**CQRS – O que é? Onde aplicar?**

## CQRS é uma daquelas siglas que está cada vez mais presente em nossas leituras, muitas vezes encontramos o CQRS sendo citado em conteúdos sobre DDD ou padrões de arquitetura escaláveis.

CQRS é um conceito muito importante e você precisa conhecer. Eu costumo dizer que todo arquiteto possui uma “caixa de ferramentas” e o CQRS é o tipo de ferramenta que precisa estar presente na sua caixa.

**O que é CQRS?**

CQRS significa Command Query Responsibility Segregation. Como o nome já diz, é sobre separar a responsabilidade de escrita e leitura de seus dados.

CQRS é um **pattern**, um padrão arquitetural assim como Event Sourcing, Transaction Script e etc. O CQRS não é um estilo arquitetural como desenvolvimento em camadas, modelo client-server, REST e etc.

**Onde posso aplicar o CQRS?**

Antes de entrar neste ponto vamos entender os cenários clássicos do dia a dia e depois veremos como o CQRS poderia ser aplicado como solução.

Hoje em dia não desenvolvemos mais aplicações para 10 usuários simultâneos, a maioria das novas aplicações nascem com premissas de escalabilidade, performance e disponibilidade. Como uma aplicação pode funcionar bem com 10 ou 10.000 usuários simultaneamente? É uma tarefa complexa criar um modelo conceitual que atenda essas necessidades.

Imagine um sistema de SAC onde diversos atendentes num call-center consultam e modificam as informações de um cadastro de clientes enquanto outra área operacional da empresa também trabalha com os mesmos dados simultaneamente. Os dados do cliente são modificados constantemente e nenhuma das áreas tem tempo e paciência para esperar os possíveis “locks” da aplicação, o cliente quer ser atendido com agilidade.

No mesmo cenário a aplicação pode possuir picos diários ou sazonais de acessos. Como impedir o tal “gargalo” e como manter a disponibilidade da aplicação em qualquer situação?

– Ah! Vamos escalar nossa aplicação em N servidores. Podemos migrar para a nuvem (cloud-computing ex. [Azure](https://azure.microsoft.com/)) e criar um script de elasticidade ([Autoscaling](https://azure.microsoft.com/en-us/documentation/articles/best-practices-auto-scaling/" \t "_blank)) para escalar conforme a demanda.

O conceito de escalabilidade da aplicação vai resolver alguns problemas de disponibilidade como, por exemplo, suportar muitos usuários simultaneamente sem comprometer a performance da aplicação.

Mas será que só escalar os servidores de aplicação resolve todos os nossos problemas?

**Problema # 1**

Deadlocks, timeouts e lentidão, seu banco pode estar em chamas.

Escalar a aplicação não é uma garantia de que a aplicação vai estar sempre disponível. Não podemos esquecer que neste suposto cenário todo processo depende também da disponibilidade do banco de dados.

Escalar o banco de dados pode ser muito mais complexo (e caro) do que escalar servidores de aplicação. E geralmente é devido o consumo do banco de dados que as aplicações apresentam problemas de performance.

**Problema # 2**

Para se obter um dado muitas vezes é necessário passar por um conjunto complexo de regras de negócio que irá filtrar a informação antes dela ser exibida, além disso existem os ORM’s que mapeiam o banco de dados em objetos de domínio realizando consultas com joins em diferentes tabelas para retornar todo conjunto de dados necessários.  
Tudo isto custa um tempo precioso até que o usuário receba a informação esperada.

**Problema # 3**

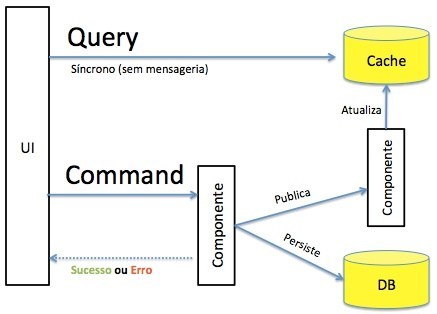
Um conjunto limitado de dados é consultado e alterado constantemente por uma grande quantidade de usuários simultaneamente conectados. Um dado exibido na tela já pode ter sido alterado por outro. Numa visão realista é possível afirmar que toda informação exibida já pode estar obsoleta.

