SOCIEDADE EDUCACIONAL DE SANTA CATARINA – SOCIESC
CENTRO UNIVERSITÁRIO SOCIESC – UNISOCIESC
CAMPUS MARQUÊS DE OLINDA

DIEGO NUNES DA SILVEIRA

UTILIZAÇÃO DE CQRS E EVENT SOURCING NO DESENVOLVIMENTO DE MICROSSERVIÇOS

Joinville
2021

DIEGO NUNES DA SILVEIRA

UTILIZAÇÃO DE CQRS E EVENT SOURCING NO DESENVOLVIMENTO DE MICROSSERVIÇOS

Trabalho de Conclusão de Curso apresentado ao curso de Sistemas de Informação do Centro Universitário Sociesc, como requisito parcial para a obtenção do Título de Bacharel em Sistemas de Informação.

Orientador: Prof. Me. Adiel Seffrin Sarates Junior

Joinville

2021

DIEGO NUNES DA SILVEIRA

UTILIZAÇÃO DE CQRS E EVENT SOURCING NO DESENVOLVIMENTO DE MICROSSERVIÇOS

Trabalho de Conclusão de Curso apresentado ao curso de Sistemas de Informação do Centro Universitário Sociesc, como requisito parcial para a obtenção do Título de Bacharel em Sistemas de Informação.

Joinville, 17 de junho de 2021.

BANCA EXAMINADORA

Prof. Me. Adiel Seffrin Sarates Junior

Prof. Me. Claudinei Dias

Prof. Me. Maicol Peterson Gandolphi de Almeida

RESUMO

Apresentam-se neste trabalho temas como desenvolvimento de software, arquitetura de software, padrões arquiteturais utilizados em microsserviços, *Cloud Computing*, banco de dados e por fim, dois *patterns* que se propõem a trazer alguns benefícios ao desenvolvimento de microsserviços, são eles o CQRS e o *Event Sourcing*. Também foi apresentado um estudo de caso envolvendo uma aplicação desenvolvida utilizando os dois *patterns*. São apresentadas vantagens e desvantagens na utilização e alguns pontos como escalabilidade, desempenho, segurança e complexidade de código são discutidos. Conclui-se então, que apesar de inúmeras vantagens, também há o aumento de complexidade no código e por isso, deve-se entender a necessidade da aplicação e comparar prós e contras antes de aplicar tais abordagens.

Palavras chave: CQRS. *Event Sourcing*. Arquitetura de software. Padrões arquiteturais. Microsserviços.

ABSTRACT

This work presents themes such as software development, software architecture, architectural patterns used in microservices, Cloud Computing, database and finally, two patterns that propose to bring some benefits to the development of microservices, they are the CQRS and Event Sourcing. A case study involving an application developed using both patterns was also presented. Usage advantages and disadvantages are presented and some points such as scalability, performance, security and code complexity are discussed. It is concluded then, that despite numerous advantages, there is also an increase in complexity in the code and therefore, the need for application must be understood and the pros and cons should be compared before applying such approaches.

Keywords: CQRS. *Event Sourcing*. Software architecture. Patterns. Microservices.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Com a atual necessidade por softwares nativos para nuvem, os quais possuem premissas de resiliência, escalabilidade, desempenho e disponibilidade, uma das questões chave no desenvolvimento destas aplicações consiste na concepção de um modelo de dados adequado a esta realidade. Mas como é possível maximizar o desempenho, escalabilidade e segurança das aplicações utilizando padrões arquiteturais que alteram o modo como os dados são manipulados pela aplicação?

Além de apresentar alguns conceitos importantes no desenvolvimento de microsserviços, o objetivo dessa pesquisa é avaliar a utilização de alguns padrões arquiteturais no desenvolvimento de microsserviços, apresentando os possíveis ganhos de desempenho, capacidade de escala e segurança de informação.

Ao combinar padrões arquiteturais como CQRS e *Event Sourcing* no desenvolvimento de aplicações, espera-se que haja algum tipo de ganho de desempenho, sejam facilmente escaláveis, garantam a segurança na camada de dados da aplicação e trilhem um caminho de eventos para auxiliar na auditoria dos dados.

Neste trabalho serão abordados temas como desenvolvimento de software, diferentes tipos de arquitetura de software, padrões arquiteturais utilizados em softwares modernos e microsserviços, antecedendo a apresentação dos padrões arquiteturais CQRS e *Event Sourcing*. Além disso, haverá apresentação de estudo de caso do desenvolvimento de uma aplicação JAVA construída utilizando os padrões arquiteturais citados, a fim de demonstrar os benefícios e as desvantagens ao utilizar tais padrões.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentados alguns conceitos importantes deste trabalho, tais como desenvolvimento de software, algoritmos de programação, CQRS, *Event Sourcing*, entre outros.

2.1 DESENVOLVIMENTO DE SOFTWARE

O desenvolvimento de software é um processo caracterizado como um conjunto de atividades utilizadas para construir um software que será aperfeiçoado e revisado durante o seu ciclo de vida.

Sommerville (2011) auxilia a entender o conceito de desenvolvimento de software ao diferenciar o desenvolvimento amador do desenvolvimento profissional de software. O autor exemplifica que ao escrever um sistema de software, para uso próprio, documentar manuais ou sua arquitetura não é uma preocupação necessária. Por outro lado, ao escrever um software onde outros desenvolvedores irão realizar alterações, algumas informações adicionais devem ser fornecidas, como documentações e manuais de utilização.

> À medida que os sistemas de software requeridos pelos usuários crescem em complexibilidade, o processo de desenvolvimento de software também torna-se mais complexo e passa a ser imprescindível a utilização de ferramentas automatizadas para apoiar suas tarefas. (FALBO e TRAVASSOS, 1996, p. 327).

Pode-se afirmar, então, que o desenvolvimento de software envolve diversas técnicas e não se trata exclusivamente do programa em si, mas sim variadas atividades de organização que em sua totalidade proporcionam o alcance dos atributos essenciais de um bom software. Segundo Machado (2004), esses atributos são funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade.

No entanto, estas técnicas têm sido aperfeiçoadas ao longo dos anos, criando padrões e diretrizes que norteiam o desenvolvimento de software dentro das necessidades e aplicações de cada sistema. Uma destas técnicas refere-se a um

modelo descritivo dos componentes de um sistema, também chamado de arquitetura de software.

2.1.1 **Arquitetura de software**

Quando o assunto é Arquitetura de software, é preciso citar o livro "*An Introduction to Software Architecture*" de David Garlan e Mary Shaw.

Para Garlan e Shaw (1994, p. 1, tradução nossa):

> À medida que o tamanho e a complexidade dos sistemas de software aumentam, o problema de design vai além dos algoritmos e estruturas de dados da computação: projetar e especificar a estrutura geral do sistema surge como um novo tipo de problema.
> Questões estruturais incluem organização bruta e estrutura de controle global; protocolos para comunicação, sincronização e acesso a dados; atribuição de funcionalidade para projetar elementos; distribuição física; composição do design elementos; escala e desempenho; e seleção entre alternativas de design.

Há que se citar também, Bahsoon e Emmerich (2003), que afirmam que a arquitetura do sistema no início do desenvolvimento é o primeiro artefato de design que atende às metas de qualidade, envolvendo diretamente questões como segurança, usabilidade, estabilidade, confiabilidade e desempenho.

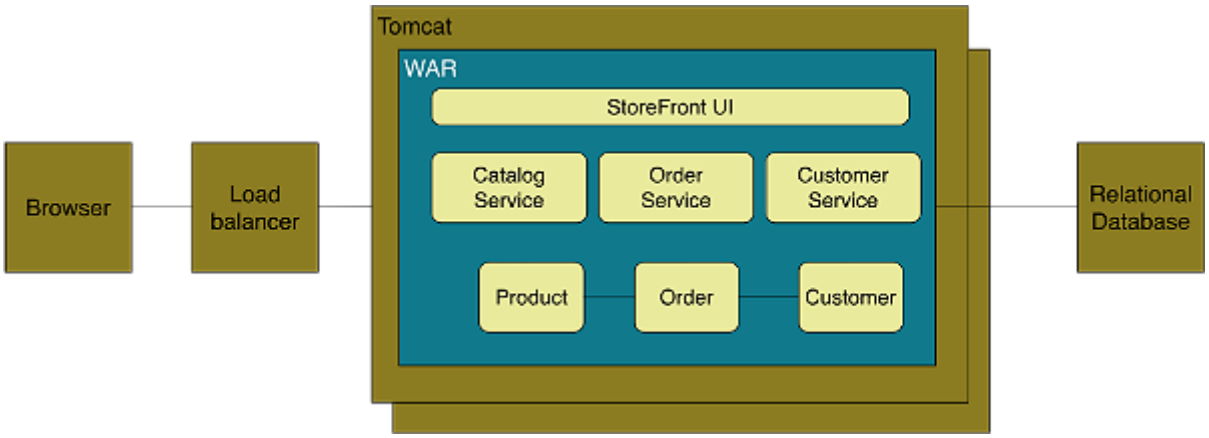
Ambos os autores concordam que, ao utilizar uma arquitetura de software definida, haverão ganhos durante o processo de desenvolvimento de software, pois solucionam problemas de design do produto. Pode-se afirmar que com uma arquitetura de software, tem-se normalmente soluções para problemas encontrados durante o desenvolvimento de software.

A seguir serão apresentados alguns modelos arquiteturais para desenvolvimento de software, como a arquitetura monolítica, cliente-servidor, orientada a serviços, arquitetura 3 camadas e arquitetura de microsserviços.

2.1.1.1 Monolítica

O pesquisador Richardson (2014) afirma que, a arquitetura monolítica empacota todos os componentes de uma aplicação em uma única unidade de implantação, em Java, por exemplo, são formadas por um arquivo *WAR* ou *EAR*.

Figura 1 - Arquitetura Monolítica



Fonte: Richardson (2014)

A arquitetura monolítica é a mais antiga e a mais utilizada até hoje nos sistemas operacionais. Tem a complexidade de um sistema quebrado em módulos, sendo projetado para um único executável, onde todos esses módulos do sistema são executados em uma mesma máquina, consumindo recursos de processamento, memória, banco de dados e arquivos.

> Uma aplicação monolítica geralmente é composta por uma única unidade de software compostas essencialmente por três partes: uma interface de interação com o usuário, uma base de dados e uma aplicação servidor para o processamento de requisições, atualização de dados e lógica de domínio. Construída sobre um único executável lógico, de modo que para qualquer alteração seja necessário a construção e liberação de uma nova versão da aplicação. (DE SOUZA e PELISSARI 2017, p. 2).

Segundo Richards (2014) uma arquitetura monolítica funciona bem para aplicativos pequenos, mas é pesada para aplicativos complexos, tornando-se difícil para os desenvolvedores entenderem e manterem a aplicação. Outro aspecto que se pode ressaltar é que uma arquitetura monolítica dificulta a adoção de novas tecnologias, é difícil experimentar uma nova estrutura sem reescrever o aplicativo inteiro, o que é demorado e arriscado.

É importante ressaltar que com o crescimento do sistema em complexidade, alguns desafios surgem na arquitetura monolítica, como a manutenção cada vez mais cara e lenta, a escalabilidade limitada, muitas funções entrelaçadas, sendo assim, qualquer mudança pode trazer riscos operacionais e instabilidade ao sistema.

2.1.1.2 Cliente-Servidor

Battisti (2001) esclarece que o modelo cliente-servidor é segmentado em processos distintos, onde um meio é encarregado da manutenção da informação (servidor) e o outro é incumbido da obtenção dos dados (cliente).

