objetivo principal é além de ter um domínio claro e coeso, utilizar na prática os padrões CQRS e *Event Sourcing* e a metodologia de desenvolvimento DDD.

O desenvolvimento desta aplicação colocará em prova toda a fundamentação teórica usada neste trabalho. Nesta aplicação exemplo é utilizada a linguagem de programação Java, primeiro pela familiaridade dos autores e segundo porque, conforme Evans cita, “para amarrar a implementação finalmente a um modelo são geralmente necessárias linguagens e ferramentas de desenvolvimento de software que aceitem um paradigma de modelagem, tal como a programação orientada a objetos.” (EVANS, 2003, p 45), que neste caso é o paradigma por trás da linguagem escolhida. Além da linguagem Java, utilizamos também o framework JSF (Java Server Faces) para desenvolvimento da aplicação para WEB e optamos por utilizar o banco de dados MySQL.

4.2 Benefícios

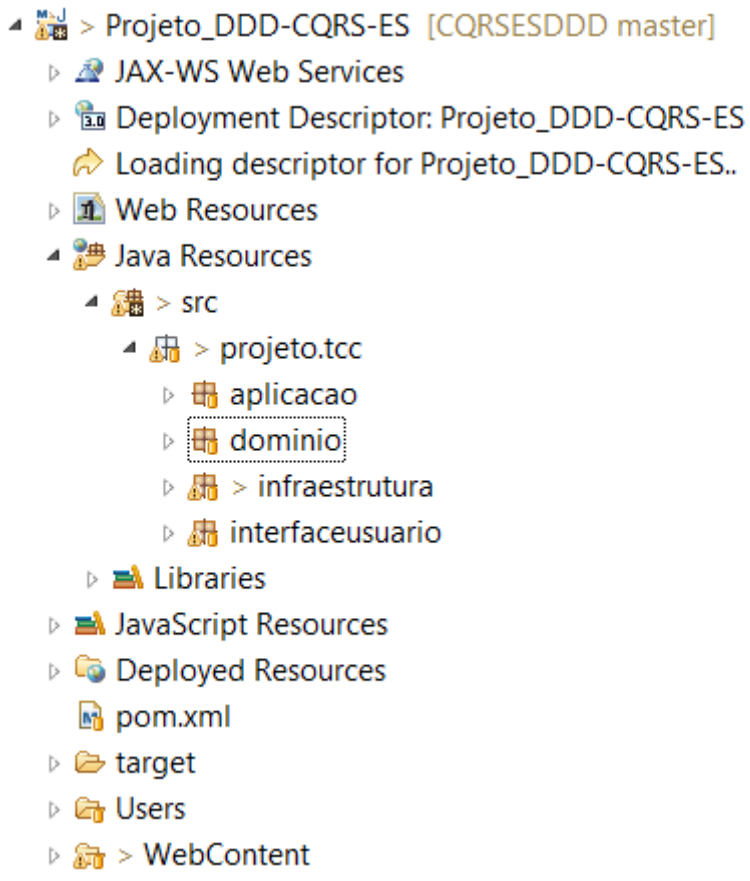
Este tipo de arquitetura não é indicado para resolver qualquer tipo de problema, ela não é uma “bala de prata”. Segundo os principais autores, os maiores retornos do uso desta arquitetura são quando estas estão inseridas em contextos para sistemas de larga escala, com domínios complexos e que necessitam de um ótimo desempenho. Os principais benefícios são:

- Separação clara entre as responsabilidades de cada camada do sistema, possuindo um baixo acoplamento e uma alta coesão;
- Aumento da eficiência nas consultas - Desnormalização da base de leitura, evitando múltiplos joins e conversões para entidades do domínio que são a causa da queda de desempenho na maioria dos sistemas atuais;
- Escalabilidade facilitada - Como esta arquitetura separa o lado de leitura do lado de escrita podemos definir objetivos específicos e que atendam cada parte da melhor maneira possível;
- Permite log e auditoria. - Como possuiremos todos os eventos que ocorreram dentro da nossa aplicação, podemos melhorar o debugging de nossas aplicações e também realizar auditorias. Além disso, este tipo de informação abre novos mercados para o nosso sistema, como por exemplo, extração de padrões comportamentais do usuário.

4.3 Estrutura

Primeiramente vamos definir a estrutura da nossa aplicação. Como a mesma é baseada em DDD, foi dividida nos seguintes pacotes: aplicação(aplicacao), domínio (dominio), infraestrutura e interface com usuário (interfaceusuario), as quais estão refletidas, no formato de pacotes, na Figura 6 a seguir.

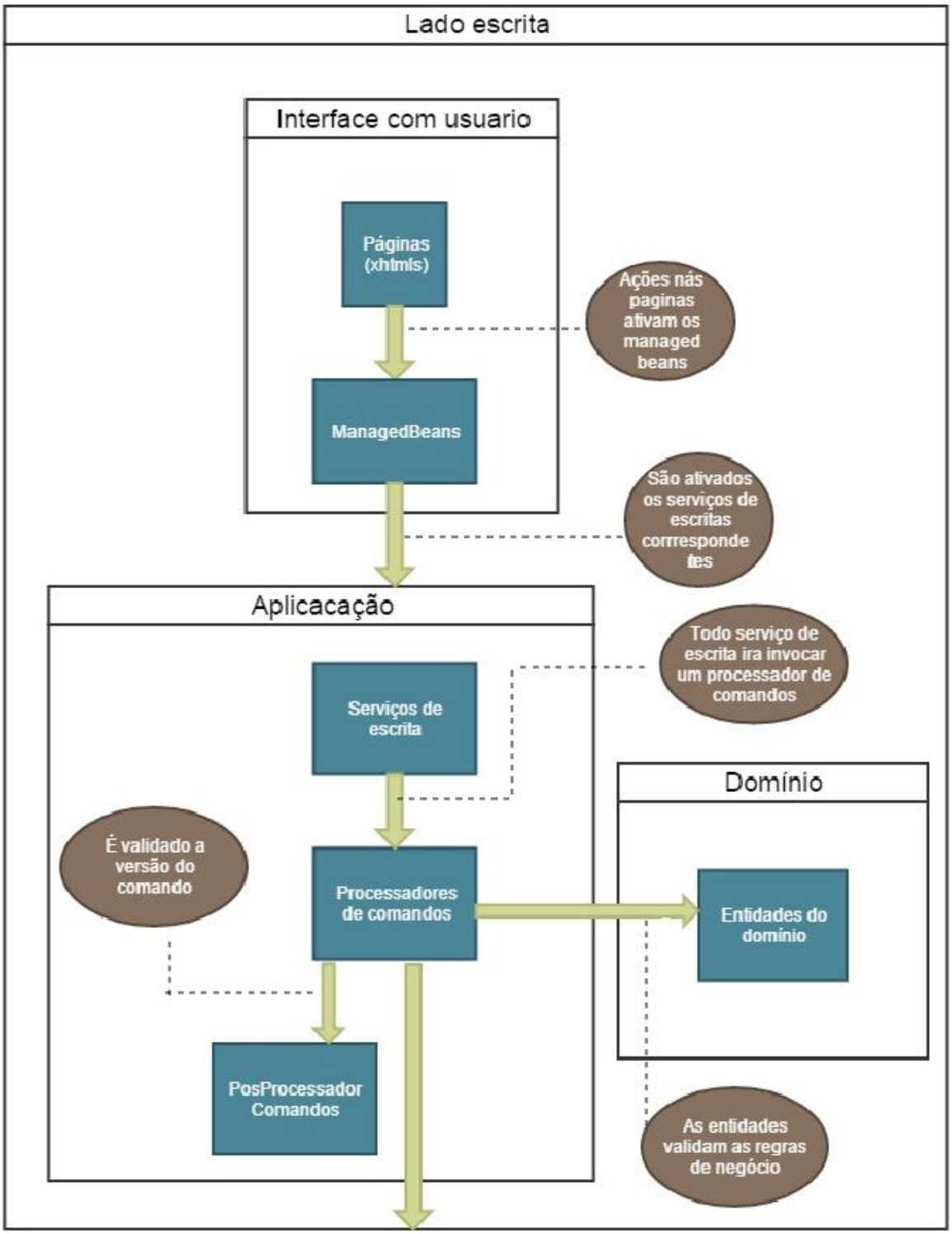
Figura 6 - Estrutura da aplicação



Fonte: Elaborado pelo autor(2015)

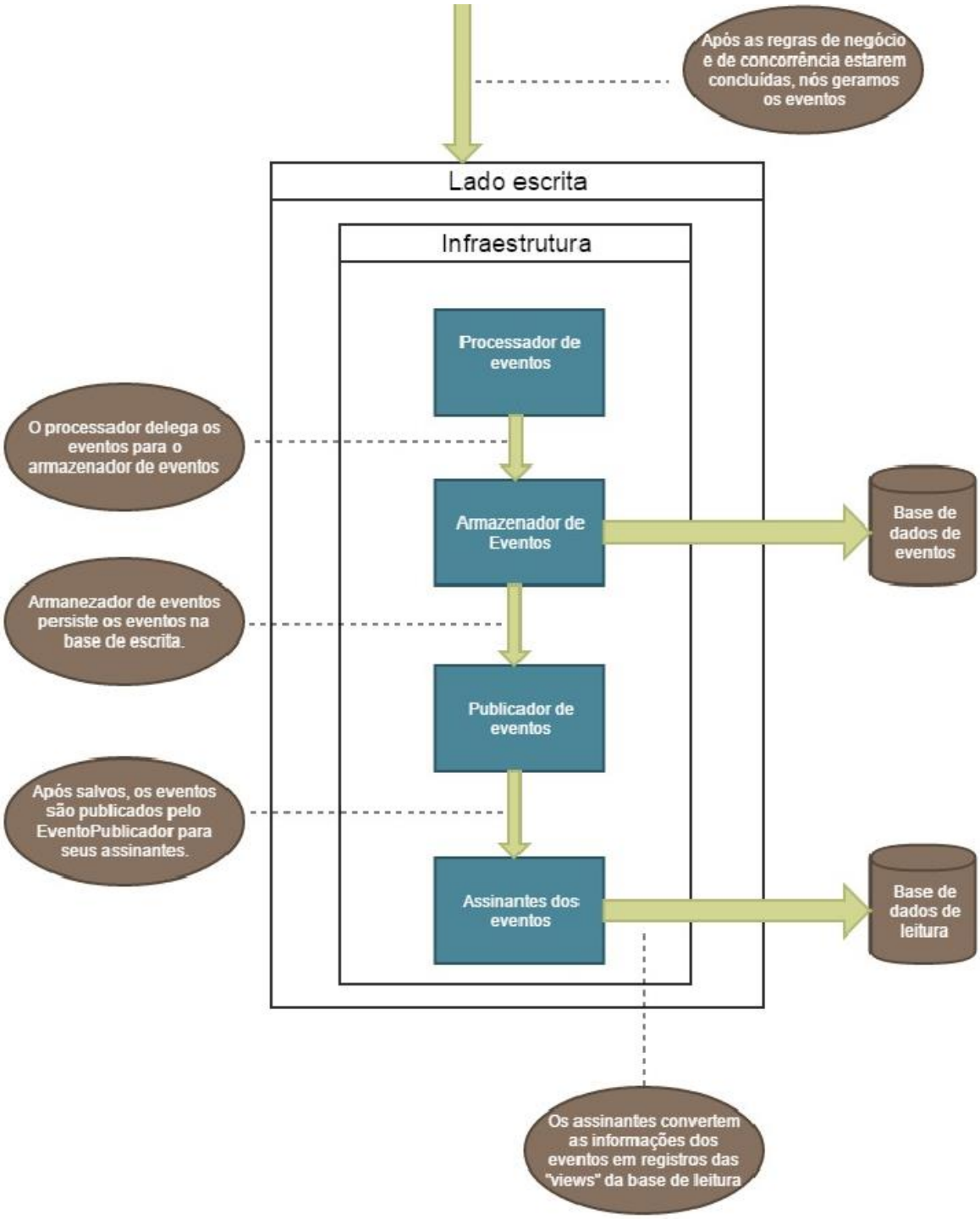
As imagens a seguir representam a arquitetura global da aplicação, nas quais os retângulos representam os principais padrões de classes do sistema e as elipses representam suas relações e comportamentos entre diferentes camadas:

