Instituto Superior de Engenharia do Porto

Padrão CQRS para Sistemas Distribuídos de Larga Escala

Carlos Miguel Brites da Silva Ferreira

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Arquiteturas, Sistemas e Redes

Orientador: Doutor Paulo Alexandre Gandra de Sousa

Júri:

Presidente:

Doutor Luís Miguel Moreira Lino Ferreira, ISEP

Vogais:

Eng.º António José Rocha de Oliveira, ISEP

Doutor Paulo Alexandre Gandra de Sousa, ISEP

Porto, Outubro, 2012

Resumo

Esta tese pretende desenvolver o estudo de um padrão que utiliza um modelo de implementação fundamentado na natureza das operações que um sistema pretende executar. Estas operações são distinguidas pelo que realizam, portanto um sistema poderá ser dividido em duas grandes áreas: uma, de leitura de dados, e outra, de manipulação de dados. A maior parte dos sistemas atuais está a progredir, com o objetivo de conseguir suportar muitos utilizadores em simultâneo, e é neste aspeto que este padrão se diferencia porque vai permitir escalar, com muita facilidade e sem sobrecarga. Além disso, este estudo deverá facultar um conjunto de boas práticas e incidir sobre o facto de se pretender desenhar um sistema de raiz e não apenas em "migrar" de um sistema já existente.

Ao estudar este padrão é essencial estudar e analisar a evolução da utilização futura dos sistemas, para determinar a utilidade e a aplicação crescente ou não, deste padrão. Interessa também saber, quem implementa atualmente este padrão, em que tipo de produtos, e enaltecer o seu sucesso de implementação, estimulando o desenvolvimento da sua utilização.

Finalmente, demonstra-se a aplicabilidade e validade do padrão proposto, através de uma implementação modelo, com a ajuda de uma *framework* de forma a determinar quais as ferramentas existentes que possam ser úteis e contribuir para a implementação deste padrão. O objetivo final será demonstrar os principais componentes do sistema, como poderá prosseguir a sua evolução e como poderá ser melhorada e simplificada a comunicação entre os seus componentes, para uma utilização mais fácil, frequente e de interesse comum para todos: utilizadores e administradores.

Palavras-chave: CQRS, Padrões, Eventos, *Messaging*, *Frameworks*, Escala;

Abstract

This thesis aims to develop the study of a standard that uses an implementation model based on the nature of the operations wanted to be carried out by a system. These operations are distinguished by what they do in the system, therefore, a system may be divided into two major areas: one for data reading and another for data manipulation. The largest part of the systems today is progressing towards supporting multi users simultaneously and it is this aspect that distinguishes this pattern, since it will allow helping to scale easily and without overload. Furthermore, this study will provide a set of good practices and focus on the purpose of designing a system from scratch and not just "migrate" from an existing one.

When studying this pattern is also essential to study and analyze the evolution of the future utilization of the systems, to determine the growing or not, use and application of this pattern. It is also important to find out who is currently implementing this pattern, in which kind of products and praise its successful implementation, thus promoting the propagation of its utilization.

Finally, the validity and applicability of the proposed standard is demonstrated by an implementation model, with the help of a framework in order to determine the existent tools that can be useful and contribute to the implementation of this standard. The ultimate goal will be to demonstrate the main components of the system, how it might evolve and how the communication between its components may be improved e simplified for an easier and more frequent utilization of common interest for all: users and administrators.

Keywords: CQRS, Patterns, Events, Messaging, Frameworks, Scale;

Agradecimentos

“O único lugar onde o sucesso vem antes do trabalho é no dicionário” (Albert Einstein).

Esta tese, para além de todo o trabalho inerente pela pesquisa, pelo estudo, pela análise, importantes para o seu desenvolvimento, exigiu sobretudo um grande esforço de disponibilidade e de abdicação social e familiar. Por isso, não posso deixar de expressar aqui, os meus sinceros e sentidos agradecimentos, pelo apoio e compreensão que sempre me manifestaram, com todo o seu afeto, família e amigos.

As pessoas mais afetadas pela minha indisponibilidade para as acompanhar e dar atenção, no que seria razoável, porque a família será sempre a nossa base de estabilidade enquanto seres humanos, foram a minha mãe, a minha tia e a minha namorada. Para elas dedico com grande orgulho e um profundo e sentido afeto, uma palavra de agradecimento, pela tolerância, paciência, compreensão, e grande estímulo que desde o início me manifestaram.

Em todo este trabalho absorvente de pesquisa e estudo, pude contar com o apoio, a orientação, os conselhos e opiniões, numa disponibilidade paciente e constante, do Doutor Paulo Alexandre Gandra de Sousa, a quem dedico uma palavra muito especial e sentida de agradecimento.

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Símbolos

Lista de Acrónimos

ABC	*Autonomous Business Component*
AC	*Autonomous Components*
ACID	Atomicidade, Consistência, Isolamento e Durabilidade
ACK	*Acknowledgement*
API	*Application programming interface*
AR	*Aggregate Root*
CQRS	*Command Query Responsibility Segregation*
CQS	*Command-query separation*
CRUD	*Create Read Update Delete*
DDD	*Domain-driven design*
DTO	*Data Transfer Object*
EPC	*Electronic Product Code*
ESB	*Enterprise Service Bus*
FInES	*Future Internet Enterprise Systems*
FTP	*File Transfer Protocol*
IaaS	*Infrastructure as a Service*
ICT	*Information Communications Tecnology*
ISBN	*International Standard Book Number*
ISU	*Interoperability Service Utility*
JSON	*JavaScript Object Notation*
JSP	*JavaServer Pages*
JVM	*Java virtual machine*

MSMQ	*Microsoft Message Queuing*
MVCC	*Multiversion concurrency control*
NFC	*Near field communication*
NoSQL	*Not Only SQL*
ORM	*Object-relational mapping*
P2P	*Peer-to-peer*
PaaS	*Platform as a Service*
POJO	*Plain Old Java Object*
QOS	*Quality-of-service*
RDBMS	*Relational database management system*
REST	*Representational State Transfer*
RFID	*Radio-frequency identification*
RPC	*Remote procedure call*
SaaS	*Software as a Service*
SLA	*Service-Level Agreement*
SOA	*Service-oriented architecture*
SoS	*System of Systems*
SQL	*Structured Query Language*
SRP	*Single Responsibiliy Principle*
UI	*User Interface*
ULS	*Ultra Large Systems*
WCF	*Windows Communication Foundation*

1 Introdução

1.1 Enunciado do Problema

Os sistemas computacionais tendem cada vez mais a serem distribuídos em numericamente e geograficamente, aliando a sua maior potência à sua utilização em larga escala. Muitos destes sistemas utilizados por grandes empresas têm como objetivo primordial conseguirem o máximo de disponibilidade, devido a fornecerem serviços aos seus clientes, e se o conseguirem poderão obter também o máximo de confiança destes. Além disso, internamente para as empresas, torna-se importante que a sua própria arquitetura possa estar dividida em pequenas partes de forma a ser mais fácil de gerir e manter. Geralmente ao efetuar a divisão consegue-se dividir e isolar do resto do sistema, o que mais uma vez tende a que exista mais disponibilidade do próprio sistema e aumento de confiança.

De forma a combater os problemas existentes nestes sistemas torna-se fundamental o conhecimento das suas características, como por exemplo, a independência e descentralização dos seus componentes, a distribuição geográfica, o aumento de falhas e cada vez mais a heterogeneidade dos seus elementos. O fator preponderante de um sistema se expandir em grande escala, é que a escala muda tudo. Estas grandes diferenças e mudanças necessitam que sejam feitos refinamentos quer a nível de manutenção de sistemas quer também do seu próprio *design*. Problemas que em sistemas de pequena dimensão eram insignificantes, em larga escala tornam-se gigantescos e podem levar ao fracasso desse sistema, o que reforça ainda mais a necessidade de conhecer e estudar os refinamentos que são necessários realizar. Estas novas formas de idealizar um sistema levam a que primeiramente seja necessário estudar quais serão os seus novos requisitos e para onde é que a tendência futura nos orienta (Carnegie Mellon University, 2006).

Hoje em dia a larga escala é um dos temas mais estudados, juntamente com os estudos de padrões, e pretende ajudar a colmatar problemas ou parte deles, existentes nestes sistemas. Estes padrões por sua vez, permitem fornecer boas práticas e assim um guia de desenvolvimento e manutenção desde início. Desta forma é possível dar respostas a problemas como o balanceamento da carga e a melhor forma de o fazer, a grande quantidade de utilizadores que tem de suportar e a grande diferença na natureza de operações que existe,

ou seja, a quantidade de pedidos de leitura tende a ser muito superior em comparação com a manipulação de dados.

Em relação aos grandes problemas enunciados, existe um padrão denominado *Command Query Responsibility Segregation* (CQRS), que é caracterizado por separar a arquitetura em dois grandes modelos: um, de leitura, denominado de *queries* e outro, de escrita, denominado *commands*. Com esta separação, é possível fornecer a estes sistemas de larga escala uma forma de conseguirem equilibrar a carga, visto que podem colocar mais processamento num modelo do que em outro. Podem também suportar grandes quantidades de utilizadores porque permitem que o modelo de leitura seja dedicado a responder o mais rapidamente possível a todas as exigências de utilização; e por fim permite delimitar os papéis e responsabilidades que cada modelo tem no sistema.

Os sistemas em larga escala cada vez mais são colocados em ambiente de *cloud computing*, o que acrescenta assim maiores desafios quanto á sua implementação mas também permite tirar partido de bastantes benefícios tais como a flexibilidade de crescimento e o controlo de concorrência, possibilitando o estudo do uso de CQRS neste tipo de ambientes. Outra mudança que o CQRS poderá trazer é a forma como as aplicações são distribuídas. Ao dividir a arquitetura em dois modelos, torna-os independentes, e assim é possível atualizar separadamente cada um deles sem comprometer a disponibilidade do sistema. (Abdullin, 2012).

Com o estudo deste padrão realizado nesta tese, é essencial salientar que não se trata apenas de boas práticas para a necessidade de refinar as arquiteturas atuais e para que sejam mais objetivas nas suas ações, mas também orientar desde a sua criação. Para esta abordagem e posterior implementação é aconselhado o uso de várias ferramentas, como *framework*s, para que o trabalho inicial seja mais facilitado e acima de tudo garantir que o padrão é cumprido.

1.2 Objetivos

Os objetivos desta tese baseiam-se no estudo e implementação de um padrão denominado CQRS, o qual poderá ser aplicado a sistemas de larga escala, no qual se pretende também demonstrar uma prova de conceito deste padrão, onde seja possível verificar como é que se poderá aplicar este conceito e ter uma melhor noção da facilidade de implementação. Além deste fator, é necessário demonstrar que problemas é que este padrão vem colmatar e de que forma ele se mantem alinhado com o futuro destes grandes sistemas. É necessário referir que poderá ser implementado numa quantidade grande de máquinas, permitindo aumentar os recursos sem ter que se quebrar ligações, alterar todos os clientes, etc., permitindo separar cada componente autonomamente. Para além do estudo do padrão, pretende-se também evidenciar como é que este poderá tirar benefícios em *cloud computing.* Exemplo: *CQRS* e *NoSQL (Not Only SQL)*. Quanto à demonstração do padrão CQRS é utilizado uma *framework*, em que se pretende evidenciar e dar um exemplo de como é que se pode construir uma aplicação, quais os componentes e respetivas classes que devem conter.

No fim do trabalho da tese espera-se ter a seguinte lista de objetivos cumpridos:

- Estudo das condições empresariais de larga escala;
- Estudo do padrão CQRS para utilização em larga escala;
- Estudo de *framework*s CQRS existentes;
- Aplicação exemplo.

No final da tese, depois de estudado o padrão CQRS e após conceção duma prova de conceito, pretende-se realçar os factos essenciais para sistemas distribuídos de larga escala, como:

- O estudo das tendências que se avizinham e saber como as utilizar no nosso sistema prevendo as mudanças necessárias.
- O estudo de uma nova arquitetura que possa ser aplicada a um sistema de larga escala ou a uma parte dele;
- A facilidade de manutenção, saber como manter;
- A demonstração da existência de casos de sucesso no mercado;
- A apresentação de ferramentas úteis para a implementação deste padrão;
- A demonstração da influência que este padrão traz para a interface do utilizador;
- A facilidade de auditoria aos dados do sistema;

Em complemento aos anteriores pontos, no fim pretende-se concluir e dar respostas a questões como:

- Será que é necessário ter sempre a informação atualizada no cliente?
- Como é possível manter a coerência dos dados?
- No caso de resposta negativa nos dois pontos anteriores, como a atualizar futuramente e com que mecanismos?
- De que forma é que os componentes comunicam?
- Que alterações trará, nas arquiteturas, a implementação do CQRS?
- Qualquer empresa deve adotar este padrão?
- Onde é que este padrão já é utilizado? Tem sucesso?

1.3 Método de trabalho

A forma de elaboração desta tese foi iniciada com um plano de trabalhos, em que se decidiu que tópicos é que seriam mais necessários e mais importantes de desenvolver, criando uma ordem de ideias para que seja possível entender em que ambientes se enquadra este padrão, quais são os principais interessados e que vantagens são adquiridas. Ao longo da sua elaboração foi-se refinando o que seria realmente importante de transmitir, como definir em que se irão basear e quais os sistemas de futuro; em que consiste em detalhe o padrão CQRS, e como o aplicar; que situações existem de sucesso. No final da tese foi elaborada uma prova de conceito com vista a exemplificar as ideias e os conceitos falados sobre este padrão, podendo também servir de início para projetos e investigações futuras.

1.4 Organização da tese

A tese é composta por sete capítulos da seguinte forma:

- O primeiro capítulo é uma introdução em que é descrito sucintamente os pontos essenciais, como por exemplo: qual é o problema que se pretende resolver, quais os objetivos finais, e qual a sua organização;

- No segundo capítulo esclarece-se o que é a larga escala e porquê que o padrão CQRS se aplica a ele, juntamente com a definição de "*Ultra Large Systems*" (ULS). É posto em realce o que é que o futuro nos reserva e quais as suas tendências de forma a se entender a razão do estudo deste padrão num ambiente empresarial;

- No terceiro capítulo pretende-se dar a conhecer o padrão em detalhe, fornecendo os conceitos inerentes á sua aplicação, a ligação com outros padrões e benefícios na sua utilização;

- O quarto capítulo pretende dar a conhecer alguns casos de sucesso da aplicação deste padrão fornecendo assim testemunhos da sua aplicação;

- O quinto capítulo é um estudo sobre quais as ferramentas, nomeadamente *framework*s que existem atualmente no mercado para ajudar a aplicar o padrão CQRS e é feita uma análise que permite saber as vantagens e desvantagens de cada uma;

- No sexto capítulo é descrita toda a prova de conceito desenvolvida para esta tese, que servirá de exemplo da aplicação deste padrão;

- No sétimo capítulo apresentam-se as conclusões, baseada nos objetivos cumpridos, limitações e trabalho futuro;

- No oitavo capítulo encontra-se a bibliografia utilizada;

- Por fim, em complemento aos capítulos anteriores, existe o primeiro anexo que pretende dar informações sobre o sistema de base de dados utilizada na prova de conceito.

2 Sistemas distribuídos de larga escala

2.1 O que é escala

Ao longo dos últimos anos a escala é cada vez mais conhecida e mais necessária nos sistemas, e por isso um sistema é considerado escalável se for possível adicionar utilizadores e recursos, sendo que as perdas com a performance e o aumento da complexidade não devem ser percetíveis. O conceito de escala tem associado três fatores em termos de dimensões: o ***numérico***, consiste no número de utilizadores, de objetos e de serviços que o sistema contém; o ***geográfico*** será a dispersão a nível global a que os sistemas estão uns dos outros; e por fim o ***administrativo*** que corresponde ao número de pessoas ou entidades que têm poder para controlar o sistema ou parte dele (Newman, 1994).

A escala afeta os sistemas em vários pontos sendo importante perceber quais são os problemas que teremos de enfrentar com os padrões estudados, de forma a conseguir colmatar e fornecer boas práticas para que desde o início o sistema seja bem desenhado. Pode afetar a nível da ***confiança*** como são distribuídos geograficamente, a capacidade de comunicarem entre si diminui, no entanto um sistema não deve deixar de funcionar apenas porque alguns nodos deixaram de estar acessíveis, apenas se deve ter em atenção que cada nodo deve ser cada vez mais autónomo e apenas afetar os recursos daquela parte do sistema.

Para combater o problema anterior existe a ***replicação*** que consiste em ter várias réplicas do mesmo recurso, ou seja, várias instâncias para que quando aconteça uma falha, ela fique o mais isolada possível, sem afetar o sistema como um todo. Na prática quando um dos nodos deixa de ficar acessível existem sempre outros a que se pode aceder. No entanto poderá ser mais lento mas o importante é que não aconteçam falhas para o utilizador.

A ***carga do sistema*** é afetada na medida em de que o sistema cresce, o número de dados também aumenta de tal forma que precisa de ser gerido através de *replicação, distribuição e cache dos dados* de modo a reduzir o número de pedidos em cada nodo. A diferença consiste em que a replicação e a distribuição servem para evitar os múltiplos saltos para chegar a uma informação, enquanto a cache dos dados serve para evitar a repetição de pedidos ao longo do

tempo. A ***manutenção (parte administrativa)*** torna-se cada vez mais complicada devido a existirem muitos nodos para que se guarde os dados de cada um deles, por isso, o mais usual é guardar os dados de um grupo de nodos que tem o mesmo objetivo no sistema. Com a distribuição consegue-se não apenas que uma pessoa fique responsável pelo sistema, mas que várias pessoas ganhem essa responsabilidade.

A ***heterogeneidade*** que a escala proporciona, leva a que cada grupo de nodos forme um subsistema e daí poderá ter tipos diferentes tanto de *hardware* como de *software*. De forma a reduzir este impacto no sistema é necessário formar um sistema coerente (*Coherence*) que consiste em ter por exemplo, uma interface comum entre todos os nodos. Outra forma é possuir computadores que possam executar um tipo de *software* que a qualquer momento possam recompilar e executar outro tipo de *software* mais conhecido como ***common execution abstraction***. Outra forma de obter esta coerência entre sistemas é a nível de protocolo: todos os nodos deverão ter suporte para um conjunto de protocolos comuns entre todos de forma a conseguirem comunicar sem qualquer problema (Newman, 1994).

Seguidamente é necessário determinar como é que a escala se aplica a sistemas, a forma como é que devem ser construídos desde o início e os ideais em que são baseados (Newman, 1994). Em termos de ***replicação*** deverão ser replicados recursos importantes para o sistema, aumentando a disponibilidade e diminuindo os saltos de cada pedido. As ***réplicas deverão ser distribuídas***, colocando-as em diferentes partes da rede para garantir a sua disponibilidade. Em casos de muito tráfego deverá existir pelo menos uma réplica para que uma área com pedidos frequentes, possa ser reencaminhada para uma réplica local de modo a balancear o tráfego.

Não é possível conseguir a ***consistência de todos os dados*** para estes sistemas de larga escala, podendo muitas vezes perder-se a consistência temporariamente. Ao longo do tempo essa consistência vai sendo reposta em todas as réplicas, existindo a necessidade de detetar se uma determinada informação está incoerente. Em relação a ***distribuição,*** esta deverá ser efetuada através de ***múltiplos servidores*** por forma a diminuir o tamanho da base de dados e assim reduzir o tempo de pesquisa em cada uma, diminuindo também o número de pedidos e por isso a sua carga.

A carga deverá ser ***proporcional à sua capacidade*** para garantir que nenhum nodo fica sobrecarregado enquanto outro está inativo. Deverá reduzir-se o tráfico e a latência através do estudo de quem usa determinados dados para que fique o mais próximo possível deles, diminuindo ao máximo o número de saltos, permitindo que a maioria das *queries* sejam locais e assim quase não precisem de sair da mesma área, e a isso é dado o nome ***explorar localmente***.

Em sistemas que contenham hierarquias, a maioria das vezes é preciso informação do pai (*root)* o que poderá ser custoso para o sistema, e a melhor forma de contornar este problema é através dos seus nodos (abaixo dele) que contêm uma cópia da informação, precisando

apenas da resposta do nodo superior e não do *root*. E a esta técnica denomina-se: ***evitar nodos superiores nas hierarquias***.

Em relação à utilização de ***caching,*** este é usado para dados que estejam sempre a ser necessários de forma a obter uma resposta muito mais rápida através de persistência local em vez de estar sempre a fazer o mesmo pedido. De acordo com a consistência dos dados a cache não deverá durar para sempre e por isso deverá ter uma ***cache timeout*** de forma a forçar a atualização dos dados. Com a existência de ***múltiplos níveis de cache*** será apenas possível pedir informação até ao *root* uma vez e depois todos os pedidos seguintes serão baseados nessa informação. Os pedidos deverão, tal como a distribuição, olhar primeiro para a ***cache local*** para evitar novamente pedidos ao servidor central, contudo deverá existir a preocupação de que a cada nível que se sobe na hierarquia, os objetos que poderão ficar em cache terão cada vez menos mudanças, senão poderão existir demasiados pedidos de atualização de cache o que poderá ter o efeito contrario ao que é pretendido (Newman, 1994).

Sobre a escalabilidade ainda existe a teoria de (Abdullin, 2011b), que tem por base uma arquitetura de *software* capaz de ***escalar de forma infinita,*** ou seja, o volume de dados e processos estão sempre em crescimento, e por isso é necessário existir uma forma que possa acompanhar este crescimento. Este tipo de arquitetura é baseada em *cloud computing* e tem algumas suposições para ser realizável. O sistema irá ter que fazer a gestão de entidades (ou *aggregators root*) que são identificadas de forma única no sistema, vejamos: Cliente. Com o crescimento do sistema estas entidades irão ser distribuídas por várias máquinas, e por isso não é possível ter a certeza em que máquina está uma determinada entidade. É pois necessário usar um sistema de mensagens para permitir a comunicação entre entidades como entre máquinas. Além disso existe a garantia de que as mensagens serão entregues pelo menos uma vez, e devido às limitações atuais e de custos, uma única entrega poderá não ser sempre conseguida (Abdullin, 2011b).

2.2 O que é um sistema de larga escala

2.2.1 Sistema de sistemas (System of Systems – SoS)

Neste tipo de arquitetura (Imagem 1) existem vários sistemas simples que formam um conjunto de sistemas, e por isso a sua composição são vários componentes de *software*, *hardware*, utilizadores e dados.

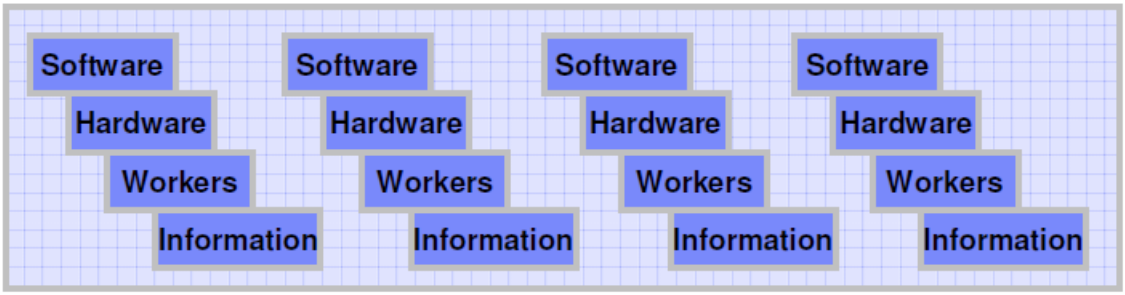

Imagem 1 – Sistema de sistemas (Eeles, 2008)

As características deste tipo de sistemas (Maier, 2009) são: ***Independência dos seus componentes*** – Todos os componentes têm de funcionar independentemente uns dos outros de maneira a que a falha de um não condicione todo o resto do sistema; ***Evolução continua*** – O sistema de sistemas não necessita de nascer logo todo 100% desenvolvido. Muitas vezes é desenvolvido e continuamente aperfeiçoado, sendo que neste caso ao ter independência permite ir evoluindo cada módulo com um risco menor de falha; ***Emergent Behavior*** – o sistema funciona como um todo e determinadas funções não estão apenas num componente do sistema, mas sim em propriedades que podem ser obtidas quando se percorre vários componentes do sistema; ***Distribuição geográfica*** – Os componentes podem ser distribuídos em qualquer zona do mundo, de maneira a reduzir a possível falha em determinadas zonas. Para além disso é necessário ter em linha de conta que as comunicações aumentam o que quer dizer que irá existir mais troca de informação entre os componentes; ***Possibilidade de aplicação de padrões*** – Como existem comunicações entre os componentes, muitas vezes a longos quilómetros de distância, torna-se necessário aplicar vários padrões para que se obtenha a máxima performance e fiabilidade do sistema.

2.2.2 Ultra Large Systems (ULS)

2.2.2.1 Definições e características

A principal característica dos ULS é serem inimaginavelmente grandes, o que provoca: número de linhas de código igualmente grande, existência de muitos utilizadores com diferentes propósitos a utilizar o sistema, grande quantidade de dados guardados, dados acedidos, manipulados ou melhorados. O mais importante destes sistemas é conseguir determinar o

conceito de "tamanho", para isso, teremos de entender os efeitos de ter um sistema em grande escala, como por exemplo, a responsabilidade das tecnologias e processos utilizados. Estes problemas que não são significantes em sistemas mais pequenos poderão ser totalmente o contrário em sistemas maiores. Por isso os sistemas de larga escala requerem uma diferente abordagem e novos conceitos de *design*, desenvolvimento e manutenção do sistema, e daí a frase "***scale changes everything***" (Carnegie Mellon University, 2006).

Os sistemas ULS têm algumas características em comum com os sistemas de sistemas, sendo que Mark Maier fez uma lista de características que distinguem grandes sistemas monolíticos de sistemas de sistemas: Funcionalidade independente dos elementos; Manutenção independente dos elementos; Evolução e interligação dos elementos que vão sendo construídas ao longo do tempo; Distribuição geográfica dos componentes (comunicação de informação). Contudo, estas características não são as desejáveis para descrever exatamente a base dos sistemas ULS, necessitando de características mais focadas na escala, ou seja (Carnegie Mellon University, 2006): ***Descentralização***, a escala dos ULS significa que devem ser descentralizados vários aspetos, como dados, desenvolvimento, manutenção e controlo; ***Dificuldade em definir os requisitos do sistema***, muitos dos *stakeholders* têm objetivos e perspetivas diferentes e por isso os requisitos irão sempre ser conflituosos, complexos e de mudança constante; ***Constante evolução e desenvolvimento***, durante a utilização do sistema será necessário integrar novas funcionalidades no sistema ULS, remover funcionalidades que já não sejam usadas, e assim o sistema não evolui em fases mas sim continuamente; ***Sistema heterogéneo***, é inconsistente e com muitas mudanças de elementos, porque o sistema não será feito com partes uniformes, poderá haver vários desajustes especialmente quando o sistema é estendido; ***Utilização das pessoas e limites do sistema***, porque as pessoas fazem parte dos sistemas ULS e por isso são elementos dele mesmo, afetando o seu comportamento; ***Falhas***, *software* e *hardware* serão a norma e não a exceção; ***Introdução de novos paradigmas*** de aquisição e política, porque o proveito de um sistema ULS será simultâneo com a operação do mesmo e por isso requer novos métodos de controlo.

As características acima referidas não são independentes e já são possíveis de verificar nos sistemas atuais de larga escala, contudo cada uma delas representa uma mudança na engenharia de *software*, desde o seu *design* e desenvolvimento até à manutenção, e daí a existência de cada vez mais padrões que consigam fornecer boas práticas para este tipo de sistemas não se tornarem insustentáveis. Cada uma das características é o resultado e uma prova que a escala dos sistemas altera tudo. Por isso, existe a necessidade de novos estudos que se foquem mais na escala e na maneira como esta deve ser gerida.

2.2.2.2 Problemas existentes e convicções atuais

Existem atualmente alguns problemas já identificados e algumas convicções sobre estes sistemas que se tornam necessárias estudar. Um exemplo é "***Todos os conflitos precisam de ser resolvidos e a sua resolução deve ser de forma uniforme***". Esta frase exprime a ideia de que deve ser tudo resolvido. Porém numa organização cada um tem os seus interesses e com

isso nascem e desenvolvem-se conflitos de desenvolvimento, o que torna impossível resolver todos os conflitos e ainda mais de forma centralizada. A melhor solução é resolver este tipo de conflitos localmente com quem tem o interesse imediato, sendo que se o mesmo problema ocorrer noutra zona do sistema ele poderá ser resolvido de outra forma mais adequada, por isso a solução não deverá nem poderá ser igual para todos como se fosse uma regra absoluta. (Carnegie Mellon University, 2006).

A ***resolução dos problemas***, tanto a nível de ***escala*** como a nível de ***complexidade*** muitas vezes acontece apenas, quando o sistema já se encontra em funcionamento, porque é frequente ser mais fácil perceber e determinar a melhor solução já com o sistema em funcionamento, fase em que os padrões têm um papel determinante pois irão permitir precaver a maioria das situações.

Em termos de ***durabilidade,*** os ULS irão fornecer os seus serviços durante um longo período de tempo, e devido ao seu tamanho é difícil retirar ou substituir uma parte do sistema. Em vez disso, e como os requisitos estão constantemente em evolução tal como a tecnologia, a necessidade de evolução passa por ter regras e políticas que sirvam para as necessidades locais sem comprometer o sistema restante e daí podermos dizer que os seus elementos são: ***Heterógenos***, os elementos do *software* irão ser heterogéneos devido à sua fonte ter linguagens diferentes, plataformas diferentes, diferentes filosofias e metodologias, e por isso nos sistemas ULS estas diferenças serão divididas por exemplo, em "*legacy systems*", e outros serão transformados em serviços, etc. A vantagem da heterogeneidade é que nem todos os elementos serão igualmente vulneráveis a falhas ou ataques; ***Inconsistentes***, devido à sua grande escala, o sistema poderá ser modificado em vários pontos, por diferentes pessoas, processos, objetivos, etc., e por isso irão existir no sistema diferentes versões do mesmo componente o que poderá levar à existência de inconsistências a nível de *design*, implementação e uso; ***Mudança constante,*** visto que uma grande parte do sistema irá estar em constante mudança, por exemplo, devido a problemas de *hardware* que foi necessário substituir, *software* a que foi necessário fazer uma atualização, ou configurações que necessitaram de modificação. Com estas mudanças o sistema deverá adaptar-se dinamicamente à sua nova missão, contudo é impossível saber exatamente qual será o seu *design* à partida, porque poderá variar constantemente.

Por fim em relação ao ***papel que as pessoas podem tomar***, é necessário existir um tipo de estatísticas que permita analisar o que os utilizadores fazem, de forma a conseguir verificar como e onde o sistema (ULS) pode melhorar. Neste aspeto existem padrões que utilizam sistemas de eventos, que também podem ser utilizados para determinar este tipo de padrões de comportamento. Nestes sistemas ULS, as ***falhas de software*** e o comportamento inesperado deixarão de ser uma exceção, e serão sim uma constante. Mesmo quando as falhas nos pequenos sistemas são raras, a nível de sistemas ULS irão ser muito frequentes e possivelmente muito mais graves e difíceis de controlar. Desta forma, as falhas que ocorrem com pouca frequência em sistemas mais pequenos, nos sistemas ULS tornam-se frequentes. Por exemplo, para transferir um ficheiro por FTP (*File Transfer Protocol*) se falhar 1 em 1000, e caso este tipo de transferência seja utilizado 1000 vezes num dia, significa que irá sempre

ocorrer uma falha num dia. Os ULS têm de ter mais em consideração a tolerância a falhas e devido à sua grande escala poderá ser necessário desenvolver novas estratégias de tolerância a falhas, devendo existir um controlo por área de maneira a que caso ocorra uma falha, ela seja contida e as consequências limitadas ao máximo, e se possível alertar antes de elas ocorrerem (Carnegie Mellon University, 2006).

Os sistemas atuais já possuem algumas destas características mas o que distingue os ULS dos sistemas atuais é que os ULS têm todas estas características, e os problemas atualmente existentes tendem cada vez mais a serem combatidos com recurso por exemplo a padrões de desenvolvimento (Carnegie Mellon University, 2006).

2.3 SQL e sistemas de larga escala (noSQL)

Os sistemas de larga escala, como por exemplo, aplicações web, atualmente já se encontram distribuídos de uma forma bastante consistente, tanto a nível de localização que pode ser a nível mundial como a nível de grandeza. Com isto, nasceu um outro problema que é a questão de como deve ser distribuída uma base de dados relacional. Seja a maneira de como dividir a tabela, seja a forma de tratamento de uma transação que pode alterar N tabelas que estão em N sítios. Para solucionar este problema foi desenvolvido o conceito de "*NoSQL*" (*Not only SQL*), que tem a característica de não ser uma base de dados relacional, retirando as propriedades de ACID (Atomicidade, Consistência, Isolamento e Durabilidade), muito usadas hoje em dia para caracterizar uma transação.

O *NoSQL* tem o grande objetivo de facilitar e resolver os problemas de escalabilidade que se tem vindo a verificar nas base de dados relacionais, visto cada tipo de base de dados ter problemas diferentes ao terem diferentes objetivos. Os sistemas tradicionais RDBMS (*relational database management system*) não são impossíveis de escalar, o problema é a complexidade associada e o custo que isso traz.

Exemplo de uma aplicação web tradicional (Imagem 2):

Imagem 2 - NoSQL - Aplicação web tradicional (Ferreira, 2010b)

Com o tempo, este tipo de aplicações que podemos ver na Imagem 2, tende a evoluir e a necessitar de mais subaplicações (divisão em módulos) e até mesmo de conseguir suportar mais utilizadores e para isso temos duas soluções (Ferreira, 2010b): ***Escalabilidade Vertical (scale up)*** – aumentar a capacidade do servidor, como por exemplo a memória, processador, disco, etc. e Escalabilidade ***Horizontal (scale out)*** – Aumentar o número de máquinas de servidores web (Imagem 3).