> É uma abordagem da computação que separa os processos em plataformas independentes que interagem, permitindo que os recursos sejam compartilhados enquanto se obtém o máximo de benefício de cada dispositivo diferente, ou seja, Cliente/Servidor é um modelo lógico. (VASKEVITCH, p.375, 1995).

Pode-se afirmar então que a estrutura essencial desta arquitetura é baseada em máquinas que atuam como servidores, disponibilizando recursos para as demais máquinas, que atuam como clientes, realizando as requisições.

Estes servidores são máquinas com maior poder de processamento, que tem uma execução contínua, atendendo diversos clientes simultaneamente. Já os clientes, são microcomputadores ligados em rede, que instauram e concluem a comunicação com os servidores.

2.1.1.3 Arquitetura orientada à serviços - SOA

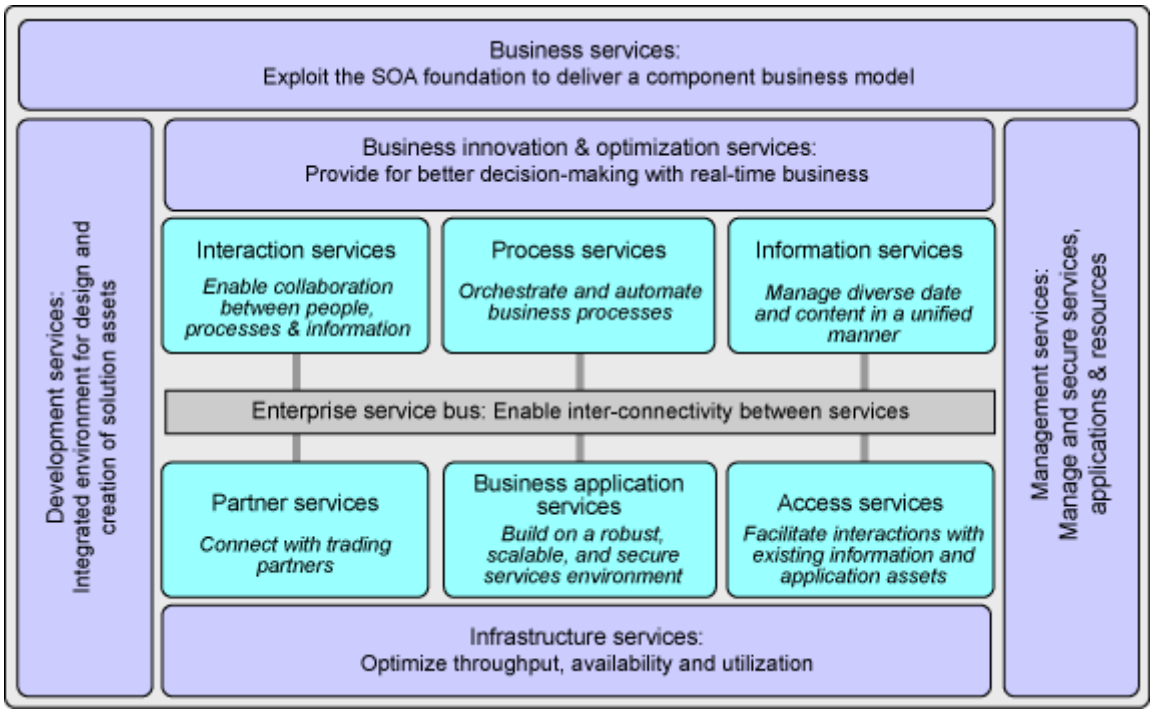
Schmutz, Welkenbach e Liebhart (2010, tradução nossa), dizem que a arquitetura orientada a serviços é um termo que descreve uma forma de implementar uma arquitetura empresarial, partindo da análise do negócio, onde os processos e áreas individuais das funcionalidades são identificados e estruturados.

Para Bieberstein (2008, p. 5, tradução nossa):

> Uma arquitetura orientada a serviços é uma estrutura para integrar processos de negócios e oferecer suporte à infraestrutura de TI como componentes seguros e padronizados - serviços - que podem ser reutilizados e combinados para atender às mudanças nas prioridades dos negócios.

Dito de outra forma, a arquitetura orientada a serviços tem o intuito de dispor as funcionalidades de um sistema como um serviço, sendo capaz de compartilhar funcionalidades e reutilizá-las entre diferentes aplicações.

Figura 2 - Modelo de referência SOA



Fonte: Portier (2006)

Há de se observar, também, o barramento de serviço na figura 2, que é o responsável por viabilizar os serviços do sistema com uma maior facilidade aos usuários e a outras aplicações, tornando o processo de integração mais eficiente.

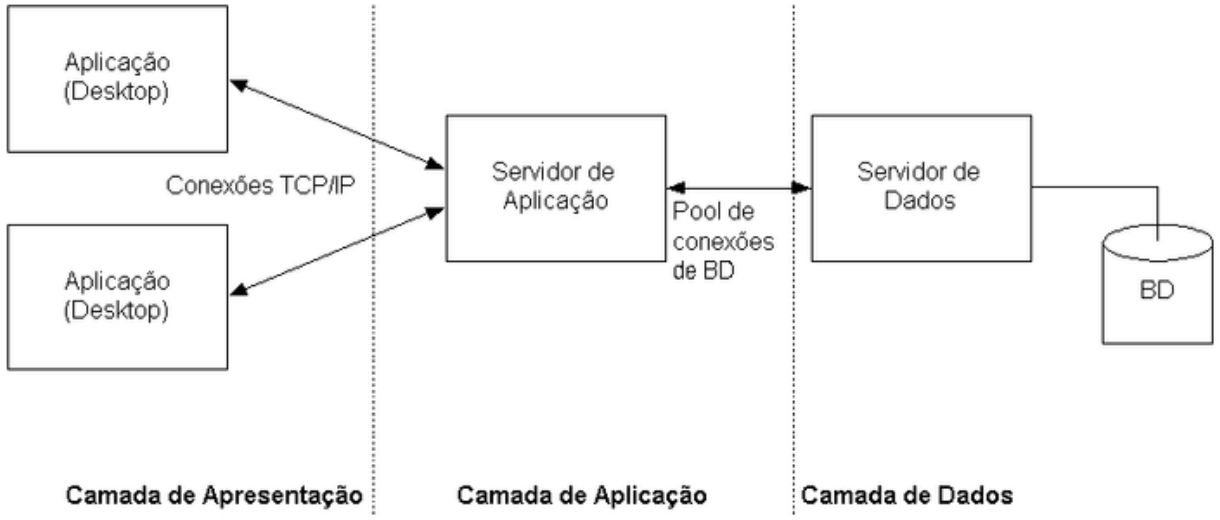
Segundo Trinugroho, Reichert e Fensli (2011), SOA tem sido utilizada para promover a interoperabilidade e a maneabilidade, já que a sua natureza de baixo acoplamento permite integrar sistemas legados e pode facilmente atender necessidades em constante evolução.

2.1.1.4 Arquitetura em 3 camadas

Segundo Kambalyal (2010) os componentes de uma arquitetura em 3 camadas são: a interface de usuário, a regra de negócio e o acesso aos dados.

Na primeira camada, conhecida como camada de apresentação é onde ocorre toda a interação com o usuário, já a regra de negócio pertence a segunda camada, onde as validações e a inteligência da aplicação acontece, e por fim, temos a terceira camada, ou camada de persistência, onde as informações geradas na segunda camada são armazenadas, nesta camada pode haver uma integração com banco de dados.

Figura 3 - Arquitetura 3 camadas



Fonte: UFCG (2019)

Albertoni (2013) ressalta que a comunicação entre essas camadas, pode ser simplificada através de protocolos abertos e APIs expostas. Outro aspecto que se pode ressaltar é que a criação dos componentes do cliente pode ser em qualquer linguagem de programação e esses clientes são executados em qualquer sistema operacional, conversando com a camada de lógica. Pode-se afirmar então que a camada de lógica é o fundamento chave desta arquitetura.

2.1.1.5 Microsserviço

Para RedHat (2019), “os microsserviços são uma arquitetura e uma abordagem para escrever software. Com eles, as aplicações são desmembradas em componentes mínimos e independentes”.

Para Richardson (2019), microsserviços refere-se a um estilo arquitetural que organiza uma aplicação de forma fracamente acoplada, com implementação de forma independente, altamente manutenível e testável e que pode ser propriedade de uma equipe pequena.

Ou seja, utilizando a arquitetura de microsserviços, as aplicações não necessitam de outras aplicações para funcionarem. Com base nisso, é possível manter equipes e infraestruturas distintas entre aplicativos, podendo inclusive, serem criados utilizando linguagens e/ou padrões arquiteturais diferentes.

Com base nesses conceitos, apresenta-se algumas vantagens em utilizar a arquitetura de microsserviços.

Para Fowler (2017, p. 21):

> Com a arquitetura de microserviços, uma aplicação pode ser facilmente escalada tanto horizontalmente quanto verticalmente, a produtividade e a velocidade do desenvolver aumentam dramaticamente e tecnologias antigas podem facilmente ser trocadas pelas mais recentes.

O que se confirma na leitura de Gertel (2019), que cita algumas das vantagens da utilização deste tipo de arquitetura como sendo a possibilidade de iniciar as aplicações de forma paralela e atualizar a aplicação de forma independente e incremental, não necessitando alterar todo o produto.

Com a atual necessidade por aplicações nativas para nuvem, as quais possuem premissas de resiliência, escalabilidade, desempenho e disponibilidade, uma das questões chave no desenvolvimento destas aplicações consiste na concepção de um modelo de dados adequado à esta realidade. Neste sentido, desenvolver aplicações utilizando a arquitetura de microsserviços tem se mostrado uma das melhores alternativas.

A seguir, será apresentado o conceito de software moderno e os padrões para desenvolvimento de softwares modernos.

2.2 DEFINIÇÃO DE SOFTWARE MODERNO

Demeyer (2008), retrata a crise do software na década de 60 como um fator determinante para a organização da primeira conferência voltada a engenharia de software, realizada pelo Comitê de Ciência da OTAN em 1960, onde foram estabelecidos princípios de engenharia para obter um software confiável, eficiente e economicamente viável.

> Os engenheiros projetam pontes e outras estruturas complexas há milênios. A engenharia civil é uma das disciplinas mais clássicas da engenharia - possui projetos, procedimentos e ferramentas bem desenvolvidos. Pontes raramente caem. Por outro lado, o design de sistemas de computador é uma das disciplinas menos clássicas de engenharia e seus produtos são frequentemente mal compreendidos, incontrolavelmente complexos e não confiáveis. (SPECTOR, 1986, p.267, tradução nossa) .

Ao longo do que tem sido analisado, percebe-se nitidamente que a crise do software foi o momento evidente onde iniciaram-se as preocupações com o desenvolvimento de software e os benefícios de projetos estruturados e bem executados.

Dentro deste contexto, foram propostos modelos de desenvolvimento de software, tal como o modelo de desenvolvimento clássico, também chamado de modelo sequencial, onde se tem um fluxo linear de atividades, que são concluídas uma após a outra.

Segundo ISTQB (2010) o modelo sequencial resulta em softwares com um conjunto completo de recursos mas demandam um grande período de tempo até estarem prontos para serem entregues.

Para Sommerville (2011) existem quatro atividades fundamentais em todos os processos de software: primeiramente a especificação de software, seguido pelo desenvolvimento de software, validação de software e a evolução de software.

Richards (2015) diz que uma prática comum dos desenvolvedores é codificar sem utilizar uma arquitetura padrão e, como resultado desta prática, produz-se um código desorganizado.