## Ponto de partida

Ter N servidores consumindo um único banco de dados que serve de leitura e gravação pode ocasionar muitos locks nos dados e com isso ocasionar diversos problemas de performance, assim como todo o processo da regra de negócio que vai obter os dados de exibição cobra um tempo a mais no processamento. No final ainda temos que considerar que o dado exibido já pode estar desatualizado.

É este o ponto de partida do CQRS. Já que uma informação exibida não é necessariamente a informação atual então a obtenção deste dado para a exibição não necessita ter sua performance afetada devido a gravação, possíveis locks ou disponibilidade do banco.

O CQRS prega a divisão de responsabilidade de gravação e escrita de forma conceitual e física. Isto significa que além de ter meios separados para gravar e obter um dado os bancos de dados também são diferentes. As consultas são feitas de forma síncrona em uma base desnormalizada separada e as gravações de forma assíncrona em um banco normalizado.



Este é um fluxo simplificado do CQRS que não leva em consideração as camadas de aplicação, domínio e infra, comandos / eventos e enfileiramento de mensagens.

O CQRS não é um padrão arquitetural de alto-nível, podemos entender como uma forma de componentizar parte de sua aplicação. Podemos entender então que a utilização do CQRS não precisa estar presente em todos os processos de sua aplicação. Numa modelagem baseada em DDD um [Bounded Context](https://www.eduardopires.net.br/2016/03/ddd-bounded-context/" \t "_blank) pode implementar o CQRS enquanto os demais não.

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. Independente de como for implementado o CQRS sempre acarreta numa complexidade extra e por isso é necessário avaliar os cenários em que realmente são necessários trabalhar com este padrão.

## Entendendo melhor o CQRS

A ideia básica é segregar as responsabilidades da aplicação em:

* Command – Operações que modificam o estado dos dados na aplicação.
* Query – Operações que recuperam informações dos dados na aplicação.

Numa arquitetura de N camadas poderíamos pensar em separar as responsabilidades em CommandStack e QueryStack.

**QueryStack**

A QueryStack é muito mais simples que a CommandStack, afinal a responsabilidade dela é recuperar dados praticamente prontos para exibição. Podemos entender que a QueryStack é uma camada síncrona que recupera os dados de um banco de leitura desnormalizado.

Este banco desnormalizado pode ser um NoSQL como [MongoDB](https://www.mongodb.com/" \t "_blank), [Redis](http://redis.io/), [RavenDB](https://ravendb.net/" \t "_blank) etc.  
O conceito de desnormalizado pode ser aplicado com “one table per view” ou seja uma consulta “flat” que retorna todos os dados necessários para ser exibido em uma view (tela) específica.

O uso de consultas “flats” em um banco desnormalizado evita a necessidade de joins, tornando as consultas muito mais rápidas. É preciso aceitar que haverá a duplicidade de dados para poder atender este modelo.

**CommandStack**

O CommandStack por sua vez é potencialmente assíncrono. É nesta separação que estão as entidades, regras de negócio, processos e etc. Numa abordagem DDD podemos entender que o Domínio pertence a esta parte da aplicação.

O CommandStack segue uma abordagem *behavior-centric* onde toda intenção de negócio é inicialmente disparada pela UI como um caso de uso. Utilizamos o conceito de Commands para representar uma intenção de negócio. Os Commands são declarados de forma imperativa (ex. FinalizarCompraCommand) e são disparados assincronamente no formato de eventos, são interpretados pelos CommandHandlers e retornam um evento de sucesso ou falha.

Toda vez que um Command é disparado e altera o estado de uma entidade no banco de gravação um processo tem que ser disparado para os agentes que irão atualizar os dados necessários no banco de leitura.

**Sincronização**

Existem algumas estratégias para manter as bases de leitura e gravação sincronizadas é necessário escolher a que melhor atende ao seu cenário:

Atualização automática – Toda alteração de estado de um dado no banco de gravação dispara um processo síncrono para atualização no banco de leitura.

Atualização eventual – Toda alteração de estado de um dado no banco de gravação dispara um processo assíncrono para atualização no banco de leitura oferecendo uma consistência eventual dos dados.