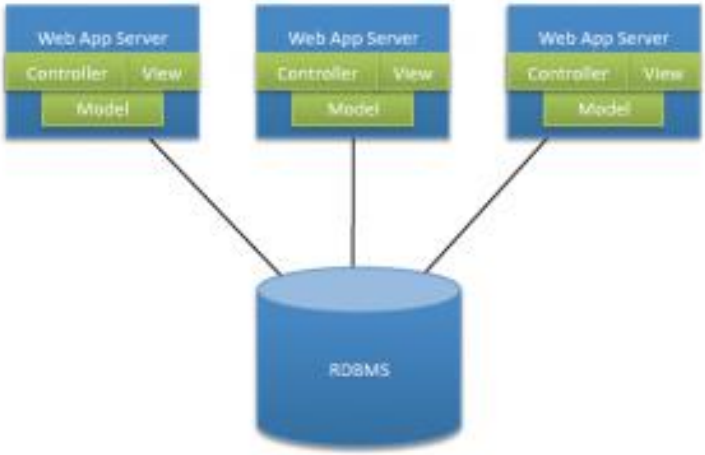

Imagem 3 - Escalabilidade horizontal (Ferreira, 2010b)

A nossa aplicação mesmo assim poderá continuar a crescer, e quando já tivermos um número bastante considerável de servidores web o *bottleneck* do sistema estará na base de dados, porque não terá a capacidade de responder a todos os pedidos. Mesmo que o tivesse, não seria em tempo útil. Nesta situação teríamos novamente duas opções: escalar verticalmente e construir uma supermáquina de base de dados com custos enormes, ou escalar horizontalmente e ter várias máquinas de base de dados a um custo muito inferior e mais fácil de escalar, como se pode ver pela Imagem 4.

Imagem 4 - Escalabilidade horizontal de base de dados (Ferreira, 2010b)

Depois de termos a situação da Imagem 4, temos de perceber que a situação mesmo assim é mais complicada do que escalar aplicações web, visto que não podemos ligar duas máquinas de base de dados e elas continuarem a funcionar sem qualquer problema. Para isso, iremos precisar de uma série de alterações tanto a nível de configurações como a nível de aplicações para que tudo possa agora funcionar de forma distribuída. Para ajudar neste processo, o *NoSQL* tem um papel importante para tornar tudo mais simples aos programadores que antes tinham de dividir da forma que achavam melhor e ficavam sempre responsáveis por essa tarefa trazendo o respetivo custo. Agora, de uma forma automática e automatizada, é possível distribuir por várias máquinas a base de dados. As técnicas são utilizadas, mas estão encapsuladas para que o desenvolvimento não tenha de ter esta responsabilidade e preocupação.

Técnicas usadas para escalar base de dados (Ferreira, 2010a): ***Replicação,*** consiste em escalar por duplicação de informações. Nesta técnica copia-se a informação em mais de uma base de dados, facilitando a recuperação de informações, ***Master-Slave***, indica quando o número de leituras da base de dados atingir um número superior de escritas, ou seja, cada escrita que chega ao Master resulta em N escritas para todos os N *slaves*, o que poderá resultar num *bottleneck*. Existe o ***Multi-Master,*** que consiste em aumentar o número de Masters no sistema e assim conseguimos aumentar a capacidade de escrita, o ***Sharding***, que consiste em escalar por divisões. Aqui os dados são divididos em várias tabelas, aumentando assim tanto o nível de leitura como o nível de escrita, e é aqui que existe uma quebra com o tradicional sistema de base de dados relacional. Com esta divisão, as aplicações têm de resolver alguma complexidade gerada pela partição dos dados, como por exemplo necessitar de executar *joins*, etc.

Pode igualmente existir uma relação entre a base de dados *NoSQL* e uma base de dados relacional. O *NoSQL* pretende facilitar a escalabilidade mas não ser uma "*silver bullet*" para todos os problemas que existem. Conjugando as duas cada uma terá para resolver parte dos nossos problemas como podemos ver pela Imagem 5.

Imagem 5 - Base de dados relacionais e NoSQL (Ferreira, 2010b)

Podemos classificar as base de dados *NoSQL* em 3 características principais (Ferreira, 2010a) (NOSQL, 2012): quanto à ***arquitetura*** poderão ser *distribuídos* (Exemplos: Cassandra, BigTable e Amazon Dynamo) ou *não distribuídos* (Exemplos: Redis, MemcacheDb e Amazon SimpleDb). Quanto ao ***armazenamento***, os dados poderão ser armazenados em *memória* (Exemplo: Scalaris e Redis), *disco* (Exemplo: CouchDb, MongoDb e Riak), ou *Configurável* (Exemplo: BigTable, Cassandra e HyperTable). Quanto ao ***modelo de dados***, os dados poderão ser: *chave/valor* (Exemplo: Amazon Dynamo, Amazon S3 e Redis), *documento* (Exemplo: Amazon SimpleDb, MongoDb e Riak), por *colunas* (Exemplo: Cassandra, BigTable) ou por *grafo* (Exemplo: Neo4j e InfoGrid).

2.4 Tendências futuras

As tendências para o futuro deste tipo de sistemas, baseado nas comunicações e na tecnologia usada (ICT - *Information Communications Tecnology*), foi alvo de um estudo por parte da *Future Internet Enterprise Systems* (FInES) (FInES, 2010). Nele se identificou o que poderia ser futurista, quais as perspetivas, que ideias fora do comum poderiam ser criadas, etc., podendo no final potenciar ideias radicais e revolucionárias na forma de pensar, implementar, produzir, processar e organizar. Neste contexto é necessário ter uma ideia de como e o que será necessário no futuro de forma a conseguir concluir o que se deverá usar e se é adequado o uso de padrões.

2.4.1 O que se prevê para 2020

Para começar é necessário perceber porquê a escolha do ano 2020. Tem essencialmente a ver com o facto de que os estudos a pequeno/médio prazo serem melhor aceites pelas empresas e indústria de consultoria. A internet é considerada hoje em dia um negócio universal na qual se pode construir novos valores, criar novas ligações entre empresas, permitindo inovar

criando benefícios sustentáveis. Em relação ao futuro de um sistema empresarial, este tende a ser mais linear, mais adaptativo, flexível e expansível, permitindo criar maior valor em inovação para a empresa, sendo nisto que o estudo a seguir se irá focar. É necessário saber o que irá evoluir e o que nunca mais irá ser usado, para saber a melhor forma de abordagem, nomeadamente que tipo e quais os padrões que se podem vir a destacar mais.

Para as empresas, o maior problema não é a obtenção de tecnologias mais sofisticadas mas antes tecnologias que possam aumentar o valor dos produtos/serviços que oferecem. Com este objetivo em mente, as tecnologias deverão ser inovadoras, ou seja, serem capazes de permitir evoluir constantemente como se fosse uma rotina. Por isso, no futuro as tecnologias vão ser capazes de fornecer valor acrescentado, através tanto da sua própria inteligência, como através de operações precisas para o utilizador final, poder ser orientada a serviços, poder ser modificada e colocada em produção *online* a qualquer momento.

A ultima tendência é denominada "*Cloud Computing*", que combina vários conceitos orientados a serviços, com por exemplo, "*infrastructure-as-a-service*", "*platform-as-a-service*", e "*software-as-a-service*". Para além da questão tecnológica é necessário ter noção da parte do negócio, ou seja, modelos de negócio, políticas de manutenção das "*cloud*", e qual a estratégia da Europa para este tipo de economia digital. Alguns valores empresariais do futuro e que já são declarados pela Google e pela Amazon, serão as receitas e lucros; reputação; uso eficiente e eficaz dos recursos (naturais e matérias-primas); pegada de carbono; relação com os colaboradores, parceiros e clientes; Exploração e uso do conhecimento; transparência negocial e responsabilidade social (FInES, 2010).

2.4.2 Grandes objetivos futuros

Este estudo tem como grande objetivo, facultar diretivas e mostrar o melhor caminho que se deve seguir, não sendo isto um conjunto de regras que se tenha de seguir mas antes um conjunto de boas práticas (*roadmap*) que devem ir sendo postas em prática ao longo do tempo. Além disso será necessário incluir novos problemas que possam existir no futuro (alguns que já existem atualmente). O foco destes objetivos será mais dedicado a empresas que podem e devem utilizar padrões.

O tipo de empresas do futuro serão ***empresas cloud***. Já existem evoluções nesta área através do *cloud computing*, que permite às empresas um desenvolvimento contínuo nos próximos anos, podendo utilizar vários padrões desde que estes sejam compatíveis com a alta disponibilidade e escalabilidade que podem ter. No caso das empresas de *software*, o *cloud computing* permite às empresas muitas vezes serem as próprias consumidoras dos serviços nomeadamente as equipas de TI. Exemplo: AppStore, Amazon Mechanical Turk, Netflix Prize, BellKor Pragmatics Chaos and Wikipedia.

As empresas que precisarão de ter conhecimento de gestão serão as ***Cognizant Enterprise***, que terão o objetivo de ganhar cada vez mais conhecimento sobre uma determinada área e conseguir usa-lo a seu favor. Para isso, muitas vezes deverão usar conhecimentos de várias

fontes, e os padrões deverão ter em consideração a necessidade de existir uma parte do sistema que necessita de algum tipo de "inteligência artificial" para funcionar. É frequente esta inteligência ser utilizada para ajudar os próprios utilizadores a determinar as suas tarefas, criando o conceito de *Cloud Knowledge.* O *Cloud Knowledge* que permite não ter este conhecimento local mas em toda a internet, podendo assim evoluir mais rapidamente, adquirindo conhecimento externo.

As empresas ganharão também preocupação com a responsabilidade social, que é designada por ***Community-oriented Enterprise.*** Para isso será necessária a existência de uma maior transparência, através de criação de mecanismos que permitam monitorizar tanto a nível legislativo como ético. Haverá uma maior preocupação com a criação de valores para as pessoas do que em lucros monetários e por isso a adaptação e a criação de novos padrões tem que estar ciente deste aspeto. Estas empresas terão de comunicar entre si, estarão em zonas geográficas diferentes e existirá a necessidade de fornecer aos clientes de forma clara o que se passa em cada momento na empresa. Este tipo de empresas ainda não estão muito bem definidas, mas prevê-se que sejam utilizadas nas áreas da medicina, bem-estar e ONGs.

Em relação às ***Green Enterprise,*** é importante referir que são empresas que têm alta preocupação com o ambiente e que querem contribuir para a "*Green Economy*" e por isso apenas estão interessadas em padrões que permitam gerir todos os recursos da melhor forma e que permitam ter por exemplo, a maior poupança energética. Para isso seria necessário um determinado padrão permitir ter uma arquitetura que permita a alocação dinâmica de recursos, o que já é ajudado hoje em dia pelo *cloud computing*.

As empresas ***Glocal Enterprise*** são empresas que pretendem ter a sua atividade exercida tanto a nível global como local ("glocal") contribuindo para dar uma melhor resposta ao seu público-alvo. Assim, será necessária a introdução de mais pessoas envolvidas no processo, como agentes, e será necessário conseguir aplicar o mesmo padrão para várias situações na mesma empresa. Isto é conseguido tanto a nível de modelação dos próprios processos da empresa como a nível da flexibilidade que existe nos padrões e na forma de eles serem extensíveis. Um exemplo será um padrão conseguir separar quem está na parte local e quem está na parte global, para que exista uma ligação entre eles mas a forma como se ligam á empresa mãe deverá ser de forma diferente (FInES, 2010).

2.4.3 Evolução das comunicações e redes

Estando a área das comunicações e redes sempre em constante evolução, é importante ter uma ideia do que é que, para 2020, se irá usar mais e de que maneira os padrões que se pretendem aplicar se enquadram. Este estudo pretende realçar apenas as partes mais importantes em termos de estrutura para aquando da projeção de *software.* Para esta arquitetura, saber quais os tipos de padrões que poderão ser aplicados sem problemas ou se é necessário haver algum tipo de ajustes.

As redes que se centralizam mais na preocupação com o QOS (*quality-of-service*) serão designadas de ***Convergent networks.*** Nas *Convergent networks* a qualidade terá de ser assegurada em todos os níveis de transportes e respetivos componentes. Existe uma grande variação de tipos de redes (wireless, P2P, etc.), com diferentes objetivos (vídeo, voz, dados, etc.), assistindo-se a uma evolução direcionada para a homogeneização dos seus protocolos e formatos. Novos desenvolvimentos irão precisar de cada vez mais aspetos da comunicação, e um exemplo é as "*identity-aware networks*", em que a infraestrutura da comunicação autentica o utilizador, usa esses dados para determinar um perfil e assim saber as suas preferências, como por exemplo, se um utilizador tem mais prioridade que outro.

Outro exemplo são as "*context-aware networks*". Aqui, o contexto em que um utilizador se encontra serve de diretiva para que cada pedido do utilizador seja redirecionado para o local correto, tornando assim a fronteira entre a aplicação e a comunicação transparente. As comunicações ***Wireless Communication*** serão cada vez mais utilizadas visto que atualmente tem vindo a aumentar os equipamentos móveis no mercado. Isto significa que a utilização da radiofrequência irá também aumentar, e disso é exemplo o RFID (*Radio-frequency identification*) e o NFC (*Near field communication*). Utilizando por exemplo o RFID, será possível criar ***Smart-Product Infrastructures*** que irão identificar e manter um rasto do produto, sabendo num preciso momento onde está e para onde vai. Este será um exemplo muito utilizado por cadeias empresariais que fazem a produção e distribuição de um produto, sabendo todos os movimentos do produto desde o seu fornecedor até ao cliente final.

A estrutura ***Real World Web*** é baseada em sensores embutidos em quase tudo que nós conhecemos, sendo que a evolução dos sistemas móveis tem vindo a dar um grande impulsionamento nesta área e por isso estarão cada vez mais presentes no nosso dia-a-dia. Estes sensores precisam de comunicar entre eles, muitas vezes sobre a forma de redes de malha, não sendo distinguida a comunicação do mundo real (fala, gestos, etc.) com a comunicação da parte técnica (tradução do mundo real para o mundo digital). Com o *Real World Web* nasce o ***Ubiquitous Communication***", que consiste na inexistência de diferenciação entre as comunicações reais e a sua correspondência no mundo digital, levando a que exista uma comunicação em que não se sinta a mudança de um tipo de comunicação para outra.

Utilizando cada vez mais as redes de sensores, em 2020 as redes ***Mesh-Sensor Networks*** serão das mais usadas, visto caminhar-se cada vez mais para uma descentralização das redes *wireless*. Esta descentralização é efetuada a nível de terminais, dispositivos móveis, etc., ficando uma arquitetura disposta como uma rede de malha de sensores, ou seja, é baseada em redes *ad-hoc* em que existem comunicações p2p da maneira mais eficiente possível. Por fim uma das grandes evoluções será baseada nas comunicações ***Communication as a Service***, em que a infraestrutura da comunicação tende a ser cada vez mais consolidada, seguindo mais padrões até ser parcialmente virtualizada, ou seja, será organizada sob a forma de comunicação orientada a serviços (FInES, 2010).

2.4.4 Roadmap para sistemas aplicacionais

Com este estudo pretende-se prever quais serão os sistemas em 2020 que serão mais usados. Isto poderá ser visto como pontos que os padrões terão de ter em conta e quais serão os tipos de sistemas empresariais usados.

A arquitetura ***Tera-architectures*** tem o conceito de captar todo o tipo de informação, gerar *reports*, etc. de forma a obter o máximo de informações possíveis. Por exemplo, tudo que seja elétrico poderá ter o seu próprio sistema de comunicação com outros elementos do seu ambiente, poderá ter informação acerca do que se está a fazer naquele momento, qual é o seu estado e o que se passa ao seu redor. Este tipo de comunicações gera múltiplos *terabytes* de informação, em tempo real, o que leva a que seja preciso ter um sistema capaz de processar esta quantidade de informação, sendo este um dos desafios em termos tecnológicos, e daí a necessidade de existirem alguns padrões que consigam lidar com este tipo de problemas.

Irão existir sistemas capazes de conseguir prever o que irá acontecer, denominados por ***Applications with proactive behaviour***. Estes sistemas consistirão em guardar e processar transações, sendo que num evento existe uma reação. De maneira a adequar melhor aos utilizadores e organizações, este tipo de aplicações e sistemas tem que prever as próximas tarefas a serem realizadas, podendo assim dar uma assistência mais rápida, muitas vezes as tarefas que ainda nem começaram. Os padrões que comunicam através de eventos poderão ter esta característica por forma a serem mais eficientes.

Em termos de plataforma existirá o ***IaaS or PaaS (Infrastructure/Platform as a Service)*** que está relacionado com *cloud computing*. Contudo este tipo de soluções em *cloud*, são oferecidas por empresas como a Amazon ou existem dentro das próprias organizações, tornando assim necessário que este tipo de mercado seja mais explorado e se torne maior e mais transparente.

Em relação a interoperabilidade existe o conceito de ***Interoperability as a Service***, que devido ao aumento das redes e ao número de componentes heterogéneos, exige a garantia duma uma interoperabilidade tecnológica. Para isso a tendência é existir a ISU (*Interoperability Service Utility*), que permite fornecer serviços para promover esta interoperabilidade. A ISU é considerada uma capacidade necessária para cada componente. Cada vez mais os sistemas serão *Software as a Service* (SaaS): em que todo o *software* será orientado a serviços, precisando também, de uma evolução a nível de fonte de descoberta, execução e garantia de fiabilidade, no qual os padrões terão um papel fundamental.

Por fim os componentes dos sistemas serão ***Intelligent and smart components*** baseados num agente, ou componente de *software* que incorpore de alguma forma inteligência. A criação de novos componentes deverá ter a representação de conhecimento e/ou inteligência artificial de maneira a que seja combinada uma série de objetivos que passem por representar e assistir em diferentes níveis, como a colaboradores de uma empresa até ao sistema de gestão

de versões, daí poder existir padrões que descrevam exatamente como deve ser estruturada esta inteligência (FInES, 2010).

2.4.5 Tecnologias existentes e investigações futuras

Em relação a tecnologias emergentes, os ***Smart Products*** irão fazer parte de quase toda a industria, ligados entre si e em que será fortemente utilizado o RFID como lançamento deste tipo de produtos. Atualmente o EPC (Electronic Product Code) está estabelecido como *standard* desta área, contudo com o avanço da tecnologia e de novas aplicações serão precisos novos *standards* de forma a regular mais o desenvolvimento. Atualmente existe o GS1 GDSN (*Global Data Synchronisation Network*) (GS1 System, 2012), que assegura um ambiente globalizado continuo e seguro, em que os dados estão sempre sincronizados para que todos os intervenientes em algum processo possam ter acesso a essa informação ao mesmo tempo. Já existem alguns projetos europeus em curso como por exemplo o BRIDGE (Bridge, 2012), CuteLoop (CUTELOOP, 2008) e o iSURF (ISURF, 2011).

Existem também preocupações com ***Security and Policies*** em que também já existem alguns projetos europeus que apenas se focam na segurança e nos problemas que podem existir como por exemplo: ANTIPHISH (*Anticipatory Learning For Reliable Phishing Prevention*) (AntiPhish, 2009), S3MS (*Security of Software and Services for Mobile Systems*) (S3MS, 2012), SECOQC (*Secure Communication based on Quantum Cryptography*) (SECOQC, 2008), OPENTC (*Open Trusted Computing*) (OpenTC, 2008), PRIME (*Privacy and Identity Management for Europe*) (PRIME, 2008) e HUMABIO (*Human monitoring and authentication using biodynamic indicators and behavioural analysis*) (HUMABIO, 2008).

Novos conceitos das organizações irão surgir, como a virtualização de organizações e a utilização de *cloud*, e o que ajudará a base destas mudanças será a ***Socialização*** das empresas, que assenta essencialmente na Web 2.0, visto que inclui serviços, aplicações web, redes sociais, vídeo conferencia, *wikis* e *blogs*. As novas tendências serão exploradas a partir daqui. As empresas serão também baseadas no conceito de "***Knowledge Society***", que assenta no contexto da própria empresa para criar uma maior envolvência por parte dos clientes no ciclo de vida dos produtos.

Em relação a sistemas aplicacionais usado nas empresas, irá ver-se cada vez mais o ***Software inteligente,*** que consegue agir proactivamente ao comportamento humano, monitorizando e respondendo, sem a supervisão constante do utilizador. Para isso o sistema deverá ser capaz de configurar, adaptar e manter-se a ele próprio, decidindo quais ações que devem ser tomadas em certas situações. A definição deste tipo de dispositivos é denominada de agentes ou multiagentes que se conseguem organizar autonomamente. Os padrões que poderão ser usados neste contexto devem ter em linha de conta de como, e onde, é que devem estar este tipo de componentes. O desafio e as investigações atuais baseiam-se na melhor forma de gerir centenas ou milhares de redes e dados que circulam, de forma a conseguir manter a

performance da rede, e ao mesmo tempo conseguir monitorizar, de maneira a que a qualquer altura, se possa dar ao utilizador uma "opinião" ou mesmo tomar a decisão por ele.

Os sistemas terão ***Operational Infrastructures*** que se baseiam na criação de organizações virtuais e "*digital business ecosystems*", ou seja, recorrem ao *cloud computing* permitindo a colaboração e modificação total, no plano de produção. Os benefícios serão o aumento ou a diminuição fácil dos recursos disponíveis e a mudança de algum modelo de produção que seja necessário, de forma quase instantânea, daí que a maioria dos padrões adverte para o uso de componentes independentes.

O ***Software-as-a-Service (SaaS)*** será o de maior crescimento, devido a permitir que as empresas consigam atingir os seus objetivos com um menor custo, tanto de desenvolvimento como de manutenção. As tecnologias usadas para o *Saas* serão o *Enterprise Service Bus (ESB)*, que é um sistema de mensagens, distribuído, baseado em *standards* que providencia a ligação, invocação e a mediação de serviços para facilitar as interligações das soluções distribuídas que existem hoje em dia. Os *Web Services*, muitos utilizados para sistemas distribuídos devido a serem independentes, auto descritivos, são aplicações modulares, o que permite publicar, localizar e expandir para toda a internet. Cada vez mais se verifica que os padrões de arquitetura têm cada vez mais preocupação com a criação de *SaaS*, podendo ser reutilizados N de vezes por todo o sistema, e muitas vezes servir de *outsourcing* (FInES, 2010).

2.4.6 Cloud Computing

De forma a perceber onde se poderá aplicar os novos conceitos de futuro e alguns padrões, torna-se necessário saber em mais detalhe em que consiste o *cloud computing* visto que os grandes sistemas de hoje em dia, são baseados neste conceito. O *cloud computing* é baseado em *hardware* apoiado em serviços, que envolvem computação e processamento, comunicações em rede e capacidade de armazenamento e têm como principais características: os ***serviços*** serem fornecidos ***on-demand***, ou seja, todos os clientes pagam à medida que vão gastando; toda a ***manutenção é abstraída*** dos clientes; toda a infraestrutura ***pode aumentar ou diminuir a escala*** facilmente; o consumo de serviços na *cloud* permite ter um melhor ***uso eficiente de recursos*** em comparação com os *data centers* utilizados hoje em dia; facilitar a ***preparação para picos do sistema***, visto que os recursos são atribuídos dinamicamente só á medida que são necessários (Abdullin, 2011a).

Do ponto de vista dos clientes, eles podem interessar-se apenas por algumas das vantagens desta tecnologia, como por exemplo, apenas a parte de armazenamento, o que exige que os fornecedores de *cloud computing* se dividam em vários tipos de serviços: ***Infrastructure as a Service (IaaS),*** em que o cliente recebe um conjunto de *hardware*, como por exemplo, máquinas virtuais com um determinado sistema operativo, exemplo da Amazon Elastic Compute Cloud (Amazon, 2012d) e Rackspace (Rackspace, 2012); ***Platform as a Service (PaaS),*** em que ao cliente é fornecido uma plataforma para colocar os seus recursos, permitindo ultrapassar e simplificar problemas tecnológicos, ou seja, fazer por exemplo o *deploy* e

configuração de determinadas aplicações em seu poder e utilizar os serviços para escalarem dinamicamente. Deste sistema são fornecedores, Microsoft Windows Azure (.NET) (Microsoft, 2012), Amazon Web Services (Amazon, 2012c), Google App Engine (Google, 2012), Amazon Elastic Map Reduce (Amazon, 2012a); ***Software as a Service (SaaS),*** o cliente tem a vantagem de se abstrair de toda a complexidade tecnológica e ter todo o suporte que seja necessário, para assim lhe ser proporcionado algo que tenha valor em termos de negócio e em que todo o *software* seja orientado para serviços, sendo assim de fácil acesso e perceção (Abdullin, 2011a).

2.5 Conclusão

Hoje em dia qualquer empresa necessita cada vez mais de alto processamento e alta disponibilidade. Para isso, é necessário saber o que é escala e em que consiste um sistema de larga escala, para se identificar quais serão as vantagens, desvantagens e problemas futuros que possam existir. Em relação a vantagens, o que é pretendido é tornar o sistema confiável ao máximo, usando replicação, balanceamento da carga do sistema, distribuindo geograficamente para o caso de catástrofes. As desvantagens com este tipo de sistemas prendem-se com os custos, manutenção que possam ser necessárias realizar, a nível de complexidade, coerência imediata dos dados e heterogeneidade dos sistemas de larga escala. Para os problemas futuros convém ter a noção de como poderão estar daqui a uns anos, as aplicações e infraestruturas, e por isso existe um complemento com um estudo do Future Internet Enterprise Systems (FInES) (Research, 2010) que nos dá uma ideia do que existe agora e como vai evoluir até 2020.

Para os problemas apresentados acima, esta tese será focalizada apenas numa parte deles, como balanceamento do sistema, coerência imediata dos dados e possível distribuição de dados. É importante realçar que não se pretende dar uma fórmula ou metodologia que resolva todos os problemas de sistemas de larga escala, mas antes ter consciência em que se baseia um sistema deste tipo e fornecer um conjunto de boas práticas que devem ser seguidas e aplicadas a uma parte dele, para mais fácil se tornar a sua expansão aumentando o valor para as empresas.

O padrão que se pretende estudar de seguida é denominado CQRS, que fornece uma grande ajuda, a nível da melhor forma de construir e gerir uma parte de um sistema de larga escala, que pode escalar "quase de forma infinita" (Abdullin, 2011b). Inicialmente poderá ser uma aplicação web simples e monolítica e passar a ser distribuída a nível mundial (Newman, 1994). Além de fornecer informação acerca de como organizar uma arquitetura completa de uma aplicação, este padrão consegue ser um grande apoio no que toca à heterogeneidade sendo este um dos grandes problemas atuais. Esta heterogeneidade de cada componente no sistema tem o objetivo de usar a melhor tecnologia que se adequa à sua função, o que leva a que mais tarde quando for necessária a interligação dos componentes se encontre um grande impedimento. Tal como referido anteriormente, este padrão, não tem a função de "*silver bullet*" em que é possível resolver todos os problemas, mas tem a melhor maneira de o fazer,

estando também preparado para a evolução a nível de crescimento e mais fácil adoção de novos requisitos do sistema. Torna-se posteriormente necessário saber a nível aplicacional as mudanças que isto trará, tanto a nível de programação como a nível de distribuição, como por exemplo, numa aplicação web a forma em grande escala da aplicação com o CQRS (e outros) e como ficará (Young, 2011).

3 CQRS - Command and Query Responsibility Segregation

3.1 Introdução

Atualmente as arquiteturas são na sua maior parte baseadas num servidor responsável por fornecer os métodos de CRUD (*Create Read Update Delete*) às aplicações, sendo que os dados numa grande parte são atualizados logo que aconteça alguma mudança. No entanto uma das coisas que se pretende mostrar com este padrão é o facto de que poderá não ser necessário fazê-lo. Um exemplo, pode ser o caso de um *website.* As estatísticas dos utilizadores do sistema, e que está sempre visível, não precisa de estar atualizado ao milissegundo e estar sempre a fazer pedidos ao servidor quando quer saber todas as estatísticas, ou seja, são dados que podem ir sendo atualizados faseadamente de forma a não sobrecarregar o sistema com informação excessiva que, para o utilizador, não é essencial.

Para além da estrutura da arquitetura que poderá ser refinada, existem outros pontos que é importante realçar a que este padrão tenta responder da melhor forma como é o caso do suporte a inúmeros utilizadores: logo á partida pode-se desenhar um sistema que poderá ser de larga escala e podem dividir-se as responsabilidades num sistema.

A definição oficial é: "CQRS é a criação de dois objetos onde antes havia apenas um. A separação tem por base a determinação de um método como um *command* ou *querie*" (Oliver, 2011b). Este padrão é caracterizado por separar os papéis e responsabilidades no sistema, de maneira a que a arquitetura se torne mais eficiente, aplicando limites muito restritos na nossa aplicação, de modo a que se torne consistente no final. É direcionado para suportar um grande número de utilizadores e saber lidar com esse facto de modo a que um sistema possa crescer para cada vez elevar mais o seu número sem grande complexidade e sem deteriorar o sistema, sendo um exemplo milhares de *browsers* a acederem a uma aplicação web.

Além do elevado número de acessos em simultâneo, este padrão pretende diferenciar a natureza das operações, ou seja, quer evidenciar que num sistema deste tipo existem muito mais leituras de dados do que manipulação de dados, e desta forma separar inequivocamente

a responsabilidade e objetivo de cada parte. Para o fazer, este padrão separa a aplicação em três grandes partes, ***comandos***, ***queries*** e (na maioria das vezes) ***eventos***, tornando a aplicação possível de desenvolver em paralelo e a manter em pequenas unidades (Tom, 2010). Em relação a um comando, este deverá realizar uma ação, ou seja, deve fazer uma alteração no sistema como uma atualização na base de dados enquanto uma *query* deve retornar dados do sistema mas não deverá alterar o estado do sistema, ou seja não pode ter efeitos secundários quando se executa uma *query*. Assim, sabendo que qualquer *query* não deverá alterar o nosso sistema, as *queries* poderão ser realizadas sem problemas, enquanto o uso de comandos já se torna refletido no sistema visto que a função é única e simplesmente alterar o estado do sistema, o que requer mais cuidados (Fowler, 2005a).

Estes modelos baseados em *queries* e comandos do CQRS é diferente de CQS (*Command-query separation*) do autor Bertrand Meyer's, em que o principio deste padrão é caracterizado por permitir a existência de dois tipos de modelos, um de leitura e outro de escrita (Young, 2011). A grande diferença para o CQS é a separação de responsabilidades e a forma como internamente os comandos e *queries,* as são geridos, como por exemplo, cada um ter a sua própria base de dados (dedicada) otimizada, sendo que as *queries* poderão ser otimizadas apenas para as leituras serem muito rápidas, podendo estar desnormalizadas (Fossmo, 2009). Em termos de escalabilidade, ao ter uma separação entre leitura e escrita, permite que se consiga, por exemplo, ter o componente de leitura em 25 máquinas enquanto o de escrita em apenas 2, e assim existe uma escala assimétrica neste padrão (Young, 2010).

Pela Imagem 6 podemos ver um exemplo de como será a comunicação entre os dois tipos de operações (comandos e *queries*). Esta comunicação é realizada através de mensagens, cada modelo tem a sua própria base de dados e não acede diretamente á base de dados de outro modelo. Quando um comando publica a mensagem, o *subscriber* do lado das *queries* vai processar essa mensagem e vai atualizar a sua própria base de dados dedicada e otimizada apenas para leitura (DAHN, 2008).