Ao longo do que tem sido analisado, percebe-se nitidamente que processos organizacionais são essenciais para a qualidade de vida do software a ser

desenvolvido e que um estilo de arquitetura melhora o particionamento e fornece soluções para problemas recorrentes.

2.2.1 **Padrões arquiteturais para softwares modernos**

Segundo Richards (2015, p. 5, tradução nossa) “Os padrões de arquitetura ajudam a definir as características básicas e comportamento de uma aplicação”.

O pesquisador Meier et al (2008) define que um estilo de arquitetura é um conjunto de princípios que visa melhorar o particionamento do software, aprimorando o design e fornecendo soluções para problemas recorrentes.

É importante ressaltar que a arquitetura de software é determinante para o sucesso do sistema, definindo sua estrutura para a implementação. Sendo assim, a organização dos componentes em um sistema impacta em sua qualidade, mas cabe lembrar que o número elevado de camadas pode prejudicar o desempenho do sistema.

2.2.2 **Metodologia Doze Fatores**

A metodologia dos doze fatores surgiu com o intuito de oferecer soluções e práticas ideais para o desenvolvimento de aplicações.

Segundo Wiggins (2011), a metodologia doze fatores é utilizada para construir aplicativos SaaS que usam formatos declarativos para automatizar a configuração inicial, tem um formato claro com o sistema operacional que o suporta, são adequados para implantação em plataformas em nuvem, minimizam a divergência entre desenvolvimento e produção e podem ser escaladas sem mudanças significativas em ferramentas, arquiteturas ou práticas de desenvolvimento.

Nesta perspectiva é importante relatar os elementos que compõem tal metodologia, de acordo com a documentação oficial do *The Twelve Factor App*:

- Base de código: onde todos os códigos são gerenciados através de um sistema de controle de versões e revisão;
- Dependências: declarar e isolar do código qualquer dependência;

- Configurações: armazenar as configurações no ambiente, não colocando configurações no código fonte;
- Serviços de apoio: tratar serviços de apoio como recursos ligados, isso traz flexibilidade caso ocorra mudanças e não será necessário alterar o código;
- Build, release, run: separar estritamente os builds e execute em estágios;
- Processos: executar a aplicação como um ou mais processos;
- Vínculo de porta: exportar serviços através de portas, criando uma URL separada para o API que o website poderá utilizar;
- Concorrência: dimensionar por um modelo de processo, cada processo trabalha de forma independente e se forem executados como processos separados podemos ter um melhor dimensionamento, executando mais processos ao mesmo tempo;
- Descartabilidade: maximizar a robustez com inicialização e desligamento rápido;
- Desenvolvimento e produção semelhantes: manter o desenvolvimento, teste e produção o mais semelhante possível;
- Logs: tratar os logs como fluxo de eventos, pois os logs são ferramentas importantes que nos permitem entender o comportamento de uma aplicação;
- Processos de gerenciamento: executar tarefas de gerenciamento como processos pontuais.

Tem-se, a partir do que foi exposto acima, uma base para projetos que podem ser escalados de forma mais fácil e atendendo requisitos.

2.3 ALGORITMOS DE PROGRAMAÇÃO

Antes de aplicar o conceito de algoritmos à programação, faz-se necessário entender o que são algoritmos. Segundo Puga e Risseti (2008), algoritmos são sequências lógicas de instruções que, ao serem seguidas, contribuem para a resolução de um problema ou para a execução de uma tarefa.

Conforme Ascencio (1999, apud Ascencio, 2012, p. 1), “algoritmo é a descrição de uma sequência de passos que deve ser seguida para realização de uma tarefa”.

Portanto, muito antes da aplicação do termo “algoritmos de programação”, os algoritmos já estavam presentes na resolução de problemas, não só em situações relacionadas à matemática.

Este é o ponto de partida para a compreensão dos algoritmos de programação.

> O termo algoritmo, do ponto de vista computacional, pode ser entendido como regras formais, sequenciais e bem definidas a partir do entendimento lógico de um problema a ser resolvido por um programador com o objetivo de transformá-lo em um programa que seja possível de ser tratado e executado por um computador. (MANZANO e OLIVEIRA, 2010, p. 25).

Há que se citar também, Puga e Risseti (2008, p. 9) “os algoritmos são amplamente utilizados na área da computação, seja na elaboração de soluções voltadas à construção de interfaces, softwares e hardware, seja no planejamento de redes”.

Conforme se verificou até aqui, algoritmos estão presentes no desenvolvimento de softwares e acabam servindo como receitas para a criação de códigos fonte com base em uma linguagem de programação. Na próxima seção será apresentado o conceito de linguagem de programação para criação de softwares a partir de algoritmos.

2.3.1 **Linguagem de programação**

De modo geral, linguagem de programação diz respeito a como o desenvolvedor de software envia instruções para um computador. Com base em regras de sintaxe e semântica, é possível criar algoritmos que são executados como softwares.

Para Silva e Melo (2014, p. 8) "uma linguagem de programação determina os recursos disponíveis e sua forma de utilização para construir máquinas abstratas específicas, de tal forma que elas possam ser simuladas adequadamente em computadores".

Neste sentido, para Willrich (2000), as linguagens de programação podem ser definidas como um conjunto limitado de instruções, associadas a um conjunto de regras que definem como as instruções podem compor programas para resolver determinados problemas.

Portanto, ao utilizar linguagens de programação específicas, consegue-se escrever códigos que sejam interpretados pelos computadores. Em adição, há algumas classificações quanto às linguagens, que podem ser de diferentes níveis, ou seja, algumas executam em camadas mais próximas ao hardware (baixo nível) e outras com uma linguagem mais parecida com a linguagem natural (alto nível).

No próximo item será apresentada de forma breve a linguagem de programação Java.

2.3.1.1 Java

Segundo a Oracle (2019), trata-se de “uma linguagem de programação e plataforma computacional”. A primeira versão da linguagem foi chamada de Oak (carvalho). Criada por James Gosling, acabou sendo uma das primeiras linguagens a oferecer portabilidade para diferentes ambientes e desenvolvimento para múltiplas plataformas.

Executar o mesmo programa em diferentes plataformas só é possível pois, ao compilar um programa escrito em Java, é gerado um compilado de *bytecodes*, que são interpretados por um ambiente de execução chamado Java Virtual Machine (JVM). Para Oracle (2019), “Java Virtual Machine (JVM) é um conjunto de programas de software que permite a execução de instruções – geralmente escritas em *bytecode* Java. Os JVMs estão disponíveis para todas as plataformas de software e hardware mais comuns”.

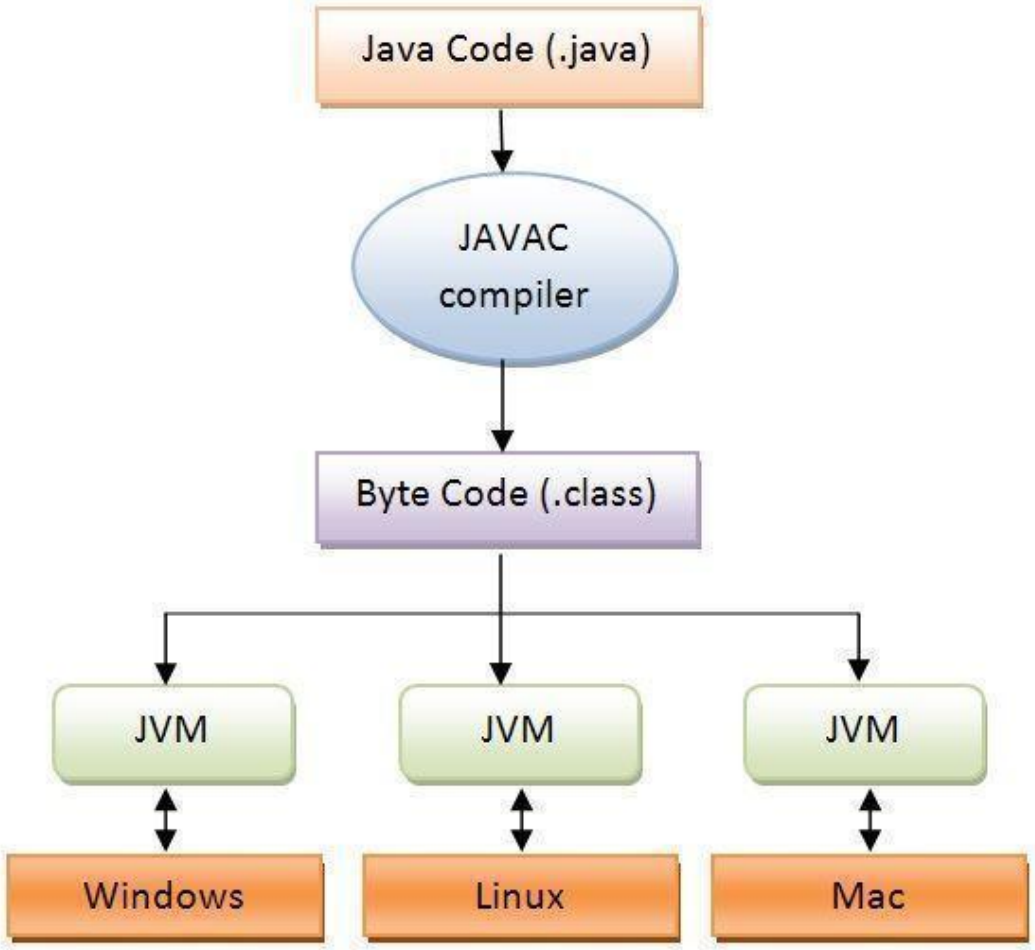
Frente a isso, para Deitel e Deitel (2010, p. 28), “um aplicativo Java é um programa de computador que é executado quando você utiliza o comando java para carregar a Java Virtual Machine (JVM)”.

Porém, antes de enviar o código para execução na JVM, é necessário entender como o código java (.java) é transformado em um *bytecode* (.class).

O compilador trabalha de forma simples, para Deitel e Deitel (2010, p. 10), “o compilador Java converte o código-fonte Java em *bytecodes* que representam as

tarefas a serem executadas na JVM". Tem-se assim o ponto de partida para a compreensão do funcionamento da JVM e do compilador javac.

Figura 4 - Funcionamento da JVM



Fonte: DevMedia (2019)

Pode-se visualizar na imagem acima que o código java (.java) primeiramente é submetido à um compilador (javac) que o transforma em um código específico que será interpretado pela JVM, o *bytecode* (.class).

Justamente pela possibilidade de ser executado em diversas plataformas distintas, o Java acaba sendo uma das linguagens mais utilizadas atualmente.

Para corroborar com tal afirmação, cita-se Ascencio e Campos (2012, p. 11) que afirmam que o Java "possui como principais características: simplicidade, orientação a objetos, portabilidade, alta performance e segurança", que atualmente são imprescindíveis para o desenvolvimento de software escaláveis, com alto desempenho e multiplataformas.

2.4 CLOUD

Para Microsoft (2019), Cloud “é um termo utilizado para descrever uma rede global de servidores, cada um deles com uma função única”.

Complementando tal afirmação, para Amazon (2019), “a computação em nuvem é a entrega sob demanda de poder computacional, armazenamento de banco de dados, aplicativos e outros recursos de TI pela Internet com uma definição de preço conforme o uso.”

Ou seja, as informações, aplicações e serviços não são armazenados localmente, mas sim em servidores que mantém tais informações na Internet. Com base nisso, várias empresas optam por ofertar serviços de hospedagem em nuvem, disponibilizando ambientes completos com hardware escalável e funcionalidades sob demanda.