Figura 7 - Representação da arquitetura no lado de escrita - parte A



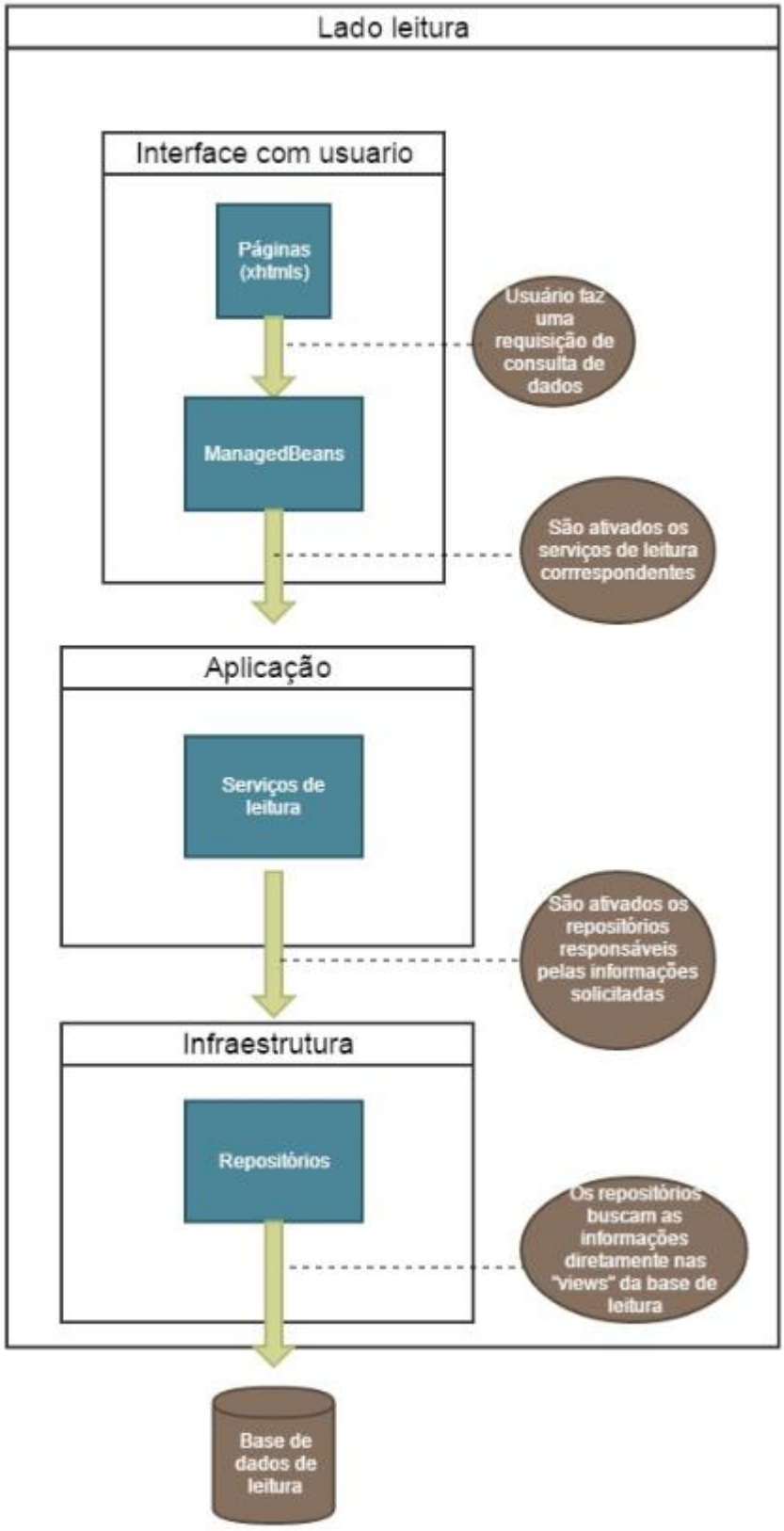
Fonte: Elaborado pelo autor(2015)

Figura 8 - Representação da arquitetura no lado de escrita – parte B



Fonte: Elaborado pelo autor(2015)

Figura 9 - Representação da arquitetura no lado de leitura



Fonte: Elaborado pelo autor(2015)

4.4 Implementação

Esta seção visa apresentar algumas funcionalidades, mais precisamente uma implementação sobre um serviço de leitura e outro sobre um serviço de escrita, de modo que facilite o entendimento para o leitor que não deseja conhecer como todas as funcionalidades do sistema foram implementadas. Vale ressaltar que o resto da implementação das outras funcionalidades pode ser vista diretamente via código com *JavaDocs* de suporte, pois, não queríamos tornar o guia maçante e repetitivo. O código fonte desta aplicação está disponível em: https://github.com/Fellipepasini/CQRSESDDD.

Além disso, vale ressaltar que, apesar de mostrarmos apenas duas funcionalidades do nosso sistema, todas as classes e métodos necessários para à geração, persistência e sincronização de eventos, que servirão de base para as demais funcionalidades, serão apresentados, de modo que esta seção seja simples porém completa.

Em uma abordagem utilizando DDD, o primeiro passo na construção de um software, mencionado no tópico sobre DDD, é o entendimento do domínio/negócio com os especialistas do negócio. Entretanto como este trabalho está voltado ao meio acadêmico, nós mesmos elaboramos o domínio e elencamos os requisitos, ou seja, nós somos os especialistas do negócio e o próprio time de desenvolvimento.

Sendo assim, antes de nos aprofundarmos na implementação, vamos elencar na seção 4.4.1 os requisitos de negócio do cadastramento do usuário:

4.4.1 Requisitos da aplicação

Como requisitos gerais da aplicação, este serviço de música digital deve:

1. Permitir o cadastramento do usuário;
2. Permitir a autenticação do usuário no sistema;
3. Permitir a alteração/atualização dos dados cadastrais do usuário';
4. Permitir a escolha/consulta de músicas do usuário;
5. Permitir a criação de playlists;
6. Permitir a adição de músicas às playlists do usuário;
7. Permitir a seleção/adição de músicas favoritas;

4.4.2 Especificação de negócio - Cadastro de usuário

As regras de negócio relacionadas ao cadastramento do usuário são:

1. No cadastramento do usuário, é obrigatório informar o CPF, nome, data de nascimento, sexo, e-mail, perfil de uso, login e uma senha;
2. O usuário só poderá se cadastrar, caso tenha idade igual ou superior a 18 anos;
3. O usuário deverá, no momento do cadastro, escolher um perfil. Os perfis com suas respectivas regras são:
    a. Gratuito - Usuários deste perfil poderão ter apenas uma "Playlist", ou seja todas as suas músicas farão parte de uma mesma lista de reprodução;

b. Intermediário - Usuários deste perfil devem pagar uma anuidade no valor de 35 reais. Além disso, este perfil disponibiliza a criação de até três “Playlists”.
c. Ilimitado - Usuários deste perfil devem pagar uma anuidade no valor de 60 reais. Além disso, este perfil disponibiliza a criação ilimitada de Playlists, de acordo com as preferências do usuário.

4. O usuário não pode cadastrar uma playlist caso já possua uma playlist com o mesmo nome.

Além da especificação de negócio, precisamos definir a estrutura das tabelas associadas à persistência dos eventos e agregados (lado escrita), pois isto será a infraestrutura mínima necessária para o correto funcionamento da aplicação.

Segundo definições dos principais autores das tecnologias são necessárias a criação de duas tabelas quando estamos construindo o lado da escrita. A primeira delas se refere às informações dos eventos. Em sua composição, faremos uso de cinco colunas, são elas:

- *ideventstore*: Apenas um sequencial (*int* (11)) gerado e incrementado automaticamente pelo banco após a inserção de registros. Serve para identificar unicamente um registro nesta tabela;
- *aggregate_id*: Esta coluna é um dos pilares da arquitetura. Serve para identificar a qual agregado este evento está associado. Definimos seu tipo como um *varchar*(40);
- *events*: Esta coluna contém as informações do evento em si. Utilizamos do artifício da serialização para escrever os dados do evento num objeto do tipo ObjectOutputStream e pegar o *array* de

bytes resultante dessa escrita. Utilizamos do tipo *BLOB* para esta coluna.

- *version*: Esta coluna é um inteiro (*bigint*(20)) que define a ordem de persistência dos eventos. Porém, vale ressaltar, que este inteiro representa o *timestamp* do momento em que o evento ocorreu. Assim, poderíamos, caso necessário, reconstituir algum agregado em qualquer instante do tempo, pois temos todos as “datas” de ocorrência dos eventos.
- *groupVersion*: Esta coluna, especificamente, foi uma criação nossa por um detalhe que verificamos que poderia causar algum problema no futuro. O “problema” é que alguns comandos podem gerar mais do que um evento. Sendo assim no momento da persistência, estes eventos são persistidos de modo transacional, ou seja, todos ou nenhum. Agora imaginemos que queremos reconstituir um objeto até um *timestamp* específico. Nós não poderíamos recuperar apenas um desses eventos, pois quando eles foram persistidos, foi de modo transacional, pois queríamos garantir um estado válido do objeto do ponto de vista do negócio. Sendo assim, quando formos persistir uma lista de eventos para um único comando, utilizamos o *timestamp* (*version*) do primeiro evento e replicamos na coluna *groupVersion* dos eventos subsequentes deste mesmo comando. Assim, poderemos utilizar essa coluna como critério na consulta dos eventos.

O script de criação da tabela de eventos é o seguinte:

```
CREATE TABLE `eventstore` (
 `ideventstore` int(11) NOT NULL AUTO_INCREMENT,
 `aggregate_id` varchar(40) DEFAULT NULL,
 `events` blob NOT NULL,
 `version` bigint(20) DEFAULT NULL,
 `groupVersion` bigint(20) DEFAULT NULL,
 PRIMARY KEY (`ideventstore`)
) ENGINE=InnoDB AUTO_INCREMENT=5172 DEFAULT CHARSET=utf8;
```

Além da coluna dos eventos, é necessária a criação de uma tabela de agregados da aplicação. Basicamente, cada agregado existente na aplicação possuirá um único registro correspondente nesta tabela. Em sua composição, fazemos uso de três colunas, são elas:

- *aggregate_id*: Identifica unicamente um agregado da aplicação e um registro na tabela. É do tipo *varchar*(40);
- *type*: Esta coluna é opcional, mas a consideramos, pois acreditamos que pode facilitar a legibilidade do registro. Sua informação contém o nome qualificado do agregado persistido (ex: 'projeto.tcc.dominio.entidades.musica.Musica'). É do tipo *varchar*(45).
- *version*: Esta coluna é um espelho da coluna *version* da tabela de eventos. Entretanto, vale ressaltar que apenas a versão do último evento persistido para um agregado é colocada aqui. Ou seja, estamos armazenando em qual versão está o agregado. É do tipo *bigint*(20).

O script de criação da tabela de agregados é o seguinte:

```
CREATE TABLE `aggregates` (
 `aggregate_id` varchar(40) DEFAULT NULL,
 `type` varchar(45) DEFAULT NULL,
 `version` bigint(20) DEFAULT NULL,
  KEY `aggregate_id_idx` (`aggregate_id`)
 ) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

4.4.3 Cadastro do usuário - Lado escrita

4.4.3.1 Interface

Inicialmente, é definido a interface de comunicação entre o usuário e o sistema. Foi elaborado um protótipo da tela de cadastro de usuário e este tem a seguinte aparência:

Figura 10 - Imagem da tela de cadastro do usuário

Fonte: Elaborado pelo autor(2015)

Para um cadastro de usuário foi definido que os campos necessários para concluir um cadastro, como visto pela imagem são: nome, CPF, email, sexo, usuario, senha, data de nascimento e um perfil. Sendo assim, no mínimo estes campos farão parte da entidade `Usuario`.

Além disso, foi utilizado dos conceitos de *`ManagedBean`* por ter sido escolhido o *framework* JSF, mas vale ressaltar que essa escolha não tem nada a ver com a proposta deste trabalho, apenas uma facilidade para tratar das questões WEB.

Inicialmente foi criado o *`ManagedBean`* `CriarUsuarioBean`. Este *bean* conterá um DTO chamado `CriarUsuarioDTO`. Este DTO encapsulará as informações (nome, CPF, login, senha, email, data de nascimento, sexo e perfil) fornecidas pelo usuário. Além disso, terá o método `criarCadastro`(), o qual será ativado pelo usuário quando for clicado no botão “Criar”. A implementação deste método é a seguinte:

Figura 11 - Implementação do método `criarCadastro()`

```
40    public String criarUsuario() throws Exception{
41        try{
42        UUID idOne = UUID.randomUUID();
43        servicoUsuarioEscrita.cadastrarUsuario(new CadastrarUsuarioComando(idOne,usuarioDTO));
44        FacesContext context = FacesContext.getCurrentInstance();
45        context.getCurrentInstance().addMessage(null, new FacesMessage("Usuario cadastrado com sucesso"));
46        context.getExternalContext().getFlash().setKeepMessages(true);
47        return "/xhtml/loginUsuarioCPF.xhtml?faces-redirect=true";
48        }catch (Exception e) {
49            FacesContext fc = FacesContext.getCurrentInstance();
50            fc.addMessage(null, new FacesMessage(FacesMessage.SEVERITY_WARN, e.getMessage(), null));
51        }
52        return "";
53
54    }
```

Fonte: Elaborado pelo autor(2015)

Primeiramente, é gerado um UUID (linha 42), o qual será o *aggregateID* do objeto usuário. Após isso, é estabelecido uma conexão com outra camada do sistema, a camada de aplicação (linha 43). Neste caso é chamado a interface `ServicoUsuarioEscrita`. Esta interface especifica o método `cadastrarUsuario`(), fornecendo uma independência entre as duas camadas. A assinatura deste método espera como parâmetro um objeto do tipo `Comando`. Porém, antes de apresentarmos este objeto específico vamos apresentar a estrutura da interface `Comando`, da seguinte forma:

Figura 12 - Interface `Comando`

```
1 package projeto.tcc.interfaceusuario.comandos;
2
3 import java.util.UUID;
4
5 public interface Comando {
6     UUID aggregateId();
7     Long getVersion();
8
9 }
```

Fonte: Elaborado pelo autor(2015)

Todo comando da aplicação deve implementar esta interface, pois, conceitualmente um comando deve ter um *aggregateID* (recomendação 3.2.1) e uma versão. Para o exemplo de cadastrar usuário, é criado o comando `CadastrarUsuarioComando`, o qual possui um construtor da seguinte forma:

Figura 13 - Construtor do comando para criação de usuário

```
24    public CadastrarUsuarioComando(UUID usuarioUID , CriarUsuarioDTO usuarioDTO) {
25        this.usuarioUID = usuarioUID;
26        this.login = usuarioDTO.getLogin();
27        this.senha = usuarioDTO.getSenha();
28        this.cpf = usuarioDTO.getCpf();
29        this.nome = usuarioDTO.getNome();
30        this.email = usuarioDTO.getEmail();
31        this.cdPerfil = usuarioDTO.getCdPerfil();
32        this.dtNascimento = usuarioDTO.getDtNascimento();
33        this.sgSexo = usuarioDTO.getSgSexo();
34        this.version = 0L;
35    }
36
```

Fonte: Elaborado pelo autor(2015)

Pelo exemplo, é possível ver que o construtor da classe `CadastrarUsuarioComando` apenas transfere as informações passadas pelo usuário (`CriarUsuarioDTO`) e o UUID gerado para os seus atributos (linhas 25 à 33). Em suma, é encapsulada a vontade do usuário neste objeto.

Voltando ao serviço de escrita do usuário, pela especificação do DDD, a camada de aplicação serve apenas como uma coordenadora das entradas do usuário e o restante de processamento da aplicação, sem conter nenhuma regra de negócio. Sendo assim, a implementação do método `cadastrarUsuario()` na classe `ServiceUsuarioEscritaImpl` se resume a:

Figura 14 - Método cadastrar usuário na camada de serviço de escrita para usuários