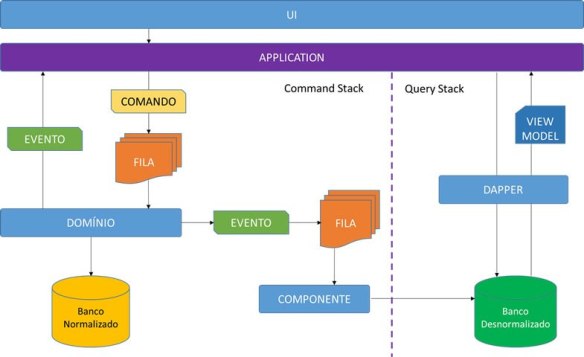
Atualização controlada – Um processo periódico e agendado é disparado para sincronizar as bases.

Atualização sob demanda – Cada consulta verifica a consistência da base de leitura em comparação com a de gravação e força uma atualização caso esteja desatualizada.

A atualização eventual é uma das estratégias mais utilizadas, pois parte do princípio que todo dado exibido já pode estar desatualizado, portanto não é necessário impor um processo síncrono de atualização.

**Enfileiramento**

Muitas implementações de CQRS podem exigir um “Bus” para processamento de Commands e Events. Nesse caso teremos uma implementação conforme a seguinte ilustração.



Existem diversas opções de Bus para .NET

* [Microsoft Azure Service Bus Queue](https://azure.microsoft.com/en-us/documentation/articles/service-bus-dotnet-get-started-with-queues/)
* [NServiceBus](http://particular.net/nservicebus)
* [Rebus](https://github.com/rebus-org/Rebus) (Free)
* [MassTransit](https://github.com/MassTransit/MassTransit) (Free)

## ****Vantagens de implementar o CQRS****

A implementação do CQRS quebra o conceito monolítico clássico de uma implementação de arquitetura em N camadas onde todo o processo de escrita e leitura passa pelas mesma camadas e concorre entre si no processamento de regras de negócio e uso de banco de dados.

Este tipo de abordagem aumenta a disponibilidade e escalabilidade da aplicação e a melhoria na performance surge principalmente pelos aspectos:

* Todos comandos são assíncronos e processados em fila, assim diminui-se o tempo de espera.
* Os processos que envolvem regras de negócio existem apenas no sentido da inclusão ou alteração do estado das informações.
* As consultas na QueryStack são feitas de forma separada e independente e não dependem do processamento da CommandStack.
* É possível escalar separadamente os processos da CommandStack e da QueryStack.

Uma outra vantagem de utilizar o CQRS é que toda representação do seu domínio será mais expressiva e reforçará a utilização da linguagem ubíqua nas intenções de negócio.

Toda a implementação do CQRS pattern pode ser feito manualmente, sendo necessário escrever diversos tipos de classes para cada aspecto, porém é possível encontrar alguns frameworks de CQRS que vão facilitar um pouco a implementação e reduzir o tempo de codificação.

Apesar da minha preferência ser sempre codificar tudo por conta própria eu encontrei alguns frameworks bem interessantes que servem inclusive para estudo e melhoria do entendimento no assunto.

* [Lokad-CQRS](https://github.com/lokad/lokad-cqrs/)
* [NCQRS](https://github.com/pjvds/ncqrs)
* [CQRS Lite](https://github.com/gautema/CQRSlite)

## Mitos sobre o CQRS

**#1 Mito – CQRS e Event Sourcing devem ser implementados juntos.**

O Event Sourcing é um outro pattern assim como o CQRS. É uma abordagem que nos permite guardar todos os estados assumidos por uma uma entidade desde sua criação. O Event Sourcing tem uma forte ligação com o CQRS e é facilmente implementado uma vez que temos também o CQRS, porém é possível implementar Event Sourcing independente do CQRS e vice-versa.

Escreverei sobre Event Sourcing em breve num outro artigo.

**#2 Mito – CQRS requer consistência eventual**

Negativo. Como abordado anteriormente o CQRS pode trabalhar com uma consistência imediata e síncrona.

**#3 Mito – CQRS depende de Fila/Bus/Queues**

CQRS é dividir as responsabilidades de Queries e Commands, a necessidade de enfileiramento vai surgir dependendo de sua implementação, principalmente se for utilizar a estratégia de consistência eventual.

**#4 Mito – CQRS é fácil**

Não é fácil. O CQRS também não é uma ciência de foguetes. A implementação vai exigir uma complexidade extra em sua aplicação além de um claro entendimento do domínio e da linguagem ubíqua.

**#5 Mito – CQRS é arquitetura**

Não é! Conforme foi abordado o CQRS é um pattern arquitetural e pode ser implementado em uma parte específica da sua aplicação para um determinado conjunto de dados apenas.