Algumas dessas empresas provedoras de soluções em nuvem são a AWS (Amazon), Azure (Microsoft) e a IBM Nuvem (IBM). Para comercializar tais serviços, Veras (2015, p. 44), cita que são três os principais modelos de serviços para o Cloud, são eles: “Infraestrutura como um serviço (IaaS), plataforma como um serviço (PaaS) e software como um serviço (SaaS).”. A seguir será apresentado os conceitos por trás das siglas citadas acima.

2.4.1 **Infraestrutura como serviço - IaaS**

Para IBM (2019), Infraestrutura como serviço (IaaS) “é uma oferta de computação em cloud na qual um fornecedor oferece aos usuários acesso a recursos computacionais, como servidores, armazenamento e redes.”

Também na mesma linha, para Microsoft (2019, tradução nossa), IaaS “é uma infraestrutura de computação instantânea, provisionada e gerenciada pela Internet”.

De modo geral, as empresas comercializam hardware para os clientes utilizarem seus próprios aplicativos dentro da plataforma. Com base nas necessidades, a infraestrutura do ambiente é altamente escalável.

Figura 5 - Modelo de serviço Cloud - IaaS

Fonte: Microsoft (2019)

Assim como mostra a imagem, somente a parte física (hardware), segurança e armazenamento é comercializado no modelo de Infraestrutura como serviço.

2.4.2 **Plataforma como serviço - PaaS**

De acordo com a IBM (2019), plataforma como serviço ou PaaS, “é uma oferta de computação em cloud que oferece aos usuários um ambiente de cloud na qual eles podem desenvolver, gerenciar e entregar aplicativos”.

Diferentemente do modelo anterior, no PaaS a empresa provedora também fornece ferramentas de desenvolvimento, testes e hospedagem. O provedor do serviço continua sendo responsável pela segurança, infraestrutura e armazenamento, e além disso, inclui outros itens, como sistema operacional, ferramentas de desenvolvimento e gerenciamento de banco de dados.

Para a Amazon (2009), proprietária da AWS (Amazon Web Services), utilizando PaaS as empresas não precisam gerenciar a infraestrutura (hardware e sistemas operacionais), permitindo que os esforços sejam concentrados na implementação e gerenciamento das aplicações.

Figura 6 - Modelo de serviço Cloud – PaaS

Fonte: Microsoft (2019)

Percebe-se que no modelo PaaS, além do já entregue no modelo IaaS, ainda há a inclusão de ferramentas de desenvolvimento, ferramentas para gerenciamento de banco de dados e o sistema operacional.

2.4.3 **Software como serviço - SaaS**

O último modelo, software como serviço (SaaS), foi definido pela Amazon (2019) como “serviço que oferece um produto completo, executado e gerenciado pelo provedor”. Trata-se de aplicativos que serão executados pela internet e que não requerem nenhum tipo de cuidado relacionado à infraestrutura. A responsabilidade de manter a aplicação no ar é do provedor de serviço.

Para IBM (2019), uma das principais vantagens do modelo SaaS, é não haver qualquer necessidade de gerenciamento, instalação ou atualização das aplicações por parte do cliente.

Alguns dos exemplos de aplicações SaaS frequentemente utilizadas são o webmail e o Office 365, da própria Microsoft.

Abaixo a figura que exemplifica a diferença entre os modelos.

Figura 7 - Modelo de serviço Cloud – SaaS

Fonte: Microsoft (2019)

Percebe-se que com o modelo de software como serviço a responsabilidade é inteiramente do provedor de serviços, incluindo aplicativos hospedados, ferramentas de desenvolvimento e gerenciamento de banco de dados, sistemas operacionais, servidores e armazenamento, segurança e planta física do *datacenter*.

2.5 BANCO DE DADOS

Segundo Korth e Silberschatz (1995), banco de dados “é uma coleção de dados inter-relacionados, representando informações sobre um domínio específico”, ou seja, são informações que se relacionam entre si, normalmente sobre o mesmo assunto. Ainda sobre o conceito de banco de dados, conforme Date (2014) um banco de dados pode ser comparado com um armário, ou seja, um repositório para uma coleção de arquivos ou informações.

O conceito de banco de dados é simples, mas é necessário para apresentar outros conceitos, como os modelos de dados e a diferença na estrutura lógica de um banco de dados.

Para um melhor entendimento da utilização de *Event Sourcing*, será necessário comentar sobre dois modelos de dados, o relacional e o NoSQL.

2.5.1 **Modelo de dados Relacionais**

O modelo de dados relacionais (SQL) foi introduzido por Edgar F. Codd no ano de 1970 em "*Relational Model of Data for Large Shared Data Banks*" como um modelo baseado em lógica e na teoria de conjuntos.

Para Caldeira (2010, p.3):

> A associação entre os dados é o ponto forte dos sistemas relacionais. As tabelas são formadas por linhas e colunas onde figuram os dados. Numa base de dados relacional os dados estão todos representados como valores nas colunas das tabelas.

Ainda conforme Caldeira (2010), o modelo relacional baseia-se em domínio, relação e atributo.

O domínio tem o fundamento teórico na matemática dos conjuntos, que é uma teoria capaz de agrupar elementos através de características comuns, formando conjuntos.

A relação compreende atributos originados de um ou mais conjuntos, cada linha da tabela, representa o valor do atributo, enquanto os domínios representam todos os valores possíveis para um atributo. Essas linhas podem variar de acordo com as circunstâncias, já os domínios e os atributos são constantes.

2.5.2 **Modelo Não-Relacional (NoSQL)**

O modelo não relacional, também conhecido como NoSQL, foi criado como alternativa ao modelo relacional e tem como foco a baixa latência, escalabilidade e a possibilidade de manter grandes volumes de dados. Como confirma Monge (2018), ao afirmar que o desafio de armazenar e processar uma grande quantidade de dados com escalas cada vez maiores fez surgir o modelo NoSQL.

Segundo Amazon (2019), "os bancos de dados NoSQL são amplamente reconhecidos por sua facilidade de desenvolvimento, funcionalidade e performance em escala". Tais vantagens facilitam algumas questões durante o desenvolvimento de microsserviços, principalmente quando diz respeito a escalabilidade.

Existem diferentes tipos de bancos de dados Não-Relacionais, como afirma Ferreira (2018):

> Uma base de dados NoSQL, tem um esquema dinâmico para dados não estruturados, e o dado é armazenado em várias formas: pode ser orientado a coluna, orientado a documento, baseado em grafos ou organizado como chave-valor.

Dentre os diferentes tipos de bancos de dados não-relacionais, um dos modelos mais utilizados é orientado a documento, modelo este que armazena coleções de documentos e que são representados, segundo Lóscio (2011, p. 7, apud Monge, 2018) por “um objeto com um identificador único e um conjunto de campos, que podem ser strings, listas ou documentos aninhados”. Tal documento, normalmente diz respeito a um JSON (*JavaScript Object Notation*) que é enviado diretamente para o banco de dados.

Outro modelo bastante utilizado é o NoSQL chave-valor. Para Amazon (2019):

> Um banco de dados de chave-valor armazena dados como um conjunto de pares de chave-valor em que uma chave funciona como um identificador exclusivo. A chave e os valores podem ser qualquer coisa, desde objetos simples até objetos compostos complexos.

Alguns exemplos de bancos NoSQL são o MongoDB (orientado a documentos), Apache Cassandra (orientado a colunas) e DynamoDB (chave-valor).

2.6 CQRS

Neste tópico, tem-se peça importante para o desenvolvimento do trabalho de conclusão de curso, será apresentado um dos temas principais, o CQRS (segregação de responsabilidade de comandos e consultas).

Command-Query Responsibility Segregation consiste em um padrão arquitetural (*pattern*). Apresentado pela primeira vez por Greg Young no artigo “*CQRS Documents by Greg Young*”, publicado em 2010, este padrão é descrito como “um padrão muito simples que permite muitas oportunidades de arquitetura que, de outra forma, não existiriam”.

Segundo Ferreira (2016), “as origens do CQRS são baseadas no CQS (*Command-Query separation*), criado por Bertrand Meyer durante o desenvolvimento da linguagem de programação Eiffel”. A ideia principal do CQS era, como cita

Nascimento (2019), “que os métodos de uma aplicação podem ser comandos ou consultas, mas nunca ambos”.

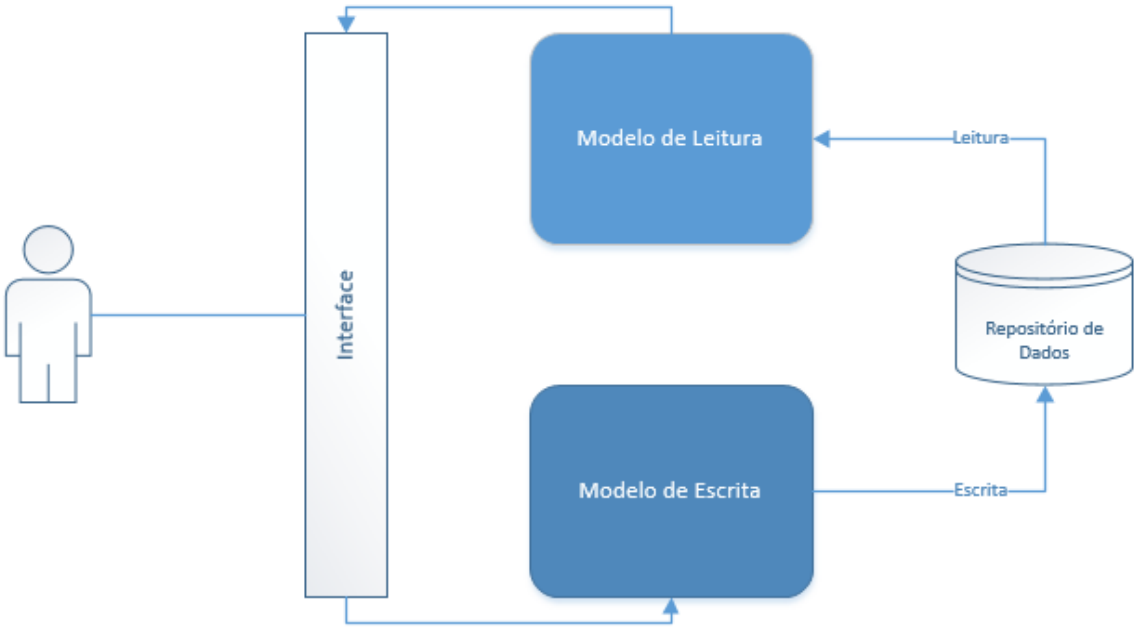
Nesse sentido, para Young (2010, tradução nossa), “CQRS é simplesmente a criação de dois objetos em que anteriormente havia apenas um. A separação ocorre com base no fato de os métodos serem um comando ou uma consulta”.

Outro nome com bastante relevância quando se diz respeito ao uso do CQRS é Martin Fowler. Em um dos seus artigos, Fowler (2011) comenta que a essência do CQRS é a possibilidade de utilizar, inclusive, modelos diferentes para atualizar e ler informações.

Com base nas afirmações dos autores, fica claro que com o CQRS divide-se comandos (*command*) que executam inclusões, alterações e exclusões no modelo de dados e consultas (*query*), que somente fazem a leitura das informações.

De forma simples, pode-se entender o modelo de segregação com a imagem a seguir.

Figura 8 – CQRS



Fonte: Ferreira (2016)

Como complementa Fowler (2011), por modelos separados, entende-se que serão modelos de objetos diferentes, executando em processos lógicos diferentes, possivelmente em hardware separados.