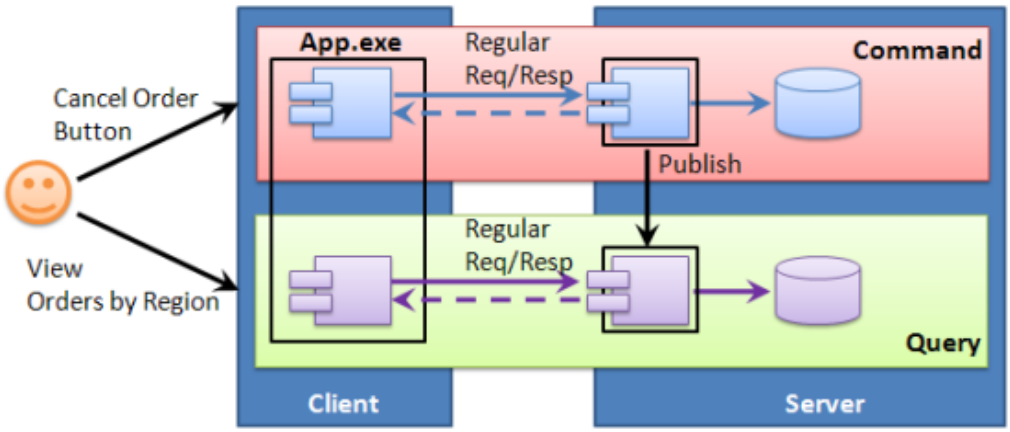

Imagem 6 - Visão de comandos e queries (DAHN, 2008)

Tal como já foi dito anteriormente, existe uma separação da natureza das operações de forma a verificar as mudanças nos objetos. Podemos ver pela Imagem 7 um exemplo prático do objeto antigo constituído pelas seguintes APIs (*Application programming interface*):

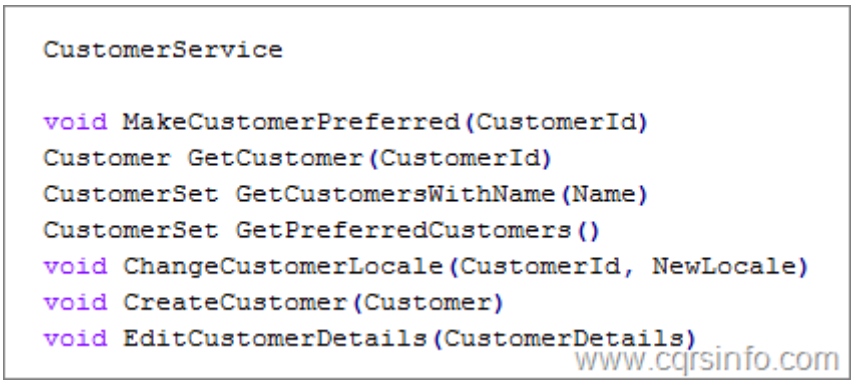

Imagem 7 - Arquitetura do objeto mais comum (Young, 2011)

Mas depois de aplicar o CQRS o resultado será:

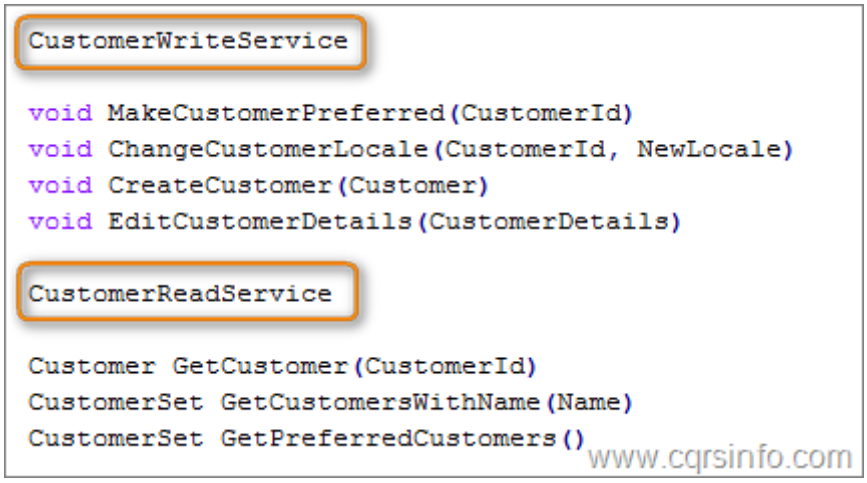

Imagem 8 - Arquitetura depois de implementar o CQRS (Young, 2011)

Pelo que conseguimos ver na Imagem 8, o processo de separação é constituído agora por dois tipos de objetos, o ***CustomerWriteService*** que é responsável por manipular o estado do sistema, ou seja, são comandos que retornam *void*, e outro objeto ***CustomerReadService*** que não deverá realizar alterações, ou seja as *queries*, ao contrário dos comandos, irá retornar um objeto. O processo de mudança é bastante simples o que permite que os clientes e mesmo quem desenvolve saibam exatamente a operação que estão a fazer e consigam separar totalmente os papéis (Young, 2011) (Young, 2010).

3.2 Caracterização de CQRS

3.2.1 Conceitos base

Após esta separação de papéis, é importante ter uma ideia geral de como ficará logicamente separado, quais os componentes base do CQRS e qual o fluxo básico do dia-a-dia de um utilizador com uma aplicação baseada em CQRS (Imagem 9).

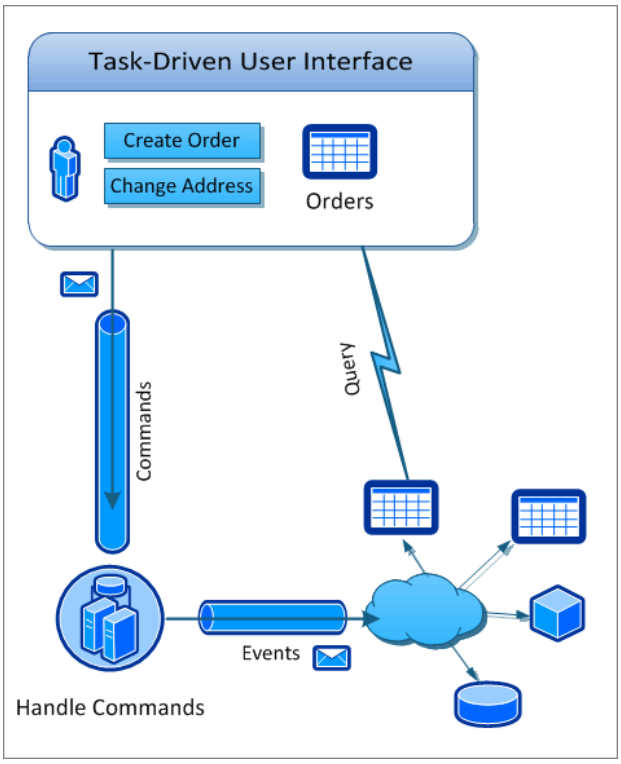

Imagem 9 – Componentes lógicos do CQRS (Abdullin, s.d.)

Pela Imagem 9 para além dos comandos e *queries* é possível verificar que após a interação de um utilizador com o sistema, ele poderá querer consultar algo no sistema (lista de encomendas) e executa uma determinada *query*, que poderá comunicar com várias bases de dados. Se o utilizador quiser modificar algo no sistema (*Change Address* ou *Create Order*) é enviado um comando com o detalhe das alterações para um *handler*, que tem a responsabilidade de identificar o pedido e saber para onde o reencaminhar para ser executado. Posteriormente poderá dar origem a um evento para o resto do sistema, persistindo depois essas mesmas alterações e podendo ser consultadas "quase" de seguida (neste tipo de sistema a consistência imediata de dados é relativa).

A Imagem 10 pretende reforçar outras características relativas à importância dos comandos e *queries* mas também quer manter presente que nem sempre existe consistência de dados.

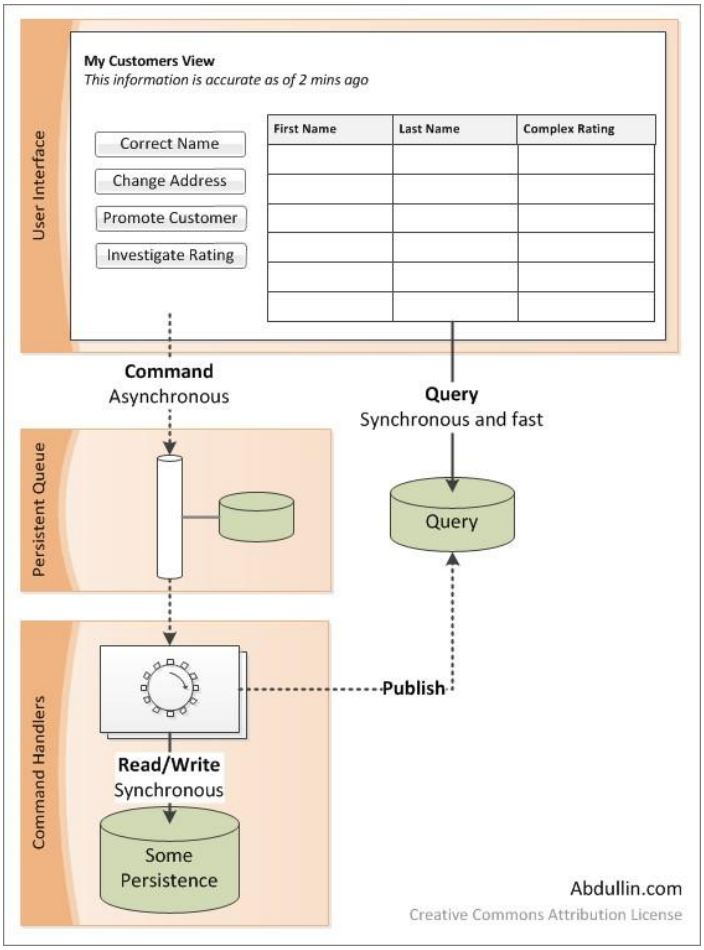

Imagem 10 – Exemplo de fluxo CQRS (Abdullin, 2010e)

Os dados que são apresentados ao utilizador poderão estar incoerentes, devido a não serem feitos pedidos constantes para manter tudo consistente com a base de dados, e daí se visualizar na interface o detalhe temporal a que os dados se referem. Outro conselho dado é que para aparecer ao utilizador todos os dados deve ser feito à base de dados um pedido do género "SELECT * FROM CustomerView Where … " de forma a que as *queries* sejam o mais simples possível e portanto que as respostas sejam mais rápidas. Em relação à alteração de dados é suposto que, por exemplo, quando apenas se quer corrigir um pormenor na morada, não seja preciso enviar sempre todos os dados do cliente juntamente com os novos dados, apenas seja necessário enviar um comando "*ChangeAddressCommand*" para que a única coisa a fazer seja identificar o utilizador em questão e os novos dados, evitando assim a passagem de dados desnecessários na rede.

Após o envio do comando de mudança de morada, este deverá ser colocado numa fila de forma a funcionar assincronamente, para que se o servidor responsável estiver desligado, possa na mesma ser processado posteriormente, sem ser preciso o utilizador fazer a mesma

ação. Quando um comando acaba de ser processado pelo *Command Handler,* então é necessário publicar essas mesmas alterações para que as *queries* possam o mais cedo possível devolver os novos dados. Em termos de tipo de comunicações é importante ter em atenção que as *queries* e os *command handlers* são ***síncronos*** porque tanto um como o outro são processados para aquele instante e necessitam os dois da informação na hora, enquanto a criação de um comando já é ***assíncrona*** pois poderá ser processado mais tarde, reduzindo a complexidade (pode-se dividir em vários comandos) e distribuindo a carga por vários servidores (Abdullin, 2010e).

Com estes factos e pelas Imagem 9 e Imagem 10, pretende-se ***dar resposta a alguns problemas empresariais e aconselhar*** para que o desenho das aplicações e arquitetura possam ser refinados, ajudando a identificar e corrigir certos ***bottlenecks*** **de performance,** visto que divide em duas grandes áreas leitura e escrita. ***Escalar facilmente***, visto que é mais fácil aumentar o número de máquinas apenas para uma determinada operação; ***resolver e prevenir problemas de concorrência***, uma vez que, funcionando com eventos, se algo grave acontecer, será possível repor; ***evitar perda de dados no sistema***, pois está tudo registado como um log detalhado através dos eventos; ***reduzir a complexidade de design***, porque ao dividir em duas áreas permite ter as melhores tecnologias e metodologias que se aplicam a cada área; e por fim torna o ***desenvolvimento e manutenção mais fácil***, visto desenvolver apenas para um modelo do sistema que não irá ter impactos no resto do sistema, sendo a nível de manutenção permitido por exemplo, atualizar faseadamente o *software*. O padrão CQRS tenta lidar com estes problemas tentando refinar o *mindset* atual e permitindo construir arquiteturas de uma forma diferente que ajudará a expansão das empresas a nível do seu sistema atual (Abdullin, s.d.).

Para além das características principais de CQRS já faladas, (Dahan, 2011) existe outra denominada ***Single Responsibiliy Principle*** (SRP), que se baseia em linguagens orientadas a objetos (OO), em que cada objeto encapsula os respetivos dados, não expondo nem permitindo alterar sem qualquer controlo. Em vez disso são utilizados métodos que permitem ter acesso e trabalhar com os dados internos do objeto. Com isto, o SRP permite conduzir-nos para que não existam os mesmos dados em diferentes objetos, ou seja, se o nome de um cliente está em dois objetos, o objetivo será de rapidamente fazer um *refactor* do código de forma a apenas estar centralizado num único sitio e quem necessitar dessa informação saber que apenas existe um único sitio onde o ir pesquisar. Contudo, poderá existir a necessidade de num sistema, o nome do cliente ter de ser exposto tanto na área pessoal do cliente, como na parte do BackOffice, e assim a mesma informação poderá estar em dois sítios, com a principal diferença que poderá estar otimizado, mais, ou menos detalhado ou até ir pesquisar informação a diferentes sítios, e é neste ponto que o CQRS explica que o SRP não deve ser utilizado como se existisse uma única forma de visualizar ou aceder a informação (Dahan, 2011).

Dependendo do tipo de sistema e negócio a que se queira aplicar este padrão, é necessário saber em que nível e até que nível, se deve aplicar estes conceitos, para que não provoque um excesso de divisões quando na realidade não existe essa necessidade ou apenas aplicado a

micro contextos. Um exemplo será a Amazon.com onde, se quiséssemos criar apenas uma única vista dos dados seria extremamente complexo, visto que na página inicial poderemos ver, os produtos mais vendidos, o que temos no carrinho, os comentários de utilizadores, produtos aconselhados, etc. Para além de ser complexo iria ser necessário muita duplicação de informação, por exemplo quando se adicionava um artigo ao carrinho, seria necessário voltar a mostrar os clientes que o compraram, os seus nomes, imagens, etc.

Para combater o problema anterior existe o conceito de ***Autonomous Business Component (ABC)*** que em vez de usar o CQRS no nível mais elevado da arquitetura, de forma a obter toda esta informação, permite usa-lo num nível abaixo, ou seja, ter um ABC responsável pelos nomes dos produtos, outro pelas imagens, outro pelos comentários, etc., utilizando na mesma o CQRS mas incorporado em cada ABC. Assim, consegue-se uma imagem para o utilizador com toda a informação, e não apenas uma vista da informação. Em termos de performance permite em cada ABC fazer *caching*, e como cada ABC tem uma área específica torna-se mais fácil criar esse cache e geri-lo, aumentando assim a performance, mesmo separando um pedido (*querie*) à base de dados, em vários. Com o CQRS em cada ABC torna-se mais fácil guardar em histórico cada alteração dos dados, visto que cada ABC é responsável por uma parte do sistema e assim não existem grandes quantidades de informação para guardar em histórico de uma só vez. O facto de guardar em histórico fornece um novo conceito denominado ***upserts*** que é muito usado em CQRS, que se baseia em fazer um *update* e um *insert* de forma a atualizar dados e introduzi-los em histórico, tornando-se muito importante porque é necessário sempre saber quando é que uma mudança aconteceu (muito usado em *event sourcing*) (Dahan, 2011).

Existem ainda duas noções importantes para a utilização de CQRS. Uma será a criação de uma ***arquitetura circular*** (*Circular Architecture*), que se baseia em ter as leituras e escritas de forma desacoplada de forma a utilizar diferentes modos de comunicações, com mensagens assíncronas (comandos e eventos) e *queries* síncronas. Outra será o recurso a eventos (*Event Sourcing*), que permite fazer persistir as mudanças através de uma forma contínua e temporal, sabendo sempre num determinado momento em que estado é que o sistema se encontrava (Abdullin, 2010h).

Após compreender os conceitos base de CQRS torna-se necessário detalhar em que consistem as *queries* e comandos, o que irá ser realizado nos próximos dois subcapítulos.

3.2.2 Queries

O modelo das *queries* está muito relacionado com o facto dos sistemas precisarem de fazer o ***report*** de dados de forma a suportar um grande número de utilizadores em simultâneo e ser o modelo com mais uso devido ao grande número de pedidos efetuados constantemente. Pedidos que poderão ser para consumo no próprio sistema como para outros serviços externos. As *queries* permitem fornecer informações acerca do estado do sistema num momento específico, que poderá estar dependente de um determinado contexto. Estas

queries deverão ter a sua própria fonte de dados, baseada na principal do sistema mas otimizada para este efeito. A forma de atualizar a fonte de dados das *queries* poderá ficar a cargo de quem fizer atualizações na base de dados principal, ou seja, aquando da passagem pelo domínio. Nesta fase, ao atualizar a base de dados principal, envia uma mensagem com os novos dados, que deverão ser consumidos mais tarde.

Quando numa situação um utilizador precisa de ver algo no ecrã, este pedido deverá ser feito através do modelo das *queries* que deverá apenas retornar dados, e neste caso teremos um DTO (*Data Transfer Object*) para o fazer. No DTO, a maioria das vezes os dados serão apresentados diretamente no ecrã do utilizador (Nijhof, 2009). A grande preocupação que existe com este tipo de abordagem é o facto dela só permitir que os dados sejam utilizados para a leitura, o que leva a serem geralmente específicos para a situação em que estão envolvidos (domínio), logo se queremos apresentar dados ao utilizador de outro domínio ou transformar os DTOs recebidos para apresentar ao utilizador, é necessário efetuar o respetivo mapeamento de dados, o que causa muitas preocupações do lado da manutenção de objetos, e por isso os DTOs poderão ter comportamento associado em vez de uma estrutura adequada (Fossmo, 2009).

Os problemas mais comuns, foram identificados em vários domínios (Young, 2011): os métodos de leitura ***expõem o estado interno dos objetos*** referentes apenas a um determinado domínio, o que faz com que os DTOs não sejam genéricos; o uso de ***caminhos pré-determinados*** na leitura de *use cases* em que é necessário fazer o carregamento de mais dados, visto poder ser usado ORM (*Object-relational mapping*); o ***carregamento de várias fontes*** para construir um DTO, leva a que o modelo das *queries* não seja o ideal para leituras, pois exige um tempo de resposta maior; um DTO poderá ficar bastante confuso e complexo se for construído mediante muitos limites (vários domínios) (Young, 2011). De forma a combater alguns problemas de performance e aumentar a separação de responsabilidades devido ao seu uso frequente, são muitas vezes ***desnormalizados os dados***, tentando que cada *querie* que o utilizador possa fazer se reflita numa tabela. Com isto poderá existir algum tipo de duplicação de dados contudo cada um está otimizado para uma determinada função específica, sendo que o responsável por fazer atualização á fonte de dados das *queries* tem uma obrigação acrescida de saber a forma correta de o fazer (Nijhof, 2009).

Por estas razões não se deve introduzir conceitos nem o próprio domínio, na elaboração dos DTOs, criando assim uma nova maneira de desenhar os DTOs.

A Imagem 11 pretende ilustrar o que as *queries* trazem de novo na arquitetura.

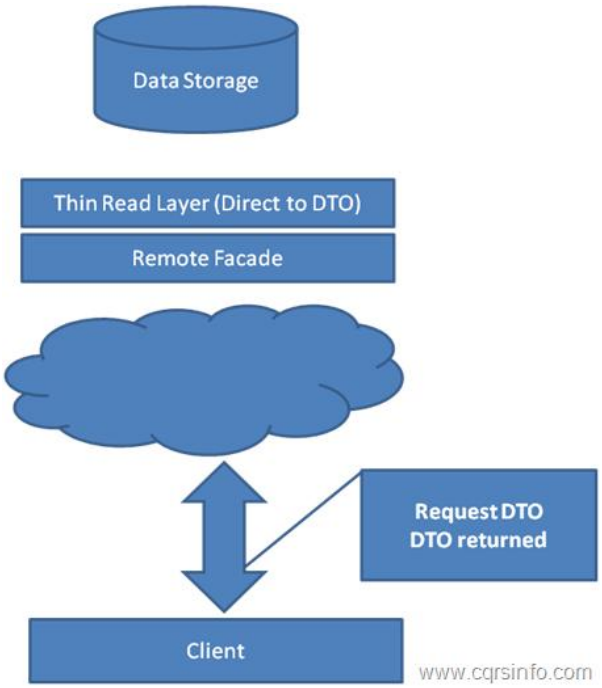

Imagem 11 - Queries, "Thin Read Layer" (Young, 2011)

Na Imagem 11, a camada "***Thin Read Layer***" tem o objetivo de ler diretamente da base de dados e criar os seus DTOs. Tal como já foi referido anteriormente, isto poderá ser feito através de ORM, ou através de uma outra abordagem em que permite fazer o mesmo objetivo que os ORMs. Em qualquer uma das opções, dever-se-á ter em linha de conta que os DTOs devem ser simplificados, o mais possível. A camada *Thin Read Layer* não tem de estar necessariamente isolada da base de dados nem é prejudicial, se estiver acoplada à base de dados, tudo depende da arquitetura que já existe ou do facto de estarmos a desenhar um novo sistema. A *Thin Read Layer* não deve ser complexa em termos de código visto que poderá com o futuro ser muito difícil de manter e o mais importante, tem de ser fácil de otimizar. Na parte de desenvolvimento do modelo das *queries*, apenas é necessário conhecer o modelo de dados não sendo obrigatório conhecer o resto do sistema nem que ORMs é que são utilizados, ***facilitando a modularidade do sistema***. (Young, 2011)

3.2.3 Comandos

3.2.3.1 Modificação de dados através de comandos

Pela estrutura do CQRS quando se deseja alterar o estado do sistema é necessário fazê-lo através de comandos, vejamos um exemplo dessa razão.

Existe um utilizador a ver o seu perfil num *site* de compras *online*. Esse utilizador deseja alterar no seu perfil, dados básicos como a morada, estado civil, etc. A certa altura no próprio *site* tem um limite de crédito que pode ter e o qual o utilizador quer aumentar para um valor superior. Mais tarde o departamento de faturação despoleta um evento em que diz que aquele mesmo cliente tem uma grande divida no mesmo *site*, e após ocorrer este evento o utilizador submete as alterações feitas no seu perfil. Após este acontecimento é necessário decidir o que é suporto fazer. Aceitam-se aceita as alterações? Tomam-se outras opções como por exemplo, bloquear a conta? Ou simplesmente, não se aceitam as alterações? O ideal poderá ser aceitar umas opções e rejeitar outras. Não poderá aumentar o limite de crédito uma vez que já é devedor, mas poderá alterar informações básicas do seu perfil, sem problema. Estas verificações em termos de negócio muitas vezes podem ser demasiado complexas ou quase impossíveis de conseguir passar para o código, daí a necessidade da existência do conceito de "***command***".

Cada comando representa um pedido para manipular o estado do domínio, ou seja, alterar alguns dados que fazem parte da aplicação (Tom, 2010). É necessário realçar que cada cliente envia o seu comando para o servidor, não o publica imediatamente em todo o sistema. A publicação do comando poderá ficar reservada para os eventos, que determinam que algo aconteceu, sendo que quem envia não se preocupa com o que o recetor recebe, quando o recebe (Dahan, 2009).

De forma a otimizar a realização dos comandos, poderá ser necessário repensar o *design* da interface com o utilizador, de forma a conseguir determinar quais são as suas intenções, ou seja, estudar o caminho de onde ele estava, para onde ele foi e para onde se pensa que ele irá, sendo isto denominado de ***Task Based UI (User Interface)***. De salientar que o CQRS *não obriga* à implementação de *Task Based UI*, podendo na mesma ser implementado sobre ações de interface CRUD. Porém, como não existe separação de modelos, o CQRS torna-se muito mais complexo. A utilização de *Task Based UI* faz com que os objetos que circulam no sistema, para além de conterem todos os dados necessários, possam conter também comportamento do utilizador. Em termos de negócio, isto pode ser importante, para que a *posteriori* se possa determinar como podem os utilizadores interagir com o sistema e também, como se pode melhorar o próprio sistema. Assim, seguindo e exemplo anterior, é possível distinguir entre o que é alteração dum limite de crédito e a alteração de dados básicos no perfil dum utilizador (Young, 2010). Ainda de forma a apoiar esta implementação poderia existir um sistema assíncrono que comunicava ao utilizador o estado real de uma ação (um determinado comando), ou seja, iria fazendo uma atualização do estado de cada pedido de alteração (comando).

Em relação aos ***comandos e o facto de serem autónomos,*** essa autonomia, deve-se ao facto de não ser necessário um processamento imediato e poderem entrar para uma ***queue***, sem necessidade de se disponibilizar uma ligação constante com o cliente. Em termos de performance é uma questão de *Service-Level Agreement* (SLA) e não uma questão de arquitetura. Com esta autonomia caso ocorram problemas (o sistema da base de dados estar em baixo por exemplo), o cliente pensa que o pedido foi totalmente processado sem se

aperceber que no sistema o pedido ficou em espera, e só será processado realmente mais tarde (Dahan, 2009).

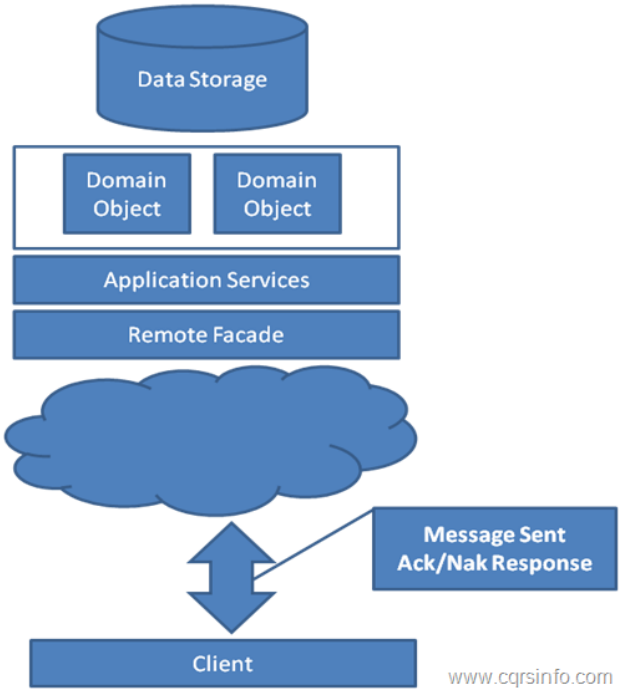

Imagem 12 – Estrutura de Commands (Young, 2011)

Pela Imagem 12, temos um exemplo de como poderia ficar uma arquitetura, sem qualquer modelo de leitura, e que permitisse apenas a escrita. Outro ponto já falado, é que o retorno é um simples ACK (*Acknowledgement*) não retornando quaisquer dados para o cliente.

Em relação aos ***domain objects***, que representam entidades lógicas no sistema e contêm lógica de negócio, eles já não precisam de expor os estados internos, visto existir agora apenas um modelo de escrita que deverá ser o único a conseguir ter-lhes acesso (Young, 2011).

3.2.3.2 Validações de comandos

Os ***comandos necessitam de validação***, e como tal a falta de validação, pode ser uma das razões para que os comandos falhem. Algumas das principais questões quanto a estas validações dizem respeito a serem apenas na UI (*User Interface*), a quem e a que pertencem e de que modo é que a validação e o negócio se incorporam no CQRS (Abdullin, 2010c). Tanto pode existir ***validação de negócio*** (dependem do contexto) ***como de dados*** (independente do contexto), daí que os comandos devem ser validados tanto do lado do cliente como do lado do servidor (Dahan, 2009). No exemplo anterior, se não fosse o departamento de faturação o

comando teria sido aceite sem problema nenhum, mesmo este comando sendo válido, ou seja tendo tipo de dados corretos. Vejamos com mais detalhe esta questão na Imagem 13:

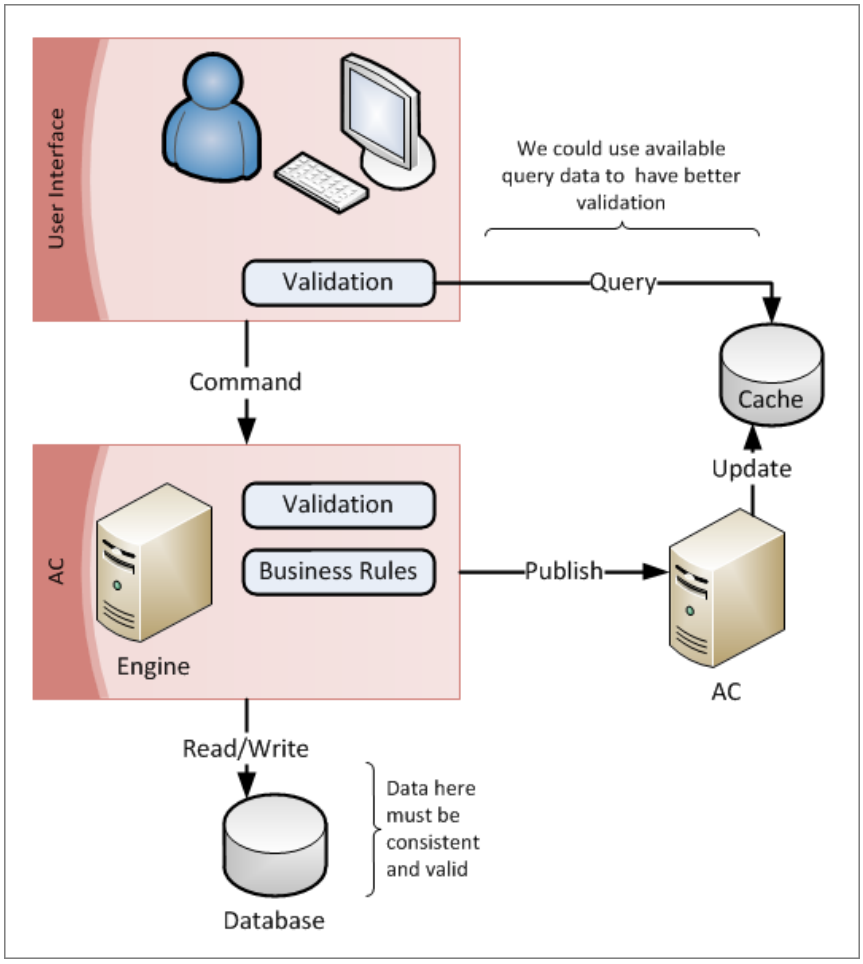

Imagem 13 – Validação de comandos (Abdullin, 2010c)

A utilização deste mecanismo torna necessário a ***reestruturação de interfaces e comandos tendo em conta validações.*** Essa reestruturação é útil para aumentar a performance do sistema. Para tal, é essencial começar pela UI e como ela comunica com o resto do sistema. Quando se fazem validações a nível de interface, podemos por exemplo, apresentar ao utilizador, quando ele está a preencher o campo código postal, apresentar-lhe um *auto-complete* para conseguir garantir uma escolha válida para posterior passagem para o comando (Dahan, 2009). Na UI apenas se consegue validar os dados no momento da introdução e com um contexto limitado, o que significa que pode não corresponder a todas as validações e requisitos da organização mas no mínimo existe a confiança que fazem sentido (Abdullin, 2010c). Além disso, se passarmos apenas o ID da rua em vez do nome completo, é possível fazer comandos mais pequenos, além de aumentarmos um pouco a complexidade para um ser humano identificar à primeira vista o que se está a passar naquele comando. Isto apenas irá falhar caso a rua que se tente introduzir já tenha sido removida antes e só nesse caso dará um erro ao utilizador (Dahan, 2009).

Em CQRS a validação pode ocorrer em várias regras da UI como através de *queries* de forma a terem maior certeza do que se está a validar, mas no final independentemente da complexidade, o objetivo será sempre ***evitar que o utilizador envie comandos inválidos*** para o servidor. Quando todas as melhorias a nível de interface estiverem concluídas, teremos uma alta probabilidade do comando estar correto. Deve-se sempre pensar em validar o máximo possível mas tendo em atenção que não têm de ser validações de consistência a 100% naquele exato momento, visto que teremos sempre que pensar que a validação na interface necessitará de ser ***simples e escalável***. Existem duas grandes razões para serem escaláveis: quase todos os cálculos são do lado do cliente e a validação com base nas *queries* é já por si escalável devido á sua natureza (devido também ao facto dos comandos serem executados com muita certeza).

Após a passagem das validações da UI, poderá passar para sistema de mensagens, chegando ao respetivo *command handler*. Aí, deverá correr novamente a validação local através de regras e se possível integrado numa transação, porque quando existem sistemas de larga escala não se pode confiar na coerência imediata dos dados, existindo a necessidade de ter uma grande certeza do comando que se vai realmente executar. Com isto, conclui-se que é necessário fazer correr tanto a ***validação a nível do cliente como a nível do servidor*** (Abdullin, 2010c).

Existem **razões para um comando válido falhar**, tomando por base o exemplo anterior, caso o evento do departamento de faturação chegasse em segundo lugar, relativamente à alteração de dados efetuada pelo utilizador, deveria existir algum mecanismo que validasse este evento, ou seja, o utilizador poderia ter alterado o limite de crédito e só depois ser-lhe comunicada a informação que não podia. Muitas vezes é enviado um *email* ao utilizador avisando-o da nova situação em que se encontra e a razão. A maioria dos sistemas prefere dar ao utilizador uma mensagem de sucesso para depois internamente, nem que seja assincronamente fazer as suas próprias validações e caso seja necessário, reverter o que foi feito (Dahan, 2009). Na maioria das vezes se a validação falhar o comando é totalmente descartado, porque não esperamos que alguns tipos de erros cheguem ao servidor, como por exemplo, o nome de um utilizador errado. Conseguindo-se dessa forma reforçar a segurança sem penalizar o lado do cliente (Abdullin, 2010c).

É importante também referir que para a validação funcionar é necessário levar em linha de conta toda a ***lógica de negócio*** associada ao sistema. Alguns exemplos serão os códigos postais dos Estados Unidos que são preenchidos apenas por pessoas que morem nos Estados Unidos, os utilizadores de um determinado país recebem um bónus monetário ao se registarem, etc. Para este tipo de validações acontecer na UI tornava todo o sistema muito complexo e pouco escalável devido às longas validações que seriam necessárias executar para o negócio e por isso, no CQRS este tipo de validações são executadas ***no lado do servidor***. Com isto, é possível, por exemplo, no caso de ser necessário dar um bónus a um utilizador, que o sistema envie um *email* com todos os requisitos e políticas do bónus no qual deve confirmar com um *link* respetivo e assim validar todo o processo a nível de lógica de negócio.

Em termos de ***performance*** esta lógica de negócio poderá trazer um aumento da carga o que se poderá tornar num *bottleneck*, só que no CQRS é possível não ter este tipo de problemas devido: *aos comandos serem postos numa queue* e serem apenas depois processados; aos comandos *poderem ser executados em diferentes servidores*, baseados nos *aggregater key* (*sharding*); *às mensagens* garantirem que os comandos irão ter a sua vez de serem processados, mesmo em casos extremos de base de dados bloqueada ou o servidor em baixo (Abdullin, 2010c).

No próximo subcapítulo são enunciadas quais as vantagens da utilização de CQRS mediante a caracterização que foi feita anteriormente.