```
1  package projeto.tcc.aplicacao.impl;
2
3  import java.io.Serializable;
19
20 public class ServicoUsuarioEscritaImpl implements ServicoUsuarioEscrita, Serializable{
21
22     private static final long serialVersionUID = -7673302045438349809L;
23     private Usuario usuario;
24
25
26     @Override
27     public void cadastrarUsuario(CadastrarUsuarioComando cadastrarUsuarioComando) {
28         try {
29             new ProcessadorCadastrarUsuarioComando().execute(cadastrarUsuarioComando);
30         } catch (Exception e) {
31             throw new RuntimeException(e.getMessage());
32         }
33
34     }
35
```

Fonte: Elaborado pelo autor(2015)

4.4.3.2 Processando Comandos

Como apresentado na Figura 14, o método `cadastrarUsuario()` apenas recebe o `Comando` gerado e coordena para um novo tipo de classe no sistema, os processadores de comando (linha 29). Todo processador de comandos deve implementar a interface `ProcessadorComandos`.

Esta interface é responsável por definir um método que deverá ser implementado por todos os processadores de comandos da aplicação. O método em questão é o `execute` e possui como argumento a interface `Comando`. Sua definição é da seguinte forma:

Figura 15 - Método execute da interface `ProcessadorComandos`

```
1 package projeto.tcc.aplicacao.comandos;
2
3 import projeto.tcc.interfaceusuario.comandos.Comando;
4
5 public interface ProcessadorComandos {
6
7     void execute(Comando comando) throws Exception;
8
9 }
```

Fonte: Elaborado pelo autor(2015)

O conceito do método *`execute()`* é englobar o carregamento da(s) entidade(s) do domínio envolvido, invocando seus respectivos métodos de negócio que tratam o comando em questão. Após a validação/execução das regras de negócio, é invocado o `PosProcessadorComandos`, o qual é responsável em validar a versão do comando (será apresentada mais para frente).

Após todas as validações serem atendidas (regras de negócio e concorrência) é gerado então o(s) evento(s) associado(s) aos comando(s). Para isto é delegada a responsabilidade para o ProcessadorEventos.

A seguir a implementação do `ProcessadorCadastrarUsuarioComando`:

Figura 16 - Implementação do método execute na classe `ProcessadorCadastrarUsuarioComando`

```
20 public class ProcessadorCadastrarUsuarioComando implements ProcessadorComandos{
21
22     @Override
23     public void execute(Comando comando) throws Exception {
24         CadastrarUsuarioComando cadastrarUsuarioComando = (CadastrarUsuarioComando) comando;
25         Map<String, Object> valores = constroiMapaParametros(cadastrarUsuarioComando);
26
27         Usuario usuario = new Usuario().criarCadastro(valores);
28
29         PosProcessadorComandos.validaVersaoComando(comando);
30
31         List<Evento> eventos = new ArrayList<>();
32         EventoProcessador eventoProcessador = new EventoProcessador();
33         Long version = ControladorVersao.getProximaVersao();
34
35         UUID usuarioID = cadastrarUsuarioComando.aggregateId();
36         UUID musicaID = UUID.randomUUID();
37
38         UsuarioCadastradoEvento usuarioCadastradoEvento =
39                 new UsuarioCadastradoEvento(usuarioID,version, usuario, version);
40
41         PlayListAdicionadaEvento playListAdicionadaEvento =
42                 new PlayListAdicionadaEvento(usuarioID,musicaID, "Default", ControladorVersao.getProximaVersao(), version);
43
44         eventos.add(usuarioCadastradoEvento);
45         eventos.add(playListAdicionadaEvento);
46         eventoProcessador.processarEventos(eventos);
47     }
```

Fonte: Elaborado pelo autor(2015)

Como é possível ver pela Figura 16, primeiramente é feito uma conversão (*cast*) do argumento `Comando` para o comando específico, neste caso o `CadastrarUsuarioComando` (linha 24). Após isso, é necessário construir um mapa (*`Map`*) com os valores vindos do comando (linha 25). Vale ressaltar que se optou por utilizar mapas como parâmetros dos métodos de negócio, mas também poderiam ser outros objetos, tais como DTOs. Portanto, o método `constroiMapaParametros`() apenas repassa as informações contidas no `CadastrarUsuarioComando` para um *`Map`* do tipo `<String, Object>` como é possível ver pela Figura 17:

Figura 17 - *Map* com os valores que serão passados como parâmetro

```
49    private Map<String, Object> constroiMapaParametros(CadastrarUsuarioComando cadastrarUsuarioComando) {
50        Map<String, Object> valores = new HashMap<String, Object>();
51        valores.put("login", cadastrarUsuarioComando.getLogin());
52        valores.put("senha", cadastrarUsuarioComando.getSenha());
53        valores.put("nome", cadastrarUsuarioComando.getNome());
54        valores.put("cpf", cadastrarUsuarioComando.getCpf());
55        valores.put("email", cadastrarUsuarioComando.getEmail());
56        valores.put("dtNascimento", cadastrarUsuarioComando.getDtNascimento().getTime());
57        valores.put("sgSexo", cadastrarUsuarioComando.getSgSexo());
58        valores.put("cdPerfil", cadastrarUsuarioComando.getCdPerfil());
59        return valores;
60    }
61
```

Fonte: Elaborado pelo autor(2015)

4.4.3.3 Domínio - Regras do Negócio

Após construído o mapa de parâmetros, nós invocamos a entidade do domínio responsável por tratar esta funcionalidade do sistema, neste caso a entidade `Usuario` (linha 27), com o método `criarCadastro`(). A implementação deste método é apresentada na Figura 18. Como a entidade do domínio é responsável por cuidar das regras do negócio, significa então que é nela que estas regras serão processadas/validadas.

Figura 18 - Implementação do método `criarCadastro()` na classe `Usuario`

```
111 public Usuario criarCadastro(Map<String, Object> valores) throws Exception {
112
113         Usuario usuario = new RepositorioUsuarioImpl().getUsuarioPorCPF(String.valueOf(valores.get("cpf")));
114         if(usuario != null){
115             throw new UsuarioJaRegistradoException("Já existe um usuário cadastrado com esse cpf.");
116         }
117
118         Date dtNascimento = new Date(Long.parseLong(valores.get("dtNascimento").toString()));
119         Calendar dtNascimentoC = Calendar.getInstance();
120         dtNascimentoC.setTime(dtNascimento);
121         Calendar dtAtual = Calendar.getInstance();
122         if(dtAtual.get(Calendar.YEAR) - dtNascimentoC.get(Calendar.YEAR) < 18){
123             throw new UsuarioMenorIdadeException("É necessário ter mais de 18 anos para utilizar o aplicativo.");
124         }
125
126         usuario = new Usuario();
127         usuario.setLogin(String.valueOf(valores.get("login")));
128         usuario.setSenha(String.valueOf(valores.get("senha")));
129         usuario.setNome(String.valueOf(valores.get("nome")));
130         usuario.setCPF( String.valueOf(valores.get("cpf")));
131         usuario.setEmail(String.valueOf(valores.get("email")));
132         usuario.setSexo(String.valueOf(valores.get("sgSexo")));
133         usuario.setDataNascimento(dtNascimento);
134         usuario.setPerfil(Integer.parseInt(valores.get("cdPerfil").toString()));
135         return usuario;
136     }
```

Fonte: Elaborado pelo autor(2015)

Inicialmente, é definido um método chamado `criarCadastro`(). Para o projeto, um cadastro básico precisa ter um nome, CPF, e-mail, data de nascimento, sexo, perfil, *login* e uma senha. Como o foco ainda está no domínio, estes serão os parâmetros do método. Uma regra simples e importante em um cadastro de usuário é verificar se o mesmo já não está cadastro no sistema. Entretanto, como a aplicação do projeto ainda está sendo iniciada, não existe nenhuma funcionalidade pronta que busca usuário por algum parâmetro. Então será apenas criado a 'casca' de uma funcionalidade que busca o usuário por seu CPF.

Depois de consultado o usuário, se o retorno não estiver vazio, significa que já existe um usuário com aquele CPF, então é lançado uma exceção (`UsuarioJaRegistradoException`) informando isto ao usuário (linhas 113 à 116). Caso o retorno da consulta fosse vazio, poderia ser concluído que este é um novo usuário no sistema e prosseguiria com a validação de outra regra de negócio.

A próxima validação é se o usuário possui idade igual ou superior a 18 anos, conforme especificado (linhas 118 à 124). Caso não, será lançada a exceção `UsuarioMedorIdadeException`, do contrário, significa que todas as regras associadas ao cadastro foram atendidas, então pode ser construído o objeto usuário com os valores passados no parâmetro de mapa (*`Hashmap`*) e retorná-lo.

Pelo exemplo, nota-se que este método do domínio não possui nada além de lógicas do domínio, nenhuma questão de infraestrutura ou regras de interface, tornando-o extremamente legível e independente e facilitando sua posterior manutenção.

4.4.3.4 Repositório - Recuperando Usuário

Agora é possível implementar a funcionalidade que verifica a existência do usuário na base. Segundo o DDD, quando se quer recuperar informações se faz uso de repositórios. Primeiramente é necessário construir uma interface para este repositório de usuário. Sua codificação é da seguinte forma:

Figura 19 - Interface do Repositório Usuário

```
1  package projeto.tcc.infraestrutura.armazenamento.repositorio;
2
3  import java.util.List;
7
8  public interface RepositorioUsuario {
9
10     Usuario getUsuarioPorAggregateID(String aggregateID);
11
12     //outros métodos omitidos
```

Fonte: Elaborado pelo autor(2015)

Basicamente, um método que recebe o `aggregateID` por parâmetro e retorna o agregado (`Usuario`) correspondente. Após a definição da interface, é necessário definir sua implementação. Assim, será construída a classe `RepositorioUsuarioImpl` que implementa esta interface. Sua codificação é da seguinte forma:

Figura 20 - Recuperando `Usuario`

```
87    @Override
88    public Usuario getUsuarioPorAggregateID(String aggregateID) {
89        try {
90            List<Evento> eventos = ArmazenadorEventos.recuperaEventos(aggregateID);
91            if (!eventos.isEmpty()) {
92                return this.constroiEntidade(eventos);
93            }
94        } catch (Exception e) {
95            e.printStackTrace();
96        }
97        return null;
98    }
99
100   private Usuario constroiEntidade(List<Evento> history)  {
101       Method method;
102       RestauradorAtributosUsuario restauradorAtributosUsuario = new RestauradorAtributosUsuario();
103       try {
104           for (Evento evento : history) {
105               method = restauradorAtributosUsuario.getClass().getMethod("aplicaMudanca", evento.getClass());
106               method.invoke(restauradorAtributosUsuario, evento);
107           }
108       } catch (SecurityException e) {
109           LOGGER.log(Level.SEVERE, e.getMessage());
110       } catch (NoSuchMethodException e) {
111           LOGGER.log(Level.SEVERE, e.getMessage());
112       } catch (IllegalAccessException e) {
113           LOGGER.log(Level.SEVERE, e.getMessage());
114       } catch (IllegalArgumentException e) {
115           LOGGER.log(Level.SEVERE, e.getMessage());
116       } catch (InvocationTargetException e) {
117           LOGGER.log(Level.SEVERE, e.getMessage());
118       }
119       return restauradorAtributosUsuario.getUsuario();
120   }
```

Fonte: Elaborado pelo autor(2015)

O método `getUsuarioPorAggregateID`() invoca a classe `ArmazenadorEventos`, chamando o método `recuperaEventos`() passando como parâmetro o *`aggregateID`* (linha 90). Este método retorna uma lista de eventos (histórico) para o agregado em questão (recomendações 3.2.1 e 3.2.3). Caso a lista não esteja vazia, invocamos o método `constroiEntidade`() passando esta lista (linha 92).

No método `constroiEntidade`(), o algoritmo itera sobre a lista e para cada evento existente, é invocado, via reflexão (recomendação 3.2.2), o método `aplicaMudanca`() da classe `RestauradorAtributosUsuario` passando o evento da iteração como parâmetro (linhas 104 à 107).

Esta classe `RestauradorAtributosUsuario`, tem a única e exclusiva responsabilidade de restaurar os valores dos atributos do usuário. Para fazer isso

ela implementa N métodos chamados de “aplicaMudanca”, porém cada um deles tem como parâmetro diferentes eventos associados ao agregado. Por exemplo, no caso da criação de usuário, o método estaria da seguinte forma:

Figura 21 - Exemplo de aplicaMudança para o evento de usuarioCadastrado

```
43    public void aplicaMudanca(UsuarioCadastradoEvento usuarioCadastradoEvento){
44        try {
45            this.usuario.login = usuarioCadastradoEvento.getLogin();
46            this.usuario.senha = usuarioCadastradoEvento.getSenha();
47            this.usuario.nome = usuarioCadastradoEvento.getNome();
48            this.usuario.CPF = usuarioCadastradoEvento.getCPF();
49            this.usuario.email = usuarioCadastradoEvento.getEmail();
50            this.usuario.dataNascimento = usuarioCadastradoEvento.getDtNascimento();
51            this.usuario.sexo = usuarioCadastradoEvento.getSgSexo();
52            this.usuario.aggregateID = usuarioCadastradoEvento.getAggregateId().toString();
53            this.usuario.setPerfil(usuarioCadastradoEvento.getCdPerfil());
54        } catch (Exception e) {
55            LOGGER.log(Level.SEVERE, e.getMessage());
56        }
57    }
```

Fonte: Elaborado pelo autor(2015)

Como apresentado na Figura 21, são apenas repassados os valores presentes do evento de cadastro para os atributos de um objeto usuário (linhas 45 à 53), o qual é retornado posteriormente. Sendo assim, no fim do percorrimento da lista de eventos do agregado, este vai estar em seu último estado, ou não, caso o mesmo não existisse na base de dados. De qualquer forma esta informação foi retornada para o método do domínio para validação das informações já apresentadas.

Antes de mostrada a implementação do método recuperaEventos(), é necessário definir as responsabilidades da classe ArmazenadorEventos. Esta classe tem a responsabilidade de tratar as interações entre a aplicação e o banco de dados, do ponto de vista dos eventos. Ou seja, ele tanto persiste informações do mesmo como também as recupera (ex: pegar a última versão de uma agregado).

Voltando a implementação do método `recuperaEventos()`, sua codificação é da seguinte forma:

Figura 22 - Método `recuperaEventos()` presente na classe `ArmazenadorEventos`