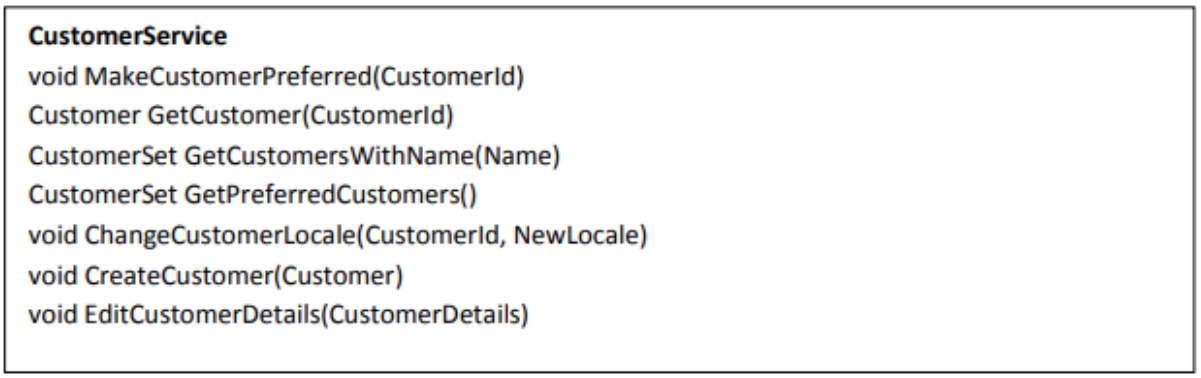
Para trabalhar com diferentes modelos para escrita e leitura se faz necessário entender e dividir corretamente as responsabilidades.

Para Young (2011, p.17, tradução nossa) "essa separação reforça a noção de que o lado do comando e o lado da consulta têm necessidades muito diferentes. As propriedades arquitetônicas associadas aos casos de uso de cada lado tendem a ser bem diferentes".

É necessário entender que no lado da consulta tem-se somente métodos para obter dados, enquanto na parte de comando, são todos os outros eventos que acabam de alguma forma alterando informações no banco de dados.

Para entender de forma prática tal divisão, pode-se verificar o exemplo a seguir. Na arquitetura padrão será criado somente um serviço que faça inclusões, alterações, exclusões e leituras:

Figura 9 - Exemplo de uso sem CQRS



Fonte: Young (2010)

Com a aplicação do *pattern*, tem-se a divisão em dois serviços, como demonstrado a seguir.

Figura 10 - Exemplo utilizando CQRS

```
CustomerWriteService
void MakeCustomerPreferred(CustomerId)
void ChangeCustomerLocale(CustomerId, NewLocale)
void CreateCustomer(Customer)
void EditCustomerDetails(CustomerDetails)

CustomerReadService
Customer GetCustomer(CustomerId)
CustomerSet GetCustomersWithName(Name)
CustomerSet GetPreferredCustomers()
```

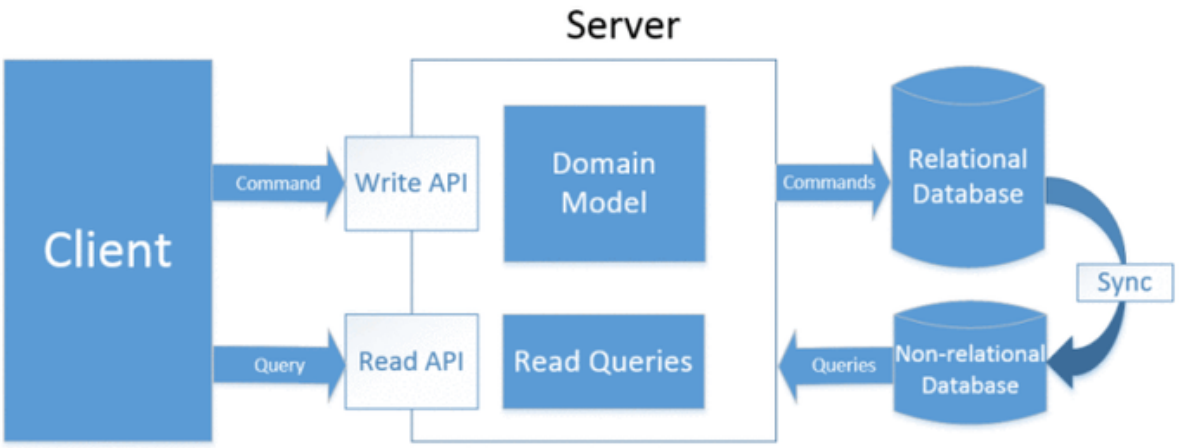
Fonte: Young (2010)

2.6.1 **Diferentes tipos de implementação**

Como sugere Pires (2016), “não existe uma única maneira de implementar o CQRS na sua aplicação, pode ser feito de uma forma simples ou muito complexa, depende da necessidade.”

A implementação pode ser feita utilizando modelos diferentes, porém, utilizando um único banco de dados ou, a infraestrutura pode ser distinta, tendo por exemplo, um banco relacional para os comandos e um banco NoSQL para consultas. A Figura 11 ilustra esta separação do modelo de dados em bancos de dados distintos..

Figura 11 - CQRS com diferentes bancos de dados



Fonte: Passos (2018)

Entretanto, separar em bancos diferentes aumenta a complexidade do projeto, trazendo outra necessidade: o sincronismo de informações.

2.6.1.1 Sincronismo de informações

O sincronismo de dados entre os bancos pode ser executado de diferentes formas. A aplicação pode implementar atualizações automáticas, ou seja, sempre que uma alteração for realizada no banco de escrita, um processo seja executado para atualizar o banco de leitura.

Outros modelos que podem ser implementados são atualizações periódicas, ou seja, diariamente ou em alguns períodos pré-definidos, serão executados processos para atualizar as informações no banco de leitura. Porém, como explica Pires (2016), “a estratégia mais utilizada é a de atualização eventual, tal estratégia parte do princípio que todo dado exibido já pode estar desatualizado”.

A atualização eventual, consiste em disparar um processo assíncrono para atualização de dados assim que uma alteração no banco de escrita seja realizada.

Tal necessidade apresenta um novo *pattern* que pode ser utilizado, o *Event Sourcing* (apresentado no item 2.7), pois com ele, tem-se a informação completa da última alteração realizada no banco de dados.

2.6.2 **Benefícios do CQRS**

Como todo padrão arquitetural criado, o uso do CQRS é recomendado em situações específicas onde são encontrados benefícios na sua utilização. Nesta subseção serão apresentados algumas das vantagens de se utilizar CQRS para desenvolvimento de aplicações.

De acordo com Microsoft (2019), um dos benefícios é a possibilidade de realizar o dimensionamento independente dos bancos, ou seja, trazendo a possibilidade de escalar da forma desejada a infraestrutura. A respeito dessa possibilidade, afirma Ferreira (2016), “tipicamente, o número de consultas num sistema é muito superior ao número de comandos realizados”, conclui-se então, que a infraestrutura disponível para o banco de leitura deve ser mais robusta do que a disponível para o banco de escrita.

O desempenho do banco também pode ser afetado com tal separação, como cita Fowler (2010, tradução nossa), “pode-se aplicar diferentes estratégias de otimização para os dois lados, sendo um exemplo disso, o uso de diferentes técnicas de acesso ao banco de dados”. Tais estratégias permitem que bancos de leitura sejam otimizados para melhor performance durante a leitura e vice-versa.

2.6.3 **Desvantagens do CQRS**

Assim como vantagem, todos os padrões arquiteturais tem desvantagens, segundo Fowler (2010, tradução nossa), o CQRS não é indicado para sistemas que necessitam ter informações atualizadas do mesmo modo que são lidas, item este, que pode ser resolvido implementando sincronismo entre os bancos de forma automática.

Já para Microsoft (2019), a complexidade é outra desvantagem do uso de CQRS. Apesar do padrão ser simples, ele acaba resultando em um design de aplicativo mais complexo. Com base nestas informações, pode-se entender que em projetos que contenham regras de negócio simples, o padrão arquitetural não é o mais indicado para o uso.

2.6.4 **Aplicação de CQRS em microsserviços**

Os benefícios da utilização de CQRS para desenvolver microsserviços foram um fator chave para a motivação do *pattern*. Pensando cada vez mais em desenvolver aplicações nativas para nuvem e que possuem premissas de resiliência, escalabilidade, desempenho e disponibilidade, o CQRS acabou ganhando espaço no desenvolvimento de microsserviços.

Para Fowler (2011), com a utilização de CQRS (*Command Query Responsibility Segregation*) e a possibilidade de segregar leitura e escrita de dados persistidos, estas operações podem ser escaladas de forma independente, o que acaba sendo um grande ganho na tentativa de criar aplicativos com alto nível de escalabilidade.

Seguindo a linha do desenvolvimento de microsserviços, com CQRS pode-se modularizar o banco de dados, possibilitando a granularização mais fina dos componentes de uma aplicação.

2.7 *EVENT SOURCING*

Event Sourcing, ou na tradução literal “Fornecimento de eventos”, trata-se de um padrão arquitetural criado para permitir que sua aplicação capture todas as alterações em um estado do aplicativo em forma de sequência de eventos.

Uma das melhores definições de *Event Sourcing* vem de Martin Fowler em seu blog pessoal.

> Podemos consultar o estado de um aplicativo para descobrir o estado atual do mundo, e isso responde a muitas perguntas. No entanto, há momentos em que não queremos apenas ver onde estamos, também queremos saber como chegamos lá. (FOWLER, 2005).

Como complementa Movile (2019), ao afirmar que o *Event Sourcing* “garante que todas as alterações em uma aplicação sejam armazenadas em uma sequência de eventos. Assim podemos montar projeções do estado atual da nossa aplicação bem como consultar os eventos”.

Utilizando *Event Sourcing* tem-se a solução para um problema típico de segurança nas aplicações: manter o histórico dos estados, ou seja, ter uma ferramenta que possibilite posteriormente a visualização de um *log* de auditoria.

Além disso, há uma facilidade em reverter alterações já que é mantido todo o histórico de alterações na base de dados da aplicação.

2.7.1 **Funcionamento do *Event Sourcing***

Para se trabalhar com *Event Sourcing* é recomendado o uso de banco de dados não-relacionais (NoSQL), como afirma Santos (2019) ao dizer que, quando se fala em arquitetura de eventos, normalmente trata-se de um modelo não-relacional, pois, não há nenhum tipo de amarração com os *schemas* do modelo de tabelas.

A ideia é simples, toda alteração de estado das informações serão enviadas como eventos por uma fila, surgindo assim, a necessidade de uma ferramenta de Event Streaming que publica e consome eventos. Alguns exemplos de ferramentas para este propósito são o Apache Kafka, Apache ActiveMQ e RabbitMQ.

Utiliza-se frequentemente o modelo de dados NoSQL orientado a documentos, permitindo que tais eventos sejam enviados em forma de JSON.

Como demonstra Movile (2019) em seus exemplos, “ao invés de uma tabela com o dado consolidado teríamos um stream de eventos dessa conta”. A seguir o exemplo de como será a estrutura no banco de dados.

Figura 12 - Modelo relacional

id	balance	status
1	R$ 15,00	CLOSED

Fonte: Movile (2019)

Figura 13 - Modelo com stream de eventos

aggregate_id	event	payload
1	AccountAddedEvent	{"initial_balance":0,"status":"ACTIVE"}
1	AccountAmountCreditedEvent	{"amount":15}
1	AccountStatusUpdatedEvent	{"status":"INACTIVE"}
1	AccountStatusUpdatedEvent	{"status":"CLOSED"}

Fonte: Movile (2019)

Percebe-se que diferente do modelo relacional, agora tem-se informações que servem como *log* para futuras auditorias, além de, servirem como ponto de restauração caso ocorra algum problema na aplicação após o envio de um evento.