3.2.4 Benefícios

Em termos gerais as vantagens da aplicação deste padrão são:

- Quando o *Domain Model* é bastante complexo, o CQRS ajuda na abordagem, porque permite dividir o que é leitura do que é escrita, criando dois modelos distintos, cada um tendo as suas próprias características;
- Aumentar a facilidade de gestão de aplicações de alta performance, visto que ao separar as leituras das escritas permite separar a carga de trabalho de cada um e isso, permite escalonar cada um separadamente;
- Utilizar otimizações apropriadas para áreas diferentes, ou seja, otimizar a parte dos comandos é diferente de otimizar a parte de leitura, portanto as respetivas técnicas deverão ser diferentes. Exemplo: Utilizar *caching* para *queries* e *queues* para processamento balanceado de comandos;
- Criar vários objetos DTO que permitam ter apenas os dados necessários para visualização, evitando os mapeamentos e os inúmeros pedidos à base de dados para obter todos os dados (Fossmo, 2009);
- O CQRS, diminui a carga para as leituras, o que permite ganhar performance na parte das escritas, e assim, há menos probabilidade de atingir um *deadlock*. (Abdullin, 2010d);
- Este padrão torna possível escalar apenas uma parte, ou seja apenas a parte da leitura que é onde geralmente recai a maior parte da carga do sistema (Fossmo, 2009);
- Possibilita o registo de todos os acontecimentos no domínio, ou seja, permite ter todos os registos dos comandos e dos eventos que aconteceram no domínio (Fossmo, 2009);

- Ao ter pequenos módulos, torna-se tudo mais fácil de testar devido aos limites existentes; (DAHN, 2008);

- Facilita a divisão dos métodos da arquitetura CRUD para CQRS, podendo isto, ser feito faseadamente;

- Possibilita a aplicação apenas a uma parte do sistema, visto que cada sistema tem vários contextos (*Bounded Context*) e cada uma deles tem os seus próprios mecanismos e modelações. (Fowler, 2011);

- O CQRS traduz-se nas propriedades que permite adicionar à arquitetura, integrando os dois tipos de modelos (Young, 2010);

- Permite escalar de forma simples, com poucos custos a nível de *deployment* na *cloud* e possibilita a adaptação das mudanças de negócio ao sistema (Abdullin, 2010g).

- Aumentando a performance, este padrão, permite aumentar também a experiência do utilizador. (Abdullin, 2010d)

- O padrão CQRS tem tendência a funcionar porque transcende a tecnologia, fazendo com que a tecnologia e o *software* nos beneficiem, e não o contrário (Oliver, 2011b);

No próximo subcapítulo será mostrada em detalhe, a arquitetura que o CQRS aconselha e quais as diferenças para outras arquiteturas.

3.3 Arquitetura CQRS

3.3.1 Introdução

Para além dos refinamentos a nível de APIs também é necessário compreender a razão da renovação do *design* dos sistemas e o que se pode melhorar noutro tipo de implementações de arquitetura. Do ponto de vista de arquitetura atual, temos a seguinte estrutura:

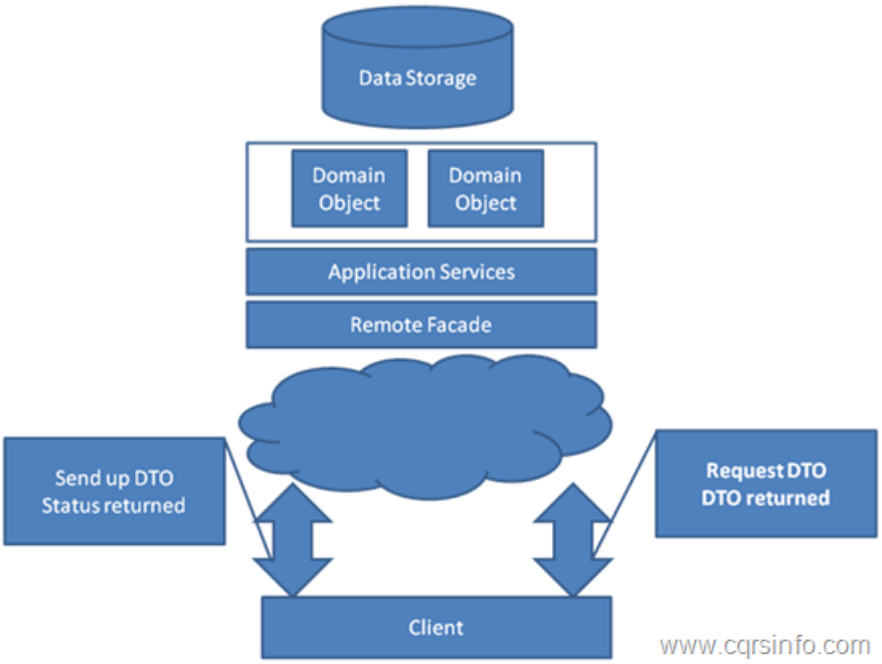

Imagem 14 – Exemplo de arquitetura (Young, 2011)

Pela Imagem 14 é possível ver que existem dois DTOs a serem usados, um deles para fazer o pedido e outro para devolver dados ao cliente. Nesta arquitetura o que se pretende é usar os típicos métodos CRUD sobre a base de dados, ou seja, não existe qualquer divisão de papéis nem responsabilidades. Quando é necessário esquemas de consulta mais complexos como agrupar vários registos, formar um registo virtual apenas para apresentar ao utilizador ou aceder a varias localizações de dados, o sistema poderá tornar-se mais complicado e moroso, visto que o cliente ao fazer o pedido, é necessário (geralmente) estar a espera que seja processado, mesmo que seja dividido em várias máquinas, e seja acedido a todos os locais onde a informação está, retornando um objeto complexo que poderá ter dados a mais que não são necessários para o que é pretendido.

Além disso a arquitetura da Imagem 14 poderá trazer algumas limitações visto que não existe a divisão de papéis bem definida, ou seja, não permite de uma forma fácil e direta otimizar por exemplo, a capacidade de consulta sem por outro lado deteriorar as escritas. O objetivo a aplicar a esta arquitetura será o de executar uma separação em comandos e *queries* de forma

a refinar o que é necessário conseguindo devolver de uma forma mais simples, utilizando o melhor de cada componente, ou seja, as *queries* poderão ter por exemplo, uma alta capacidade de *caching* enquanto os comandos têm uma fila para os processar tentando garantir ao máximo que serão executados (Young, 2011).

O refinamento das arquiteturas para aplicarem o padrão CQRS tem como um objetivo, para quem as aplica, conseguir ***baixar os custos***, tanto de implementação como da manutenção, sendo este um dos principais fatores decisivos para as empresas. Este fator poderá ser alcançado através da criação de duas grandes áreas, uma de leitura e outra de escrita, permitindo aplicar a cada uma delas uma forma específica de implementação, para que possam ser realmente competentes naquilo para que estão destinadas. Ao serem aplicadas as melhores formas a cada área isso vai permitir entre outras possibilidades, que as *queries* possam aumentar a performance, o que reproduzirá uma diminuição de necessidade de recursos para o funcionamento do sistema.

Conseguindo umas arquiteturas mais aprimoradas, consegue-se ter um melhor ***isolamento dos conceitos de negócio*** associados, permitindo a quem desenvolve abstrair-se da aplicação de certas regras de negócio. Nos piores casos o custo inicial e a curto prazo é igual a ter a antiga arquitetura. Para que se consiga compreender em detalhe como será a alteração da arquitetura antiga (Imagem 14) para uma com aplicação de CQRS (Imagem 15), é necessário perceber como é que a divisão da natureza das operações funciona e se devemos ou conseguimos tratar estas questões como dois sistemas separados (Young, 2011).

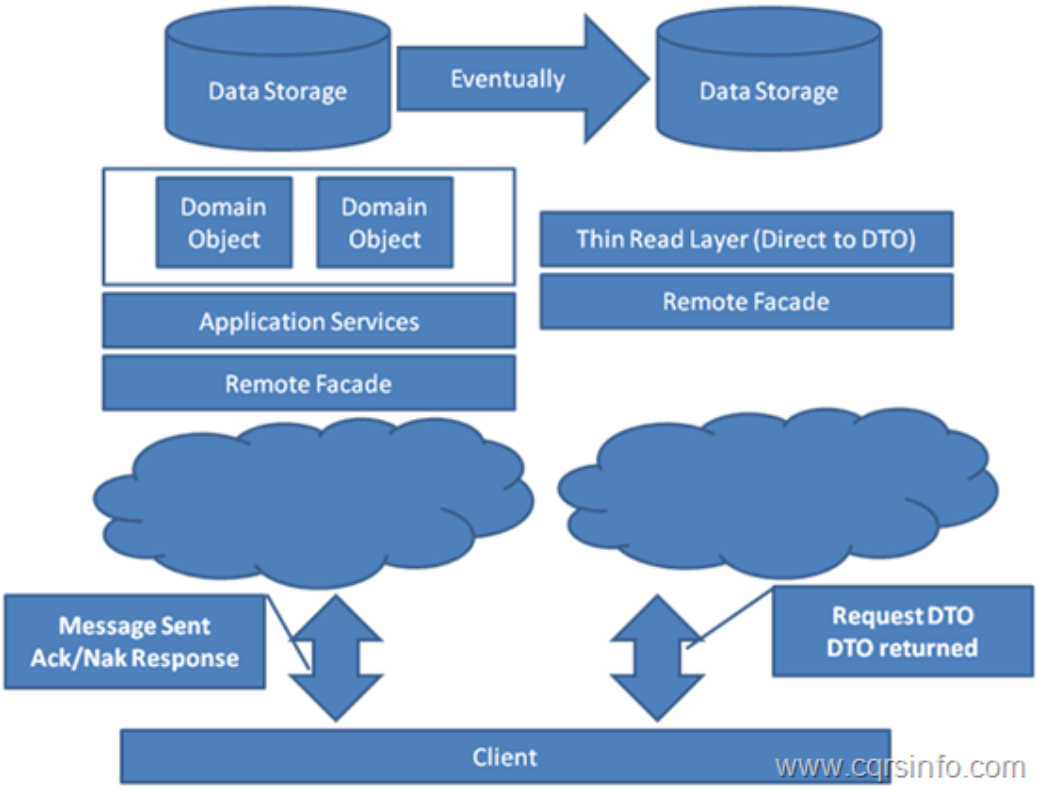

Imagem 15 - Arquitetura final dos commands e queries. (Young, 2011)

Pela Imagem 15 é possível confirmar duas grandes áreas: do lado esquerdo, a parte dos comandos, visto para o cliente não retornarem dados e serem aplicados conceitos de negócio e do lado direito, a parte das *queries*, a qual retornam DTOs. Na área das *queries* é de notar a existência de um novo conceito denominado "*Thin Read Layer*", no qual reside o objetivo de ler dados diretamente da base de dados e projetar esses dados em DTOs, e além disso também é de salientar que o domínio foi ultrapassado ficando mais simples assim o retorno de dados. O "*Thin Read Layer*" não é um conjunto de código complexo que dificilmente é possível sustentar, existindo assim dois benefícios: a sincronização direta com o tipo de dados e a otimização facilitada das *queries*.

Alguns problemas que eram comuns a toda a arquitetura agora passaram a estar bem definidos e delimitados, o que faz com que o modelo dos comandos sofra uma diminuição ou até extinção de alguns problemas. Um exemplo seria o número elevado de leituras que eram precisas fazer para a construção dos DTOs baseados em objetos de domínio, o que aumentava a complexidade e o processamento de conceitos de negócio; e a necessidade de carregar várias entidades de negócio para construir um único DTO, o que traz problemas de performance ao não serem construídas *queries* otimizadas. Desta forma apenas os comandos lidam diretamente com os "*Domain Object*", permitindo que exista um menor *overhead* do sistema.

Em relação à interligação da base de dados para sincronização, existe um tipo de implementação de outro padrão de forma a interligar duas fontes de dados, denominado de ***Event Sourcing***, como se pode ver na Imagem 15 (parte superior). Este padrão, consegue manter as bases de dados sincronizadas e otimizadas cada uma para o seu objetivo. Com isto é possível possuir uma base de dados na 1FN (Primeira forma normal) e outra na 3FN (Terceira forma normal) de maneira a otimizar o processo de leitura ou escrita. (Young, 2011) Com esta separação é ainda possível possuir uma base de dados real-time para as *queries*, que irá funcionar como uma base de dados de ***reporting*** (Fowler, 2011).

3.3.2 Aplicação detalhada

De seguida é apresentado numa arquitetura, um fluxo de trabalho de um utilizador, a qual pretende evidenciar mais detalhes sobre as transformações para CQRS e o que se deverá ter em conta.

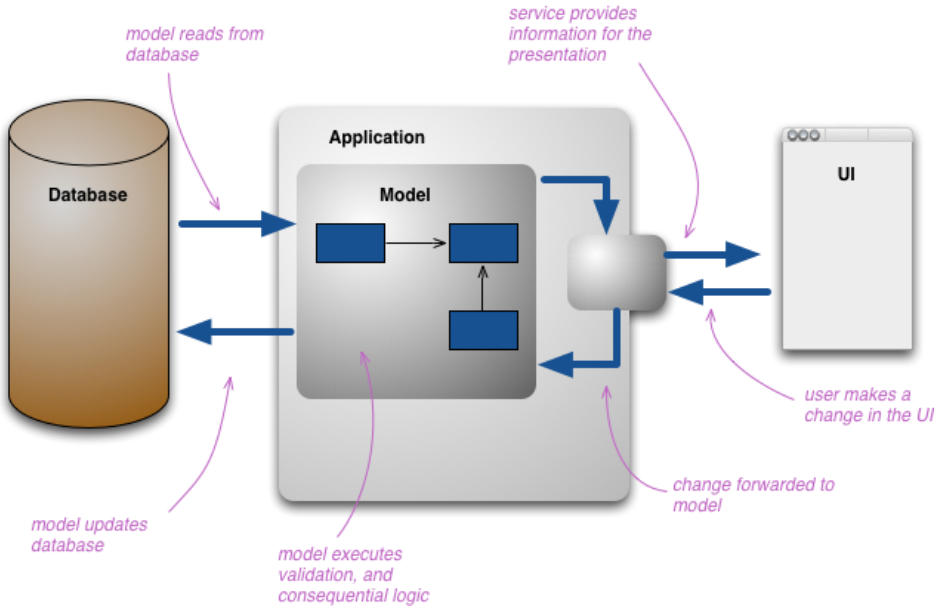

Imagem 16 - Arquitetura baseada em CRUD (Fowler, 2011).

Pela Imagem 16, pressupomos que poderão existir várias representações da mesma informação, por exemplo a representação na base de dados, sendo esta organizada de forma ORM e a que é fornecida ao utilizador final, sendo toda a informação disposta de forma organizada e humanamente percetível. Quando é utilizado o *Domain Model* (como neste caso), geralmente a base de dados é muito semelhante ao modelo conceptual, ou seja, todos os conceitos/entidades são representados na base de dados da mesma forma que eles existem no *Domain Model* (incluindo as suas relações) ou é o mais parecido possível. Por isso, este tipo de arquitetura torna muitas vezes complicado o trabalho de manutenção, porque se por um lado tentamos que os dados estejam tão bem organizados quanto possível, a nível de base de dados, a nível de interface e de pesquisa não está otimizado para tal (Fowler, 2011).

Tal como foi referido anteriormente, as mudanças que o CQRS introduz passam pela separação total deste modelo, em dois, e assim a interface com o utilizador passará a utilizar um modelo para visualizar os dados e outro para fazer persistir as mudanças nos dados. Esta diferenciação baseia-se no facto de em ambientes com domínio bastante complexo se tornar muito difícil possuir apenas um modelo para duas naturezas de operações totalmente diferentes.

A Imagem 17 pretende ilustrar em que consiste a mudança na arquitetura.

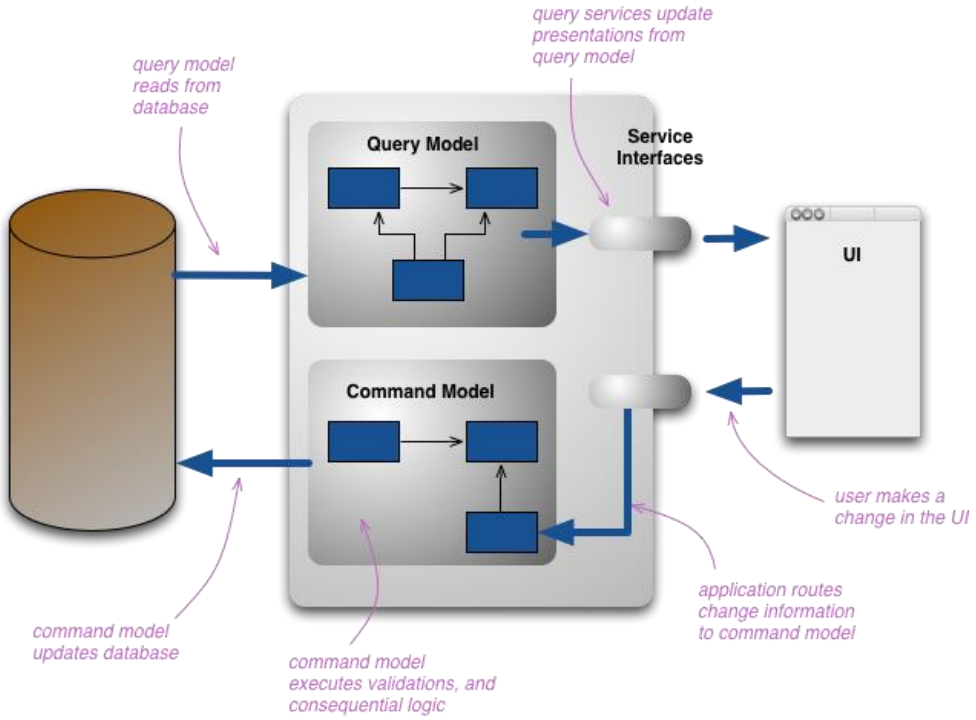

Imagem 17 - Aplicação de CQRS a uma arquitetura CRUD (Fowler, 2011).

Podemos ver pela Imagem 17 um exemplo da aplicação de CQRS à arquitetura da Imagem 16. Aí, verificamos que podemos ter um *site* web em que um utilizador vê a página inicial usando para isso as *queries*. Quando decide alterar algo no seu perfil, o pedido de alteração é direcionado para um comando e o seu resultado é comunicado novamente ao utilizador através de uma *querie*. (Fowler, 2011) Cada operação tem necessidades específicas, a nível de ***armazenamento de dados*** os *comandos* podem ter um gestor de transações para que possam dividir os trabalhos por várias máquinas e normalmente a organização dos dados na base de dados são sobre a 3FN enquanto as *queries* requerem o acesso aos dados o mais rápido possível, ou seja, sem utilizar processamento complexo, como os *joins* das base de dados e por isso a 1FN é o ideal. A nível de ***escalabilidade***, os *comandos* na maioria dos sistemas e dando o exemplo de *websites*, têm um número de transações baixo e geralmente são pequenas por isso a escalabilidade dos comandos não é tão crítica. Ao contrário das *queries*, que são a operação de leitura mais frequente e por isso têm a maior percentagem de pedidos, pelo que a sua escalabilidade é muito importante.

Outro exemplo de uma situação comum será a representada na Imagem 18.

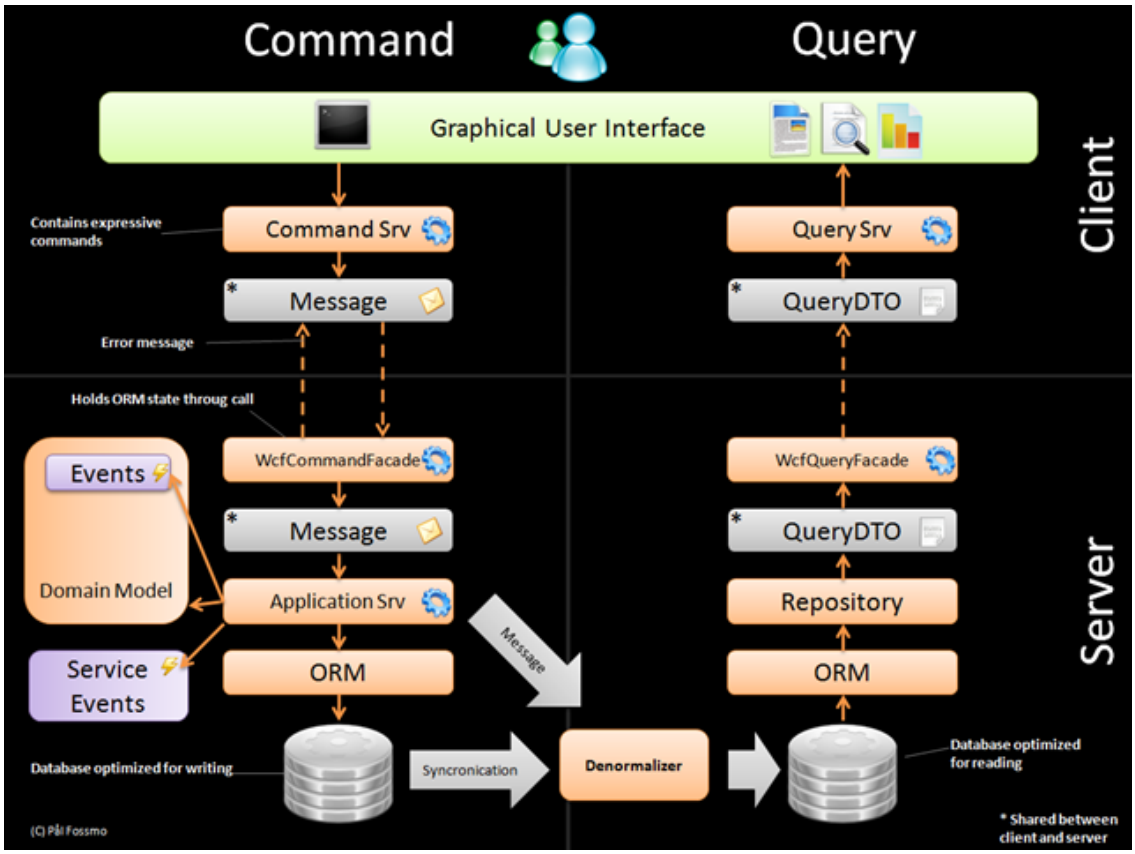

Imagem 18 – Exemplo de arquitetura CQRS (Fossmo, 2009)

Na Imagem 18 é possível observar que um utilizador poderá entrar na aplicação e ver a página inicial, através da execução das *queries* necessárias. Neste caso utiliza os serviços WCF (*Windows Communication Foundation*) e lê da base de dados (desnormalizada para promover as leituras rápidas) transformando um DTO, em que apenas que retorna os dados necessários para a UI. A navegação do utilizador entre todas as páginas utilizará o mesmo processo. A determinada altura o utilizador decide alterar algo, e assim é criada uma mensagem para um comando contendo os dados que serão necessários alterar na vista (lado esquerdo da Imagem 18). Posteriormente será necessário validar o comando e para isso terá de seguir para o *Domain Model* para ser validado consoante as regras de negócio. Caso dê erro uma mensagem é retornada sem dados, apenas com o respetivo erro. Se não der erro, então a mensagem persistirá na base de dados e o pedido de sincronização para as bases de dados de leitura é enviado. Com isto, torna-se mais fácil o *logging* de tudo que se passou com um determinado comando, ficando facilmente visível o que o utilizador quis fazer não se limitando apenas ao resultado das suas ações. Na maioria dos casos é apenas necessário acrescentar uma parte à arquitetura com as *queries* (Fossmo, 2009).

Visto que ao implementar o CQRS não será necessário uma mudança radical no sistema existente, é necessário analisar como este se consegue interligar com outros padrões e sistemas já existentes: A ***mudança da representação em CRUD para os comandos e queries***,

permite que se evolua também para uma interface baseada em tarefas, ou seja, consegue-se uma solução simples, orientada a tarefas em que o utilizador consiga visualizar o que está a fazer e o que irá fazer a seguir, sem ter dúvidas, sobre como o fazer, ou do que está realmente a ver (Corporation, Microsoft, 2011); A ***atualização e mudanças de estado*** ao serem feitas pelo modelo dos comandos permitem aplicar um sistema de eventos, ou seja, guardar sequencialmente todas as modificações de uma aplicação criando no fim uma série de eventos, que poderá permitir reconstruir o passado através do log de eventos, etc. (Fowler, 2005b); Ao ter ***dois modelos separados*** poderá criar alguma dificuldade em garantir a consistência dos dados, visto estarmos a lidar com sistemas distribuídos (Fowler, 2011); A utilização de ***Domain Model***, é usada porque contém lógica de domínio complexa, sejam as validações, as consequências e as derivações. A maioria das vezes é apenas necessária quando estamos a usar um comando, eventualmente a atualizar dados, tornando-se necessário simplificar a parte das *queries*.

3.3.3 Eager Read Derivation

O modelo de leitura deverá evitar ligar-se diretamente à base de dados principal, passando a ligar-se a uma, apenas para esse efeito, denominada de *reporting*, introduzindo o conceito de ***Eager Read Derivation.*** Este conceito determina que quando é recebido um pedido de leitura, esse pedido é direcionado para a base de dados principal para devolver dados, percorrer a lógica de domínio (caso exista, mas a evitar) e retornar o resultado, não existindo ligação à base de dados principal, ligando-se alternativamente às bases de dados de *reporting* que já estão estruturadas para leituras. Qualquer pedido de leitura irá ser tratado por estas base de dados para ler os dados diretamente sem ter que passar por lógica de domínio (Fowler, 2011).

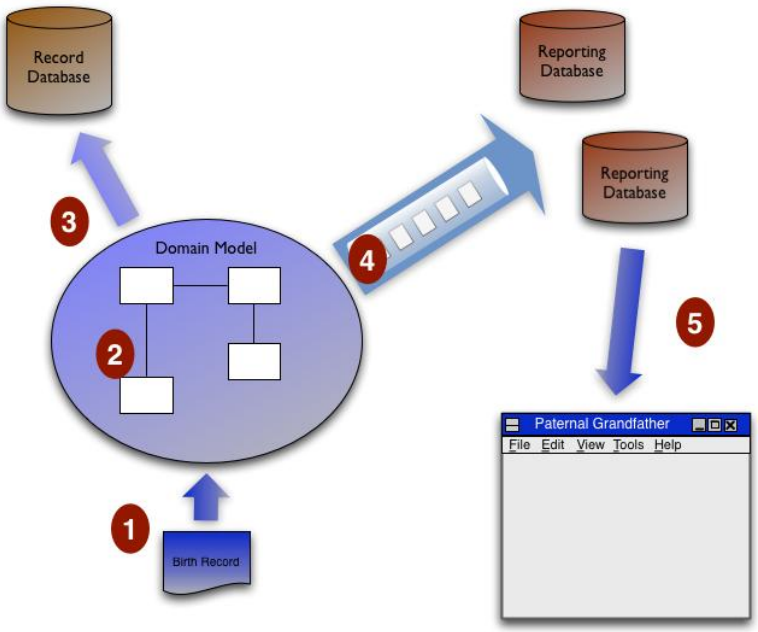

Imagem 19 - Eager Read Derivation (Fowler, 2009)

Pela Imagem 19 podemos ver um exemplo de *Eager Read Derivation.* Aqui, um pedido de alteração de dados, recebido (1), eventualmente despoletado por uma interface, é validado e processado pelo *Domain Model* (2), ficando registado na base de dados principal (3), a qual contém todos os dados devidamente estruturados para este efeito. No entanto o *Domain Model* ainda tem outra ação no sistema: enviar mensagens de atualização para uma *queue* para serem consumidas pelas bases de dados de *report* (4), as quais são responsáveis por consumir a sua parte para atualização dos dados. Aquando de um pedido do utilizador para visualizar esse mesmo registo, este é diretamente pedido às bases de dados de *report* (5) o que torna assim mais simples e eficazes todas as *queries* efetuadas no sistema.

Depois da caracterização de CQRS, incluindo conceitos base e arquitetura, é importante salientar a importância que a orientação a eventos têm no mundo do CQRS e é nisso que o próximo capítulo se irá focar.

3.4 *Event* sourcing e Dados imutáveis

3.4.1 Introdução

Event sourcing é ter a capacidade de saber detalhar todas as alterações que uma aplicação realiza, guardando-as numa sequência de eventos. Isto permite descobrir num determinado momento qual é o "estado" do sistema ou qual foi o estado em que estivemos numa determinada altura, conseguindo assim, traçar um percurso da situação atual e de como chegamos até ela. O padrão dos eventos é reconhecido por garantir a sincronização do sistema e a sua integridade (Young, 2011). Os *reports* mais ricos em informação, eram previamente mais difíceis de obter porque a arquitetura era de tal forma complexa, que era necessário passar por todo o sistema para ter algum tipo de informação mais específica e completa, mas agora, poderemos ter *reports* mais ricos em informação mais facilmente.

Um exemplo de um *report* que se tornou muito mais "trivial" com eventos, foi saber a média temporal que um utilizador teve para adicionar um artigo específico no seu carrinho de compras depois de ter recebido um cupão de desconto no *email*. Anteriormente seria necessário compilar toda a informação do utilizador, saber em que data é que foi enviado o *email*, do qual muitas vezes apenas se regista o envio do número de cupão para mais tarde ser validado, não guardando certo tipo de informação como a hora de envio, saber em que compra é que o utilizou e por fim verificar a diferença temporal. Ao aplicar este padrão basta para um utilizador saber se tem o evento de envio do *email* de desconto, que tem logo acesso à data de envio e saber se tem outro evento de consumo de cupão de desconto, e com isso o cálculo da diferença temporal é quase imediato.

Outra das funcionalidades do *event sourcing* é ser capaz de repor o sistema em estados passados, através de ***logging*** por exemplo, para situações de possíveis catástrofes. (Fowler, 2005b). Além disso, é possível com os eventos, ajudar no desenvolvimento de aplicações, por

exemplo, caso exista um erro em produção, é possível copiar os eventos para desenvolvimento, replicar todos os eventos e verificar mais facilmente qual é o erro. Torna muito mais fácil o *debug* de possíveis erros da aplicação.

Uma das vantagem que também não parece muito óbvia é o facto de os eventos serem ***imutáveis***, ou seja, o que acontece no dia-a-dia é existir sistemas concorrentes e querer a qualquer altura ter a certeza que quando se quer retornar um determinado dado ele é verdadeiro. Com isto podemos dizer que é o seu valor final e não está a ser alterado, sendo caracterizado por a partir do momento em que um evento é aceite, ele se tornar um facto aceite e definitivo, ou seja, imutável. Um exemplo prático: se numa conta bancária for depositado um euro a mais, a única maneira de retificar é colocando um movimento de um euro negativo, contrariando assim o evento anterior (Oliver, 2011a). Com a ligação de ***event sourcing e os dados/eventos imutáveis*** é quase eliminada a hipótese de grande perda de dados que as grandes companhias hoje em dia enfrentam, como é o caso do Google, que perdeu temporariamente contas de GMail e respetivos dados. Utilizando este tipo de padrões foi possível recuperar todos os dados (Musil, 2011).

Em termos de ***performance***, os nodos guardam esses eventos em memória e replicam por todos os nodos permitindo que a performance aumente, contudo tem a desvantagem de ocupar mais memória, mas a verdade é que os eventos são raramente modificados, e por isso temos quase a garantia total que estamos a comunicar com a última versão do mesmo. Para além disso a tarefa de realização de *backups* tornou-se muito mais simples porque basta guardar todos os eventos novos que não existem depois do último *backup* (Oliver, 2011a). Quando a ***quantidade de eventos*** a determinada altura, já se tornar demasiado grande, existe a possibilidade de transferir estes mesmos eventos para outra localização, aumentando apenas a latência de acesso a estes eventos. Em termos de custos, transferir para outra localização pode-se tornar bastante económico visto que existem possibilidades como o **S3** (*Amazon Simple Storage Service*) (Amazon, 2012b) ou **Azure Blobs** (Microsoft, 2012) que permitem alugar espaço em disco a um preço muito competitivo (Oliver, 2011a).

No próximo subcapítulo são realçadas as diferenças entre os comandos e os eventos de forma a se compreender quais as diferenças de conceitos e como poderão ser aplicados.

3.4.2 Comandos vs. Eventos

Num sistema de larga escala, irão existir muitos eventos e por isso todo o sistema poderá estar particionado. Quando existe um novo evento que possa ser consumido por várias partições do sistema, torna-se demasiado pesado e poderá tornar-se um *bottleneck* no tráfego de rede quando é enviado para todos sem se diferenciar quem é que terá interesse no respetivo evento. No que diz respeito aos comandos é totalmente o contrário, visto serem diretamente encaminhados apenas para um local, tendo os recetores identificados unicamente, através de um esquema de partições que define exatamente por onde um

comando deve seguir. Desta forma podemos dizer que os comandos e os eventos têm ***significados diferentes na arquitetura***.

De forma a compreender melhor os comandos, caracterizemo-los por: serem uma mensagem que comunica com uma entidade específica para fazer uma determinada ação; todo o sistema de mensagens está contextualizado com a identidade do comando e por isso sabe para onde o redirecionar; a mensagem do comando poderá ser ela mesma considerada uma entidade, visto que permite ser identificada unicamente e permite ao recetor ter a certeza que poderá ser executada N vezes que o resultado será sempre o mesmo, filtrando inequivocamente mensagens parecidas que possam ser duplicadas pela infraestrutura; Enquanto as mensagens dos eventos são: geralmente no passado, ou seja, algo que se passou e irá ser utilizado para o histórico e não poderá ser mudado; são associadas com quem envia e uma respetiva versão, identificando de forma única cada evento para conseguir identificar também eventos que possam ser duplicados; não tem um recetor associado, são publicadas no sistema de mensagens, para serem reencaminhadas para todos os subscritores;

Com estas características a forma de os comandos e os eventos se interligarem num sistema escalável é enviando comandos para o respetivo *handler*, de forma a ser identificado unicamente para qual é a entidade a que se destina, sem o trabalho do encaminhamento passando a ser de uma forma direta. Ao mesmo tempo são publicadas mensagens de eventos com os resultados desses comandos, com informação de quem executou esse comando e todo o contexto que seja necessário para quem subscreve os eventos (Abdullin, 2010g).