```
128    public static List<Evento> recuperaEventos(String id) {
129        List<Evento> eventos = new ArrayList<>();
130        Evento evento = null;
131        ObjectInputStream objectIn = null;
132        connection = Conexao.getConectionEventSource();
133        try {
134            PreparedStatement pstmt = (PreparedStatement) connection.
135            prepareStatement("select events from eventstore where aggregate_id = ? order by version");
136
137            pstmt.setString(1, id);
138            ResultSet rs = pstmt.executeQuery();
139            while (rs.next()) {
140                byte[] buf = rs.getBytes("events");
141                if (buf != null) {
142                    objectIn = new ObjectInputStream(new ByteArrayInputStream(buf));
143                    evento = (Evento) objectIn.readObject();
144                    eventos.add(evento);
145                }
146            }
147            return eventos;
148        } catch (Exception e) {
149            e.printStackTrace();
150        }finally{
151            Conexao.fechaConexao();
152        }
153        return null;
154    }
155
```

Fonte: Elaborado pelo autor(2015)

Este método tem a responsabilidade de ir consultar na base de dados de escrita, tabela *eventstore*, os eventos associados a um agregado, ordenando-os por sua versão. O *script* para realizar essa consulta é:

`SELECT events FROM eventstore WHERE aggregate_id = ? ORDER BY version`

Após realizar a consulta e retornar algum resultado (linha 138), o algoritmo itera sobre a mesma (linha 139) e cria um objeto evento (linha 143) para cada registro dessa consulta através da deserialização dos dados presentes na coluna *events*, através do método *`readObject()`* da classe *`ObjectInputStream`* e é

adicionado estes eventos em uma lista de eventos (linha 144), a qual será retornada no fim da iteração.

4.4.3.5 Tratando a concorrência

Após serem apresentadas as implementações necessárias para o escopo do método de domínio (processamento/validações de regras de negócio), é necessário voltar ao `ProcessadorCadastrarUsuarioComando` para validar a versão do comando (linha 29), pois um outro usuário pode ter tentado acessar concorrentemente o mesmo serviço, ou seja, este pode ter gerado um comando o qual venceu a disputa para a geração do(s) evento(s) correspondente(s) . Para isto, entra em cena um novo tipo de classe, o `PosProcessadorComandos`.

Figura 23 - `PosProcessadorComandos` invocando o método `validaVersaoComando`

```
22    @Override
23    public void execute(Comando comando) throws Exception {
24        CadastrarUsuarioComando cadastrarUsuarioComando = (CadastrarUsuarioComando) comando;
25        Map<String, Object> valores = constroiMapaParametros(cadastrarUsuarioComando);
26
27        Usuario usuario = new Usuario().criarCadastro(valores);
28
29        PosProcessadorComandos.validaVersaoComando(comando);
30
31        List<Evento> eventos = new ArrayList<>();
32        EventoProcessador eventoProcessador = new EventoProcessador();
33        Long version = ControladorVersao.getProximaVersao();
34
35        UUID usuarioID = cadastrarUsuarioComando.aggregateId();
36        UUID musicaID = UUID.randomUUID();
37
38        UsuarioCadastradoEvento usuarioCadastradoEvento =
39                new UsuarioCadastradoEvento(usuarioID,version, usuario, version);
40
41        PlayListAdicionadaEvento playListAdicionadaEvento =
42                new PlayListAdicionadaEvento(usuarioID,musicaID, "Default", ControladorVersao.getProximaVersao(), version);
43
44        eventos.add(usuarioCadastradoEvento);
45        eventos.add(playListAdicionadaEvento);
46        eventoProcessador.processarEventos(eventos);
47    }
```

Fonte: Elaborado pelo autor(2015)

Esta classe tem a única e exclusiva responsabilidade de validar a versão do comando gerado. Para isso é consultada, na base de eventos, a versão do último

evento persistido para o *aggregateID* associado ao comando (linha 12 – Figura 24).

Depois de consultada, verifica-se se a versão do comando buscado na base é diferente da versão do comando que chegou (linha 13 – Figura 24). Caso sim, é lançada a exceção `VersaoComandoInvalidaException` indicando que ocorreu um problema de concorrência (linha 14 – Figura 24), ou seja, algum outro comando venceu a disputa para gerar o próximo evento para o agregado em questão. Em outras palavras, quando um comando é construído, este tem setado como atributo *version* a versão que está relacionada com o último evento para o agregado em questão, portanto, antes de um novo comando ser aceito é necessário verificar se a versão que está em seu atributo *version* continua a mesma que está no último evento da base de escrita para este mesmo agregado, assim é possível garantir que não houveram alterações neste durante o seu processamento.

Vale ressaltar que esta classe está centralizando o processamento dos comandos, do ponto de vista de concorrência, pois seu método `validaVersaoComando()` está sincronizado (*synchronized*), portanto apenas uma *thread* será atendida por vez, enfileirando os comandos que estão chegando (recomendação 3.4.1). Esta classe está implementada da seguinte forma:

Figura 24 - Implementação do método `validaVersaoComando(comando)`

```
1  package projeto.tcc.aplicacao.comandos;
2
3  import projeto.tcc.infraestrutura.armazenamento.ArmazenadorEventos;
5
6  public class PosProcessadorComandos {
7
8      public static synchronized void validaVersaoComando(Comando comando)
9              throws VersaoComandoInvalidaException {
10
11         String aggregateID = comando.aggregateId().toString();
12         Long ultimaVersaoAgregado = ArmazenadorEventos.getUltimaVersaoAgregado(aggregateID);
13         if (!comando.getVersion().equals(ultimaVersaoAgregado)){
14             throw new VersaoComandoInvalidaException();
15         }
16     }
17
18 }
```

Fonte: Elaborado pelo autor(2015)

Como apresentado anteriormente, é necessário realizar uma consulta na base de escrita para recuperar a última versão ao agregado em questão. Neste caso, é utilizado o método `getUltimaVersaoAgregado`(), passando como parâmetro o `aggregateID` (linha 12). A implementação deste método está da seguinte forma:

Figura 25 - Implementação do método `getUltimaVersaoAgregado(agregado)`

```
158    public static Long getUltimaVersaoAgregado(String aggregateID) {
159        connection = Conexao.getConectionEventSource();
160        try {
161            PreparedStatement pstmt = (PreparedStatement) connection
162                    .prepareStatement("select version from aggregates where aggregate_id = ?");
163            pstmt.setString(1, aggregateID);
164            ResultSet rs = pstmt.executeQuery();
165            if (rs.next()) {
166                return rs.getLong("version");
167            }
168        } catch (Exception e) {
169            e.printStackTrace();
170        }finally{
171            Conexao.fechaConexao();
172        }
173        return 0L;
174    }
```

Fonte: Elaborado pelo autor(2015)

Como apresentado anteriormente, no início do capítulo, além da tabela de eventos, a base de escrita possuí uma tabela de agregados (*aggregates*) a qual contém, além de outras informações, a última versão associado ao `aggregateID` em questão. Apesar disso, como ainda está sendo cadastrado um usuário no sistema, não haverá nenhum registro para este `aggregateID`, portanto foi convencionado que neste caso é atribuído zero ao seu atributo *version*, justamente pela ausência do registro. O *script* para esta consulta é o seguinte:

```
SELECT version FROM aggregates WHERE aggregate_id = ?
```

Tendo em mãos a última versão associada ao agregado e desconsiderando que outro usuário tenha gerado um comando neste momento, o valor do atributo *version* retornado da base de escrita é zero, portanto tanto a versão do comando que chegou quanto a versão que está no último evento da base associado ao agregado em questão é a mesma. Sendo assim, nenhuma exceção será lançada e o serviço continuará seu fluxo correto.

4.4.3.6 Gerando e Armazenando Eventos

Voltando ao `ProcessadorCadastrarUsuarioComando`, percebe-se que já foram validadas as regras de negócio e as questões de concorrência, portanto não existem mais validações a serem executadas. Sendo assim, o fluxo está apto para gerar os eventos associados a este comando.

Antes de serem apresentados os eventos deste comando, é necessário primeiramente definir algumas questões. Neste projeto, é adotada a seguinte terminologia para um evento: nome da ação no passado + Evento. Esta padronização é importante por causa da linguagem ubíqua, facilitando o

entendimento de todos os membros da equipe. No caso de criar um cadastro, os eventos gerados serão `UsuarioCadastradoEvento` e `PlaylistAdicionadaEvento`.

Um evento irá conter um identificador único ou UUID (*Universally unique identifier*). Mas isso não significa que cada evento possuirá um UUID próprio, esse identificador será utilizado para recuperar todos os eventos de determinado agregado.

Voltando ao exemplo, o primeiro evento é o `UsuarioCadastradoEvento`. Este evento tem como parâmetros de seu construtor um `aggregateID`, uma *version*, um objeto *usuario*, e um *groupVersion*. Para gerar a versão deste evento, e do subsequente, foi implementado um método chamado `getProximaVersao`(), na classe `ControladorVersao`, apenas para não replicar múltiplas vezes a instanciação de um `TimeStamp`. A implementação deste método é a seguinte:

Figura 26 - Implementação do método `getProximaVersao()`

```
30⊖     public static Long getProximaVersao(){
31          return new Timestamp(new Date().getTime()).getTime();
32      }
33  }
34
```

Fonte: Elaborado pelo autor(2015)

No caso do `UsuarioCadastradoEvento`, tanto o atributo *version* quanto o *groupVersion* tem o mesmo valor, pois este evento é o primeiro da lista de eventos para o comando em questão e conforme havíamos apresentado antes, o groupVersion é replicado para os eventos subsequentes para garantir a atomicidade do comando e a consistência do agregado.

Apesar de não ter sido falado anteriormente, existe outra regra associada à criação do usuário, que especifica que quando for criado um usuário no sistema,

deve ser criada também uma playlist “default” para o mesmo. Poderia ter sido encapsulada todas as informações no evento `UsuarioCadastradoEvento`, porém não seria interessante misturar conceitos/agregados, portanto é preferido deixar o mais claro possível essa separação.

Para esta arquitetura, a classe `Usuario` é o agregado raiz da classe *`Playlist`*, pois não existe uma *playlist* sem nenhum usuário. Além disso, um usuário poderá ter uma ou N *playlists*, dependendo do perfil escolhido. Portanto é adicionado mais um atributo à classe `Usuario`, uma lista de *`Playlists`*.

Voltando ao evento, como já foi apresentado que uma *`Playlist`* é agregada de um usuário, faz-se necessário adicionar em seu construtor tanto o seu `aggregateID` quanto o `aggregateID` do agregado raiz. Além disso, em seu construtor também é passado o nome da *playlist* (neste caso *“Default”*), e também sua versão, um *timestamp* mais recente que o evento anterior e, muito importante, utilizar a versão do evento anterior como valor do parâmetro *`groupVersion`* deste evento, justamente para ser mantida uma lista de eventos para este comando.

Após terem sido criados estes dois eventos, é necessário adicioná-los a uma lista de eventos e repassá-los ao processador de eventos. A classe `EventoProcessador` tem as seguintes implementações:

Figura 27 - Classe `EventoProcessador`

```
1  package projeto.tcc.dominio.eventos;
2
3  import java.util.List;
6
7  public class EventoProcessador  {
8
9
10     public void processarEvento(Evento e) throws Exception{
11         ArmazenadorEventos.salvarEvento(e);
12     }
13
14     public void processarEventos(List<Evento> evs) throws Exception{
15         ArmazenadorEventos.salvarEventos(evs);
16     }
17
18 }
```

Fonte: Elaborado pelo autor(2015)

Basicamente, é atribuída a esta classe apenas a responsabilidade de organizar as chamadas para o `ArmazenadorEventos`, sendo disponibilizado dois métodos. Um que recebe apenas um evento a ser armazenado e o outro que recebe uma lista de eventos, o qual está sendo utilizado para este exemplo. Vamos apresentar a implementação do método `salvarEventos()` do `ArmazenadorEventos`. A Figura 28 apresenta sua estrutura:

Figura 28 - Implementação do método `salvarEventos()`

```
180    public static void salvarEventos(List<Evento> evs) throws EventosNulosExceptions {
181        validaListaEventosNulasOuVazias(evs);
182
183        connection = Conexao.getConectionEventSource();
184        PreparedStatement pstmt1 = null;
185        try {
186            connection.setAutoCommit(false);
187            pstmt1 = (PreparedStatement) connection
188                    .prepareStatement("insert into eventstore(aggregate_id,events, version, groupVersion) values(?,?,?,?)",
189                            PreparedStatement.RETURN_GENERATED_KEYS);
190            processaEventos(evs, pstmt1);
191
192            connection.commit();
193            pstmt1.close();
194            publicador.publicaEventos(evs);
195            ControladorVersao.removeIDAgregadoCache(evs.get(0).getAggregateId().toString());
196        } catch (Exception e) {
197              trataExcecao();
198        } finally {
199            Conexao.fechaConexao();
200        }
201    }
202
```

Fonte: Elaborado pelo autor(2015)

Este método inicia com a validação para verificar se a lista de eventos passada não está nula ou vazia (linha 181), para evitar que sejam lançadas exceções em tempo de execução. Ou seja, o método `validaListaEventosNulasOuVazias()` apenas faz esse tratamento, por isso não se faz necessário apresentar sua codificação.

Após isso, é desabilitado o *autoCommit* (linha 186), pois é necessário comitar todos os eventos passados na lista em uma única transação, justamente por eles sempre terem que ocorrer juntos. O *script* para inserção dos eventos na tabela de eventos, baseado em sua estrutura (recomendação 3.1.1), é o seguinte:

```
INSERT INTO eventstore(agrregate_id, events, version, groupVersion)
values(?, ?, ?, ?)
```

Após criar o *statement* com o *script* de inserção (linha 187), é invocado o método `processaEventos()` (linha 190), passando como parâmetros a lista de eventos e o *statement* já configurado. A implementação deste método é a seguinte:

Figura 29 - Implementação do método `processaEventos()`

```
204  private static void processaEventos(List<Evento> evs,
205          PreparedStatement pstmt1) throws IOException, SQLException {
206      for (Evento evento : evs) {
207          if (evento.getAggregateId() != null && evento.getVersion() != null) {
208              insereValoresInsert(pstmt1, evento);
209              pstmt1.executeUpdate();
210              salvaOuAtualizaAgregado(evento.getAggregateId(), evento.getClazz(), evento.getVersion());
211
212          }
213      }
214  }
215
```

Fonte: Elaborado pelo autor(2015)

Explicando-o, para cada evento presente na lista de eventos (linha 206), é verificado se tanto o `aggregateID` quanto a versão estão presentes (linha 207), pois são informações extremamente importantes. Caso sim, é invocado o método `insereValoresInsert()` (linha 208), o qual tem a seguinte implementação:

Figura 30 - Implementação do método `insereValoresInsert()`

```
226    private static void insereValoresInsert(PreparedStatement pstmt1,
227            Evento evento) throws IOException, SQLException {
228        ByteArrayOutputStream bos = new ByteArrayOutputStream();
229        ObjectOutputStream oos = new ObjectOutputStream(bos);
230        oos.writeObject(evento);
231        oos.flush();
232        oos.close();
233        bos.close();
234        byte[] dadosEvento = bos.toByteArray();
235        pstmt1.setString(1, evento.getAggregateId().toString());
236        pstmt1.setObject(2, dadosEvento);
237        pstmt1.setLong(3, evento.getVersion());
238        pstmt1.setLong(4, evento.getGroupVersion());
239    }
240
```

Fonte: Elaborado pelo autor(2015)

Resumidamente, este método apenas seta os valores que serão colocados no *statement* de inserção. Um dos valores mais peculiares diz respeito à serialização (linha 234) dos dados presentes no evento, o qual corresponde ao parâmetro de número dois.

Voltando ao método `processaEventos()`, é executada a linha `"pstmt1.executeUpdate()"`, ou seja, o código deve rodar o *script* de *insert*, porém esta inserção não está concluída, pois foi desabilitado o *autoCommit*, portanto deve ser ordenado o momento do *commit*.

Após inserir o evento, é necessário verificar se deve ser inserido ou atualizado o agregado na tabela de agregados (linha 210), pois como havia sido apresentado antes, todo agregado no sistema deverá ter um registro nesta tabela. Neste caso, sabe-se que quando foi rodado o *script* de inserção do evento `UsuarioCadastradoEvento`, não existiria nenhum registro na tabela de agregado, pois o mesmo ainda não foi criado. O método `salvaOuAtualizaAgregado()` tem a seguinte implementação :

Figura 31 - Implementação do método `salvaOuAtualizaAgregado()`

```
78  public static void salvaOuAtualizaAgregado(UUID aggregateID, Class<?> clazz, Long version) throws SQLException {
79      boolean jaExisteAgregado = jaExisteAgregado(aggregateID.toString());
80      if (jaExisteAgregado) {
81          atualizaUltimaVersaoAgregado(aggregateID, version);
82      }else{
83          insereAgregado(aggregateID, clazz, version);
84      }
85  }
```

Fonte: Elaborado pelo autor(2015)

Como dito anteriormente, é realizado uma consulta para verificar a existência do agregado na tabela de agregados, utilizando como filtro o `aggregateID` (linha

79). Esta consulta retornará um `booleano`, dizendo se o mesmo existe ou não. Sua implementação está da seguinte forma:

Figura 32 - Implementação do método `jaExisteAgregado()`

```
101 private static boolean jaExisteAgregado(String aggregateID) throws SQLException{
102     PreparedStatement pstmt2 = null;
103         pstmt2 = (PreparedStatement) connection.prepareStatement(
104                 "SELECT 1 from aggregates WHERE  aggregate_id = ?");
105         pstmt2.setString(1, aggregateID);
106         ResultSet rs = pstmt2.executeQuery();
107         if (rs.next()) {
108             return true;
109         }
110     return false;
111 }
```

Fonte: Elaborado pelo autor(2015)

Como pode ser visto na Figura 32, o método acima apenas roda o *script*:

```
SELECT 1 FROM aggregates WHERE aggregate_id = ?
```

Caso a consulta retorne alguma linha, significa que existe algum registro para este `aggregateID`, então é retornado *true* (linha 108), do contrário, *false* (linha 110). Portanto existem dois fluxos diferentes neste método. Um deles se refere quando ainda não existe nenhum registro nesta tabela, então é invocado o método `insereAgregado()` para cadastrar um novo registro com esse `aggregateID` na base. Sua implementação é a seguinte:

Figura 33 - Implementação do método `insereAgregado()`

```
88  private static void insereAgregado(UUID aggregateID, Class<?> clazz,
89          Long version) throws SQLException {
90      PreparedStatement pstmt2 = null;
91      pstmt2 = (PreparedStatement) connection.prepareStatement(
92              "insert into aggregates(aggregate_id,type, version) values(?,?,?)",
93              PreparedStatement.RETURN_GENERATED_KEYS);
94
95      pstmt2.setString(1, aggregateID.toString());
96      pstmt2.setString(2, clazz.getName());
97      pstmt2.setLong(3, version);
98      pstmt2.executeUpdate();
99  }
```

Fonte: Elaborado pelo autor(2015)

Pela Figura 33, é possível notar que apenas está sendo criado o *statement* de inserção (utilizando como referência a estrutura da tabela de agregados - recomendação 3.1.1) passando os valores vindos por parâmetro (`aggregateID`, nome qualificado da classe e versão). Caso o retorno tivesse sido *true*, então significa que já existia algum agregado com este `aggregateID`, portanto deve-se apenas atualizar a coluna *version*, utilizando a versão do evento que acabou de ser persistido. A implementação deste método é a seguinte:

Figura 34 - Implementação do método `atualizaUltimaVersaoAgregado()`

```
114 public static void atualizaUltimaVersaoAgregado(UUID aggregateID, Long version) {
115     PreparedStatement pstmt2 = null;
116     try {
117         pstmt2 = (PreparedStatement) connection.prepareStatement(
118                 "UPDATE aggregates SET version = ? WHERE aggregate_id = ? ");
119         pstmt2.setLong(1, version);
120         pstmt2.setString(2, aggregateID.toString());
121         pstmt2.executeUpdate();
122     } catch (Exception e) {
123         e.printStackTrace();
124     }
125
126 }
```

Fonte: Elaborado pelo autor(2015)

Como já dito, apenas é atualizado a coluna *version*, portanto, este método gera o *statement* contendo um *UPDATE* para a coluna *version* usando como condição o `aggregateID`. Agora, é necessário voltar para o método `salvarEventos()`, continuando seu fluxo normal.

Figura 35 - Concluir a transação, fechar o *statement*, publicar eventos

```
180  public static void salvarEventos(List<Evento> evs) throws EventosNulosExceptions {
181      validaListaEventosNulasOuVazias(evs);
182
183      connection = Conexao.getConectionEventSource();
184      PreparedStatement pstmt1 = null;
185      try {
186          connection.setAutoCommit(false);
187          pstmt1 = (PreparedStatement) connection
188                  .prepareStatement("insert into eventstore(aggregate_id,events, version, groupVersion) values(?,?,?,?)",
189                          PreparedStatement.RETURN_GENERATED_KEYS);
190          processaEventos(evs, pstmt1);
191
192          connection.commit();
193          pstmt1.close();
194          publicador.publicaEventos(evs);
195          ControladorVersao.removeIDAgregadoCache(evs.get(0).getAggregateId().toString());
196      } catch (Exception e) {
197            trataExcecao();
198      } finally {
199          Conexao.fechaConexao();
200      }
201  }
```

Fonte: Elaborado pelo autor(2015)

Como pode ser visto nas linhas 192, 193, 194, da Figura 35, é necessário executar três tarefas importantes. A primeira delas é, de fato, concluir a transação (linha 192), fazendo para isto um *commit* de todos os *scripts* de *inserts* gerados para os eventos. Em caso de falha, é executado o método `trataExcecao()`, o qual apenas faz um *rollback* de tudo que foi inserido neste método. A segunda, em caso de sucesso, é fechar o *statement* (linha 193).

Por fim, entra em cena um novo elemento da arquitetura e que tem papel fundamental para o correto funcionamento da mesma, o `Publicador` (*Publisher*) (linha 194).

4.4.3.7 Publicando Eventos

O Publicador, como o próprio nome diz, tem a única responsabilidade de publicar os eventos para seus assinantes. Nesta aplicação, a classe `EventoPublicador` é a implementação da interface `IPublicador`, a qual tem a seguinte definição:

Figura 36 - Interface do Publicador

```
1 package projeto.tcc.infraestrutura;
2
3 public interface IPublicador<E> {
4
5     public void assina(IAssinante<E> assinante);
6
7     public void publica(E arg);
8
9 }
```

Fonte: Elaborado pelo autor(2015)

Esta interface define dois métodos que devem ser implementados pelo `EventoPublicador`, um método que registra um assinante passado por parâmetro, o qual será apresentado em parágrafos subsequentes, e um método chamado `publica()`, o qual terá um argumento a ser passado (neste caso, um evento).

Basicamente, toda vez que um evento for persistido ele é colocado numa fila de eventos para ser publicado. Utilizando da tecnologia *Future* do Java, é criado uma promessa de publicação, ou seja, assim que for possível essa classe notificará todos os assinantes responsáveis por determinado evento, fazendo isso para todos os eventos na fila de eventos a serem publicados (recomendação 3.3.2).

Após os eventos da fila serem publicados, a mesma é limpada para evitar notificações posteriores.

Voltando ao código, o método `publicaEventos()`, da classe `Publicador`, tem a seguinte implementação:

Figura 37 - Implementação do método `publicaEventos()`

```
70⊖    public void publicaEventos(List<Evento> eventos) {
71         adicionaEventos(eventos);
72         inicializaPublicacao();
73     }
```

Fonte: Elaborado pelo autor(2015)

O primeiro método (linha 71) apenas adiciona todos os eventos que vieram numa lista para uma fila de eventos (variável `filaEventos`) a serem publicados. Ou seja, o primeiro evento que chegou será o primeiro a sair. Em outras palavras, este método está implementado da seguinte forma:

Figura 38 - Implementação do método `adicionaEventos()`

```
51⊖    public void adicionaEventos(List<Evento> eventos) {
52         for (Evento evento : eventos) {
53             filaEventos.add(evento);
54         }
55     }
```

Fonte: Elaborado pelo autor(2015)

Após construir a fila de eventos a serem publicados, é iniciada a publicação dos mesmos, invocando o método `inicializaPublicacao()` (linha 72 da figura 37). Sua implementação é a seguinte:

Figura 39 - Implementação do método `inicializaPublicacao()`

```
30    public Future<String> inicializaPublicacao(){
31
32        return pool.submit(new Callable<String>() {
33            @Override
34            public String call() throws Exception {
35                publicar(filaEventos);
36                return "Processado";
37            }
38        });
39
```

Fonte: Elaborado pelo autor(2015)

Este método tem características bem interessantes e que favorecem muito a arquitetura. É neste método que é feito uso da tecnologia *Future* já mencionada anteriormente. Cria-se um pool a ser submetido, entretanto isto não é um fato e sim uma promessa, portanto quando for possível, será invocado o método `publicar()` (linha 35). Ou seja, a própria tecnologia define o momento disponível para essa ação ser executada.

Antes de ser mostrada a publicação de fato, é necessário apresentar um trecho importante do código que ainda não havia sido mostrado, mas que é necessário para a publicação dos eventos. Este trecho se refere ao construtor da classe `ArmazenadorEventos`, a qual em sua construção, além de instanciar o `EventoPublicador`, também carrega todos os assinantes do sistema, como pode ser visualizado na Figura 40:

Figura 40 - Instanciando publicador e carregando os assinantes do sistema

```
33      static {
34          publicador = new EventoPublicador();
35          carregaAssinantes();
36      }
37
38
39    private static void carregaAssinantes() {
40        publicador.assina(new MusicaAdicionadaManipulador());
41        publicador.assina(new MusicaAdicionadaFavoritoManipulador());
42        publicador.assina(new UsuarioCadastradoManipulador());
43        publicador.assina(new UsuarioEditadoManipulador());
44        publicador.assina(new MusicaAdicionadaManipulador());
45        publicador.assina(new UsuarioLogadoManipulador());
46        publicador.assina(new PlayListAdicionadaManipulador());
47    }
48
```

Fonte: Elaborado pelo autor(2015)

Porém ainda não são mostrados os detalhes dos assinantes, apenas apresentar que os mesmos já foram carregados (linhas 40 à 46), ou seja, todas as classes apresentadas acima estão esperando alguma notificação por parte do `EventoPublicador`. As características dos assinantes são apresentadas mais adiante.

Voltando a publicação, quando chegar o momento de serem publicados os eventos, o método `publicar()` é invocado, e após todos estes eventos serem publicados (linha 43), é feita uma limpeza na fila de eventos para evitar futuras inconsistências (linha 45). Abaixo a implementação do método `publicar()`.

Figura 41 - Implementação do método `publicar()`