3 METODOLOGIA

A seguir será apresentada a metodologia utilizada para a presente pesquisa.

3.1 CARACTERIZAÇÃO DA PESQUISA

A presente pesquisa caracteriza-se como uma pesquisa exploratória, pois o principal objetivo é aprofundar o tema de desenvolvimento de microsserviços utilizando dois padrões arquiteturais, CQRS e *Event Sourcing*. Para isso, foi realizada uma pesquisa bibliográfica acerca dos teóricos que tratam da temática Desenvolvimento de Microsserviços. Em seguida, foi realizado um estudo de caso com base na utilização de CQRS e *Event Sourcing* no desenvolvimento de microsserviços e seus ganhos.

3.2 AMBIENTE DA PESQUISA

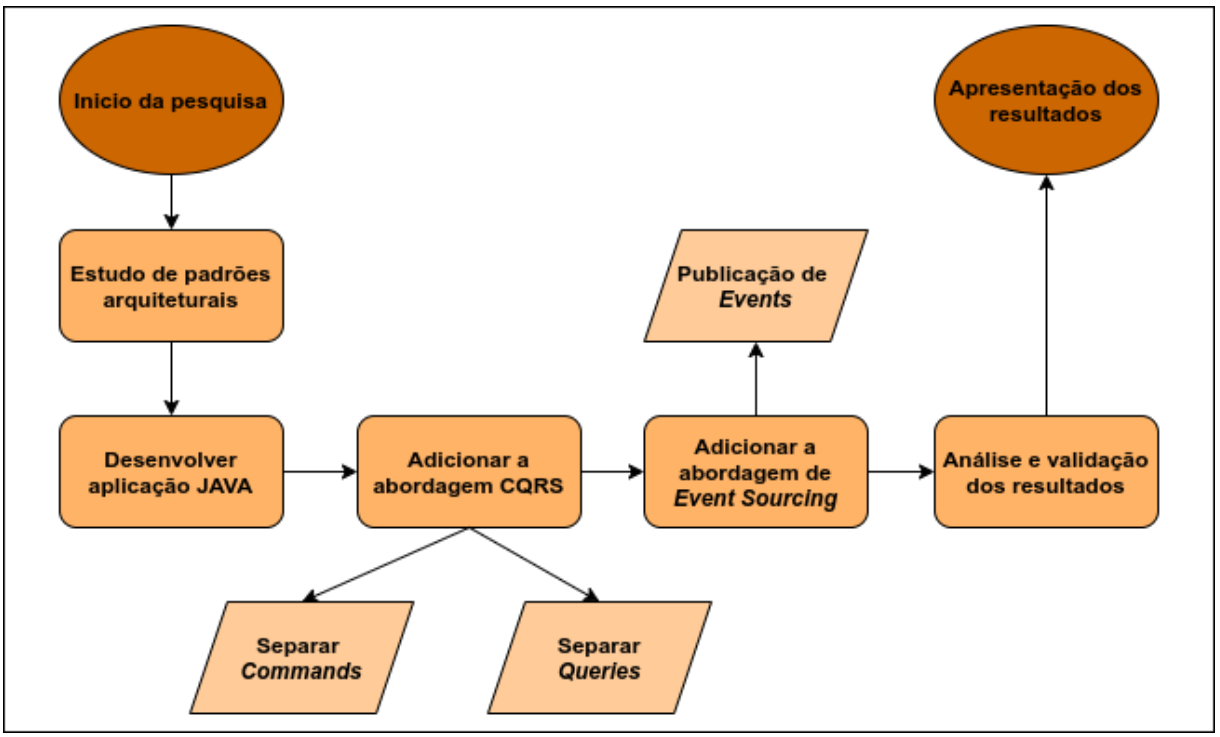
O estudo foi realizado em ambiente local com o desenvolvimento de aplicações que utilizam os recursos citados na monografia. O computador utilizado roda o sistema operacional Windows 10 e contém a seguinte configuração: 16GB de RAM, Processador i7-8700 @ 3.20GHz e SSD 240GB.

Para o desenvolvimento das aplicações foi utilizado o ambiente de desenvolvimento integrado (IDE) Eclipse IDE na versão 2021-03, além do OpenJDK na versão 13 ou superior. Todo o projeto foi desenvolvido utilizando JAVA e alguns *frameworks* para facilitar o desenvolvimento, como *Spring Framework, Hibernate* e *Axon Framework*, além da integração com o banco de dados *PostgreSQL* e o *Axon Server* via imagem *Docker* para visualização dos comandos e eventos.

3.3 ETAPAS DA PESQUISA

No fluxograma da figura a seguir, estão ilustradas as etapas de desenvolvimento da pesquisa.

Figura 14 - Fluxograma



Fonte: O Autor (2021)

Primeiramente foi realizado estudo de padrões arquiteturais conforme demonstrado no referencial teórico. Após esse estudo, foi desenvolvida a aplicação JAVA com *endpoints* REST para inclusão, leitura, alteração e exclusão de informações em banco, utilizando o *PostgreSQL*. Nas etapas a seguir, foram adicionados tanto o padrão CQRS, quanto a abordagem de *Event Sourcing* no projeto.

Para adicionar os padrões arquiteturais citados, foi utilizado o Axon Framework para facilitar a visualização dos Comandos e Eventos.

Na última etapa do projeto, foram analisados os resultados relacionados ao ganho de desempenho, capacidade de escala e segurança da aplicação.

3.4 DESENVOLVIMENTO DA APLICAÇÃO

A aplicação foi desenvolvida em JAVA, utilizando alguns *frameworks,* como o *Spring Framework* e o *Axon Framework*.

Para criar um exemplo de fácil entendimento, foi desenvolvida uma aplicação que de forma simples inclui e remove viagens, com destino e distância e nessas viagens podem ser incluídos e removidos passageiros.

Utilizando o *pattern* CQRS, há a divisão entre *commands* e *queries,* por isso, primeiramente será criado um agregado, responsável por tratar nossos comandos de inclusão e remoção de viagens e passageiros. Com isso é possível enviar os eventos para serem armazenados e posteriormente recebidos pela parte da aplicação que trata da leitura dos registros, a seguir serão apresentados alguns pontos da aplicação para melhor entendimento dos *patterns*.

Na aplicação desenvolvida os eventos foram enviados diretamente para o *Axon Server*, que acaba armazenando o agregado e servindo como banco NoSQL da aplicação, pois mantém as informações salvas em formato JSON. Além disso, o *Axon Server* tira a necessidade de manter um *RabbitMQ* para troca de mensagens e o sincronismo de dados.

Figura 15 - Exemplo de agregado com comandos e eventos

```
@Log4j2
@NoArgsConstructor
@Aggregate
public class TripAggregate {

    @AggregateIdentifier
    private String tripId;
    private String destination;
    private int distance;
    private Set<String> passengers;

    @CommandHandler
    public TripAggregate(CreateTripCommand command) {
        log.info("Send command: {}", command.getClass());

        AggregateLifecycle
                .apply(new TripCreatedEvent(command.getTripId(), command.getDestination(), command.getDistance()));
    }

    @CommandHandler
    public void handle(DeleteTripCommand command) {
        log.info("Send command: {}", command.getClass());

        AggregateLifecycle.apply(new TripDeletedEvent(command.getTripId()));
    }

    @CommandHandler
    public void handle(JoinPassengerCommand command) {
        if (!passengers.contains(command.getName())) {
            log.info("Send command: {}", command.getClass());

            AggregateLifecycle.apply(new PassengerJoinedTripEvent(tripId, command.getName()));
        }
    }

    @CommandHandler
    public void handle(LeavePassengerCommand command) {
        if (passengers.contains(command.getName())) {
            log.info("Send command: {}", command.getClass());

            AggregateLifecycle.apply(new PassengerLeftTripEvent(tripId, command.getName()));
        }
    }
```

Fonte: O Autor (2021)

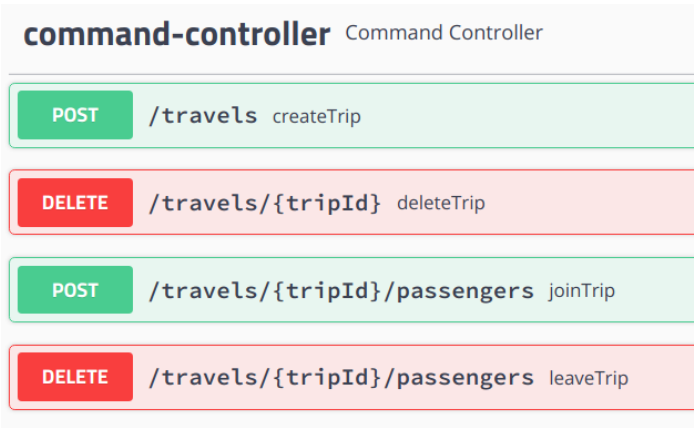
Figura 16 - Exemplo de agregado com comandos e eventos

```
68      @EventSourcingHandler
69      protected void on(TripCreatedEvent event) {
70          log.info("Send event: {}", event.getClass());
71
72          this.tripId = event.getTripId();
73          this.destination = event.getDestination();
74          this.distance = event.getDistance();
75          this.passengers = new HashSet<>();
76      }
77
78      @EventSourcingHandler
79      protected void on(PassengerJoinedTripEvent event) {
80          log.info("Send event: {}", event.getClass());
81
82          this.passengers.add(event.getName());
83      }
84
85      @EventSourcingHandler
86      protected void on(PassengerLeftTripEvent event) {
87          log.info("Send event: {}", event.getClass());
88
89          this.passengers.remove(event.getName());
90      }
91
92      @EventSourcingHandler
93      protected void on(TripDeletedEvent event) {
94          log.info("Send event: {}", event.getClass());
95      }
96  }
97
```

Fonte: O Autor (2021)

A execução dos comandos está diretamente ligada com a utilização dos *endpoints REST.*

Figura 17 - *Endpoints* de comandos



Fonte: O Autor (2021)

Conforme os comandos são enviados, são disparados eventos para que seja possível aplicar o outro *pattern* proposto no documento, o *Event Sourcing*.

Para manter o sincronismo das informações, os eventos automaticamente são lidos para gravar as informações no banco de dados relacional. É possível

visualizar os eventos no *Axon Dashboard*, a seguir o exemplo de como fica o registro de um evento para criação de viagem.

Figura 18 - Exemplo de evento

token	0
eventIdentifier	93434783-5d90-41b4-8930-44a663a14688
aggregateIdentifier	9c19d779-02cc-4fae-9995-41d3a5be3183
aggregateSequenceNumber	0
aggregateType	TripAggregate
payloadType	com.github.diegonsilveira.sample.events.TripCreatedEvent
payloadRevision	
payloadData	<com.github.diegonsilveira.sample.events.TripCreatedEvent> <tripId>9c19d779-02cc-4fae-9995-41d3a5be3183</tripId> <destination>Joinville</destination><distance>20</distance> </com.github.diegonsilveira.sample.events.TripCreatedEvent>
timestamp	2021-06-13T22:47:04.975Z

Fonte: O Autor (2021)

Na parte da aplicação responsável pela leitura das informações, os dados estão armazenados no banco *PostgreSQL*.

Além disso, foram publicados *endpoints* específicos para leitura e que permitem a criação de projeções para maior desempenho nas *queries.*

4 RESULTADOS E DISCUSSÕES

A seguir serão apresentados os resultados obtidos na execução de uma aplicação criada utilizando *CQRS* e *Event Sourcing*. A aplicação foi desenvolvida pelo próprio autor do trabalho e está disponível no Github: https://github.com/diegonsilveira/cqrs-eventsourcing-sample.