O próximo subcapítulo destina-se a explicar o objetivo dos *command handlers* e a forma como eles se interligam com os eventos.

3.4.3 Command Handlers

Um ***command handler*** é um processo responsável por receber comandos (normalmente através de uma *queue*) e aplicá-los nas respetivas entidades. Todos os comandos têm um recetor associado e por isso a aplicação de um ***comando passa por três fases***: numa primeira fase é necessário carregar a respetiva entidade à qual se irá aplicar o comando de forma única, que poderá já estar em memória; numa segunda fase um comando passa pela sua aplicação no comando da entidade, ou seja, invoca um determinado método para a alteração; por fim, numa terceira fase e caso tenham sucesso é necessário comunicar à lista de comandos que aquele já foi processado (na maioria das vezes é retirar da lista da *queue*) e fazer persistir todas as alterações do próprio comando.

As alterações podem acontecer de duas formas no sistema, através de ***event sourcing***, ou seja, guarda-se todo o detalhe da alteração (guardar as alterações de domínio é primordial) para posteriormente poder ser reprocessado se for necessário ou então ***state persistence*** (alteração direta do estado do sistema mas pouco utilizada), ou seja, não existe a persistência para algum tipo de log da alteração que vai ser feita no sistema sendo logo feita no domínio. Nos dois casos, é necessário manter a persistência dos dados e para isso é preciso garantir

que o comando **tem sempre o mesmo resultado**, ou seja, para qualquer número de execuções do mesmo comando o resultado terá de ser sempre o mesmo, podendo na mesma acontecer situações como mensagens duplicadas (muito em *cloud* ou sistemas particionados) ou o processo bloquear depois de ter feito as alterações, mas antes de dar o ACK do processamento da mensagem atual. Logo, quando existe alteração de dados utilizando *event sourcing,* simplesmente são publicados todos os eventos dando conhecimento às partes interessadas do sistema, das últimas alterações (Abdullin, 2010a).

Um conceito associado aos ***command handlers*** é o de ***subscrições de eventos*** sendo estas, muitas vezes usadas quando aplicadas a uma arquitetura CQRS. São caracterizadas por serem ***publicadas por entidades*** interessadas num determinado evento. Cada entidade é algo que pode ser identificado de forma única em todo o sistema e que se situa num ***command handler***, como *Aggregate Root*, *Sagas*, modelo de leitura; A entidade publica a sua lista de ***queries de eventos***, ou seja, as *queries* são públicas e podem ser replicadas para todas as partições, de forma a se poder escalar facilmente; as subscrições são uma forma de ***comunicar o interesse*** para um determinado tipo de mensagens e que a partir daquele momento quer receber tudo o que diz respeito àquele tipo de mensagens; alguns exemplos serão eventos de ***timeouts*** e de ***domínio***.

Em CQRS existe um termo denominado de *Aggregate Root*, e que é usado em muitos sistemas. De forma a se perceber melhor e fazendo um paralelismo com a programação orientada a objetos, em que uns objetos têm referências para outros objetos, por exemplo, o *Cliente* tem referência para uma *Encomenda*, mas os AR são um pouco diferentes. Os AR promovem uma melhor separação, ou seja, enquanto o *Cliente* e *Encomenda* podem existir independentes no sistema, algumas entidades e respetivos valores já não fazem sentido serem separados, como é o caso da *Encomenda* e *LinhaEncomenda*. Neste caso a *LinhaEncomenda* já não faz sentido existir sem *Encomenda* nem fazer parte de outra *Encomenda*, e por isso *LinhaEncomenda* possivelmente será um *Aggregate* e *Encomenda* um *Aggregate Root*. Para ajudar neste raciocínio, bastará pensar no *Cascading Delete* de uma determinada entidade e verificar se faz sentido que ao apagar uma entidade outra relacionada também seja apagada. Por exemplo, se se desejar apagar a *Encomenda* deverá apagar-se também as *LinhaEncomenda*, porém, este pensamento já não é válido para a relação entre *Cliente* e *Encomenda* (Charlton, 2009).

Existe um tipo de entidade, muito falado no mundo do CQRS denominado ***Saga*** que também pode fazer parte de um *command handler*, sendo caraterizado por ser um *message handler* a correr no servidor, com a função de ajudar a gerir transações de negócio muito grandes, conseguindo gerir o passar do tempo com gestão de *timeouts*. Permite também proporcionar consistência nos dados mas não pode ir além da entidade em que se encontra (AR - *Aggregate Root*), daí, a um sistema deste tipo poderem na mesma chegar mensagens duplicadas ou fora de ordem.

As Sagas podem-se ligar às subscrições de eventos e receber comandos diretamente, e por isso muitas vezes são confundidas mesmo com os *command handlers*. Porém existem algumas

diferenças entre as sagas e os tradicionais *Aggregate Root* nos *command handlers* que são: as ***funções de negócio***, e as intenções do domínio.

Quanto às funções de negócio, os AR são responsáveis pelo *house-keeping* enquanto as sagas refletem as especificações todas do negócio como a forma de lidar com a inconsistência dos dados e probabilidades existentes;

Já no que se refere às ***intenções do domínio,*** os AR agem em limites contextuais, enquanto as sagas interagem entre eles, com o tempo e com o inesperado;

Tecnicamente, as sagas subscrevem eventos e recebem os comandos diretamente, enquanto os *command handlers* apenas recebem comandos.

Com a ajuda da Imagem 20, é possível perceber melhor a ligação entre estes conceitos, e compreender todo o fluxo (Abdullin, 2010g).

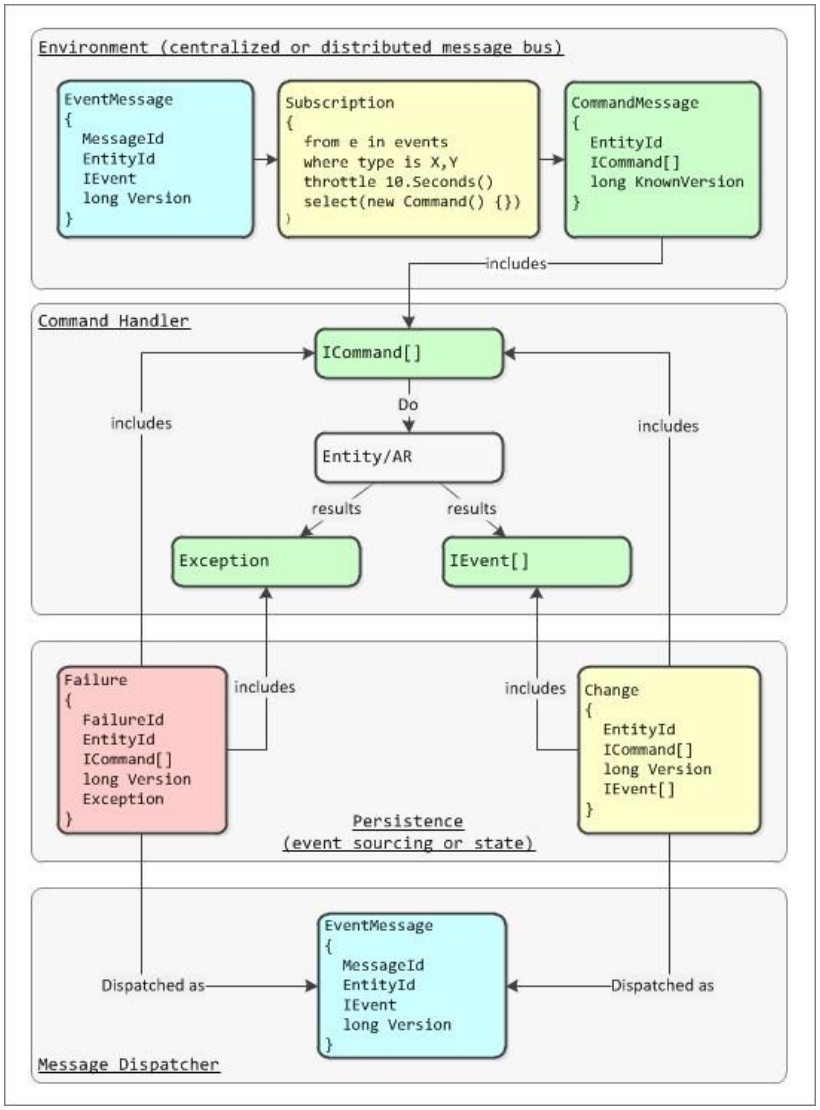

Imagem 20 – Command handling, Sagas, Event Subscription (Abdullin, 2010g)

Aqui, vemos como se relacionam os três conceitos, sendo que na realidade o cenário poderá ser diferente (mais complexo), sendo este apenas a relação base que deverá estar sempre presente em qualquer sistema que os use. Com isto podemos tornar os sistemas mais escaláveis, ricos em informação, confiáveis se usarmos ***event sourcing*** e ***message dispatcher***, redundantes, e com permissão de reordenação de mensagens para cenários de *cloud*. De notar que nestes cenários, é necessário saber a versão com que se está a trabalhar para dessa forma, diminuir os problemas de concorrência (Abdullin, 2010g).

Na Imagem 21 já a seguir, fica demonstrado e em detalhe, como ficará a integração de comandos num sistema com eventos.

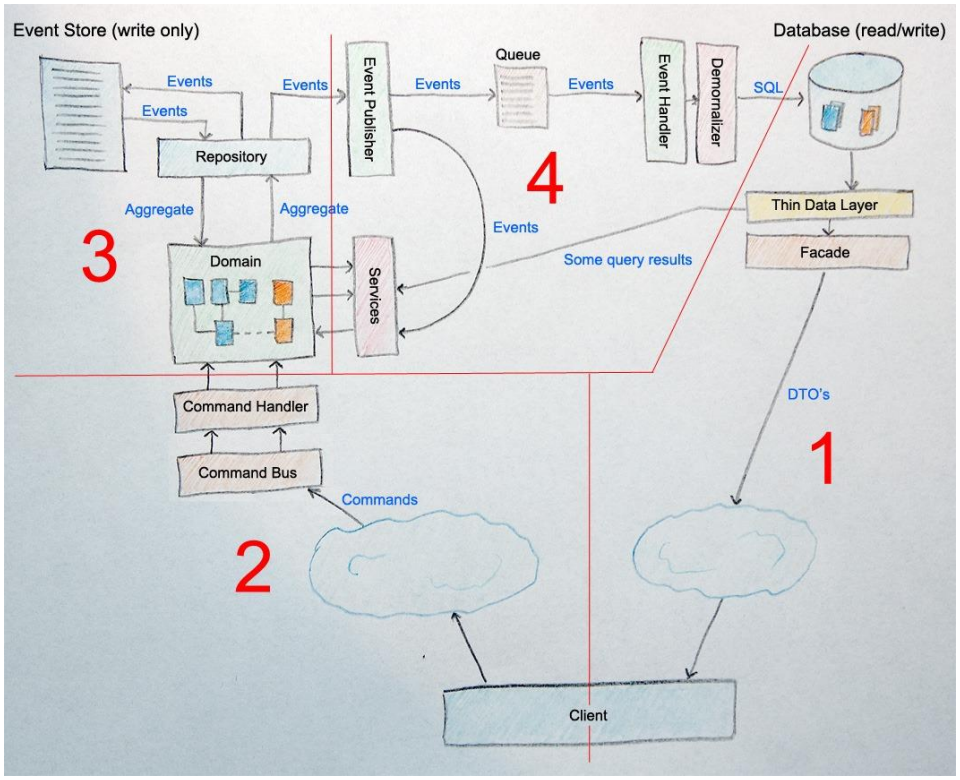

Imagem 21 - Arquitetura de CQRS (Ashton, 2011)

Com a Imagem 21 podemos ver que cada comando é consumido sempre por um ***Command Handler*** através do *command bus* (zona 2) e os seus nomes devem ser sempre imperativos, como por exemplo "CancelOrder". Cada *Command Handler* converte o comando numa chamada a funções num ***Aggregate Root (AR)*** (zona 3), que por sua vez contém eventos públicos (Fossmo, 2009). Cada evento representa uma mudança de estado no domínio, e todos eles são registados num repositório para posterior consulta. Após esse registo são consumidos por ***Event Handlers*** (zona 4) que devem ter sempre os seus nomes no passado, por exemplo, "OrderCanceled". Cada *event handler* aplica a mudança de estado ao respetivo destino, por exemplo, o *view model*, que é a representação dos dados na UI (Tom, 2010).

No próximo subcapítulo será apresentada a comunicação entre modelos e componentes da arquitetura, de forma detalhada.

3.4.4 Messaging

No CQRS o facto de se dividir responsabilidades traz grandes vantagens, porque permite dividir o sistema em várias partes, focando-se cada uma numa área e num problema específico, permitindo assim usar a melhor tecnologia e implementação. Um exemplo será, que um *website* não tem que ser todo desenvolvido em C#, e em vez disso podemos ter exposto JSON (*JavaScript Object Notation*) *endpoints* em que o servidor poderá receber pedidos do *browser* e depois assincronamente, fazer o reencaminhamento das mensagens para processamento. Além disso podemos ter os sistemas de armazenamento de mensagens mais adequado para cada tipo, dependendo da aplicação que o vai usar e da tecnologia. No fundo a utilização de *messaging* abre um novo leque de opções, visto estar mais dividido e estruturado (Oliver, 2011b).

Um primeiro exemplo (Abdullin, 2010f), em que foi utilizado *messaging* na *cloud* com CQRS, demorou apenas 10 minutos a fazer o *deploy* da aplicação para testar o número de mensagens e a capacidade das máquinas, de forma a se conseguir perceber também os impactos na performance. Foi conseguida uma média de 300 mensagens por segundo (o que é pouco para um grande sistema) devido a ter sido feito o teste com o código cliente apenas a correr numa *thread* num computador portátil e num café na Rússia, enquanto a aplicação estava nos Estados Unidos com uma máquina virtual básica (256 MB RAM) (Abdullin, 2010f). Já nos casos aplicados à produção foi possível obter os seguintes números:

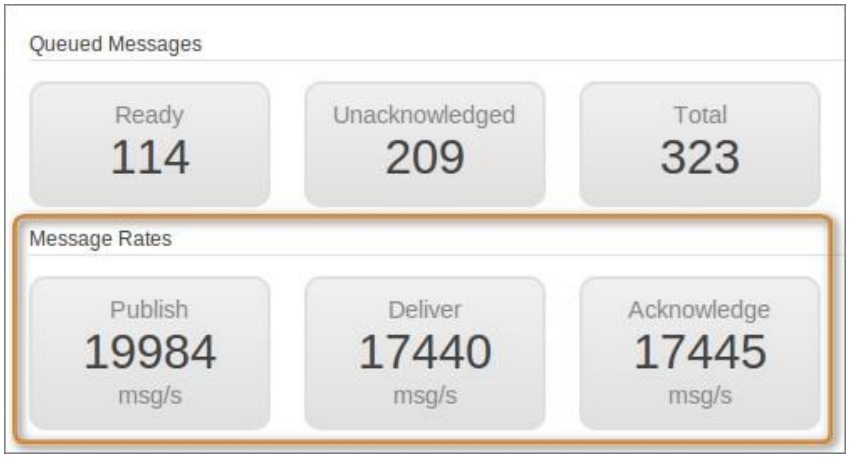

Imagem 22 – Messaging na cloud (Abdullin, 2010f)

Pela Imagem 22 é possível verificar com mais precisão o número de mensagens que serão trocadas e ter uma melhor perceção da realidade, daí hoje em dia as mensagens serem essenciais para construir um sistema distribuído e robusto.

A utilização de *messaging* nos sistemas é caracterizada por cada mensagem ser uma ***estrutura de dados*** que pode ser transmitida entre componentes, que poderão estar em qualquer parte do mundo. Um paralelismo será o tradicional *email*, em que tem um emissor, um assunto e detalhe do assunto e um ou vários recetores, sendo que o processo desde o envio até à chegada ao destinatário poderá demorar (pode ser rápido mas não é instantâneo), muitas vezes devido ao tempo que passam nas *queues* para serem processados. A principal desvantagem das mensagens/*email* é ser um ***sistema assíncrono***, ou seja, não saber quando é que se irá receber a resposta. Ao contrário destes sistemas existe a chamada síncrona, como por exemplo, uma chamada telefónica, em que quando se consegue estabelecer comunicação a troca de informações torna-se quase imediata e garantida e por isso é importante saber que ***vantagens e desvantagens*** trazem cada uma delas:

Com ***chamadas diretas*** é possível: obter ***resposta imediata*** uma vez que seja estabelecida a comunicação; as duas partes interessadas têm que estar ***sempre disponíveis***, é necessário aceitar a comunicação numa altura específica; quanto mais carga houver, mais tempo demorará a comunicação a ser aceite e ***não há garantia que alguma vez seja aceite***. Com ***mensagens*** consegue-se: enviar a mensagem e ***retornar logo ao trabalho*** que se estava a fazer; organizar o trabalho de forma mais ***eficiente***, visto que não se espera por uma resposta da outra parte; enviar mensagens a qualquer altura (a outra parte irá receber e mais tarde responder); quanto mais carga houver, mais tempo irá demorar a resposta, contudo existe uma ***maior certeza*** que mais tarde irá ser processada e enviada a resposta. A existência de *queues* para darem prioridades e evitar descartar pedidos que não possam ser logo processados são uma mais-valia (Abdullin, 2011c).

Com este conceito de comunicação é possível muito mais facilmente escalar os sistemas, diminuindo o *stress* dos sistemas e atrasos nas respostas. As mensagens (assíncronas) tornam-se melhores que as chamadas diretas (síncronas) porque permitem desacoplar os sistemas, fazer o balanceamento do sistema, tornando-se importante em termos de negócio, reprocessar mensagens falhadas, guardar as mensagens para efeitos de auditoria e redirecionar ou balancear entre vários recetores. Contudo existem situações pontuais em que as chamadas diretas são melhores, como por exemplo, quando se pretende obter informação de uma cache em memória, que já por si é rápida e por isso queremos a resposta imediata (Abdullin, 2011c).

No próximo subcapítulo será abordado a forma como o CQRS trata a colaboração de dados e como lida com a incoerência de dados.

3.5 Colaboração e incoerência no CQRS

O CQRS tem associado dois conceitos à sua utilização: ***colaboração*** e ***incoerência***. A ***colaboração*** vem do facto de poderem existir vários atores que podem modificar os mesmos dados ao mesmo tempo, e o que se pretende é que existam meios para manter a coerência nos dois casos, ou seja, até podem ser aplicadas as duas alterações (caso façam sentido) mas nunca se irá perder informação ficando as duas sempre registadas. A ***incoerência*** refere-se ao facto que a partir do momento que um utilizador vê uma determinada informação ela pode ser alterada por outro utilizador e por isso o que o primeiro está a ver, já não é o mais atualizado. Isto acontece muitas vezes devido à existência de caches no sistema para os tornar mais rápidos, mas ao mesmo tempo torna o sistema pouco colaborativo entre si. Estas incoerências podem ser drásticas visto que muitas vezes são usadas para tomadas de decisão, e quando estas situações acontecem, a pessoa pode ter de tomar decisões baseadas em informação errada, daí poderem ser fornecidas prioridades e tempos de validade de dados (Nijhof, 2009).

Em termos técnicos, o mais simples (e menos correto) seria colocar tudo na mesma base de dados e assim garantia-se a colaboração porque tudo estava no mesmo sítio, sendo usada a cache apenas em termos de performance. Com a Imagem 23 podemos ver como é que um utilizador comunica com o sistema e internamente por que passos é que passa cada pedido, seja de consulta seja de alteração.

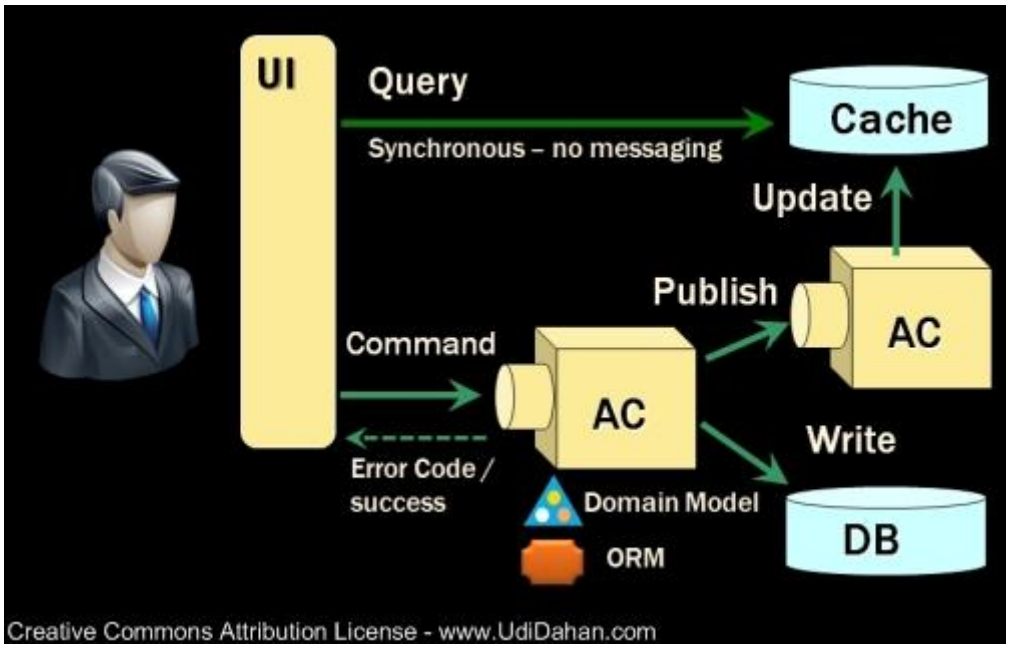

Imagem 23 - Sincronização de dados no CQRS (Dahan, 2009)

O que é visualizado é através de ***queries*** e muitas vezes está incoerente e por isso surgem alguns problemas possíveis em relação aos dados, como por exemplo, como saber se existe a necessidade de consultar a base de dados principal para extrair os dados, se é preciso

realmente transformar em objetos DTO ou porque é que muitas vezes se transforma os DTOs em objetos modelados da base de dados.

Assim e partindo do princípio que reutilizar o código é mais fácil do que resolver o problema novamente, não querendo fazer trabalho desnecessário vamos tentar outra abordagem, que consiste em ter um ***armazenamento de dados extra*** em que este pode estar *dessincronizado* com a informação real, mas no qual seja possível fazer *queries* muito simples e diretas ou seja, fazer por exemplo, "SELECT * FROM PRODUCT" e apresentar logo essa informação ao utilizador. Podemos fazer uma ***remote facade*** para o usar, ou outros mecanismos mas o que interessa é que seja o mais parecido possível com o que se deseja mostrar ao utilizador. Com isto é possível poupar o trabalho de usar DTOs complexos ou outros objetos para a leitura. Se for possível ter um esquema em que fazemos uma *query* direta e não tendo a necessidade de a transformar em novos objetos, muitas vezes não é necessário a existência de uma base de dados relacional (Dahan, 2009).

Em termos de ***escalabilidade de Queries***, atualmente já temos um armazenamento de dados separado do principal e já se sabe que poderá existir uma parte incoerente, que não impedirá a utilização do mesmo mecanismo em várias instâncias e assim não existirá a preocupação que todas tenham exatamente os mesmos dados. O mecanismo de atualização de dados numa delas irá ser igual para todos, e dessa maneira garantimos escalabilidade. Outro aspeto muito importante é a realização da escalabilidade de forma horizontal e por isso a latência diminui. Além disto consegue-se ter o código muito mais simples e se é mais simples, é mais rápido (Dahan, 2009). Na Imagem 23 podemos observar os componentes autónomos, em que os comandos são encaminhados todos para o mesmo AC (*Autonomous Components*), mas poderemos ter vários AC com a sua própria *queue* para processamento, permitindo assim ver qual é a *queue* que demora mais a processar cada pedido. Com isto pode-se alocar mais recursos nos nodos que tem as *queues*, para que apenas se neces*site* de fazer o escalonamento da parte mais importante e que é a mais lenta (Dahan, 2009).

O ***domain model***, na arquitetura demonstrada na Imagem 23, está ao lado do AC que na realidade não passa agora de um detalhe de implementação. Com o CQRS nada obriga a que todos os comandos tenham de comunicar com o *domain model*, conseguindo assim filtrar e simplificar os que não precisam propriamente, tornando assim mais rápido o seu processamento. O *domain model* não é utilizado para as *queries*, evitando assim os relacionamentos entre todas as entidades que existem nas arquiteturas relacionais. As relações que existem entre, por exemplo, um sistema com utilizadores e as suas encomendas, pode ser dividido, não existindo esta dependência entre a entidade utilizador e a encomenda, visto que os comandos apenas devem conter os identificadores necessários. Se realmente este tipo de relações for necessário então poderá fazer parte das propriedades da entidade pai ou ser pré-calculado, fazendo apenas uma referência nos filhos, mantendo assim os comandos a processar uma única entidade (Dahan, 2009).

Para ***manter os dados das queries sincronizados*** após um comando ter sido executado e aceite pelo componente autónomo, existe a necessidade de fazer persistir os dados (como já

era anteriormente) mas também de publicar (*publish*) as partes interessadas, ou seja, fazer *publish* da informação para atualizar a cache, chamando para isso um evento. Este evento tem de ocorrer em simultâneo com a transação de modificação para que caso ocorra alguma falha não ser enviado o evento. Por sua vez, o componente autónomo que trata deste tipo de eventos, é bastante simples, visto que o que necessita de fazer é receber o evento a comunicar o que foi alterado e escrever na estrutura usada as alterações (cache) (Dahan, 2009). É determinante ter em conta a performance aquando da sincronização de dados, visto que quando o sistema começa a tomar grandes proporções é difícil tudo estar sincronizado, ou seja, a *event store* estar igual a base de dados de *report*, e por isso haverá uma altura "mais tarde" que tudo irá ficar igual, daí serem conhecidos por ***eventualmente consistentes*** (Nijhof, 2009).

No subcapítulo seguinte, será analisada uma forma de implementar CQRS em ambiente *cloud* de forma a se perceber nos conceitos falados até agora como é que se poderão encaixar em ambientes de grande dimensão.

3.6 Cloud computing e utilização de CQRS

Este padrão com a utilização do *cloud computing*, vem acrescentar vários desafios e benefícios para os sistemas distribuídos: o desenvolvimento torna-se muito mais flexível pois existe a possibilidade de fazer o *deploy* num ou em vários sistemas ao mesmo tempo; causa uma considerável redução de custos, aumenta ou diminui a escala de forma dinâmica e por fim, não se preocupa com *backups* complexos, configurações e a própria manutenção.

Na maioria das vezes existem bases de dados relacionais que são difíceis de colocar na *cloud* sem haver algum tipo de tratamento antes, para que seja uma base de dados totalmente escalável, o que poderá ser uma limitação a uma implementação imediata. É neste tipo de limitações que o CQRS pretende dar respostas, aproveitando também todas as vantagens do *cloud computing*. Visto que o CQRS visa ter no sistema um modelo dedicado a leituras, uma opção de implementação em *cloud* seria: ***Could Views*** armazenadas no *Windows Azure Blob Storage* (ou Amazon S3) sob a forma de ficheiros simples, ou seja, com a capacidade de utilizar *key-value*, que sendo fornecida uma chave torna possível obter uma *view*; ***View Handlers*** são consumidores de eventos de domínio, responsáveis por atualizar e manter as *views* sempre atualizadas. Estes *views handlers* poderão ser implementados através dum processo que está subscrito num sistema de mensagens e assim é possível ter toda a informação relativa a cada atualização. Poderá ser alojado em *Azure Worker Roles*, Amazon EC2, etc.; do ***lado do cliente*** para ver o que está nas *cloud views* é apenas necessário saber quais os contratos de visualização e ser capaz de os ir pesquisar (normalmente estão disponíveis sob a forma de REST (*REpresentational State Transfer*)) num ambiente *cloud* (Abdullin, 2011d).

Existem algumas questões quanto às *cloud views* serem baseadas em *key-values*, como por exemplo, como é que será feita a sua gestão, a atualização das listas de *views*, etc. Por isso, as

principais diferenças que se sentem, quando se utiliza as *views* em *cloud* são: ***existe mais código***, mas é mais simples, organizado e desacoplado do resto do sistema; ***quem desenvolve não precisa de ter determinadas preocupações***, como scripts de *update* para o lado da leitura, o *deploy* fica mais simples em todo o sistema, existindo assim mais liberdade para fazer *refactor* e experimentar novos desenvolvimentos; as ***views são consideradas descartáveis***, devido a serem facilmente refeitas e colocá-las quase de imediato em produção, permitindo que a UI possa evoluir muito mais rapidamente; em termos de ***escalabilidade*** deixa de haver essa preocupação devido à existência de serviços como o *Azure blob storage*, que permite escalar automaticamente e otimizar tudo para as leituras; ***fica reduzida a complexidade*** entre o cliente e o servidor, não sendo necessário existirem *APIs* complexas e muitas vezes sendo utilizado apenas POJO (*Plain Old Java Object* (Fowler, et al., 2012)) (Abdullin, 2011d).

Quando se fala em CQRS em *cloud*, é importante referir o tema da ***concorrência***. Quando uma *view* é acedida ao mesmo tempo por mais que uma *thread* na maioria das vezes não é considerado um problema visto que os serviços de *cloud* já tratam desse tipo de problemas por nós, o problema começa a surgir quando queremos atualizar uma *view*. A forma mais simples de abordar este problema é através de cada *view entity* (instância de um tipo de *view*) estar limitada a uma única *thread*, por exemplo, podemos no início apenas ter uma *thread* que processa todos os *updates* daquela *view* e à medida que a aplicação cresce em termos de carga e disponibilidade, são divididos os *updates* de cada *view entity* entre várias *threads*. Poderá no entanto acontecer que em determinadas situações que seja preciso fazer um *update* em simultâneo numa determinada entidade e é aqui que entra o ***controlo otimístico de concorrência***.

A melhor forma de solucionar este problema é alterar a implementação da parte que escreve, de forma a ter um atributo denominado *ETag*, ou seja, é como se fosse o número da versão quando se vai escrever. Após ter executado o *update* local, é necessário enviar os dados novamente para o servidor com o atributo *ETag* e poderão acontecer duas situações: o *update* poderá ser aceite porque tem a mesma versão de *ETag*, ou o *update* irá falhar porque alguém já alterou primeiro a vista, e por isso deverá existir a possibilidade de tentar novamente (Abdullin, 2011d).

De seguida no subcapítulo seguinte será apresentado como é que o CQRS poderá facilitar as operações de *deployment* e quais as vantagens que advêm daí.

3.7 Deployment em CQRS

Para além das vantagens do CQRS em termos de arquitetura, existem constantemente novos desenvolvimentos de *software* e isso implica que se tenha que fazer várias vezes *deployment* das aplicações ou até mesmo de uma grande parte do sistema. Com isto torna-se necessário saber o que o ***CQRS nos pode trazer para facilitar esta tarefa***, que hoje em dia é de elevada importância visto que existem muitas dificuldades em por exemplo, atualizar um sistema, sem que os clientes tenham que parar completamente ou simplesmente conseguir evoluir o *software on-demand*, ou seja, os clientes nem notam que o sistema foi alterado. Para que isto aconteça, é preciso saber em que contexto é que vamos aplicar a alteração e em que ambientes é que os vamos aplicar. Um exemplo será uma empresa que fornece *Software-as-a-Service* (SAS), em que é cobrada uma taxa à medida que o *software* é usado e é baseado em três pequenos contextos. Subscrições, Serviços de integração em *Cloud* e um determinado produto (Produto 1).

Pela Imagem 24 podemos ver a parte de subscrições, que tem por funções: manter todos os dados dos clientes, identificar quais os planos é que eles têm ativos, etc.

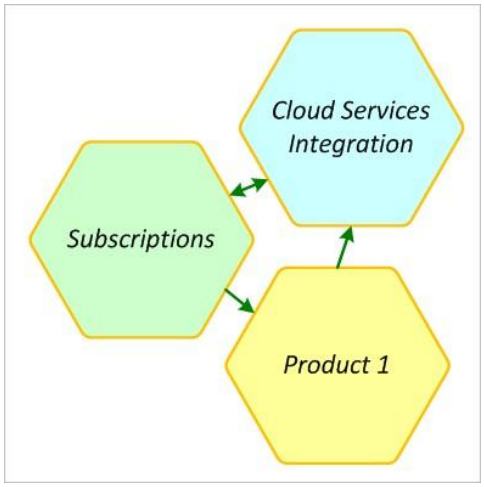

Imagem 24 – Deployment CQRS (Abdullin, 2012)

Este sistema pode utilizar o *NoSQL* com *event sourcing* e ser *deployed* num servidor dedicado. Os serviços de integração com a *Cloud*, poderão ser distribuídos pelo *Windows Azure* e fornecem serviços (*APIs*) para terceiros (através de *queries*) mas também para efeitos internos. Este tipo de projetos não tem tendência a ser mudado frequentemente. Por fim, existe o novo produto, denominado de Produto 1, que é um produto novo na empresa e que foi desenvolvido *standalone* para melhorar a experiência de um utilizador usando as APIs da *cloud*. A ligação deste novo produto é feita com a parte de subscrições para se conseguir autenticar e gerir utilizadores sempre através de APIs.

O fluxo e as interações de um sistema deste tipo acontecerão por exemplo, quando um novo utilizador for adicionado no componente de subscrições. Aí, as suas credenciais devem ser rapidamente replicadas entre todos os serviços de *cloud* de forma a ter acesso às APIs (isto,

poderá ser feito através do enviado de eventos), ou se a conta de um cliente estiver bloqueada, todos os nomes de utilizadores também devem ser bloqueados em todo o sistema. Para que este tipo de sistemas funcione desta maneira, devem funcionar de forma independente uns dos outros de tal forma que se um deles estiver em baixo o resto possa continuar a realizar o seu trabalho, para assim poder disponibilizar por exemplo, o sistema em modo *read-only* sem ir abaixo totalmente. Assim, as mudanças que são feitas pelo utilizador no sistema, devem ficar pendentes e à espera que possam ser enviadas/processadas para que mal o sistema fique novamente disponível, ele possa tornar definitivas as alterações feitas pelo utilizador.