```
40⊖    public void publicar(Queue<Evento> filaEventos){
41
42         for (Evento evento : filaEventos) {
43             publica(evento);
44         }
45         filaEventos.clear();
46
47     }
```

Fonte: Elaborado pelo autor(2015)

Pela Figura 41, é possível notar que para cada evento, na fila de eventos, é invocado o método `publica()`, passando o evento atual. Este método `publica()` é o mesmo método definido na interface `IPublicador`, o qual tem a seguinte implementação:

Figura 42 - Implementação do método `publica()`

```
61⊖    @Override
62     public void publica(Evento arg) {
63         for (IAssinante<Evento> assinante : assinantes) {
64             assinante.getPublicacao(arg);
65         }
66     }
```

Fonte: Elaborado pelo autor(2015)

Como é possível notar, o método `publica()` literalmente notificará todos os assinantes do evento atual.

4.4.3.8 Assinante de Eventos

Assinantes de eventos, ou também manipuladores de eventos, são classes responsáveis por transformar as informações presentes nos eventos (lado escrita) em informações para as *views* (lado leitura). *Views*, para esta arquitetura, segundo artigo da Microsoft (CQRS Journey, 2012) são tabelas não normalizadas adaptadas às interfaces e requisitos da aplicação, as quais ajudam a maximizar a apresentação e o desempenho da consulta.

Cada manipulador trata exclusivamente um evento, mas pode haver mais de um manipulador para o mesmo evento. Esse manipulador tem conhecimento de quais *views* o evento está relacionado.

Todos os manipuladores são assinantes do publicador, pois quando um evento é persistido, posteriormente ele deve ser publicado, então todos os assinantes são notificados, mas só os que tratam aquele evento específico prosseguirão com o processamento.

Toda classe que deseja ser um assinante deverá implementar a interface `IAssinante`, a qual tem a seguinte estrutura:

Figura 43 - Interface `IAssinante`

```
1 package projeto.tcc.infraestrutura;
2
3 public interface IAssinante<E>{
4
5     public void getPublicacao(E argumento);
6
7 }
```

Fonte: Elaborado pelo autor(2015)

Pela Figura 43, é possível perceber que esta interface define o método `getPublicacao()`, o qual espera um argumento, que nesta aplicação é um evento a ser publicado.

Como foram gerados dois eventos, o `UsuarioCadastradoEvento` e `PlaylistAdicionadaEvento`, é possível presumir que existem dois assinantes envolvidos neste fluxo, um para cada evento.

Antes de ser dada continuidade à apresentação dos manipuladores, é necessário definir duas *views* que são necessárias nesta aplicação. Uma delas que contém os dados do usuário e a outra que contém as *playlists*. A primeira delas tem o nome de "`dadosusuario`" e seu script de criação é o seguinte:

```
CREATE TABLE `dadosusuario` (
 `id` int(11) NOT NULL AUTO_INCREMENT,
 `login` varchar(20) NOT NULL,
 `senha` varchar(100) NOT NULL,
 `nome` varchar(50) NOT NULL,
 `CPF` varchar(11) NOT NULL,
 `email` varchar(40) NOT NULL,
 `dataNascimento` date DEFAULT NULL,
 `sexo` varchar(1) DEFAULT NULL,
 `aggregateID` varchar(40) DEFAULT NULL,
 `dataUltimoLogin` date DEFAULT NULL,
 `cdPerfil` int(11) DEFAULT NULL,
 PRIMARY KEY (`id`)
    ) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
```

Como é possível ver pelo script de criação, foram colocadas todas as informações solicitadas no cadastro, além de uma coluna (`dataUltimoLogin`) que é utilizada para outra funcionalidade (*login* no sistema).

A segunda *view* é a relacionada às *playlists* do usuário (`playlistsusuario`) e seu script de criação tem a seguinte estrutura:

```
CREATE TABLE `playlistsusuario` (
 `id` int(11) NOT NULL AUTO_INCREMENT,
 `aggregateIDUsuario` varchar(45) NOT NULL,
 `aggregateIDPlayList` varchar(45) NOT NULL,
 `nome` varchar(45) NOT NULL,
 PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
```

Pelo script de criação apresentado, é possível notar além de um `id` sequencial, a presença do `aggregateID` do agregado raiz (neste caso `Usuario`), um `aggregateID` da própria *`playlist`* e um nome para a *`playlist`*.

Agora que estão definidas as duas *views* necessárias à funcionalidade, é necessário voltar aos manipuladores, os quais irão popular essas duas tabelas.

O primeiro deles vai tratar o evento de `UsuarioCadastradoEvento` e é nomeado de `UsuarioCadastradoManipulador` e implementa a interface `IAssinante`, ou seja, terá que codificar o método `getPublicacao()`. Para este evento, a codificação do método `getPublicacao()` está da seguinte forma

Figura 44 - Implementação do método `getPublicacao()`

```
44⊖     @Override
45      public void getPublicacao(Evento arg) {
46          if (arg instanceof UsuarioCadastradoEvento) {
47              insereViews((UsuarioCadastradoEvento) arg);
48          }
49      }
50
51  }
```

Fonte: Elaborado pelo autor(2015)

Como pode ser visto pela Figura 44, nota-se que o método está recebendo um argumento do tipo Evento e como todos os eventos da aplicação implementam

esta interface, é possível fazer o *cast* para um evento específico, graças ao polimorfismo presente na linguagem. Por isso, antes de serem inseridas de fato as informações nas *views*, é preciso verificar se o evento que veio por parâmetro é do tipo `UsuarioCadastradoEvento` (linha 46), pois só este tipo é tratado por esta classe. Esta validação é necessária, pois quando um evento é publicado todos os assinantes do sistema recebem essa notificação, mas só os que têm responsabilidade sobre o mesmo continuarão o processamento.

Após ser verificado que o evento que veio por parâmetro é do tipo esperado, é invocado o método `insereViews()` passando este evento (já convertido) (linha 47).

O método `insereViews()` apenas pega as informações presentes no objeto evento e coloca nas *views* que necessitam desta informação, que nesta aplicação corresponde a tabela “dadosusuario”. A implementação deste método é a seguinte:

Figura 45 - Implementação do método `insereViews()`

```
17  private void insereViews(UsuarioCadastradoEvento usuarioCadastradoEvento) {
18      Connection conexao = Conexao.getConectionReader();
19      try {
20          PreparedStatement pstmt1 = null;
21          pstmt1 = (PreparedStatement) conexao.prepareStatement("insert into " +
22                  "dadosusuario(aggregateID,login, senha, nome, CPF, email, dataNascimento, sexo, cdperfil) " +
23                  "values(?,?,?,?,?,?,?,?,?)",
24                  PreparedStatement.RETURN_GENERATED_KEYS);
25          pstmt1.setString(1, usuarioCadastradoEvento.getAggregateId().toString());
26          pstmt1.setString(2, usuarioCadastradoEvento.getLogin());
27          pstmt1.setString(3, usuarioCadastradoEvento.getSenha());
28          pstmt1.setString(4, usuarioCadastradoEvento.getNome());
29          pstmt1.setString(5, usuarioCadastradoEvento.getCPF());
30          pstmt1.setString(6, usuarioCadastradoEvento.getEmail());
31          pstmt1.setDate(7, new Date(usuarioCadastradoEvento.getDtNascimento().getTime()));
32          pstmt1.setString(8, usuarioCadastradoEvento.getSgSexo());
33          pstmt1.setInt(9, usuarioCadastradoEvento.getCdPerfil());
34          pstmt1.executeUpdate();
35          pstmt1.close();
36      } catch (Exception e) {
37          e.printStackTrace();
38          return;
39      }finally{
40          Conexao.fechaConexao();
41      }
42  }
```

Fonte: Elaborado pelo autor(2015)

O método apresentado na Figura 45 apenas executa um *script* de inserção das informações pertinentes ao cadastro de um usuário no sistema (nome, idade, sexo, perfil, etc). Assim, quando se faz necessário realizar certas consultas é possível ir buscá-las diretamente nessa *view*, sem necessidade de *joins* ou reconstrução de eventos.

O próximo manipulador envolvido neste fluxo é o `PlaylistAdicionadaManipulador`. Esta classe tem a responsabilidade de tratar o evento `PlaylistAdicionadaEvento`, como é possível ver em sua implementação do `getPublicacao()` :

Figura 46 - Implementação do método `getPublicacao()`

```
39@Override
40 public void getPublicacao(Evento arg) {
41     if (arg instanceof PlayListAdicionadaEvento) {
42         insereViews((PlayListAdicionadaEvento) arg);
43     }
44
45 }
```

Fonte: Elaborado pelo autor(2015)

Neste caso, apenas a "*view*" "`playlistsusuario`" é modificada pela geração desse evento. É gerado um *script* de inserção, persistindo as informações `aggregateIDUsuario`, `aggregateIDPlaylist`, nome e gerando uma chave sequencial pelo banco, como pode ser confirmado pela imagem abaixo sobre a implementação do método `insereViews()` :

Figura 47 - Inserindo valores na *view* `playlistusuario`

```
18⊖private void insereViews(PlayListAdicionadaEvento playListAdicionadaEvento) {
19      Connection conexao = Conexao.getConectionReader();
20      try {
21          PreparedStatement pstmt1 = null;
22          pstmt1 = (PreparedStatement) conexao.prepareStatement("insert into " +
23                  "playlistsusuario(aggregateIDUsuario,aggregateIDPlayList,nome) " +
24                  "values(?,?,?)",
25                  PreparedStatement.RETURN_GENERATED_KEYS);
26          pstmt1.setString(1, playListAdicionadaEvento.getAggregateId().toString());
27          pstmt1.setString(2, playListAdicionadaEvento.getPlayListId().toString());
28          pstmt1.setString(3, playListAdicionadaEvento.getNomePlayList());
29          pstmt1.executeUpdate();
30          pstmt1.close();
31      } catch (Exception e) {
32          e.printStackTrace();
33          return;
34      }finally{
35          Conexao.fechaConexao();
36      }
37  }
```

Fonte: Elaborado pelo autor(2015)

4.4.3.9 Finalizando a Escrita

Após terem sido publicados os eventos na base de leitura, pode-se dizer que está finalizado o fluxo completo de um exemplo de escrita de eventos. Percebe-se que é um fluxo bem extenso de operações e diferentes classes/camadas envolvidas, tendo um esforço um pouco maior para uma correta estruturação da arquitetura.

Por outro lado, é nítida a separação de responsabilidades entre as diferentes classes envolvidas, portanto é possível ter um código mais legível e uma maior segurança nas modificações, pois é minimizado o impacto entre as camadas.

4.4.4 Lado leitura

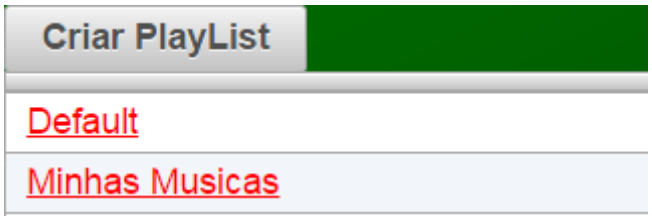
Esta parte do sistema é responsável apenas por recuperar informações e apresentá-las ao usuário, atendendo aos requisitos de consultas executados. Outra consideração importante é que, no lado leitura, nenhum método deve alterar o estado dos objetos.

Nesta seção é apresentada a implementação de um exemplo de funcionalidade relacionado à consulta de informações em nosso sistema. Mais exemplos podem ser visualizados diretamente no código fonte desta aplicação.

4.4.4.1 Listagem de *playlists* e músicas

A funcionalidade escolhida para ser apresentada é a listagem das *playlists* e suas músicas correspondentes. Na imagem a seguir é mostrada a listagem das *playlists* do usuário logado. No caso, este possuía apenas duas *playlists*, a que vem junto na criação de um usuário, a *playlist* '*Default'*, e uma *playlist* criada por ele, Minhas Musicas.

Figura 48 - Listagem de *playlists*



Fonte: Elaborado pelo autor(2015)

Na Figura 49 a seguir é apresentado o método invocado da interface gráfica para a listagem das *playlists* do usuário. Isto é feito através de um `ManagedBean`

do framework JSF chamado `OuvirMusicasBean`, mas poderia ter sido utilizado qualquer outra tecnologia.

Figura 49 - Implementação do método `listarMinhasPlayList()`

```
91  public String listarMinhasPlayList() {
92      minhasPlayLists = servicoPlayListLeitura.buscarPlayLists(aggregateIDObject.toString());
```

Fonte: Elaborado pelo autor(2015)

Para realizar a listagem das *playlists* é necessário capturar o agregado do usuário logado. É através dele que é realizado a busca das suas *playlists*. Este agregado está gravado na sessão no momento que o usuário faz login, no caso do método acima, é a variável chamada `aggregateIDObject`. Esta variável é passada como parâmetro para a camada de serviço de leitura representada como `servicoPlayListLeitura` (linha 92). A classe `ServicoPlayListLeitura` representa todas as consultas que são relacionadas às *playlists*, diferentemente do serviço de escrita para *playlist,* apresentado na seção 4.4.3.

O serviço de leitura se comunica com o repositório do usuário. O repositório é responsável por capturar informações referentes aos usuários do sistema. Ele se comunica com a base de dados responsável apenas pela leitura (recomendação 3.2.1), através da *view* `playlistsusuario`, passando como parâmetro o agregado do usuário logado. O repositório traz todas as informações necessárias para a listagem das *playlists*. No caso deste exemplo, as informações capturadas são apenas o agregado da *playlist* e o seu nome, refletidas através da seguinte consulta:

```
SELECT AGGREGATEIDPLAYLIST, NOME FROM BASELEITURA.PLAYLISTSUSUARIO WHERE AGGREATEIDUSUARIO = ?
```

As informações capturadas refletem o nome da *playlist* que é mostrada na tela ao usuário, e o agregado como informação necessária para a listagem das músicas associadas à *playlist* selecionada. Consulta esta que é executada no método getListaPlayList() na Figura 50 a seguir :

Figura 50 - Buscando nome e agregado na *view* playlistsusuario.