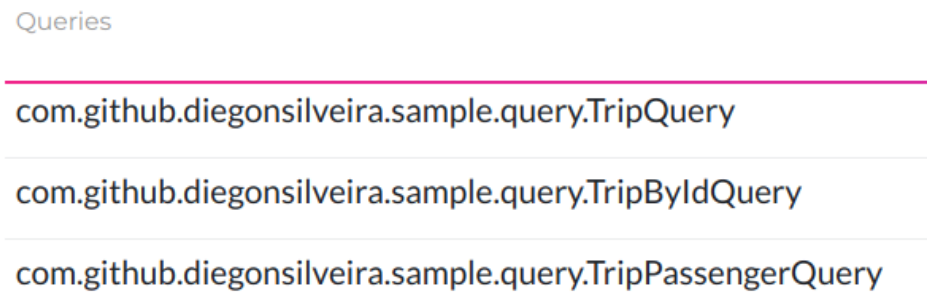
4.1 ANÁLISE DOS RESULTADOS OBTIDOS

A seguir alguns pontos levantados durante o projeto e que foram vistos ao criar uma aplicação de exemplo utilizando CQRS e *Event Sourcing*.

4.1.1 **Escalabilidade e desempenho**

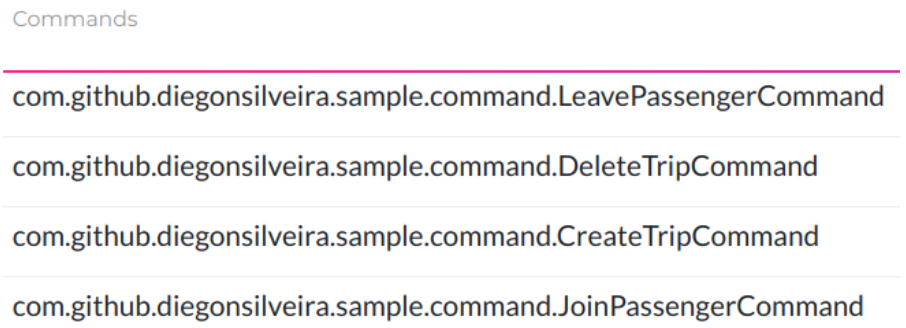
Uma das vantagens facilmente identificadas na aplicação é a possibilidade de escalonar individualmente a leitura e a gravação dos dados.
Como podemos ver, há uma separação entre *Commands* e *Queries* na aplicação.

Figura 19 - *Queries*



Fonte: O Autor (2021)

Figura 20 - *Commands*



Fonte: O Autor (2021)

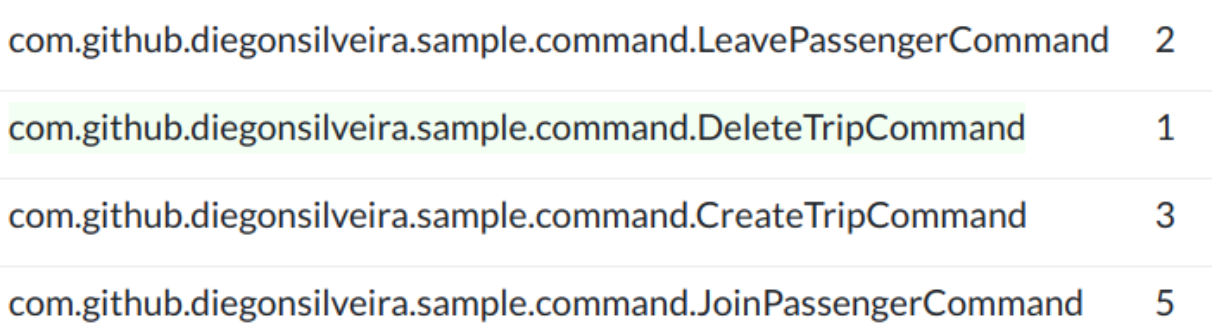
Por esse motivo é possível, por exemplo, manter várias instâncias da aplicação *Query* já que normalmente são os *endpoints* mais acessados nas aplicações.

Outro ponto a se ressaltar é a possibilidade de criar projeções específicas para melhorar o desempenho das buscas. Utilizando a aplicação, foram incluídos 10 mil registros que foram retornados pelo *endpoint getAll* em 1 segundo. Além disso, a busca por um único registro foi feita em menos de 1 segundo.

4.1.2 **Segurança e auditoria**

Todos os comandos e eventos são armazenados possibilitando que sejam auditados posteriormente. O exemplo a seguir apresenta a quantidade de comandos de inclusão e remoção foram realizados pela aplicação:

Figura 21 - Comandos executados



Fonte: O Autor (2021)

Além disso, é possível verificar a lista de eventos armazenados, conferir a ordem de execução e qual foi o objeto recebido.

Figura 22 - Eventos executados

token	eventIdentifier	aggregateIdent...	aggreg...	aggregateType	payloadType	payloa...	payloadData	timestamp	metaData
7	d382277a-3...	654ebdfd-3c...	4	TripAggregate	com.github.diegonsilveira.sa...		<com.github.diegonsilveira.sample.events....	2021-06-1...	{traceId=...
6	a9496b71-b...	654ebdfd-3c...	3	TripAggregate	com.github.diegonsilveira.sa...		<com.github.diegonsilveira.sample.events....	2021-06-1...	{traceId=...
5	c171748b-8...	654ebdfd-3c...	2	TripAggregate	com.github.diegonsilveira.sa...		<com.github.diegonsilveira.sample.events....	2021-06-1...	{traceId=...
4	5ecbfa9e-c5...	654ebdfd-3c...	1	TripAggregate	com.github.diegonsilveira.sa...		<com.github.diegonsilveira.sample.events....	2021-06-1...	{traceId=...
3	27390100-7...	654ebdfd-3c...	0	TripAggregate	com.github.diegonsilveira.sa...		<com.github.diegonsilveira.sample.events...	2021-06-1...	{traceId=...
2	a68c1d3d-0...	1e0ea426-5...	1	TripAggregate	com.github.diegonsilveira.sa...		<com.github.diegonsilveira.sample.events...	2021-06-1...	{traceId=...
1	380f9d28-6...	1e0ea426-5...	0	TripAggregate	com.github.diegonsilveira.sa...		<com.github.diegonsilveira.sample.events...	2021-06-1...	{traceId=...
0	00571c86-2...	38acb784-7...	0	TripAggregate	com.github.diegonsilveira.sa...		<com.github.diegonsilveira.sample.events...	2021-06-1...	{traceId=...

Fonte: O Autor (2021)

No primeiro evento do agregado “654ebdfd-3cc6-4c85-a0fc-56172f49bfe8” é realizada a criação de uma viagem conforme a seguir:

Figura 23 - Evento de criação

Name	Value
token	3
eventIdentifier	27390100-7d35-442b-81ae-a4a635de9173
aggregateIdentifier	654ebdfd-3cc6-4c85-a0fc-56172f49bfe8
aggregateSequenceNumber	0
aggregateType	TripAggregate
payloadType	com.github.diegonsilveira.sample.events.TripCreatedEvent
payloadRevision	
payloadData	<com.github.diegonsilveira.sample.events.TripCreatedEvent> <tripId>654ebdfd-3cc6-4c85-a0fc-56172f49bfe8</tripId> <destination>Joinville</destination><distance>2</distance> </com.github.diegonsilveira.sample.events.TripCreatedEvent>
timestamp	2021-06-14T01:11:08.326Z

Fonte: O Autor (2021)

Na imagem anterior, é possível verificar algumas informações sobre o registro adicionado, como o Identificador único do registro, quantos eventos foram enviados para aquele registro que no caso é 0 pois trata-se de uma inclusão. Também há a classe de agregado, no caso a "*TripAggregate",* o tipo de evento que originou o registro "*TripCreatedEvent"* e o conteúdo da alteração, o *payloadData*.

Analisando melhor o *payload* do registro, nota-se que são armazenadas as informações da classe que serve como repositório de agregados.

Figura 24 - *Payload* do evento

```
<com.github.diegonsilveira.sample.events.TripCreatedEvent>
        <tripId>654ebdfd-3cc6-4c85-a0fc-56172f49bfe8</tripId>
        <destination>Joinville</destination>
        <distance>2</distance>
</com.github.diegonsilveira.sample.events.TripCreatedEvent>
```

Fonte: O Autor (2021)

Sendo assim, cada registro de evento contém as informações que foram alteradas, incluídas ou removidas, bastante útil quando se espera identificar o estado e gerar um log para auditoria dos registros. Alguns exemplos:

Figura 25 - Evento de remoção

Name	Value
token	8
eventIdentifier	3dd986e3-0fea-4808-9072-0b965f363e93
aggregateIdentifier	654ebdfd-3cc6-4c85-a0fc-56172f49bfe8
aggregateSequenceNumber	5
aggregateType	TripAggregate
payloadType	com.github.diegonsilveira.sample.events.TripDeletedEvent
payloadRevision	
payloadData	<com.github.diegonsilveira.sample.events.TripDeletedEvent> <tripId>654ebdfd-3cc6-4c85-a0fc-56172f49bfe8</tripId> </com.github.diegonsilveira.sample.events.TripDeletedEvent>
timestamp	2021-06-14T01:30:35.709Z

Fonte: O Autor (2021)

4.1.3 **Complexidade de código**

Como já citado nas desvantagens do *pattern,* uma das consequências da utilização é o aumento na quantidade de classes e linhas de código para geração de comandos e eventos.

No exemplo abaixo, está a apresentação das classes utilizadas para a criação de um exemplo simples, contendo todos os comandos, eventos e classes responsáveis pela segregação proposta pelo CQRS.

Figura 26 - Implementação com CQRS e *Event Sourcing*

Fonte: O Autor (2021)

5 CONSIDERAÇÕES FINAIS

Conclui-se então que além de benefícios, também existem desvantagens na utilização dos *patterns* CQRS e *Event Sourcing* para desenvolvimento de microsserviços. Com a demonstração via exemplo, fica comprovado os ganhos relacionados a escalabilidade, desempenho e segurança das aplicações, porém, aumentando consideravelmente a complexidade do código.

A possibilidade de separar os microsserviços em aplicações que alteram os dados e aplicações que somente fazem a leitura, permite escalar de forma individual os serviços disponibilizados. Além disso, a criação de modelos e projeções para leitura, aumentam consideravelmente o desempenho das aplicações que realizam somente consultas.

Outro ponto importante é a geração das trilhas de eventos e a facilidade para implementar mecanismos de auditoria nas aplicações. O custo destes ganhos é visível à medida que o microsserviço aumenta e a quantidade de classes necessárias para implementação de *commands* e *events* cresce, como consequência, temos um código mais complexo que o normal.

Não existe somente um modo de aplicar CQRS e *Event Sourcing* em aplicações e apesar da utilização conjunta facilitar o desenvolvimento, ela não é obrigatória. Além disso, assim como em outras abordagens, é preciso entender as necessidades de cada aplicação antes de definir sua arquitetura.

Por fim, há muito material disponível sobre o assunto e apesar do conceito ser antigo, cada vez mais ganha espaço no desenvolvimento de microsserviços para o modelo nativo em nuvem.

6 REFERÊNCIAS

ALBERTONI, Fabio et al. **WebSphere Application Server V8. 5 Concepts, Planning, and Design Guide**. Konzeptanleitung: IBM, 2013.

AMAZON. **Modelos de computação em nuvem**. **AWS Amazon**. 2019. Disponível em: https://aws.amazon.com/pt/types-of-cloud-computing/. Acesso em: 4 Nov. 2019.