Utilizando o CQRS, quem desenvolve, tem a possibilidade de saber à partida, que quando se está a construir um componente ele terá que comunicar com o resto do sistema, seja via APIs clássica ou *event sourcing* para que se possa propagar qualquer alteração feita no próprio componente. Com isto, é necessário ter sempre em mente que não se deve depender de outros componentes e que é necessário ser tolerante a falhas para que, caso aconteça algo que não era suposto, o sistema conseguir comportar-se de forma controlada para o utilizador sem passar a ideia de que algo está realmente em baixo.

Quando existe um novo módulo desenvolvido e o queremos colocar na *cloud,* por exemplo, temos de ter sempre em atenção se não são quebradas nenhumas ligações existentes e se com quem vamos comunicar está à espera de receber a nossa informação. Ao ser colocado na *cloud*, ele passará a estar acessível sem comprometer outros componentes e por isso não é necessária a paragem do sistema para o efeito. Por isso, o CQRS permite ganhar outra experiência e sensibilidade quando se está a desenvolver para que estejamos preparados para o pior e para comunicar com qualquer componente a qualquer altura sempre sem comprometer o restante sistema. Além disso o conceito de *queries* e comandos permite também desenvolver um componente, isoladamente de outro, mantendo sempre a independência quase obrigatória (Abdullin, 2012).

3.8 Conclusão

O padrão CQRS trata-se de um refinamento a nível de arquitetura, que pretende dar ênfase à divisão da natureza das operações. Desta forma, a arquitetura é baseada numa área de leitura e outra de manipulação de dados, ou seja, onde antes havia uma localização que permitia ler e escrever dados no sistema, agora passam a existir duas localizações que podem estar completamente isoladas uma da outra. Neste padrão, a área da leitura é denominada de *queries* e a área de manipulação de comandos. Esta separação permite também a separação de responsabilidades, ou seja, permite que cada área trate da melhor forma as suas limitações. Tomando como exemplo a área das leituras (*queries*), e sendo esta, a mais utilizada tendo que suportar imensos utilizadores ao mesmo tempo, poderá optar por implementar um sistema de cache, para responder mais rápido a cada pedido. Na área da escrita (comandos) um bom exemplo, seria a implementação de uma *queue* que permitisse o processamento faseado, para se conseguir balancear a carga do sistema.

A utilização deste padrão, permite tornar as aplicações e sistemas mais simples e mais facilmente escaláveis, e pelo exemplo dado, ao ter cache para as *queries* e uma *queue* para os comandos faz com que os dados não estejam sempre atualizados, ou seja, quando existe uma atualização por parte dos comandos, nem todo o sistema ficará coerente e todo com a mesma informação, mas existe a garantia de que "mais tarde" tudo irá ficar coerente através das atualizações faseadas e da validade das caches. Com os sistemas de grande dimensão esta desatualização poderá ser preocupante, no entanto, pelo estudo feito, podemos concluir que ao darmos validade aos dados, validade as caches utilizadas e prioridades à informação, consegue-se que todo o sistema consiga dar mais ênfase a uma determinada informação do que a outra. Verifica-se que numa tomada de decisão, torna-se necessário que alguma informação seja o mais atualizada possível, e por isso prioriza-se essa informação como sendo a mais relevante, conseguindo um sistema escalável, simples e confiável nos dados.

Para a comunicação no sistema, existe o *messaging* que é caracterizado pela troca de mensagens, que podem conter uma estrutura de dados, entre vários componentes que poderão estar dispersos por todo o mundo. Esta aplicação no CQRS traz grandes mais-valias, visto as mensagens poderem ser reprocessadas, por exemplo, caso existam erros devido a um servidor estar desligado. Isso acontecendo, a mensagem poderá ser processada mais tarde sem qualquer problema. Além disso as mensagens são facilmente usadas assincronamente, ou seja, ao entrarem para uma fila, para mais tarde serem processadas, quem envia a mensagem fica logo disponível para continuar o seu trabalho tendo uma garantia que a mensagem será entregue. Por fim, as mensagens facilitam o desacoplamento do resto do sistema, visto que não dependem de tecnologias nem de componentes específicos, podem ser deslocalizadas para qualquer sítio ou podem enviar as mensagens para um novo local sem grande complexidade.

Para além do *messaging* existe o recurso a *Event Sourcing*, que aliado ao CQRS torna-se um elemento crucial para o seu funcionamento e manutenção. O padrão CQRS não necessita obrigatoriamente de usar eventos, no entanto pelos casos estudados grande parte dos eventos fazem uso das suas características. Estes eventos são caracterizados por serem uma mensagem com o detalhe de todos os dados que passam no sistema, servindo mais tarde para auditoria e estatísticas. Ao serem guardados sobre a forma de uma sequência irão permitir que mais tarde seja possível saber a uma determinada altura qual seria o estado do sistema, podendo servir para auditoria a um determinado erro ou simplesmente para a realização de *reports* específicos de negócio. Além disso estas auditorias podem ser baseadas em *logging*, ou seja, ao guardar todos os passos realizados no sistema será muito mais fácil restringir num espaço temporal o nosso sistema e saber exatamente quais foram as alterações efetuadas.

A interface com o utilizador será uma outra questão de significativa importância. A interface poderá tornar-se mais um componente de ajuda à validação dos dados de entrada, ou seja, um utilizador ao interagir com a interface do sistema, poderá enviar vários comandos e de vários tipos e por isso é necessário conseguir validar logo à partida quais é que estarão corretos, de modo a não enviar comandos desnecessários e reduzindo assim a carga. O

mesmo se passa com as *queries* que ao guardarem certas informações em cache permitem uma resposta mais rápida ao cliente.

Finalmente, podemos concluir que com o padrão CQRS obtemos uma grande ajuda para desenhar sistemas desde o seu início e não saímos beneficiados apenas ao migrar de sistemas atuais. A sua implementação auxilia a usufruir de *código sustentável*, código de *alto desempenho, simplicidade, escalabilidade*, e o facto de poder ser distribuída em *cloud*, torna o seu valor comercial muito mais elevado.

Após a elaboração de todo o detalhe do padrão CQRS, é importante conhecer quais as aplicações atuais deste padrão e se haverá interesse e sucesso, na sua implementação. Sobre este tema se debruçará o capítulo que se segue.

4 Exemplos de utilização de CQRS no mundo real

Neste capítulo da tese pretende-se estudar a existência de casos de sucesso na aplicação deste padrão. Torna-se importante perceber a forma como eles se aplicam e o que disponibilizam visto que para obter este tipo de modelo para grandes empresas poderá ser necessário a utilização de serviços externos, por exemplo a Amazon fornece serviços a nível de *cloud computing*, etc. Outro fator importante é saber a nível de base de dados, se existem alguns motores de SQL que permitam ou utilizem também padrões, seja a nível de funcionamento de eventos ou separação de escrita e leitura.

4.1 Amazon Simple Queue Service (Amazon SQS)

Existe um serviço fornecido pela Amazon que utiliza uma parte de conceitos associados ao CQRS em ambiente *cloud*, o que permite posteriormente interligar com outros sistemas. Este serviço é designado de Amazon *Simple Queue Service* (Amazon SQS) sendo caracterizado por ser uma *queue* que pode ser facilmente escalada e é delineada para armazenar todo o tipo de mensagens ao longo de um sistema para que possam ser processadas por qualquer componente. Com isto, permite-se que as mensagens possam transitar entre componentes distribuídos de cada aplicação sem qualquer problema, garantindo que não existe perda de mensagens e que se uma aplicação ficar indisponível temos a certeza que mais tarde o sistema e irá processar a mensagem.

Este é um serviço que pode ser interligado com outros serviços, como serviços web (*Amazon Web Services*) podendo assim ser parte de um componente ou de um sistema que use o CQRS. A vantagem deste tipo de serviços é que pode ser fornecido a toda a internet, ou seja, qualquer cliente se poderá ligar para ler ou escrever mensagens na fila, sem precisar de dependências para a usar nem *software* específico. Tal como qualquer fornecedor de serviços tem as suas próprias características sendo que as mensagens poderão ficar 14 dias à espera de serem consumidas ou de poderem ser lidas e escritas ao mesmo tempo. Aquando da receção de uma mensagem, ela ficará temporariamente bloqueada por forma a controlar o

acesso de dois computadores à mesma mensagem, visto que poderão os dois processar a mesma.

A utilização deste tipo de serviços para qualquer aplicação empresarial, e ainda utilizando a *cloud*, permite uma evolução tanto a nível de recursos disponíveis como a nível económico permitindo uma redução muito acentuada. As principais características são ser ***confiável***, devido a ser executado em *Datacenters* de ***alta disponibilidade*** para poder aumentar ou reduzir sempre que as aplicações assim o obriguem, guardando também cópias dessas mensagens para garantir sempre a sua entrega; É um sistema simples contendo apenas cinco APIs: **CreateQueue**, **SendMessage**, **ReceiveMessage**, **ChangeMessageVisibility,** e **DeleteMessage**; ao ser escalável permite que um número ilimitado de clientes e de mensagens possam ler e escrever ao mesmo tempo; é seguro, devido a fornecer mecanismos de autenticação para assim garantir que a mensagem é entregue a quem deve; e por fim é económico, devido a não haver custos fixos, apenas se irá pagar o que se usa, dependendo da quantidade de dados transferidos.

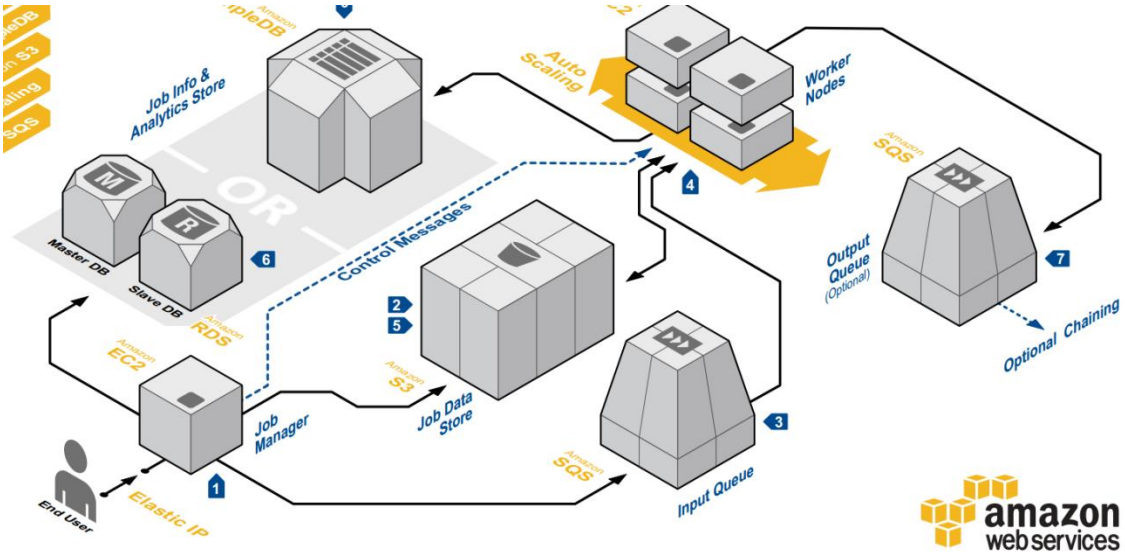

Imagem 25 – Amazon SQS (Amazon, 2012c)

Na Imagem 25 podemos ver um utilizador a interagir com o sistema (1) submetendo o seu trabalho para um Job Manager que será responsável por saber o que é necessário realizar para o utilizador e, como muitas vezes é o caso, gerar as respetivas mensagens para serem distribuídas pelo resto do sistema. Ao enviar uma mensagem, esta chegará a uma SQS (3) que a guardará durante um determinado tempo até ser consumida por um *Worker Node* (4), em que poderá ser uma aplicação responsável por realizar algum trabalho com uma mensagem. No fim, poderá existir ainda outra SQS (7) para ainda executar outra parte da mensagem, ou responsável por enviar para o histórico com todo o detalhe da mensagem processada, em que poderá ser utilizado o conceito de evento. Esta poderá ser uma *queue* com prioridade baixa e poderá ter associada uma aplicação também com baixa prioridade no sistema visto tratar-se de recolha para histórico.

Com este sistema, podemos sem qualquer problema utilizar na mesma o nosso sistema CQRS e delegar a responsabilidade de troca de mensagens entre componentes no serviço de SQS. Visto que no CQRS é utilizada intensamente a parte de leitura, poderemos utilizar por exemplo, dois sistemas de SQS para assim separar, as responsabilidades de escrita e de leitura, sendo sempre necessário ter a noção de que isso poderá trazer mais custos. Com isto podemos ainda reduzir trabalho em termos de gestão de infraestrutura e aumentar a confiança do nosso próprio sistema, tanto para os clientes, como para quem desenvolve o nosso sistema (Amazon, 2012c).

4.2 CouchDB in the Cloud - Document Persistence Powered by CQRS

Num futuro próximo e com a evolução da programação em *cloud computing*, torna-se indispensável que as bases de dados progridam, e acompanhem essa mesma evolução. As bases de dados relacionais para grande escala, não têm as características que se pretende tanto a nível de performance como a nível de facilidade de escalabilidade. Para resolver esses problemas existem as bases de dados orientadas ao documento que pretendem ajudar a evoluir tanto a nível de tamanho como a nível de performance, e um dos exemplos é o CouchDB da Apache (*Cluster Of Unreliable Commodity Hardware*). É conhecido por funcionar num *hardware* que muitas vezes é conhecido no mercado por ser falível, devido a ter sido originalmente desenvolvido em C++ mas em Abril de 2008, existiu um novo projeto que levou a ser reescrito em *Erlang* reforçando ainda mais a ideia de tolerante a falhas (Lennon, 2009).

À semelhança dos outros motores de base de dados deste género, neste não existe o conceito de relação entre documentos, sendo necessário sempre que se quer relacionar dados criar uma nova *view* de forma a poder utilizar na mesma os benefícios do SQL típico, como *joins*, mas sem a responsabilidade de ter que definir todas as relações na camada da base de dados. O ChouchDB é conhecido por ser altamente distribuído, ser fácil de replicar, ter uma grande performance a nível de leitura e ser *schema-less* quanto aos seus documentos (o mesmo tipo de documento pode ter campos diferentes), podendo-se verificar as características do CQRS no ChouchDB. O *schema-less* do ChouchDB equivale a no CQRS não se dar importância à coerência imediata dos dados (***eventualmente consistente***), vistas múltiplas no CouchDB que equivale no CQRS à ***subscrição de eventos*** nas tabelas desnormalizadas, a alta performance a nível de leitura deve-se à utilização de ***queries desnormalizadas e otimizadas*** para leitura, tal como no CQRS, e por fim o facto de se poder ter autorreplicação de dados equivale ao típico ***publish/subscribe*** do CQRS. Assim temos um motor de base de dados com todas as características essenciais do CQRS, e a prova que também é possível de aplicar internamente às aplicações.

Para além da utilização dos conceitos deste padrão no seu sistema, este motor de base de dados ainda utiliza uma API baseada em REST, organização e caracterização dos documentos em JSON e a possibilidade de se construir várias *views* em diferentes linguagens, fornecendo

assim um motor leve e de elevada performance orientado a documentos. O CouchDB poderá ser utilizado em aplicações web mas também poderá servir para reorganizar e estruturar através de CQRS todos os documentos de domínio.

A nível de negócio este motor proporciona uma grande mais-valia, devido a implementar o ***controlo de versões de dados,*** através do uso de ***Multiversion concurrency control (MVCC)*** que antes de utilizar os sistemas de *locking* para gerir a concorrência entre vários clientes, fornece um *snapshot* da última versão da base de dados até que as alterações feitas por outro cliente sejam *commited*, e só assim serão visíveis por todos, aplicando então o controlo de versão falado em CQRS. Ainda em termos de valor para o negócio temos o facto de cada alteração feita em cada documento, não ser considerado um típico *update* como numa BD relacional, porque quando se atualiza um campo no CouchDB o que é feito internamente é uma *revision* do documento, e com isso é criada uma outra versão do documento, conseguindo manter todo o histórico pelo qual um documento passou, não havendo a necessidade de existir implementação por parte dos programadores e não afetando a performance do sistema visto que está imediatamente no nível da base de dados (lembrando os *upserts* utilizados em CQRS) (Lennon, 2009).

Além disso permite distinguir o estado dos dados da mudança de dados. O estado é a versão do documento em questão, sendo uma representação de algo no sistema, enquanto a mudança descreve os passos que se realizaram, como por exemplo, como é que chegou à versão 2 de um documento a partir da 1, ou o que mudou em concreto entre cada versão. Com isto permite enriquecer o nosso sistema com informação importante a nível de negócio, pois permite mais tarde fornecer um contexto total das mudanças que se passaram em todo o sistema e assim proporcionar algo essencial que se baseia em padrões de eventos, como o ***event sourcing*** (Abdullin, 2010b).

Para além das características essenciais, é necessário também saber onde é que o CouchDB está a ser usado, para que se consiga dar mais confiança e suporte à sua implementação. Em termos de *websites* temos *iWantMyName* (http://iwantmyname.com/) que é um *registrar* de domínios web e que atualmente usa 100% CouchDB em produção; Em termos de aplicações móveis temos para Android o *SpreadLyrics* que fornece letras de músicas e as permite partilhar; Em *software* temos o *Group Complete* (http://www.groupcomplete.com/) que é um agregador de dados para android que usa o CouchDB tanto a nível do cliente como a nível do servidor; Para uso interno existe o *Engine Yard* (http://www.engineyard.com/) que usa o CouchDB para saber métricas e fazer o *tracking* de instalação de pacotes no sistema (BenoitC, 2012).

4.3 Enterprise workflow - Um ano de aplicação de CQRS

Durante o ano de 2011, John Teague desenvolveu juntamente com a sua equipa uma aplicação baseada em CQRS com recurso a *Event Stores* para persistência de dados. A primeira versão disponibilizada para os utilizadores foi em Junho de 2011 e era bastante básica, fazendo apenas o que os utilizadores mais precisavam para o seu dia-a-dia. Depois deste primeiro *deployment* essa aplicação foi sendo aperfeiçoada ao longo do tempo com funcionalidades cuja inclusão tinha sido esquecida ou que seriam precisas apenas mais tarde, pelo que, ainda hoje se trabalha na aplicação. Tal como qualquer nova aplicação de CQRS passou pelas fases iniciais, desenvolvimento e manutenção tendo tido tanto decisões boas como más ao longo do percurso, o que permitiu uma grande aprendizagem deste padrão.

Concretamente a aplicação é um ***workflow*** desenhado para a empresa onde os utilizadores têm uma série de tarefas para realizar, que podem passar por diferentes estados, o que por sua vez contextualiza no padrão CQRS, visto que diferentes estados geram diferentes eventos em objetos diferentes o que irá determinar qual a ação que deverá ser tomada e qual será a seguinte. A definição escolhida de CQRS distingue modelos separados, um para ler, outro para escrever, não fazendo parte da definição trocas de mensagens, arquitetura distribuída ou *event sourcing*. No entanto ele utiliza *event sourcing* para distinguir claramente a parte que escreve da parte que lê os dados. Este conhecimento da separação de papéis permite ter um impacto profundo no desenho e desenvolvimento da aplicação desde o início, visto solucionar de forma diferente o mesmo problema, como por exemplo, em termos de performance, será melhor construir uma *queue* para receber novos comandos ou mesmo, dar prioridades aos comandos de forma a serem rapidamente executados, enquanto a nível de *queries* pode-se aumentar a performance desnormalizando as tabelas de base de dados.

Nem tudo tem de ser desenhado utilizando o CQRS, e isso poderá ser fatal para o desenvolvimento e manutenção da aplicação. Se já existirem partes da aplicação de forma bastante simples então não será necessário introduzir esta parte no CQRS porque só irá criar ruído para o sistema. Para se ter noção do que é necessário desenvolver em CQRS, e se se quiser introduzir um conceito simples em CQRS é preciso passar por determinados passos: cada comando ser criado, validado, executado, cada evento ser despoletado e gerido por um *handler*, etc., e por isso considera que o CQRS pode ser complexo como um todo, e que se uma parte é simples então devemos mantê-la assim. Um conselho fornecido é que para se começar a desenvolver com CQRS não se deve utilizar logo um ambiente distribuído, porque eleva para um nível ainda mais complexo, obrigando quase à utilização de técnicas de utilização de *queues*, como o MSMQ (*Microsoft Message Queuing*) ou RabbitMQ.

Mais tarde quando o sistema necessitar de ser escalado, deve-se nesse momento sim, aumentar a complexidade e testar num ambiente distribuído. Na aplicação em questão a execução de um comando e dos eventos associados são executados na mesma *thread* visto ainda ser uma aplicação web pouco complexa, em que para um determinado evento, tem 4 a 6 *handlers* que podem atualizar 4 a 5 tabelas da base de dados.

Com este padrão, foi possível ter um feedback mais rápido dos erros e mais facilmente descobrir quais eram os bugs e corrigi-los, conseguindo assim construir a aplicação mais rapidamente e dando aos utilizadores finais resultados mais rápidos. Em termos de *design*, o autor dá o conselho de ser criativo quanto ao desenho do domínio e do modelo de leitura, visto que ele apenas aplicou um *aggregate root* o que levou a que com o tempo ficasse demasiado grande e o considere agora um erro. Neste momento ainda está desta forma, não tendo ainda uma ideia de como resolver o problema visto também não ser impeditivo para o sistema.

Em termos de *event sourcing*, foi onde o autor teve mais dificuldade tanto em entender o conceito como fazer passar a ideia do que realmente era, até à estratégia de implementação que deveria tomar. Não sendo um requisito o uso de *event sourcing,* ele optou por usá-lo, e concluiu que não teve grandes benefícios com a sua utilização mas também não trouxe grandes custos de implementação em relação a outras soluções propostas, atribuindo a principal vantagem em poder repetir eventos e conseguir refazer o estado dos modelos de leitura, ou seja, existe a possibilidade de criar novas *views* na aplicação e que fica disponível para todos os dados já existentes como se lá estivesse desde início.

Por fim o autor revela que ficou satisfeito com a aplicação concebida e que conseguiu instruir-se em muitos novos conceitos e novas práticas a tomar no futuro. Teve grande impacto no processo de negócio e conseguiu mudar e criar um novo sistema bastante mais rápido para o que estava à espera, atribuindo esta mais-valia à capacidade de independência de todos os componentes, *views* e objetos de domínio. No futuro irá continuar a construir novos projetos baseados em CQRS, sempre com refinamentos que vai aprendendo com o tempo (Teague, 2012).

5 Frameworks existentes

5.1 Introdução

Este capítulo da tese pretende evidenciar que ferramentas é que existem no mercado para que se possa iniciar de uma forma mais simples a implementação do padrão CQRS, e para isso irão ser estudadas três *framework*s: NServiceBus (.NET), NCQRS Framework (.NET) e Axon Framework (JAVA). O objetivo deste estudo será também fornecer uma lista de ferramentas com as tecnologias mais utilizadas no mercado, para que se possa ter o primeiro contacto com uma nova forma de desenhar arquiteturas desde o início. Foram escolhidas estas três *framework*s visto na comunidade de CQRS serem muito bem cotadas e já utilizadas por algumas grandes empresas. Para cada *framework* pretende-se saber em detalhe, em que tecnologias é que se baseiam, de que forma é que aplicam o CQRS, se como um todo ou em pequenas partes do sistema, a forma que usam para comunicar os eventos, se já trazem *templates* próprios para ajudar a implementação de *queries* e comandos, etc.

No final de cada uma é apresentada uma pequena demonstração de aplicação para que se consiga ter uma ideia sobre a conceção de uma aplicação com o padrão CQRS.

5.2 NServiceBus

5.2.1 Introdução

Esta *framework* é destinada para colaboração entre serviços orientados e geralmente usados em projetos SOA (*Service-oriented architecture*) e DDD (*Domain-driven design*). Esta *framework* na maioria das vezes não é usada sozinha mas combinada com outras tecnologias. O NServiceBus tem algumas parecenças com o WCF (*Windows Communication Foundation*) em que o *bus* é algo virtual, ou seja, é como se fosse uma rede *peer-to-peer* que corre ao lado do nosso código (objetos da *framework* a correr em vários processos). O modo de distribuição continua inalterado, mesmo com a inclusão de vários processos *standalone* na *framework*, como *distributor*, *timeout manager*, *proxy* e *gateway* (DAHAN, 2012b). Ao longo destes anos

as tecnologias têm vindo a ser alteradas constantemente, tanto devido à sua evolução como à sua descontinuação, e esta *framework* conseguiu-se manter bastante estável e robusta devido a ter acompanhado estas mesmas mudanças. A forma de comunicação é através de uma ***fila de mensagens*** (*queue*d *messaging*) e isso faz com que o NServiceBus não tenha problemas de chamadas bloqueantes ao sistema. A diferença do NServiceBus para o WCF é que o WCF obriga a utilização deste paradigma o que torna as implicações na arquitetura mais profundas. A utilização deste paradigma de utilização de troca de mensagens permite aos sistemas atuais serem de mais confiança em relação às chamadas tradicionais de RPC (*remote procedure call*), e assim ganhar mais valor para a empresa. Para além disso muitas empresas que o adotaram tiveram apenas um custo inicial de perceber quais os novos conceitos, e posteriormente o tempo de implementação foi célere e possibilitou desfrutarem de mais perspetivas futuras em relação à nova forma de comunicar e à sua aplicação.

A ***nível de desenvolvimento***, quando os sistemas existentes usam o tradicional RPC em que o WCF também o suporta, é direto e simples de conseguir pôr tudo a funcionar, contudo, em termos de escalabilidade e tolerância a falhas o RPC vai contra este tipo de conceitos e paradigmas e por isso torna-se um grande problema. Ao usar RPC torna-se quase impossível resolver esse tipo de problemas, mesmo adicionando *hardware* para tentar diminuir o efeito negativo. Enquanto o WCF não obriga nem previne a quem desenvolve, de utilizar RPC, o NServiceBus é quem direciona desde inicio quem está a desenvolver para longe destes problemas (DAHAN, 2012b).

5.2.2 Arquitetura

Um dos problemas de hoje em dia ao construir um sistema distribuído é que muitas vezes ele é frágil desde início, e caso uma parte do sistema fique lenta ou deixe de responder, o resto do sistema tende a ficar também lento ou deixa de conseguir funcionar normalmente. Um dos objetivos do NServiceBus é eliminar desde o início esse problema, guiando quem desenvolve para escrever num código mais robusto, prevenindo perda de dados, etc. Para se conseguir implementar eficazmente esta *framework* é necessário perceber qual é a arquitetura do sistema em que o vamos implementar e quais as diretivas da *framework*, para que nos possa tornar a vida muito mais fácil no dia-a-dia. Para que seja possível implementar necessidades específicas do negócio, é possível estender as características de modo a que possamos tirar partido de todas as vantagens e adicionar a nossa própria lógica ao sistema. O padrão de comunicação é ***one-way-messaging***, também conhecido como ***fire and Forget*** e para a passagem de mensagens entre máquinas é utilizado o ***Store-and-Forward***.

No modelo ***Store-and-Forward*** o objetivo é que quando um cliente chama uma API, ou seja envia uma mensagem para um servidor, a API retorna o controlo para a *thread* que a chamou antes de ser enviada a mensagem, ou seja, depois de a *thread* ter o controlo a mensagem foi enviada para a rede e a partir deste ponto a gestão da mensagem passa a ser responsabilidade do sistema de mensagens como se pode ver pela Imagem 26.

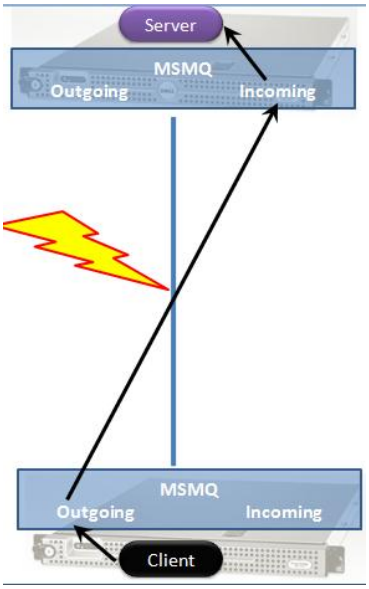

Imagem 26 - NServiceBus Store & Forward Messaging (Dahan, 2012a)

O padrão ***Request/Response*** é conhecido por ser um RPC síncrono enquanto o ***One Way Messaging*** o faz de maneira diferente, não deixando a *thread* que fez o pedido em espera de resposta à mensagem enviada. Pela Imagem 27 podemos ver como funciona em detalhe:

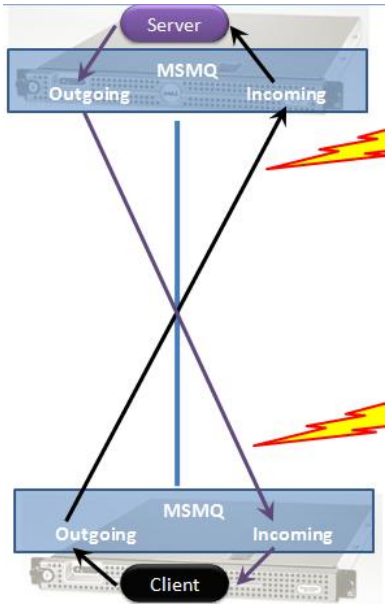

Imagem 27- NServiceBus Messaging (Dahan, 2012a)

Uma das grandes vantagens deste padrão verifica-se, quando um cliente deixa de responder e o servidor está pronto para enviar a mensagem, aí, o servidor não necessita de estar vários minutos à espera até dar *timeout*. Se a isto for adicionado o conceito de ***Durable Messaging*** então, é possível ainda aumentar a robustez do sistema, para que o sistema de mensagens deixe de ser *store-and-forward* e as mensagens passem a persistir no disco local antes de serem enviadas. Isto permite à *thread* que fez o pedido e que já tem novamente o controlo, se o processo em que está por acaso der um erro, a mensagem não será perdida e poderá ser reenviada mais tarde. É o que acontece frequentemente na comunicação entre servidores e então, caso uma transação dê erro será possível continuar no mesmo ponto em que deu erro, sem ser necessário reprocessar todo o pedido (Dahan, 2012a).

Neste sistema de troca de mensagens, quem envia a mensagem não sabe quem a irá receber ou não terá a certeza de quem a quererá receber e por isso é necessário subscrever (***Publish/Subscribe***) de forma a receber todas as novas mensagens. Com isto torna-se também o sistema menos desacoplado como se pode ver pela Imagem 25:

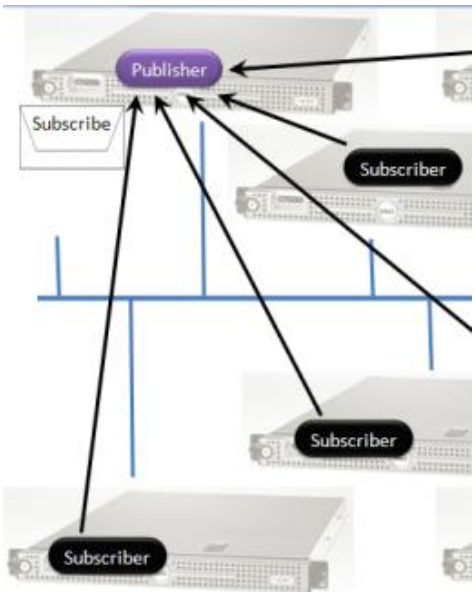

Imagem 28 - NServiceBus Publish/ Subscribe (Dahan, 2012a)

A comunicação entre vários componentes da rede é feita através de um contrato entre um *subscriber* e um *publisher* (***Subscriptions***) (Imagem 28). O *subscriber* envia um pedido de subscrição com o seu endereço de resposta para posteriormente o publisher saber a quem tem que mandar uma determinada mensagem para um determinado endereço (*endpoint*) do *publisher*. Muitas vezes o *subscriber* não comunica diretamente com o *publisher* para subscrever, mas comunica através de uma central de comunicações que guarda o endereço de retorno e essa central é que trata de subscrever todos os serviços necessários. Além disso o *publisher* apenas sabe que tem de enviar uma mensagem para um determinado *subscriber*, mas esse pode na realidade ser dividido em vários, permitindo que exista um balanceamento da rede a nível de gestão de mensagens.

Quanto ao Publishing, ele é definido por o *publisher* enviar uma mensagem para todos os *subscribers*, como se pode ver pela Imagem 29.

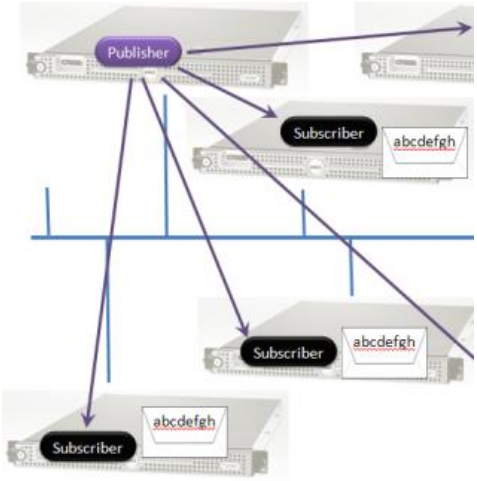

Imagem 29 - NServiceBus publishing (Dahan, 2012a)

Quando as mensagens são recorrentes denominam-se de ***eventos***, como por exemplo, uma encomenda cancelada, demora no envio da encomenda, etc. Visto que num sistema muito utilizado este tipo de mensagens podem ter de ser enviadas demasiadas vezes num curto espaço de tempo, isso, pode trazer para o sistema, uma degradação da performance. Uma melhor solução é um publisher apenas enviar determinadas mensagens num período de tempo, que é definido aquando da celebração do contrato da troca de mensagens. Depende da necessidade de obter os ***dados logo atualizados***, ou conseguirem apenas ser atualizados mais tarde. Um exemplo será a situação de atualização do saldo num sistema bancário que terá de ser de 10ms enquanto numa situação de e-commerce talvez 10 segundos seja aceitável. Outra vantagem será poder ***delegar a entrega das mensagens*** a outros servidores aquando do envio tardio de forma a não sobrecarregar o servidor principal (Dahan, 2012a).