```
173@Override
174 public List<PlayList> getListaPlayList(String aggregateID) {
175     List<PlayList>playlists = null;
176     PlayList playlist = null;
177     try {
178         playlists = new ArrayList<PlayList>();
179         PreparedStatement pstmt =  (PreparedStatement)Conexao.getConectionReader().
180                 prepareStatement("SELECT aggregateIDPlayList, nome from baseleitura.playlistsusuario where aggregateIDUsuario = ?");
181         pstmt.setString(1, aggregateID);
182         ResultSet rs = pstmt.executeQuery();
183         while (rs.next()) {
184             playlist = new PlayList();
185             playlist.setAggregateID(rs.getString("aggregateIDPlayList"));
186             playlist.setNome(rs.getString("nome"));
187             playlists.add(playlist);
188         }
189     } catch (SQLException e) {
190         e.printStackTrace();
191     }
192 
193     return playlists;
194 }
```

Fonte: Elaborado pelo autor(2015)

A *view* `playlistusuario` possui a estrutura da Figura 51. Neste caso, é selecionado apenas as colunas `aggregateIDPlaylist` e nome, utilizando para isto a coluna `aggregateIDUsuario` como filtro. A coluna *id* é gerada automaticamente.

Figura 51 - *View* `playlistusuario`

	id	aggregateIDUsuario	aggregateIDPlayList	nome
▶	19	af75f7e8-65e5-401d-b1f3-86c3d1990031	01085289-bc71-411f-a2cc-2b9b4cc1c03b	Default
	20	d5db2626-58e1-4eaa-b609-745df6b2f6e7	a89eda86-430b-441a-b2af-91154a07904d	Default
	21	7e633b64-f5b3-4d58-b12e-58297a609b8d	39c9bdc9-79b2-4351-91e1-3d4dbb1ba1ee	Default
	22	7e633b64-f5b3-4d58-b12e-58297a609b8d	473761fb-55bf-4984-9d23-bc309123b479	Minhas Musicas
	23	7e633b64-f5b3-4d58-b12e-58297a609b8d	b2d16f73-9e05-4590-9631-22eb4fed8e49	aif
	24	7e633b64-f5b3-4d58-b12e-58297a609b8d	e96b87bc-9fa0-4277-8871-bf6e5fac4497	nao entendi
	25	96b870f2-abfa-4f73-b58a-9ba757fdfc88	8a95ab2c-2a47-4013-a7ba-7bee5a4f148b	Default
	26	96b870f2-abfa-4f73-b58a-9ba757fdfc88	fb0c1c9d-3103-4bec-a73e-82676efc1cd5	teste
	27	96b870f2-abfa-4f73-b58a-9ba757fdfc88	fa816e33-b9f8-442a-93f6-719c87a422b3	dsf dsf sd fds fd
	28	e0771cf4-a93e-4925-af5b-24b5ab95509a	049be332-f7da-409d-925a-288fbce84b22	Default
	29	11ab56cf-301f-4f30-8bfd-edd6dddab931	9a095d64-cf3d-43fb-b850-781bec863236	Default
	30	e5d7098d-f6ac-4546-8a65-80b43ce78381	a1a93756-b9d7-4b34-be62-5f40511ac152	Default
	31	408e239b-ee00-40da-a5b0-2fce1cc8b4b0	83bae9be-60b7-4c49-8a24-b4c54f44d9ae	Default
	32	408e239b-ee00-40da-a5b0-2fce1cc8b4b0	1f66f124-d522-4ee6-9ec5-119eb34ec1b6	a
	33	408e239b-ee00-40da-a5b0-2fce1cc8b4b0	65aa057e-631f-45b1-b915-be82652d69ed	f
	34	e5d7098d-f6ac-4546-8a65-80b43ce78381	624b7a7a-979f-4cfb-bb36-1911a733ca33	a
	35	c4404ec7-00ec-4b56-85d8-df05d0309691	84836eb1-48a1-4b03-96c4-97563c714b47	Default

Fonte: Elaborado pelo autor(2015)

Cada *playlist* possui um agregado próprio. Quando o usuário seleciona a *playlist Default*, é feita uma listagem de todas as músicas pertencentes ao seu agregado. No caso da *playlist Default*, havia duas músicas, como pode ser visto na Figura 52 abaixo:

Figura 52 - Listagem de músicas

Fonte: Elaborado pelo autor(2015)

A captura do agregado correspondente à *playlist* é feita via interface. Ao clicar em seu nome é invocado o método para listagem de músicas, mostrado pela Figura 53:

Figura 53 - Implementação do método `listarMinhasMusicasPlayList()`

```
83    public void listarMinhasMusicasPlayList(String aggregateID) {
84        minhasMusicas = servicoMusicaLeitura.listarMinhasMusicas(aggregateID);
85    }
```

Fonte: Elaborado pelo autor(2015)

O parâmetro `aggregateID` é capturado no momento em que o usuário clica na *playlist* desejada. Este é então passado como parâmetro para o serviço de leitura correspondente a músicas (linha 84), representado pela classe `ServicoMusicaLeitura`, a qual é responsável apenas por consultas referente a músicas.

Como no exemplo anterior, o serviço de leitura se comunica com o repositório. Este repositório contém informações relacionadas apenas a músicas. Para este caso, há o interesse em listar as músicas baseado no agregado da *playlist* selecionada. Uma vez escolhida, resta apenas fazer uma busca na view `musicasusuario` utilizando o agregado obtido como filtro.

A busca é realizada através de uma simples consulta: "`SELECT NOME FROM BASELEITURA.MUSICASUSUARIO WHERE AGGREGATEID = ?`". É capturado apenas os nomes das músicas para retornar na tela como lista. Esta busca pode ser visualizada na Figura 54:

Figura 54 - Método no repositório buscando musicas na *view* `musicasusuario`

```
71⊖public Set<Musica> recuperarMinhasMusicas(String aggregateID) {
72      Set<Musica> minhasMusicas = null;
73      try {
74
75          PreparedStatement pstmt = (PreparedStatement) Conexao
76                  .getConectionReader()
77                  .prepareStatement(
78                          "SELECT * from baseleitura.musicasusuario where aggregateId = ?");
79          pstmt.setString(1, aggregateID);
80          ResultSet rs = pstmt.executeQuery();
81          if(rs.isBeforeFirst())
82              minhasMusicas = new HashSet<Musica>();
83          while (rs.next()) {
84              Musica musica = new Musica();
85              musica.setNome(rs.getString("nome"));
86              minhasMusicas.add(musica);
87          }
88      } catch (SQLException e) {
89          e.printStackTrace();
90      }
91
92      return minhasMusicas;
93  }
```

Fonte: Elaborado pelo autor(2015)

Nesta *view*, é selecionado a coluna nome e usado como filtro o `aggregateId` da *playlist*. A coluna *id* é gerada automaticamente. Para a criação desta *view* é necessário executar o seguinte script:

```
CREATE TABLE `musicasusuario` (
 `id` int(11) NOT NULL AUTO_INCREMENT,
 `nome` varchar(45) NOT NULL,
 `duracao` varchar(45) DEFAULT NULL,
 `aggregateId` varchar(45) NOT NULL,
 PRIMARY KEY (`id`)
     ) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8;
```

4.4.4.2 Finalizando a leitura

Normalmente cada *view* representa informações associadas a uma tela ou parte dela. Então, no caso da listagem de músicas e das *playlists*, são buscadas todas as informações pertinentes em sua listagem através da(s) *view(s)* correspondente(s). Ao contrário de uma busca por eventos para a reconstituição de um objeto, aqui é feito um consulta bem simples capturando diretamente os dados necessários para o preenchimento da tela, dados estes que estão desnormalizados e que não necessitam de *joins*. É possível concluir que o fluxo de leitura é um fluxo bem mais simples do que o de escrita, em que ele tira proveito de *selects* diretos através das *views*, a fim de conseguir um tempo de resposta mínimo para sua consulta.

5. Testes Realizados com JMeter

5.1 JMeter

JMeter é um software de código aberto, que tem com o objetivo realizar testes de comportamento funcional e medição de performance. Esta ferramenta pode ser usada para simular testes de carga nos serviços de rede: um servidor, grupo de servidores, rede ou objeto para testar a sua resistência ou para analisar o seu desempenho geral sobre diferentes tipos de carga. (APACHE, 2015)

Como o número de usuários com acesso a web tem aumentado muito nos últimos anos, a preocupação com a performance que uma aplicação terá com o acesso múltiplo e simultâneo de vários usuários é importante. O JMeter tem como objetivo possuir cenários de testes o mais próximo possível da realidade que simulem o uso do sistema testado para garantir que a aplicação e todo o ambiente

de produção tenham condições de suportar um grande volume de acesso, sem riscos de sua aplicação ficar fora do ar .

5.2 Características

Segundo o site do JMeter, algumas de suas características são:

- Consegue realizar testes de carga e performance com diferentes tipos de protocolo ou servidor: web(http,https), soap/rest, ftp, database via jdbc , ldap, mom via jms, mail(smtp, pop3, imap), mongodb(nosql), shell scripts, tcp;
- Desenvolvida utilizando 100% Java;
- É multithread;
- Possui uma GUI que ajuda a construção e depuração mais rápida de planos de teste;
- Permite o armazenamento e análise ou repetição do resultado dos testes realizados;
- Tem um núcleo extensível, possuindo a funcionalidade de utilizar plugins.

5.3 Análise realizada e resultados obtidos

5.3.1 Objetivo do teste

Para os testes realizados, o objetivo é acompanhar o desempenho geral da aplicação no momento da reconstrução de um objeto através de eventos. No eixo y é o tempo em milissegundos de resposta do servidor, no eixo x são os horários durante o teste.

Os passos da utilização do sistema são gravados pela ferramenta até o momento do *login*, a reconstrução do objeto por eventos será feita no momento que o usuário realiza seu *login*. São colocados os parâmetros necessários para que o JMeter simule a navegação destes passos e possa realizar as medições. O objetivo é acompanhar o tempo de resposta à medida que o número de eventos aumenta. Isso sempre sobre um estresse de 100 usuários acessando simultaneamente a aplicação, característica que o JMeter consegue simular. Vale ressaltar que as requisições feitas por esses usuários foram executadas utilizando a mesma conta, o mesmo *login*. Portanto os testes realizados se iniciam com dois eventos apenas, o evento do cadastro do usuário, e da criação de sua *playlist*. Os demais eventos gerados são feitos pela realização do *login*. A cada teste, é monitorado a quantidade de eventos no momento.

O ambiente para a execução dos testes foi realizado em um servidor web Tomcat 7.0, num computador com as seguintes configurações: processador i5-2500 CPU 3.3GHz, 16 GB de memória ram, sistema operacional Windows 64 bits.

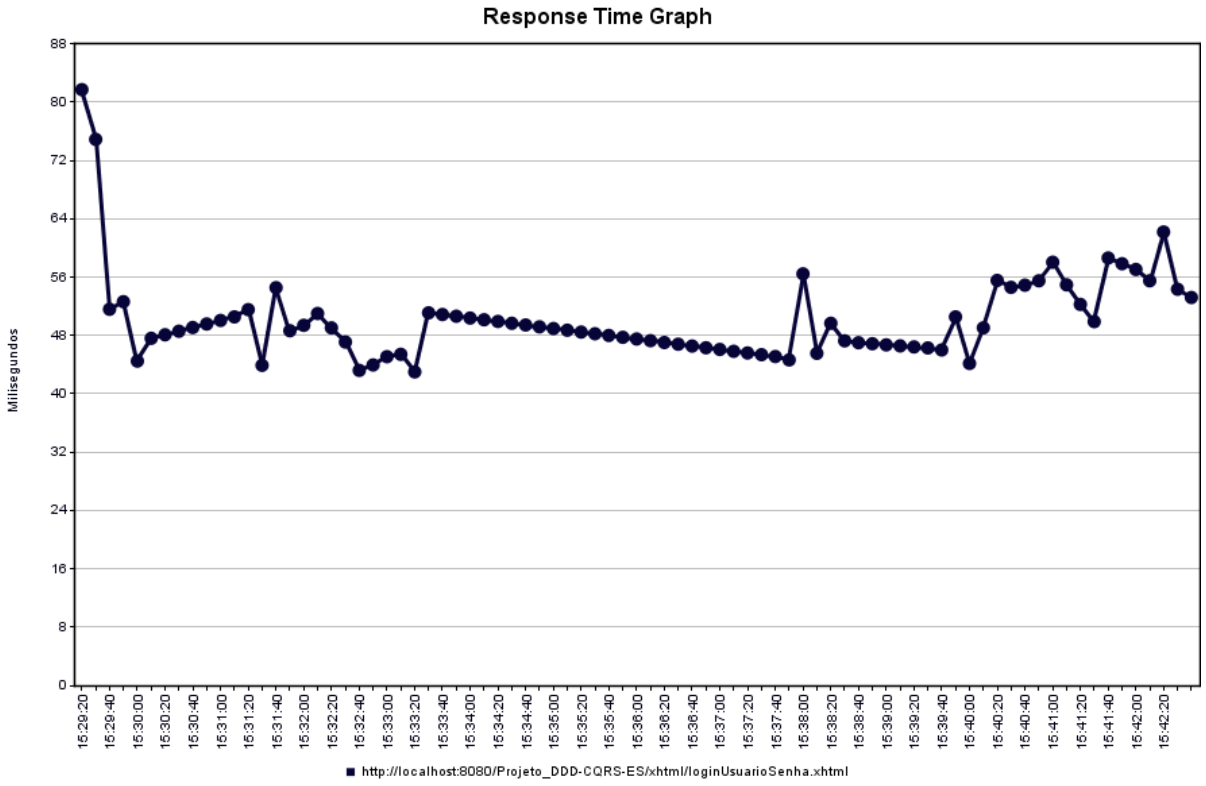
5.3.2 Resultados obtidos

Os testes iniciaram com apenas dois eventos. Estes eventos são gerados no momento da criação do usuário. No gráfico da Figura 55 abaixo é possível perceber como o tempo de resposta do servidor da aplicação se comporta à medida que o número de eventos aumenta.

Por exemplo, com dois eventos, a aplicação demorou 80 milissegundos, que é bem o começo do gráfico. Prosseguindo, é possível verificar que com o decorrer dos testes e com o número de *logins* realizados, o número de eventos aumenta, a

aplicação se mantém em uma média de 40 e 60 milissegundos, até o final do gráfico já com 450 eventos gerados.

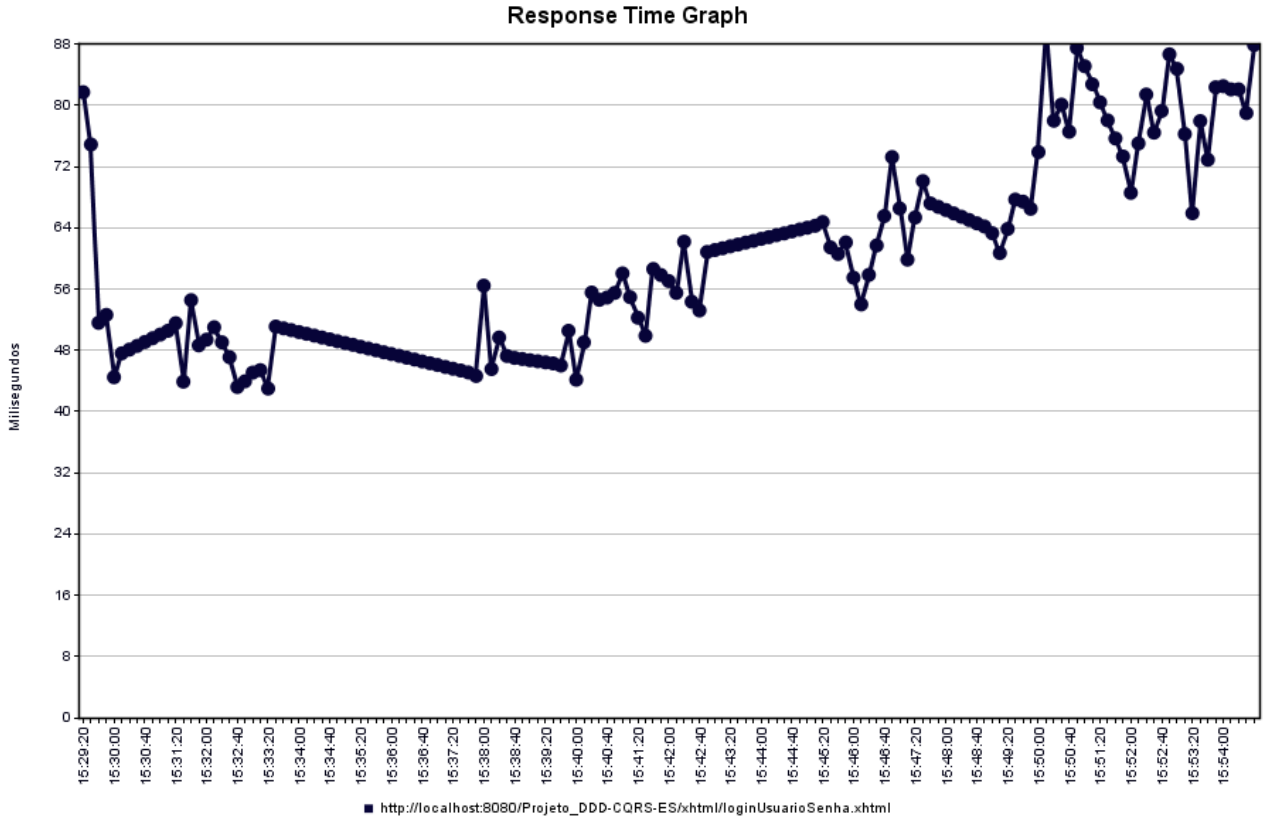
Figura 55 - 2 a 450 eventos



Fonte: Elaborado pelo autor(2015)

Continuou-se a produzir eventos a partir do momento do horário 15:38, o qual é o tempo final do gráfico anterior. Como pode ser visto, a aplicação mantém a média de milissegundos até o momento 15:46. Após isso, a aplicação começa a levar mais tempo para lidar com eventos, no final deste gráfico, com 800 eventos, a aplicação está levando 88 milissegundos como tempo de resposta para reconstituir um objeto.

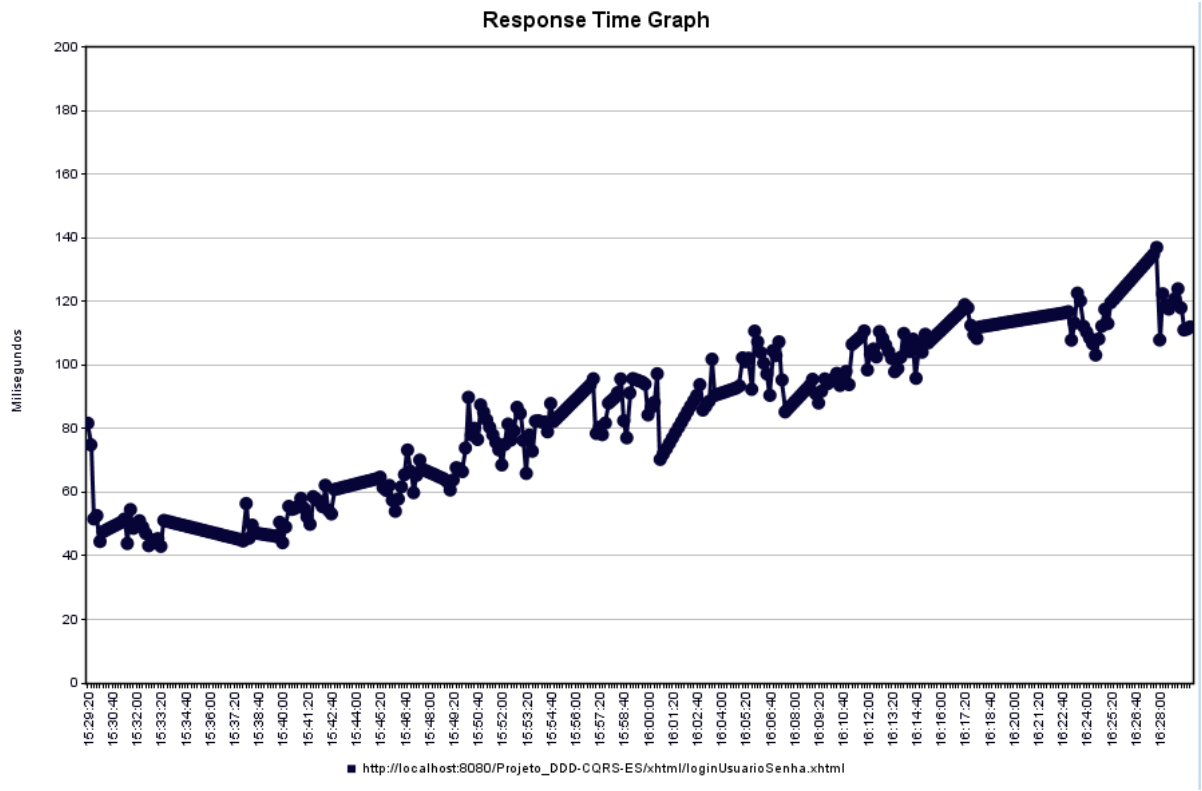
Figura 56 - 2 a 800 eventos



Fonte: Elaborado pelo autor(2015)

Agora, dobra-se o número de eventos, fazendo a análise com 1600 eventos. Pode ser visto pelo gráfico da Figura 57 que vai aumentando o tempo de resposta aos poucos, ficando com o tempo de resposta no final do gráfico entre 100 e 120 milissegundos.

Figura 57 - 2 a 1600 eventos



Fonte: Elaborado pelo autor(2015)

Embora com dois eventos o tempo de resposta tenha sido de 80 milissegundos, isso foi uma exceção, logo se estabilizou entre 40 e 60 milissegundos até perto de 800 eventos onde o tempo de resposta alcançou 88 milissegundos, dobrando o valor de eventos, o tempo de resposta foi aproximadamente 100 milissegundos.

Com as informações obtidas é possível perceber uma degradação no tempo de resposta, mas isso só acontece quando se passa a trabalhar com milhares de eventos na reconstrução. Devido às limitações de infraestrutura do ambiente de teste, não é possível aplicar estes testes com uma quantidade muito maior de eventos, o que aconteceria num ambiente mais próximo do real. De qualquer forma, caso a degradação fosse muito grande na reconstrução de milhares de eventos,

talvez fosse uma opção utilizar a heurística do Snapshot para minimizar o tempo de resposta.

6. Conclusão e trabalhos futuros

6.1 Considerações finais

Uma arquitetura integrando as três tecnologias/padrões pode ser vista como uma ótima alternativa quando está se lidando com aplicações que necessitam ter um ótimo desempenho na recuperação das informações por parte dos usuários e também quando está se tratando com domínios complexos.

É possível perceber que a união destas tecnologias pode fornecer uma arquitetura mais organizada e independente. Tem-se independência entre as camadas, mantendo questões de interface separadas de regras de negócio assim como lógicas de infraestrutura. Além disso, existe uma larga independência entre os "lados" da aplicação. O lado de leitura tem suas regras e implementações específicas diferentemente do lado escrita, podendo assim garantir maior segurança aos desenvolvedores, implementar otimizações e também, num caso real, poderia ser dividida a implementação de cada lado para equipes e, até quem sabe, empresas diferentes, na forma de componentes, por exemplo.

Acreditamos que a curva de aprendizado para toda tecnologia/padrão apresentando neste artigo não difere muito do tempo necessário para aprender a arquitetura "clássica", pois na verdade é apenas uma abordagem diferente para problemas tão conhecidos. Vale ressaltar também que existiu um esforço maior para o funcionamento desta arquitetura haja vista que se optou por não usar nenhum tipo de componente/framework relacionado com a mesma, como por exemplo, o AXON.

De nossa parte, acreditamos no potencial da arquitetura proposta, pois a mesma pode e poderá oferecer ótimas alternativas para diversos problemas conhecidos, que tornam o desenvolvimento de aplicações mais complexo e custoso

por serem necessárias diversas soluções de contorno, mas que não vão à raiz do problema.

6.2 Trabalhos futuros

Existem alguns pontos que necessitam ser mais trabalhados e explorados. Um deles é a dificuldade em achar informações claras e completas sobre como tratar certas questões. Existem muitas informações espalhadas, com poucos artigos disponíveis e alguns exemplos em blogs.

Um tema específico que se teve bastante dificuldade e que não foi encontrado muito material de apoio foi em relação a uma interface orientada a tarefas. Este tipo de característica, num primeiro momento, não parece ser tão relevante, entretanto ela é o primeiro passo na identificação e criação dos comandos por parte do usuário. Com uma boa interface, fica mais fácil elaborar e implementar os comandos. Sendo assim, acreditamos que este tema poderia ser melhor trabalhado e estudado em trabalhos futuros.

Outro trabalho futuro está relacionado a realização de um comparativo mais aprofundado sobre a utilização de bases relacionais vs arquivos no armazenamento de eventos, uma vez que as conclusões feitas aqui sobre esse tema específico não tiveram o suporte de métricas.

7. Referências Bibliográficas

1.BETTS, D;DOMINGUES, J;MELNIK, G; et al."CQRS Journey". Disponível em: http://msdn.microsoft.com/en-us/library/jj554200.aspx

.Acesso em: 15 mai. 2015

2. FERREIRA, C, M, B. "Padrão CQRS para Sistemas Distribuídos de Larga Escala". Porto, 2012. Disponível em:

http://recipp.ipp.pt/bitstream/10400.22/2690/1/DM_CarlosFerreira_2012_MEI.pdf

Acesso em: 15 mai. 2015

3. "Domain-Driven Design". [S. l.] : [s.n.]. Disponível em:

http://www.cqrs.nu/Faq/domain-driven-design

Acesso em: 15 mai. 2015

4. "Event sourcing". [S. l.] : [s.n.]. Disponível em:

http://www.cqrs.nu/Faq/event-sourcing

Acesso em: 15 mai. 2015

5. " Command/Query Responsibility Segregation". [S. l.] : [s.n.]. Disponível em:

http://www.cqrs.nu/Faq/command-query-responsibility-segregation

Acesso em: 15 mai. 2015

6. EVANS, E. "Domain-Driven Design". 2.ed. [S. l.] : Prentice Hall, 2003, 498p. Disponível em: http://www-public.int-evry.fr/~gibson/Teaching/CSC7322/ReadingMaterial/Evans03.pdf

Acesso em: 20 mai. 2015

7. YOUNG, G. "CQRS, Task Based UIs, Event Sourcing agh!" Disponível em : http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh.Acesso em: 20 mai. 2015

8.DAHAN,U.”Clarified CQRS”Disponível em: http://www.udidahan.com/wp-content/uploads/Clarified_CQRS.pdf.
Acesso em: 25 mai. 2015

9.BUILDING an Event Storage. Disponível em: https://cqrs.wordpress.com/documents/building-event-storage/.
Acesso em: 10 jul. 2015

10. GILBERT, S; LYNCH,N. “Perspectives on the CAP Theorem “. Disponível em: http://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf
Acesso em: 12 jul. 2015.

11. FOWLER,M; “Eventing Sourcing”. Disponível em: http://martinfowler.com/eaaDev/EventSourcing.html.
Acesso em: 10 ago. 2015

12. APACHE JMeter. Disponível em: http://jmeter.apache.org/.
Acesso em: 15 ago. 2015