AMAZON. **O que é a computação em nuvem?**. **AWS Amazon**. 2019. Disponível em: https://aws.amazon.com/pt/what-is-cloud-computing/. Acesso em: 1 Nov. 2019.

AMAZON. **O que é NoSQL?**. **AWS Amazon**. 2019. Disponível em: https://aws.amazon.com/pt/nosql/. Acesso em: 1 Nov. 2019.

AMAZON. **O que é um banco de dados chave-valor?**. **AWS Amazon**. 2019. Disponível em: https://aws.amazon.com/pt/nosql/key-value/. Acesso em: 2 Nov. 2019.

ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi. **Fundamentos da programação de computadores**: Algoritmos, Pascal, C/C++ e Java. 3. ed. Pearson, 2012.

BAHSOON, Rami; EMMERICH, Wolfgang. **Evaluating software architectures: Development stability and evolution**. Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications: IEEE Computer Society Press, 2003.

BATTISTI, Júlio. **SQL server 2000: administração & desenvolvimento: curso completo**. Axcel Books, 2001.

BIEBERSTEIN, Norbert et al. **Service-oriented architecture compass: business value, planning, and enterprise roadmap**. FT Press, 2006.

CALDEIRA, Carlos Pampulim. **Introdução ao Modelo de Dados Relacional**. 2019. Disponível em: http://www.di.uevora.pt/~ccaldeira/e/sp/imd/docs/BD1.pdf. Acesso em: 14 Nov. 2019.

DATE, Christopher J.. **Introdução a sistemas de bancos de dados**. Elsevier Brasil, 2004.

DE SOUZA, Ítalo Andrade; PELISSARI, William Roberto. Microserviços como alternativa de arquitetura monolítica. **Revista de Pós-Graduação da Faculdade Cidade Verde**, v. 3, 2017.

DEITEL, Paul; DEITEL, Harvey. **JAVA: COMO PROGRAMAR**. 8. ed. São Paulo: Pearson Educación, 2010.

DEMEYER, Serge; MENS, Tom. **Software Evolution**. Springer, 2008.

DEVMEDIA. **Introdução ao Java Virtual Machine (JVM)**. **DevMedia**. 2019. Disponível em: https://www.devmedia.com.br/introducao-ao-java-virtual-machine-jvm/27624. Acesso em: 13 Nov. 2019.

DEVMEDIA. **Introdução ao Java Virtual Machine (JVM)**. **DevMedia**. 2019. Disponível em: https://www.devmedia.com.br/introducao-ao-java-virtual-machine-jvm/27624. Acesso em: 30 Out. 2019.

FALBO, Ricardo de Almeida; TRAVASSOS, Guilherme Horta. **A integração de conhecimento em um ambiente de desenvolvimento de software**. II Congreso Argentino de Ciencias de la Computación, 1996.

FERREIRA, Guilherme. **O que esconde o CQRS**. **Revista Programar**. 2016. Disponível em: https://www.revista-programar.info/artigos/o-que-esconde-o-cqrs/. Acesso em: 8 Nov. 2019.

FERREIRA, Lauren. **As diferenças entre SQL e NoSQL: MySQL x MongoDB**. **Medium**. 2018. Disponível em: https://medium.com/devtranslate/diferencas-entre-sql-e-nosql-51311f9069bd. Acesso em: 28 Out. 2019.

FOWLER, Martin. **CQRS**. 2011. Disponível em: https://www.martinfowler.com/bliki/CQRS.html. Acesso em: 14 Nov. 2019.

FOWLER, Martin. **Event Sourcing**. 2005. Disponível em: https://martinfowler.com/eaaDev/EventSourcing.html. Acesso em: 16 Nov. 2019.

GARLAN, David; SHAW, Mary. **An introduction to software architecture**: Advances in software engineering and knowledge engineering. 1993, p. 1-39.

GERTEL, Lucas. **Padrões de Arquitetura Web — Monolítica ou Micro Serviços?**. **Medium**. 2019. Disponível em: https://medium.com/@lgertel/padrões-de-arquitetura-web-monolítica-ou-micro-serviços-7b3f0c9394fe. Acesso em: 16 Nov. 2019.

IBM. **Entenda os modelos de serviço IaaS, PaaS e SaaS do IBM Cloud**. 2019. Disponível em: https://www.ibm.com/br-pt/cloud/learn/iaas-paas-saas. Acesso em: 30 Out. 2019.

ISTQB. **Certified Tester Foundation Level Syllabus**. 2018. Disponível em: https://www.bstqb.org.br/uploads/syllabus/syllabus_ctfl_2018br.pdf. Acesso em: 30 Out. 2019.

KAMBALYAL, Channu. **3-tier architecture**. Retrieved On, v. 2, 2010.

KORTH, Henry F.; SILBERSCHATZ, Abraham. **Sistemas de Bancos de Dados**. São Paulo: Markron Books, 1995.

MACHADO, Marcio P.; SOUZA, Sotério F.. **Métricas e qualidade de software**. Departamento de Informática – Universidade Federal do Espírito Santo, 2004.

MANZANO, José Augusto N.G.; OLIVEIRA, Jayr Figueiredo. **Algoritmos lógica para desenvolvimento de programação de computadores**. Saraiva, 2010.

MARK, Richards. **Software Architecture Patterns-Understanding Common Architecture Patterns and When to Use Them**. 2015.

MARZULLO, Fabio Perez. **SOA na Prática**. Novatec Editora, 2009.

MEIER, J. D. et al. **Application Architecture Guide 2.0**: patterns & practices. 2008.

MICROSERVICES. **What are microservices?**. **Microservices**. 2019. Disponível em: https://microservices.io/. Acesso em: 28 Out. 2019.

MICROSOFT. **O que é Cloud?**. **Microsoft Azure**. 2019. Disponível em: https://azure.microsoft.com/pt-pt/overview/what-is-the-cloud/. Acesso em: 1 Nov. 2019.

MICROSOFT. **O que é IaaS**: Infraestrutura como serviço. **Microsoft Azure**. 2019. Disponível em: https://azure.microsoft.com/pt-br/overview/what-is-iaas/. Acesso em: 29 Out. 2019.

MICROSOFT. **O que é o SaaS**: Software como serviço. **Microsoft Azure**. 2019. Disponível em: https://azure.microsoft.com/pt-br/overview/what-is-saas/. Acesso em: 31 Out. 2019.

MICROSOFT. **O que é PaaS**: Plataforma como serviço. **Microsoft Azure**. 2019. Disponível em: https://azure.microsoft.com/pt-br/overview/what-is-paas/. Acesso em: 30 Out. 2019.

MICROSOFT. **Padrão CQRS**: Segregação de responsabilidade de consulta e comando. **Microsoft**. Disponível em: https://docs.microsoft.com/pt-br/azure/architecture/patterns/cqrs. Acesso em: 27 Out. 2019.

MONGE, Wellington. **BANCO DE DADOS - NoSQL**. **Medium**. 2018. Disponível em: https://medium.com/técnicas-de-modelagem-de-dados-nosql/banco-de-dados-nosql-58726ce6886c. Acesso em: 15 Nov. 2019.

MOVILE. **CQRS + Event Sourcing – Vantagens de uma Arquitetura Orientada a Eventos**. **Movile**. 2019. Disponível em: https://movile.blog/cqrs-event-sourcing-vantagens-de-uma-arquitetura-orientada-a-eventos/. Acesso em: 15 Nov. 2019.

NASCIMENTO, Wellington. **CQS — Command Query Separation**. 2019. Disponível em: https://medium.com/tableless/cqs-command-query-separation-4085ec41e3a4. Acesso em: 17 Nov. 2019.

ORACLE. **Glossário de Conceitos e Definições úteis**. 2019. Disponível em: https://www.java.com/pt_BR/download/faq/helpful_concepts.xml. Acesso em: 3 Out. 2019.

ORACLE. **O que é a tecnologia Java e por que preciso dela?**. **Oracle**. 2019. Disponível em: https://www.java.com/pt_BR/download/faq/whatis_java.xml. Acesso em: 29 Out. 2019.

PASSOS, Clayton K. N.. **CQRS — Command Query Responsability Segregation**. **Medium**. 2018. Disponível em: https://medium.com/codigorefinado/cqrs-command-query-responsability-segregation-98848c274233. Acesso em: 30 Out. 2019.

PIRES, Eduardo. **CQRS o que é e onde aplicar**. 2016. Disponível em: https://www.eduardopires.net.br/2016/07/cqrs-o-que-e-onde-aplicar/. Acesso em: 15 Nov. 2019.

PORTIER, Bertrand. **Service, architecture, governance, and business terms**. 2006. Disponível em: https://www.ibm.com/developerworks/library/ws-soa-term1/. Acesso em: 15 Nov. 2019.

PUGA, Sandra; RISSETTI, Gerson. **Lógica de programação e estruturas de dados, com aplicações em Java**. Pearson Educación, 2008.

REDHAT. **O que são os microsserviços?**. **RedHat**. 2019. Disponível em: https://www.redhat.com/pt-br/topics/microservices. Acesso em: 27 Out. 2019.

RICHARDSON, Chris. **Microservices: Decomposing Applications for Deployability and Scalability**. **InfoQ**. 2014. Disponível em: https://www.infoq.com/articles/microservices-intro/. Acesso em: 4 Out. 2019.

SANTOS, Lucas. **Event Sourcing - A arquitetura que pode salvar seu projeto**. **iMasters**. 2019. Disponível em: https://imasters.com.br/banco-de-dados/event-sourcing-arquitetura-que-pode-salvar-seu-projeto. Acesso em: 13 Nov. 2019.

SCHMUTZ, Guido; LIEBHART, Daniel; WELKENBACH, Peter. **Service-oriented Architecture**: An Integration Blueprint a Real-world SOA Strategy for the Integration of Heterogeneous Enterprise Systems. Packt Publishing Ltd, 2010.

SILVA, Flávio Soares Corrêa; MELO, Ana Cristina Vieira. **Princípios de linguagem de programação**. 3. ed. São Paulo: Blucher, 2014.

SOMMERVILLE, Ian. **Software engineering 9th Edition**. 9. ed. ISBN-10137035152, 2011.

SPECTOR, Alfred Z. et al. **The camelot project**. 1986.

TRINUGROHO, Yohanes Baptista Dafferianto; REICHERT, Frank; FENSLI, Rune . **A SOA-based health service platform in smart home environment**. 2011 IEEE 13th International Conference on e-Health Networking, 2011.

UFCG. **Introdução e Motivação: Arquiteturas em n Camadas**. **UFCG**. Campina Grande. Disponível em: http://www.dsc.ufcg.edu.br/~jacques/cursos/j2ee/html/intro/intro.htm. Acesso em: 17 Nov. 2019.

VASKEVITCH, David. **Estratégias cliente/servidor**. Berkeley Brasil Editora, 1995.

VERAS, Manoel. **Computação em Nuvem**. Brasport Livros e Multimídia Ltda, 2015.

WIGGINS, Adam. **The Twelve-Factor App**. 2011. Disponível em: https://12factor.net/. Acesso em: 17 Nov. 2019.

WILLRICH, Roberto. **Introdução à Informática**. **Florianópolis**. UFSC, 2000. Disponível em: http://www.inf.ufsc.br/~roberto.willrich/Ensino/INE5602/. Acesso em: 22 Out. 2019.

YOUNG, Greg. **CQRS Documents by Greg Young**. 2010. Disponível em: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf. Acesso em: 1 Nov. 2019.

YOUNG, Greg. **CQRS, Task Based UIs, Event Sourcing agh!**. 2010. Disponível em: http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/. Acesso em: 16 Nov. 2019.