Esta *framework* utiliza o ***CQRS*** para retornar e atualizar informação através dos comandos e *queries*. A razão para tal é a melhor organização das operações CRUD e a utilização de mensagens devido a um utilizador comum olhar muitas vezes e durante algum tempo para a informação que está a ver e por isso muitas vezes os dados não precisam de ser atualizados ao segundo. A vantagem da utilização de comandos e *queries* é ainda mais notável quando o sistema tem uma carga muito elevada, pois como existem dois modelos para obter dados e outro para os atualizar isso permite que um modelo nunca degrade a performance do outro. A nível de persistência de dados também permite a otimização duma base de dados para as *queries* e para os comandos para que cada modelo aceda apenas à sua e nunca interfira com o restante sistema (Dahan, 2012a).

5.2.3 Exemplo

Um exemplo básico é um *website* que nos apresenta um campo de texto em que é pedido um número, e caso esse número seja par a resposta será “Fail” caso seja impar será “None”, como podemos ver pela imagem seguinte:

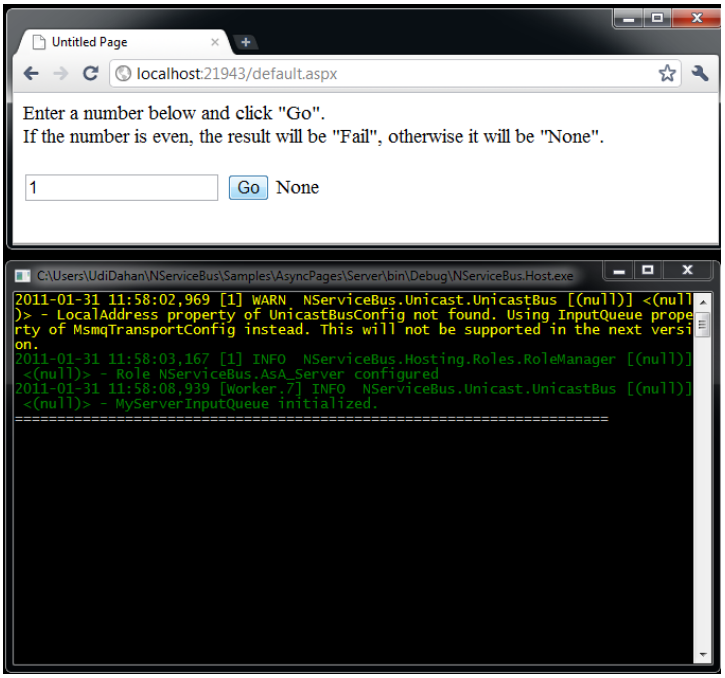

Imagem 30 – Exemplo de aplicação NServiceBus (Dahan, 2012c)

Para o utilizador, ao introduzir o número e carregar no botão, parece que a chamada ao servidor é síncrona e bloqueante, mas no entanto todo o NServiceBus utiliza *messsaging* assincronamente, sendo necessário analisar como é que é composto em termos de projetos:

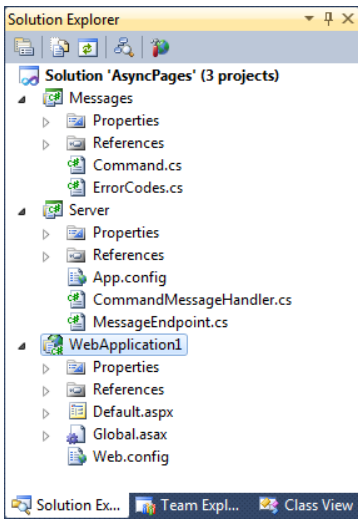

Imagem 31 – Projetos da aplicação exemplo NServiceBus (Dahan, 2012c)

Na Imagem 31 é possível ver os três projetos utilizados, *Messages*, *Server* e *WebApplication1*. O *WebApplication1* é um projeto web que envia mensagens para o projeto *Messages* e de seguida para o projeto *Server* que contém a consola da aplicação. Em mais detalhe para o envio de uma mensagem existe o "Default.aspx.cs" que contém o seguinte método (Imagem 32):

```
int number = int.Parse(TextBox1.Text);
var command = new Command { Id = number };

Global.Bus.Send(command).Register<ErrorCodes>(
    code => Label1.Text = Enum.GetName(typeof (ErrorCodes), code)
    );
```

Imagem 32 – Código de envio de mensagem NServiceBus

Na Imagem 32 podemos ver que a primeira linha guarda o valor introduzido pelo utilizador, a segunda é responsável por criar uma nova mensagem do NServiceBus através do tipo "*Command*", em que é inicializada a propriedade "*id*" com o valor que foi passado pelo utilizador e a terceira faz referência para o Bus interno da aplicação, que é chamado para encaminhar o comando que queremos executar (O método "*Register*" é falado mais à frente). A interface do comando (Imagem 33) é muito simples neste caso pelo código abaixo:

```
[Serializable]
public class Command : IMessage
{
    public int Id { get; set; }
}
```

Imagem 33 – Interface de mensagem NServiceBus

Posteriormente ao envio da mensagem é necessário tratá-la, e isso é feito na classe "*CommandMessageHandler*" (Imagem 34) que contém o seguinte código:

```
public class CommandMessageHandler : IHandleMessages<Command>
{
    public IBus Bus { get; set; }

    public void Handle(Command message)
    {Console.WriteLine("================================================");

        Thread.Sleep(TimeSpan.FromSeconds(1));

        if (message.Id % 2 == 0)
            Bus.Return(ErrorCodes.Fail);
        else
            Bus.Return(ErrorCodes.None);
    }
}
```

Imagem 34 – Código de Command Handler NServiceBus

Neste caso a interface que implementa a denominada "*IHandleMessages<T>*" permite ao NServiceBus saber que deverá fazer a sua gestão, e assim quando uma mensagem chega, é

colocada na respetiva *queue*, fazendo a sua decomposição e por fim instancia a classe a que corresponde o tipo de mensagem invocando o método "*Handle*" da respetiva classe passando como parâmetro o objeto recebido.

Por fim, existe a resposta para a aplicação web, em que o *bus* já tinha invocado uma *callback* (Imagem 35) ou seja:

```
Global.Bus.Send(command).Register<ErrorCodes>(
    code => Label1.Text = Enum.GetName(typeof (ErrorCodes), code)
    );
```

Imagem 35 – Registo de callback NServiceBus

No método "*Register*" é registado o código e comunica com o *bus* fornecendo os dados para ser invocado quando for recebida uma resposta. Já dentro da *callback*, temos a atualização do campo de texto com o valor "*None*" ou "*Fail*" consoante o resultado obtido (Dahan, 2012c).

5.3 Ncqrs Framework

5.3.1 Introdução

Esta é uma *framework* que é aplicada em .NET e ajuda a construir aplicações escaláveis, extensíveis e de fácil manutenção, através da utilização do padrão CQRS. O objetivo é fornecer uma infraestrutura e a implementação para a execução de comandos, modelação do domínio, gestão de eventos e eventos do domínio, permitindo a quem está a usar focar-se apenas no código de negócio adicionando assim valor ao sistema. Com isto ainda se facilita a criação de código isolado e testável (Ncqrs, 2012).

A *framework* não é uma solução para todos os problemas, e para sistemas que usem apenas o CRUD sem acrescentar algum valor, esses então, não tirarão grandes vantagens das características deste tipo de *framework*. Contudo existem algumas características de aplicações que poderão beneficiar. São elas: (Ncqrs, 2012): ***Grande duração*** – o projeto já tem muito tempo ou será suposto ter um grande crescimento; ***Necessidade de ser escalável*** – Precisar de crescer a qualquer altura sem problemas; ***Contém lógica de negócio*** – necessidade de dar manutenção a toda a lógica de negócio; ***Integração*** – necessidade de integrar a nossa aplicação com outras; ***Várias views*** – quando existem vários tipos de outputs, *reports* ou forma própria de visualizar os dados***; Log de auditoria*** – necessidade de guardar todas as alterações que são feitas para mais tarde serem consultadas ou repostas em caso de catástrofe.

5.3.2 Arquitetura

Esta *framework* permite aplicar o CQRS mais facilmente, pois fornece todas as ferramentas necessárias para o fazer, como a criação de comandos e de *queries*. Pela Imagem 36 podemos ver a arquitetura geral:

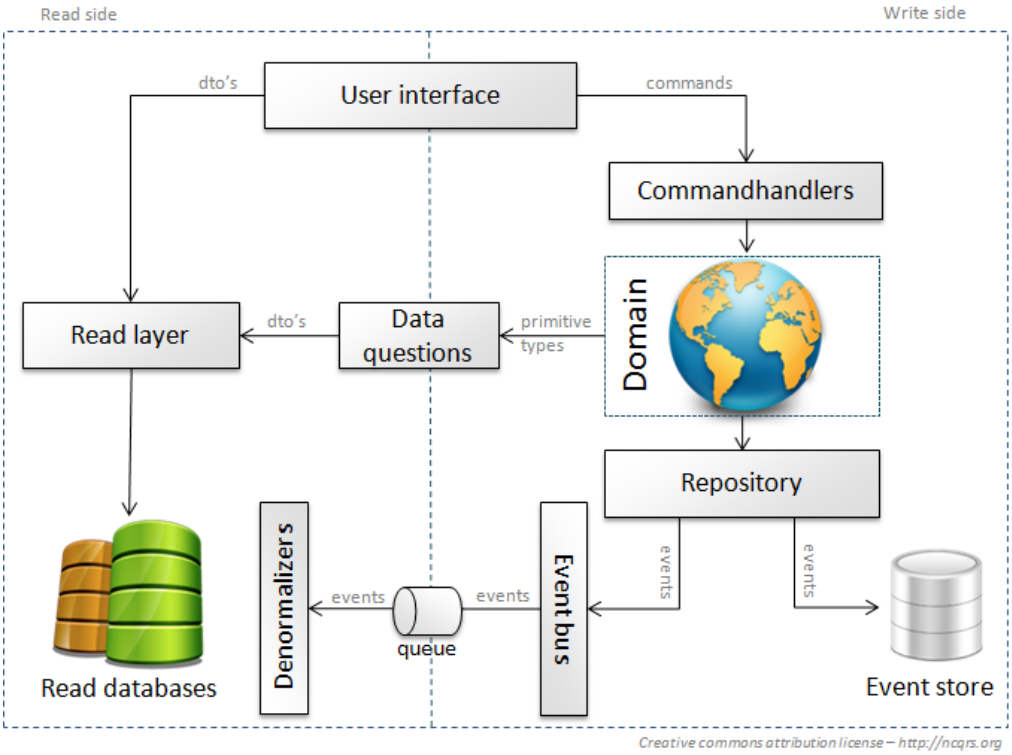

Imagem 36 - Arquitetura geral do NCQRS (Ncqrs, 2012)

Tal como o esperado com a implementação do CQRS existe uma diferenciação entre o retorno de dados e a sua escrita. Na Imagem 36 podemos ver o ***Read Side*** e o ***Write Side***. Pelo fluxo normal de um utilizador temos a componente da interface que no diagrama podemos ver que executa comandos, comunicando com os respetivos *handlers*, que por sua vez irão fazer mudanças no domínio. Estas mudanças irão dar origem a eventos e à sua possível persistência num meio de armazenamento, que serão depois publicados através de um *bus* para a *queue* de ***eventos***. Depois da passagem pela *queue* passará pelos ***Denormalizares*** que são responsáveis por ler os eventos e refletir as suas alterações para os modelos de leitura. Por outro lado a interface para ler a informação será através da camada de leitura, ou seja, das *queries*.

Os comandos são caracterizados por serem objetos que têm os dados necessários para proceder a uma alteração e são processados pelos ***command handlers***, e esse pedido de alteração é feito através de um ***Command Service***, ou seja, é um serviço que faz o reencaminhamento aquando da receção de um comando para o correspondente executor desse comando. Todos estes ***command executors*** respondem a um comando específico e executam uma ação baseada no conteúdo recebido e não devem conter lógica de negócio,

apenas fazer as mudanças para que estão destinados e para garantir que isto acontece, o NCQRS suporta o mapeamento de um comando para o respetivo objeto do domínio.

Esta *framework* já fornece as ferramentas que ajudam a implementar este tipo de serviços permitindo que quem desenvolve não se tenha de preocupar tanto com a arquitetura. Alguns exemplos são as permissões ou o *logging* (Ncqrs, 2012). Os *command executors* fazem as alterações no respetivo ***domínio***, contendo toda a lógica de negócio que não deverá ser usada nas *queries*, podendo assim ser otimizada. Quando existe uma alteração no estado do domínio então é criado um evento, e cada evento é representativo de cada detalhe da alteração, sendo este padrão denominado de ***event sourcing***. Geralmente os nomes dos eventos são dados no passado, como por exemplo, "*UserMovedToNewAddress*".

Em relação ao ***repositório*** este é utilizado para ler e guardar todos os *aggregate roots*, ou seja, a ação de guardar resulta em fazer persistir todos os eventos que não foram *committed* enquanto a ação de ler é fazer retornar os eventos fornecendo todo o caminho até ao último estado. Os eventos são guardados no ***event store*** quando um *aggregate root* é guardado ou quando existe alguma mudança sendo possível construir o estado atual apenas a partir dos eventos. Neste momento o NCQRS pode ***guardar os eventos*** em *Memória*, *MongoDB*, *Microsoft SQL Server* e *Microsoft Windows Azure*. O repositório publica os eventos através do ***Event Bus***, e este *bus* publica em todos os que se registaram nele, que poderá ser de modo ***síncrono ou assíncrono***. No modo síncrono quem gere os eventos da parte dos subscritores bloqueia o que está a ser executado e espera que todos os subscritores complementem a ação antes de retornar. De modo assíncrono, não espera que todos os subscritores completem antes de retornar, ou seja, comunica o evento e sai, o que torna os comandos mais rápidos mas estes acontecimentos não ocorrem em simultâneo, apenas passado algum tempo é que ficam todos sincronizados.

Existem diferentes tipos de ***Event Handlers***, os mais comuns são os ***denormalizers*** que são responsáveis por fazer as mudanças no modelo de leitura, mas também poderão fazer alterações em sistemas externos ou enviar um correio eletrónico quando um determinado evento ocorre. Além disso estes *handlers* poderão enviar também um novo comando, sendo muito utilizados para estender o sistema, adicionando novas funcionalidades sem fazer mudanças no mesmo.

Uma das partes mais importantes de uma aplicação são os dados e a forma como são manipulados. Os ***modelos de leitura*** são otimizados para retornarem dados para uma determinada situação, por exemplo, um utilizador poderá querer ver sempre todos os produtos com o respetivo nome e preço, mas outro apenas quererá ver os primeiros dez e apenas o nome. Em termos práticos uma *query* apenas deverá pesquisar num único sítio e devolver apenas os dados, não fazendo *joins*. Isto é conseguido muitas vezes através da criação de uma tabela para cada *view* e assim saber onde ir exatamente pesquisar a informação. A leitura, porém, ***não contém necessariamente uma base de dados relacional*** podendo ser sob a forma de documento por cada *view*. O NCQRS apenas fornece uma classe

base para os *denormalizers* que é possível subscrever através do *event bus*, sendo assim simples e prático (Ncqrs, 2012).

5.3.3 Exemplo

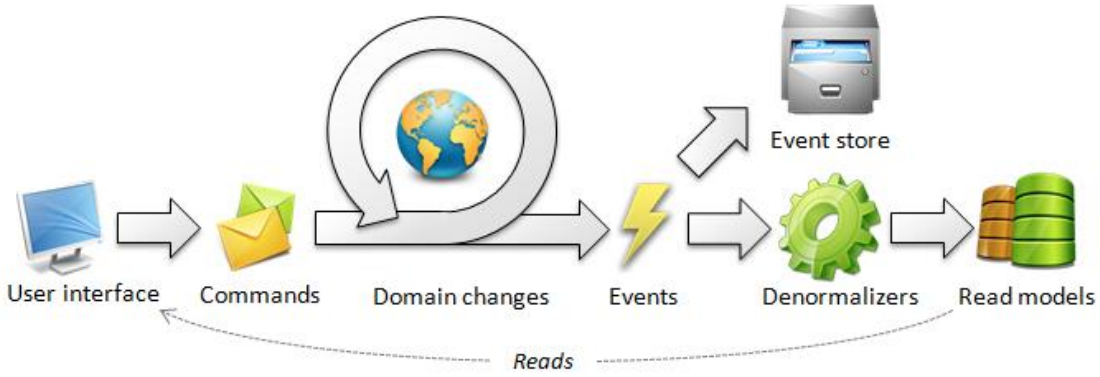

Imagem 37 - Resumo da framework NCQRS (Ncqrs, 2012)

Pela Imagem 37 é possível verificar que tudo começa na ***user interface***, passando pela mudança no sistema, ou seja através de ***comandos***. Estes comandos efetuam mudanças no ***domínio*** que por sua vez irão gerar ***eventos*** para que fique tudo registado. Estes eventos vão para a ***event store*** para serem persistidos, e posteriormente passam pelos ***Denormalizers*** para que as mudanças feitas na *user interface* possam ser refletidas no ***modelo de leitura***, ou seja, quando se fazem novas ***queries*** será possível já ter acesso a estas alterações. Esta *framework* disponibiliza assim, um conjunto de módulos para que se possa usar na maioria dos sistemas existentes de forma a torná-los cada vez mais expansíveis e escaláveis.

5.4 Axon Framework

5.4.1 Introdução

Esta *framework* pretende ajudar a construir aplicações que possam ser escaláveis, extensíveis e de fácil manutenção, através da aplicação do padrão de ***CQRS***. A *framework* fornece a implementação para os blocos mais importantes como os agregadores, repositórios e mecanismos de troca de eventos. A linguagem utilizada para a construção é o ***JAVA 6***, o que permite recorrer às anotações de forma a que quem está a desenvolver consiga focar-se na lógica de negócio sem ter que se preocupar com a lógica da *framework*, mantendo assim o código mais fácil de ler e mais isolado. A Axon garante também a quem está a desenvolver, a garantia de entrega dos eventos pela ordem correta, e o seu processamento que poderá ser concorrencial.

Um dos principais problemas que ocorre hoje em dia é o facto de as arquiteturas serem ***multi-threading*** o que leva muitas vezes a que a aplicação falhe totalmente caso aconteça algum

bug inesperado, sendo que esta *framework* já foi intensamente testada de forma a prevenir este tipo de bugs, ajudando assim numa implementação mais célere. Em termos técnicos, uma das maiores preocupações será com a utilização de apenas uma ***JVM (Java virtual machine)***, porém, para a construção de aplicações escaláveis uma simples JVM não chega. Por essa razão, esta *framework* utiliza um módulo designado ***axon-integration*** que permite que os eventos sejam enviados via "*Spring Integration*",conseguindo enviar os eventos para os componentes em diferentes máquinas (Buijze & Coenradie, 2012a).

Esta *framework*, tal como o CQRS, ***não trará contributos a todos os sistemas***, aqueles que apenas se baseiam em CRUD e que não se espera que precisem de escalar, irão beneficiar pouco com esta implementação. Os tipos de sistemas e aplicações que irão tirar partido da aplicação do *Axon* serão: ***sistemas que precisem de ser estendidos*** com novas funcionalidades durante um longo período de tempo, como por exemplo, um sistema que apenas guarda a localização de determinadas encomendas e em que a certa altura passa a ser preciso também ter o respetivo inventário. Mais tarde poderá novamente ser preciso tirar estatísticas e assim sucessivamente; ***Sistemas com muitas leituras e poucas escritas***, também irão beneficiar e muito com o *Axon* visto que existem diferentes localizações para as *queries* e para os comandos, o que permite ter otimizações próprias para cada contexto; ***Aplicações com dados em diferentes formatos***, aqui, uma aplicação poderá ter o seu próprio *report* para apresentar a informação mas outra aplicação poderá ter a mesma fonte de dados, porém, apenas facultar os dados necessários para o público-alvo a que se destina.

Com a *Axon* cada *datasource* pode ser atualizada independentemente e em tempo real; ***Aplicações com componentes bem definidos e totalmente independentes***, como por exemplo, numa loja *online*, os empregados da loja irão atualizar os detalhes de um produto e o stock enquanto os clientes irão fazer encomendas e saber o estado de cada uma delas. Com o Axon estes componentes poderão ser distribuídos (*deployed*) em máquinas diferentes e escaladas de diferentes maneiras. A forma de manter o sistema atualizado é através dos eventos que a *framework* irá enviar a todos os subscritores; A ***integração com outras aplicações*** é garantida através da disponibilização de *APIs* que usam comandos e eventos, permitindo a qualquer aplicação enviar comandos ou receber os eventos gerados pela aplicação (Buijze & Coenradie, 2012a).

5.4.2 Arquitetura

Sendo o *Axon* baseado em CQRS teremos uma arquitetura nesse sentido, em que existe a normal separação de comandos e *queries* e além disso fornece ferramentas que permitem a interligação do CQRS com outros padrões também importantes para o *design* de uma aplicação. Pela Imagem 38 podemos verificar que existe um componente de UI que interage com o resto da aplicação através do envio de comandos para a aplicação (parte superior do diagrama) e o pedido através de *queries* à aplicação (parte inferior do diagrama) (Buijze & Coenradie, 2012b).

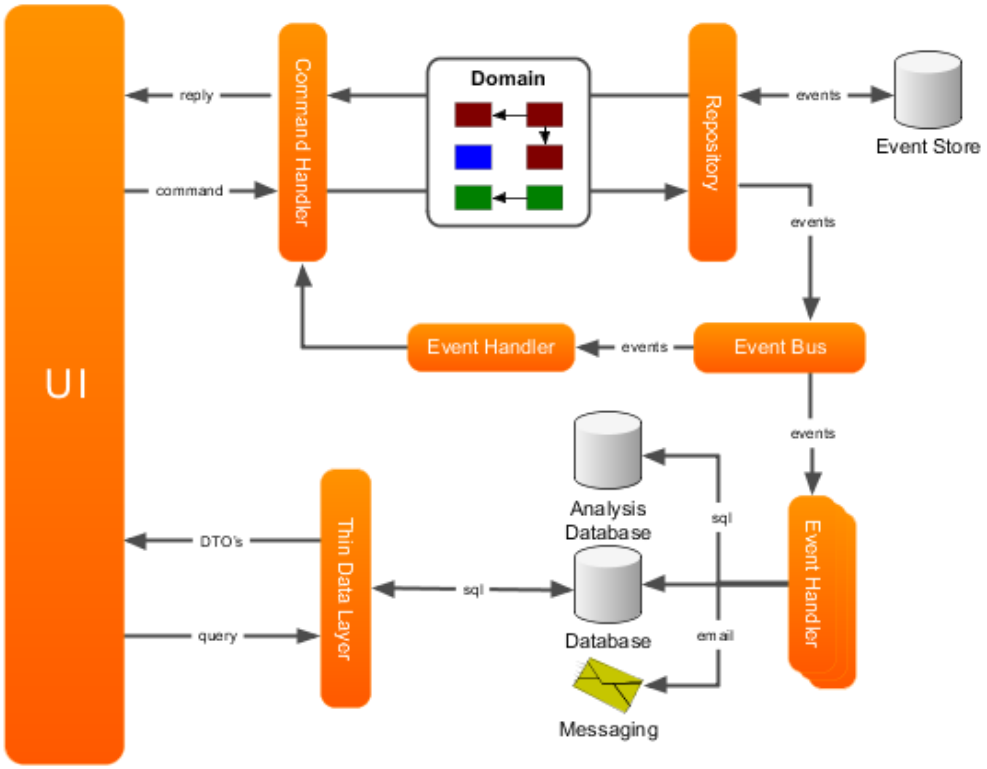

Imagem 38 – Arquitetura Axon (framework java) (Buijze & Coenradie, 2012b)

Com a Imagem 38 conseguimos também verificar as partes essenciais da arquitetura como o ***Command Bus*** que recebe os comandos e os reencaminha para um *Command Handler*. O ***Command Handler***, é responsável por reencaminhar objetos que contêm os dados necessários para um comando ser executado. Em java o nome da classe é usado para saber o que é necessário fazer e os atributos do comando contêm a informação necessária para o executar. Cada um responde a um comando específico e executa a lógica associada e em muitos casos é necessário também executar *validação*, *logging* e *autorização*, no qual o Axon já providencia blocos para ajudar na sua implementação.

Em relação ao ***Domain Modeling*** este permite aos *Command* Handlers retornarem objetos do domínio (***Aggregates***) dos repositórios e executar métodos alterando o seu estado. Estes *aggregate*s contêm a lógica de negócio e o seu estado quando se altera produz ***eventos de***

Domínio, em que o Axon dá suporte a classes que ajudam a construir um *domain model* desde o inicio. Os ***repositórios*** e os ***event stores*** são responsáveis por fornecer acesso aos *aggregate*s, tanto a nível de pesquisa como a nível de persistência de alterações. O Axon neste aspeto fornece suporte tanto para alterações diretas nos *aggregate*s (usando ORM) como através de eventos.

O ***processamento dos eventos*** poderá ser feito *sincronamente* (é mais simples, pois permite executar vários eventos na mesma transação) ou *assincronamente* (retorna o controlo de imediato para o utilizador, mais rápido na resposta, e processa em segundo plano). Muitas vezes o processamento de eventos leva a que seja necessário enviar mais comandos para a aplicação, suponhamos, quando uma ordem de encomenda é recebida, quer dizer que ao cliente deverá ser cobrado esse mesmo valor e a encomenda deverá ser preparada para ser enviada. Em muitas aplicações a lógica de negócio torna isto muito mais complexo, ou seja, o cliente poderá não pagar logo ou demorar demasiado tempo a pagar e é preciso decidir o que fazer com a encomenda, se se envia de imediato a encomenda ou se se espera primeiro pelo pagamento. Este tipo de complexidade e a sua gestão fica a cargo do CQRS e do tratamento das devidas transações.

Em relação à ***obtenção de dados*** existe entre a UI e as *datasources* uma camada "***Thin Data Layer***" que retorna *DTOs* em modo apenas de leitura com os resultados das pesquisas efetuadas. A maioria das vezes estes *DTOs* correspondem ao que o utilizador irá ver na interface de modo a que exista um mapeamento direto, tal como nos diz o padrão (Buijze & Coenradie, 2012b).

5.4.3 Exemplo

Além de saber os conceitos base e como irá funcionar o "core" do sistema, é necessário saber também, como é que se poderá criar a nossa própria aplicação e a melhor forma de a usar. É preciso definir quais serão os objetivos detalhadamente, ou seja, a definição dos comandos e os respetivos eventos, como construir os *command handling* responsáveis pelo consumo de comandos procedendo a eventos, e finalmente ter disponível um componente para fazer as respetivas *queries* ao sistema. O exemplo será mais focalizado na utilização do core do sistema do que na UI (Buijze, 2012).

Para termos um exemplo simples em que se utilize todas as partes essenciais do sistema, iremos criar um sistema de gestão de encomendas de uma empresa. Esse sistema deverá permitir criar encomendas, que poderão conter um ou mais artigos. As encomendas poderão ser canceladas ou confirmadas pelos clientes. Depois de uma encomenda ser cancelada já não será possível confirmá-la. Neste sistema torna-se assim possível ter três tipos de "operações" que serão denominadas de ***CreateOrderCommand***, ***ConfirmOrderCommand*** e ***CancelOrderCommand***. De notar que todos os comandos são imperativos, de forma a perceber-se claramente o que irá acontecer aquando da execução de cada comando.

Exemplo do *CreateOrderCommand* (Imagem 39):

```
public class CreateOrderCommand {

    private final String orderId;
    private final String productId;

    public CreateOrderCommand(String orderId, String productId) {
        this.orderId = orderId;
        this.productId = productId;
    }

}
```

Imagem 39 – Exemplo de CreateOrderCommand AXON

O *CancelOrderCommand* e *ConfirmOrderCommand* são idênticos à Imagem 39, com diferenças apenas na remoção do atributo "*productId*".

É importante salientar que o que o cliente necessita de enviar será o *orderId* para cancelar e confirmar, o que permitirá por exemplo executar mais tarde a operação pretendida. De qualquer forma tudo depende da forma de como a empresa já tinha estruturado o seu esquema de negócio e as regras de negócio existentes. Depois de executar qualquer um dos comandos é gerado um evento respetivo, ou seja, ***OrderCreatedEvent***, ***OrderConfirmedEvent*** ou ***OrderCancelledEvent***. Visto todos os eventos terem o atributo de identificação *orderID* será criada uma classe pai *AbstractOrderEvent* (Imagem 40):

```
public abstract class AbstractOrderEvent extends DomainEvent {
    public String getOrderId() {
        return getAggregateIdentifier().asString();
    }
}
```

Imagem 40 – Classe pai AbstractOrderEvent dos eventos

O exemplo de *OrderCreatedEvent* (o nome será sempre no passado Imagem 41):

```
public class OrderCreatedEvent extends AbstractOrderEvent {

    private final String productId;

    public OrderCreatedEvent(String productId) {
        this.productId = productId;
    }

    public String getProductId() {
        return productId;
    }
}
```

Imagem 41 – Evento OrderCreatedEvent

Seguidamente é necessário existir um *Command Handler* que receba os pedidos via comandos que no AXON recebem a anotação de @CommandHandler para permitir ao AXON saber que deverá receber comandos, exemplo Imagem 42:

```
@CommandHandler
public void createOrder(CreateOrderCommand command) {
    orderRepository.add(new Order(command.getOrderId(),
command.getProductId()));
}

@CommandHandler
public void confirmOrder(ConfirmOrderCommand command) {
    Order order = orderRepository.load(new
StringAggregateIdentifier(command.getOrderId()));
    order.confirm();
}
```

Imagem 42 – CommandHandlers createOrder e confirmOrder

Para os dados persistirem no repositório não é necessário nenhum tipo de *commit* porque o Axon trata disso por nós. Após o *command handling* estar criado teremos que passar para outro componente, o *Order*, sendo este um *aggregator*. É importante ter em atenção que estamos e queremos usar *Event Sourcing*. O método *apply* será responsável por guardar o evento e prepará-lo para publicação, chamando também o *event handler* correto dentro do próprio *aggregator*. Por fim o método *onCreate* é responsável por aplicar as mudanças de estado no sistema. Como ficaria a classe *Order* Imagem 43:

```
public Order(String orderId, String productId) {
    super(new StringAggregateIdentifier(orderId));
    apply(new OrderCreatedEvent(productId));
}

@EventHandler
private void onCreate(OrderCreatedEvent event) {
    status = Status.OPEN;
    productId = event.getProductId();
}
```

Imagem 43 – Classe Order

No final do fluxo será necessário confirmar a encomenda e por isso será necessário ter um método de confirmação, que faz a validação e por fim outro que realmente muda o estado do sistema, ou seja, ficaria da seguinte forma na Imagem 44:

```
public void confirm() {
    if (status == Status.OPEN) {
        apply(new OrderConfirmedEvent());
    }
}
@EventHandler
private void onConfirm(OrderConfirmedEvent event) {
    status = Status.CONFIRMED;
}
```

Imagem 44 – Confimar encomenda

Depois dos comandos estarem definidos com os respetivos *command handlers*, torna-se necessário desenvolver também a parte das *queries*. Para isso impõe-se também conhecer os eventos que tratam das encomendas, e por isso teremos diferentes *handlers* para criar, cancelar e confirmar. Cada *handler* terá a responsabilidade de alterar, o estado da encomenda persistindo de seguida as alterações. No caso do exemplo que estamos a analisar vejamos a Imagem 45:

```
@Component
public class OrderEventHandler {
    @PersistenceContext
    private EntityManager entityManager;
    @EventHandler
    public void handleOrderCreated(OrderCreatedEvent event) {
        Order order = new Order(event.getOrderId(), event.getProductId());
        order.setStatus("We're working on your order");
        entityManager.persist(order);
    }
    @EventHandler
    public void handleOrderCancelled(OrderCancelledEvent event) {
        Order order = entityManager.find(Order.class, event.getOrderId());
        order.setStatus("Your order was cancelled");
    }
    @EventHandler
    public void handleOrderConfirmed(OrderConfirmedEvent event) {
        Order order = entityManager.find(Order.class, event.getOrderId());
        order.setStatus("Your order was confirmed. You'll get it soon.");
    }
}
```

Imagem 45 – Código do OrderEventHandler

Querendo testar o sistema todo integrado poderemos criar uma classe de teste em que são chamados os métodos pela ordem que geralmente uma encomenda é submetida e assim saber por onde passa e se realmente funciona (Imagem 46).

```
public class Runner { (…)
        commandBus.dispatch(new CreateOrderCommand("1", "Chair"));
        commandBus.dispatch(new CancelOrderCommand("1"));
        commandBus.dispatch(new ConfirmOrderCommand("1"));
        commandBus.dispatch(new CreateOrderCommand("2", "Table"));
        commandBus.dispatch(new ConfirmOrderCommand("2"));
        commandBus.dispatch(new CancelOrderCommand("2"));
        commandBus.dispatch(new CreateOrderCommand("3", "Lamp"));
        commandBus.dispatch(new ConfirmOrderCommand("3"));
        commandBus.dispatch(new CreateOrderCommand("4", "Sofa"));
        List<Order> orders = queryRepository.findOrders();
        for (Order order : orders) {
            System.out.println(String.format("Order [%s] for a [%s] has
status: %s",
            order.getOrderId(),
            order.getProductId(),
            order.getStatus()));
        }
```

Imagem 46 – Classe de teste de execução

O resultado obtido seria (Imagem 47):

```
Order [1] for a [Chair] has status: Your order was cancelled
Order [2] for a [Table] has status: Your order was confirmed. You'll get it
soon.
Order [3] for a [Lamp] has status: Your order was confirmed. You'll get it
soon.
Order [4] for a [Sofa] has status: We're working on your order
```

Imagem 47 – Resultado do teste

De notar que na encomenda 1 seria emitido um aviso declarando que uma encomenda, não pode ser cancelada e de seguida confirmada, e na encomenda 2 aconteceria o contrário, ou seja, não se pode cancelar uma encomenda já confirmada pelo sistema.

5.5 Análise comparativa

Seguidamente será apresentada uma tabela na qual se pretende demonstrar as principais diferenças entre as *framework*s estudadas.

Tabela 1 – Tabela comparativa de frameworks

	NServiceBus	**NCqrs**	**Axon**
Tempo mercado	2 anos	2 anos	2,5 anos
Documentação	Detalhada	Pouca	Detalhada
Linguagem	.Net	.NET	Java
Utilização de Messaging	Sim	Sim	Sim
Persistência de dados (base de dados/sistema ficheiros incluído na base)	B.D. No SQL - RavenDB	Várias	B.D. *NoSQL* - MongoDB
Utilização de *queues*	Sim	Sim	Sim
Templates desenvolvimento (comandos, *queries*, enventos, etc.)	Não	Não	Sim
Exemplos práticos	Sim	Sim (muito básicos)	Sim
Open Source	Não	Sim	Sim

Com a Tabela 1, é possível verificar a respetiva análise e comparação dos fatores mais importantes com as respetivas frameworks. Pretende-se demonstrar a alguém que deseje iniciar a implementação do padrão CQRS, quais os fatores que deverá ter em conta. É importante salientar que estes fatores comparados na tabela, foram estudados devido a serem uma mais-valia na prova de conceito.

Começando pelo tempo de mercado, existe a *framework* Axon, que há cerca de dois anos e meio já possuía alguns testemunhos com experiências de alguns utilizadores que tentavam implementá-la nas suas aplicações, enquanto o NserviceBus e NCQRS têm cerca de dois anos.

Outro fator importante foi a documentação que existia, visto ser um ponto crucial para que nos seja possível tomar como base e iniciar um trabalho de implementação. Neste caso, a AXON e o NServiceBus foram as que demonstraram possuir mais detalhe e organização.

Um fator importante seria a forma como comunicavam os componentes nas *framework*s e desta forma foi possível concluir que todas utilizam *messaging* o que as torna numa mais-valia.

Visto existir a necessidade de mais tarde elaborar uma prova de conceito que possa ser utilizada futuramente, tornou-se necessário decidir como seria a persistência dos dados, se seria guardada em sistema de ficheiros ou base de dados, sendo que em relação ao NserviceBus foi possível testar numa máquina local e verificar que a base de dados *default* utilizada era a RavenDB que tem a característica de ser *NoSQL*. Por outro lado temos a AXON que utiliza MongoDB, sendo esta também *NoSQL*, mas os documentos guardados são através de BSON (JSON binário). Por fim o NCQRS tem vários *templates* de ligação no seu projeto base mas não foi possível testar a sua interação com estes motores.

Para o balanceamento do sistema e garantia de entrega, é sugerido que exista algum tipo de mecanismo para o fazer, e como já tinha sido mencionado, a existência de uma *queue* poderia resolver esse problema. O ideal seria que não existisse a necessidade de ter que implementar esse mecanismo de raiz, e que as três *framework*s fornecessem este tipo de sistemas, seja ele para gestão de comandos, eventos, etc.

Ao desenhar um novo sistema em larga escala, muitas vezes existem determinados problemas que se não tiverem algum tipo de controlo fará com que se criem conflitos e confusões de conceitos a quem implementa um sistema com recurso a este padrão pela primeira vez. Para que essa hipótese seja reduzida, será uma mais-valia a criação duma *framework* que forneça um controlo aquando do desenvolvimento de por exemplo, a criação de um comando. A *framework* conseguirá guiar quem desenvolve pelo caminho correto sem confundir conceitos, a qual apenas foi possível verificar na AXON *framework*. De salientar que este controlo poderá ou não ser utilizado, visto que para quem desenvolve há mais tempo, conseguirá saber exatamente o que pretende e como deve fazê-lo.

Um fator muito importante a levar em conta seria os exemplos práticos que existem, de forma a ser possível iniciar com uma ideia das ligações entre componentes e como tudo funciona. Com isto a Axon sai beneficiada visto que apresenta exemplos mais completos com todos os detalhes e documentação, sendo que a sua atualização é constante (através do GITHub). O NServiceBus também possui os seus exemplos, mas devido às suas restrições apenas é permitido utilizar duas instâncias de cliente o que leva a que não possamos futuramente testar. Em relação à NCQRS possui três exemplos muito básicos da sua aplicação o que para esta tese não se adequava.

Por fim, em termos de licenças a AXON e NCQRS são *open source* enquanto a NServiceBus tem licenças quanto ao número de mensagens trocadas por segundo e número de clientes utilizados.

Devido aos fatores e mais-valias enunciados, foi optado por utilizar a AXON *framework* para a implementação de prova de conceito, a qual será descrita em detalhe no próximo capítulo.

6 Aplicação exemplo

6.1 Introdução

É necessário conceber uma prova de conceito, para que, na prática, se consiga verificar o funcionamento e a interligação das *queries*, comandos e eventos, por forma a clarificar os conceitos apresentados de CQRS. Para exemplificar o funcionamento do padrão CQRS foi utilizada a *framework* AXON, devido a usufruir de uma base de código muito diversificada e que possibilita a ligação com outras tecnologias. Para esta escolha também pesou o facto de a *framework* ser rica em exemplos de terceiros, que a usam com sucesso. A ideia geral deste protótipo é de ser uma loja *online* de venda de livros com o nome "XopXop CQRS", que poderá ser acedida por milhares de *browsers* (*queries*) para ver o catálogo de livros ou o detalhe de um livro específico, mas onde as operações de inserção ou atualização (comandos) são mais raras.

Para ter uma melhor perceção da separação de conceitos e de quantas *queries* e comandos já foram executados ou, quantos é que são executados de cada vez que é aberta a aplicação, neste caso, o catálogo, foi criado um contador onde se consegue visualizar a utilização de *queries* e comandos. Além disso, também foi implementado o padrão de eventos (que não é obrigatório em CQRS) para que se possa também comprovar como é que o CQRS se interliga com outros padrões utilizados em grande escala.

Em termos técnicos é utilizado o IDE eclipse, a linguagem Java com a versão 1.6 e para a definição e instanciação é utilizado o Spring versão 3.1.1. Para que se consiga executar este sistema foi utilizado o servidor TomCat versão 7. O código base da AXON Framework foi retirado de https://github.com/AxonFramework/AxonFramework. Para a persistência de dados utilizou-se uma base de dados *NoSQL* de forma a, além de se conseguir demonstrar os conceitos de CQRS, também demonstrar as interações e forma de funcionar, na realidade de uma base de dados *NoSQL*. Foi escolhida a MongoDB por duas razões importantes, em primeiro lugar, o MongoDB utiliza método de compressão de dados como o JSON, e em segundo lugar porque é bastante fácil de se trabalhar, como por exemplo, quando se insere um registo se ainda não existir nenhum daquele tipo é automaticamente criado (10gen,

2012e). Os detalhes deste tipo de base de dados podem ser encontrados no Anexo 1 (MongoDB na AXON).

6.2 Requisitos

Neste subcapítulo é sistematizado os requisitos a considerar, tanto funcionais como não funcionais para esta prova de conceito, seguindo-se *use cases* e diagramas de sequência de explicação.

6.2.1 Requisitos funcionais

Uma loja *online* de venda de livros tem que permitir ao utilizador efetuar as seguintes funcionalidades:

- Deverá aceder à aplicação por meio de um login. Sem login, o utilizador só terá acesso à página inicial com informação genérica.
- O utilizador terá a sua área pessoal (depois de fazer login), onde poderá ver que livros comprou, qual o saldo disponível e quanto já gastou;
- Poderá aceder a um catálogo de livros e desses poderá fazer opções para a compra.
- Depois de escolher um livro do catálogo poderá ver o seu detalhe.
- Ao ver o detalhe do livro, ser-lhe-á permitido comprar esse mesmo livro, sendo então direcionado para uma área de "checkout".
- Na área de checkout o utilizador poderá verificar quantas unidades deseja comprar, quanto ainda tem em carteira e qual o preço (total) do(s) livro(s) que deseja comprar.
- O utilizador ao comprar deverá ver na sua área pessoal o registo dessa compra juntamente com o resumo das outras compras.
- Para esta tese, o utilizador deve ter um contador no rodapé da aplicação, com o número de *queries* e comandos executados no sistema, servindo apenas de informação e sensibilizando o utilizador para o que é despoletado em cada ação. (Numa aplicação real este contador não traria interesse para um utilizador, mas sim para o administrador do sistema);
- O utilizador poderá consultar a base de dados em modo informático, ou seja, não muito amigável a nível de interface, apenas com os dados tratados da base de dados, de forma a conseguir gerir todas as entidades como livros ou informações que circulam no sistema, como por exemplo, que eventos é que são criados aquando da compra de um livro. (Numa aplicação real apenas teria interesse para o administrador);

A Imagem 48 demonstra o diagrama de uso de um utilizador e um administrador a interagir com o sistema como o requerido.

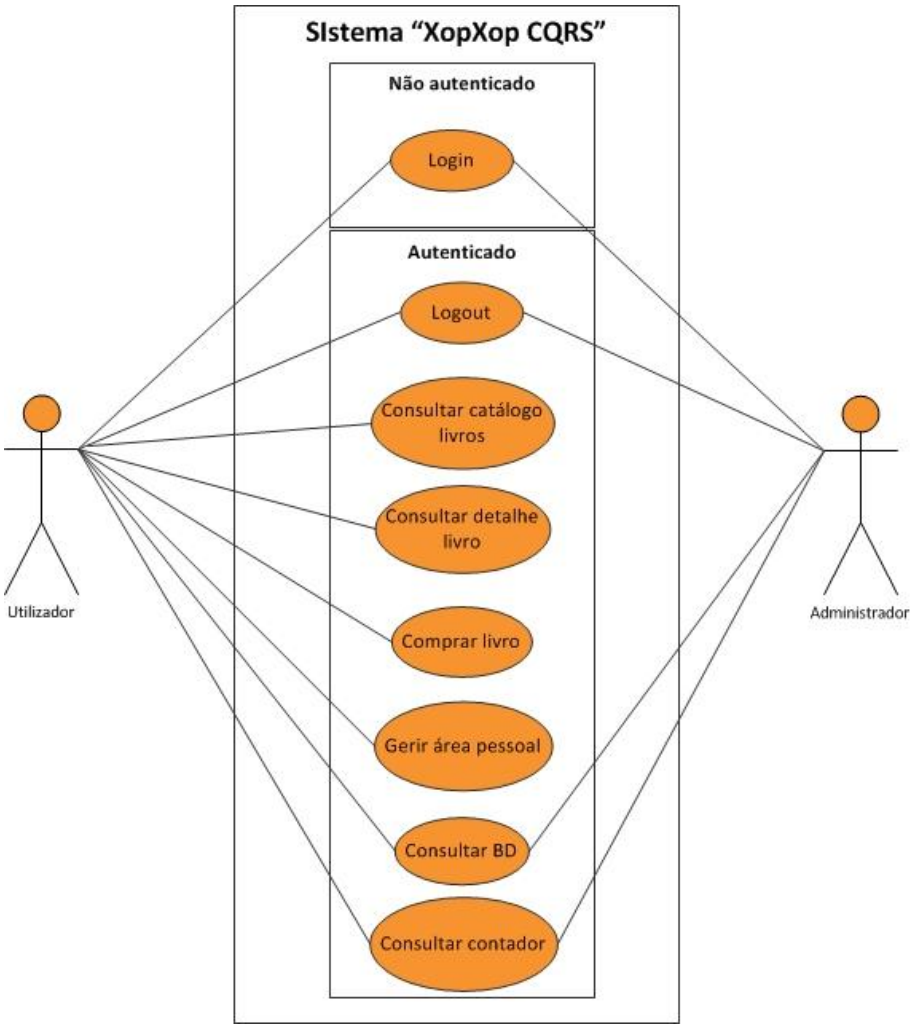

Imagem 48 – Interação do utilizador e administrador com a loja online

6.2.2 Requisitos não funcionais

A aplicação deverá sempre respeitar o padrão CQRS por forma a conseguir revelar na sua totalidade, que cada ação despoletada tem um objetivo bem definido na nova arquitetura demonstrada ao longo da tese.

Além de CQRS, deverá implementar sempre que possível eventos, para que seja possível ao administrador visualizar todas as ações que decorrem no sistema, de forma a poder ser aproveitado tanto para estatísticas como para auditorias.

Qualquer que seja a nova funcionalidade, ela deverá ser passível de implementar através de *queries* e/ou comandos, conseguindo assim utilizar sempre este modelo proposto.

Não sendo muito aprofundado nesta tese o tema de *NoSQL*, a prova de conceito deverá utilizar um motor de base de dados deste formato, para que ela possa ser utilizada e desenvolvida no futuro.

6.3 Design

Neste capítulo pretende-se apresentar o *design* do *software* proposto para responder aos requisitos identificados anteriormente. Este *design* baseia-se em diagramas de classes e de sequências de forma a ser possível compreender como é que o padrão CQRS define cada parte da arquitetura.

6.3.1 Visualizar catálogo de livros e ver detalhe

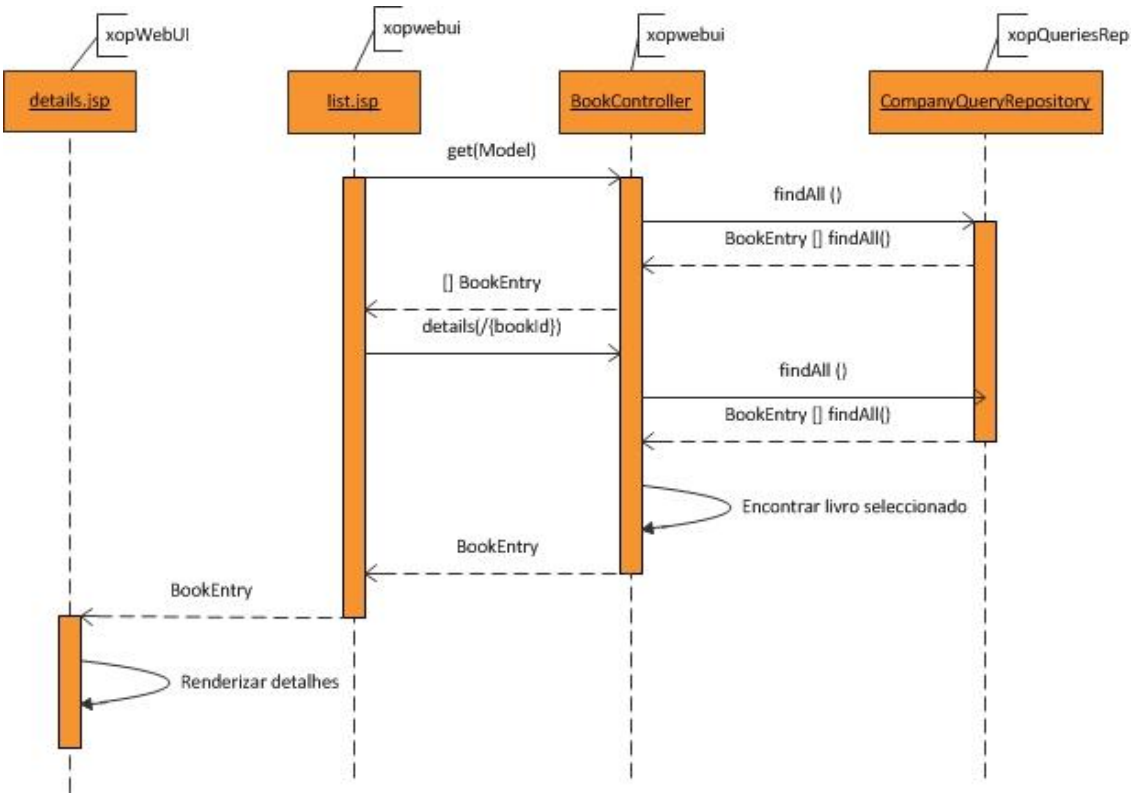

Imagem 49 – Diagrama de sequência do catálogo e detalhe de um livro

Na Imagem 49 é possível ver que o utilizador ao interagir com a aplicação web, acede a uma página denominada "list.jsp" a qual tem a responsabilidade de invocar o gestor de livros, denominado "*BookController*". Este gestor é que é capaz de invocar as *queries*, e desta forma invocar também, o método *findAll()* no repositório da empresa denominado

"*CompanyQueryRepository*", que permite retornar todos os dados existentes na base de dados sobre a forma de *BookEntry* (representa um registo de um livro). Após existir uma lista, o utilizador deseja ver o detalhe do livro, como por exemplo o ISBN (*International Standard Book Number*) ou a descrição total do livro e para isso, é feita novamente uma chamada ao repositório de forma a retornar todo o detalhe de um determinado livro, o qual depois é processado no gestor (*BookController*) para apresentar ao utilizador toda a informação estruturada. Por fim, a JSP details.jsp processa os dados e apresenta-os ao utilizador que poderá optar por comprar.

6.3.2 Atualizar contador

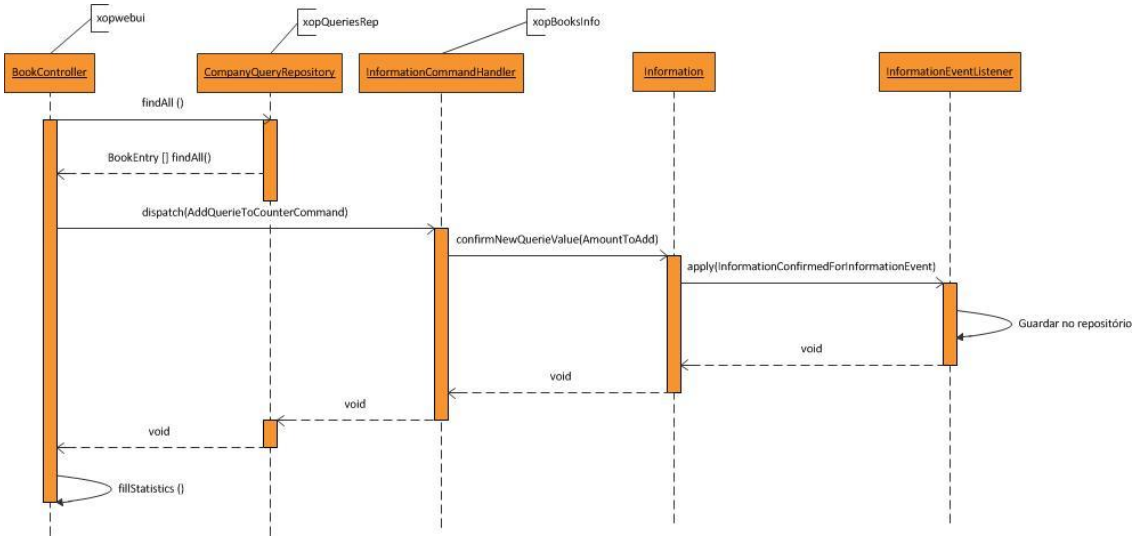

Imagem 50 – Diagrama de sequência do contador

Para esta tese importa realçar o número de *queries* e comandos que são executados no sistema, a cada ação do utilizador, mas também o total de todo o sistema. Como se pode ver pela Imagem 50, tudo é despoletado no controlador de gestão de livros (neste exemplo, todos os outros gestores tem a mesma capacidade) em que após consultar todos os livros existentes na base de dados, o controlador irá enviar um comando com o valor de quantas *queries* é que foram executadas. Desta forma, a informação é processada pelo *InformationCommandHandler.* Este comando, ao receber a informação envia para o objeto de domínio *Information* a informação da quantidade que é para adicionar. Esta operação dá origem a um evento denominado *InformationConfirmedForInformationEvent*, de forma a esta atualização ficar registada sobre a forma de uma espécie de log. Ao enviar este evento, é depois consumido pelo *InformationEventListener*, que o processa e guarda retornando *void* até ao gestor de livros novamente.

Por fim, o *BookController* processa os dados para serem apresentados na interface. A atualização do contador não acontece apenas nesta situação de pesquisar todos os livros, mas sim sempre que é executada qualquer *querie* no sistema. No entanto a título exemplificativo,

apenas é apresentada uma das situações possíveis. Em relação à atualização do número de comandos executados, a sequência de passos é parcialmente igual. Apenas é mudado o comando enviado, passando a ser o "*AddCommandToCounterCommand*" e o evento "*AddCommandCountConfirmedFromCounterEvent*".

6.3.3 Compra de livro

6.3.3.1 Registo da compra

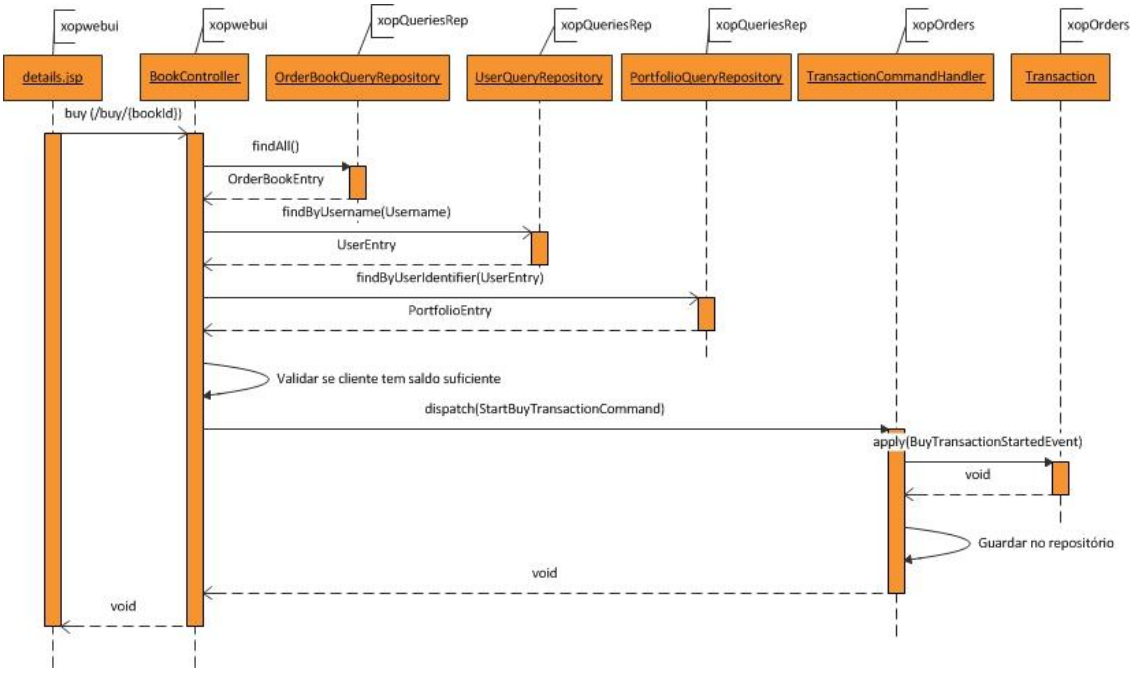

Imagem 51 – Diagrama de sequência do registo de compra

Ao ver os detalhes de um livro, o utilizador pode comprar esse mesmo livro e nesse caso são necessárias várias validações Na Imagem 51, o processo é iniciado na JSP dos detalhes. Aí, começa por ser enviado um pedido com o ID do livro, para o gestor de livros. O *BookController* pesquisa por esse mesmo livro de forma a obter todos os detalhes, para que possa ter informação atempadamente, nomeadamente, se existe em *stock* o livro naquele preciso momento. Após esta confirmação, verificamos a segurança da aplicação web e de quem está a utilizá-la, e com isso, verifica-se se o utilizador que está autenticado no *site* é o mesmo a quem estamos a tentar enviar o pedido de compra. Para esta verificação é utilizado o método *findByUsername* e retornado o *UserEntry*.

Seguidamente são pesquisados os detalhes do utilizador através do *findByUserIdentifier*, para que possamos ter todo o historial do utilizador incluindo todas as compras, através do objeto *PortfolioEntry*. Um ponto importante será verificar novamente se o utilizador tem saldo suficiente para fazer a compra através do seu detalhe retornado anteriormente, no entanto a primeira ação neste processo é a nível de interface consistindo em verificar imediatamente se

o utilizador tem saldo, para que não seja enviado nenhum comando desnecessariamente, tal como diz o padrão CQRS.

Depois da segunda verificação, é enviado o comando *StartBuyTransactionCommand* que permite registar a compra com todos os detalhes no *TransactionCommandHandler*, o qual envia o evento denominado *BuyTransactionStartedEvent* passando para o objeto de domínio *Transaction*. Tal como indica o padrão CQRS depois deste processo é retornado *void*.

6.3.3.2 Atualização de saldo

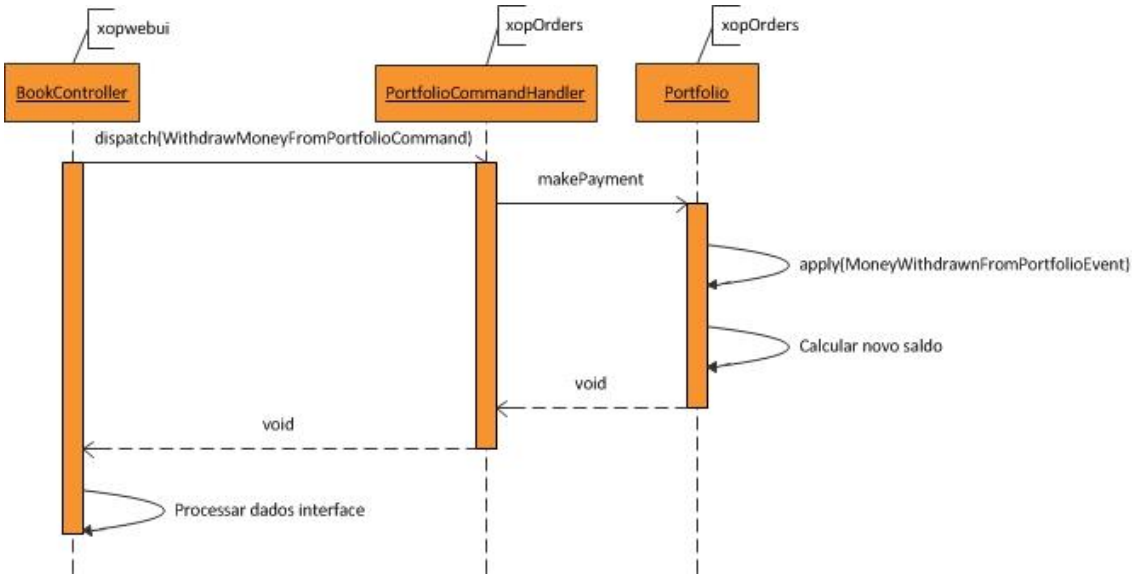

Imagem 52 – Diagrama sequência da atualização do saldo

Depois de realizado o registo da compra (Imagem 51), é necessário atualizar a conta do utilizador com o novo saldo, e pela Imagem 52 é possível verificar que é enviado um comando denominado "*WithdrawMoneyFromPortfolioCommand*" e que é interpretado pelo *PortfolioCommandHandler*. Desta forma, o *handler* envia a ordem de pagamento para o objeto de domínio Portfolio a qual envia e processa o próprio evento. Neste caso o evento *MoneyWithdrawnFromPortfolioEvent* fornece informação para a entidade sobre a quantidade que se pretende retirar da conta. De seguida é processado o novo saldo com essa informação. Ao chegar novamente ao *BookController*, este envia a nova informação para a interface depois de processada. Esta sequência de passos acontece, de uma forma geral, imediatamente a seguir à compra de um livro, no entanto poderá ser usada separadamente.

6.3.4 Área pessoal

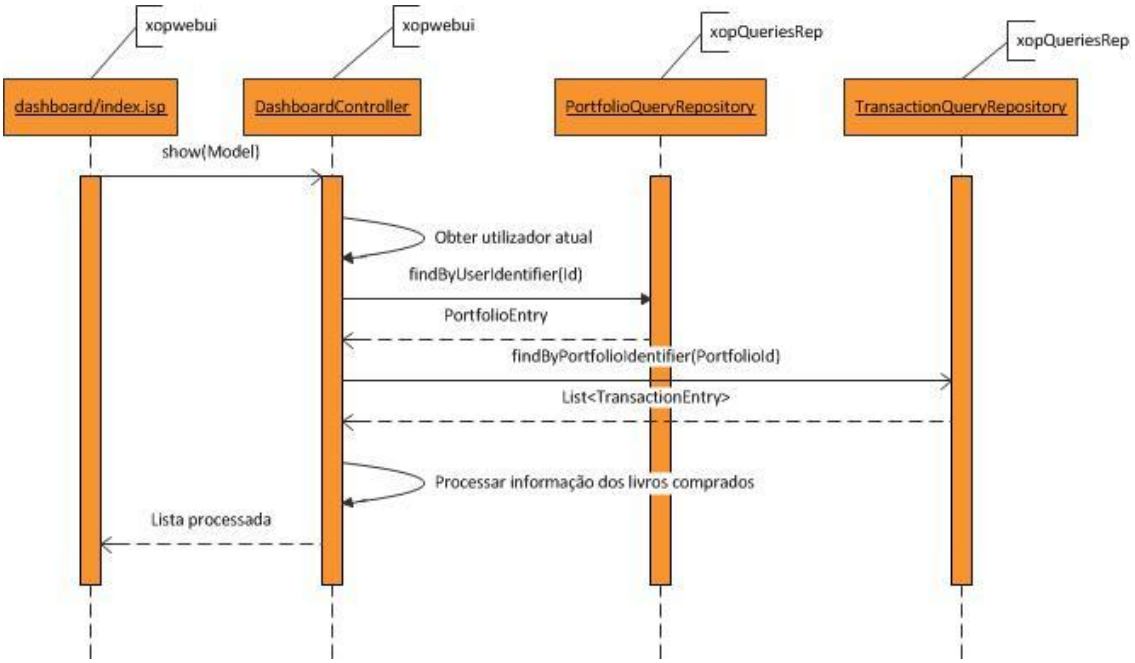

Imagem 53 – Diagrama sequência do acesso a área pessoal

Outro requisito definido anteriormente foi a capacidade do sistema ter uma área pessoal para o utilizador conseguir verificar quanto é que tem de saldo, quais os livros que já comprou, quanto é que já gastou, etc. Desta forma e pela Imagem 53, existe a nível de interface uma JSP no grupo da "*Dashboard*", que começa com o fluxo. Este pedido é feito através de um método utilizado "*show*", que solicita informações sobre um determinado utilizador. Isto faz com que o processamento passe para o *DashboardController* que tem a responsabilidade de determinar qual é o utilizador atual e de seguida enviar um pedido ao repositório de Portfolios (*querie*) através do método "*findByUserIdentifier*" para obter os detalhes do utilizador (*PortfolioEntry*).

Após estes passos, é necessário determinar tudo o que ele já comprou no *site* e para isso é necessário fazer uma *querie* ao repositório de transações através do método *findByPortfolioIdentifier*, que devolve uma lista com todas as transações. Por fim e novamente no *Controller* da área pessoal, é processada toda a informação, ou seja os detalhes das transações efetuadas no *site, mas detalhes pessoais, como o saldo,* são devolvidos para a JSP para serem apresentados ao utilizador.

6.3.5 Resumo

De forma a interligar todos os subcapítulos anteriores e para se perceber a ligação entre as partes integrantes do sistema, apresenta-se de seguida, um diagrama exemplificativo de como funcionará o circuito de compra de um livro.

Imagem 54 – Diagrama de atividade

Com o diagrama de atividade da Imagem 54, é possível perceber que o utilizador começa por aceder à aplicação web na qual necessita de realizar o respetivo login para aceder à lista de livros disponíveis para compra. Para esta ação são realizadas *queries* ao sistema, e por isso é necessário atualizar o contador para que o utilizador tenha a perceção de que foi necessário fazer algo para que aquela determinada informação aparecesse no ecrã.

De seguida o utilizador poderá verificar que livro é que deseja comprar, e dessa forma terá interesse em visualizar detalhes desse livro, como a descrição mais alargada, o ISBN, a data de publicação, etc. Da mesma forma são necessárias *queries* para o fazer e por isso atualiza-se de seguida o contador de *queries* e comandos. O utilizador ao ter a certeza do livro que pretende comprar terá à sua disposição, um botão, no qual lhe é permitido comprar o livro, seguindo-se a escolha da quantidade desejada. Mais uma vez é preciso, neste caso, realizar comandos, tendo em vista que se pretende adicionar uma nova compra, retirar o saldo ao utilizador, adicionar ao seu histórico, etc., e por isso é necessário voltar a atualizar os contadores. Internamente existe uma confirmação da compra que não é usada a nível de interface.

Por fim o utilizador poderá aceder à sua área pessoal para ver tudo o que comprou e a situação da sua conta, dos seus movimentos., após a última compra.

6.4 Implementação

6.4.1 Projetos base

Para que toda a *framework* trabalhe, existe um projeto denominado "*axon-core*" (https://github.com/AxonFramework/AxonFramework) que tal como o nome indica, é toda a base da *framework*. O *axon-core*, contém, para além de *templates* que exemplificam a forma mais correta de implementação, todos os mecanismos de gestão, onde se inclui a gestão de repositórios, comandos, eventos e sagas.

Na Imagem 55 é possível verificar o projeto base do Axon.

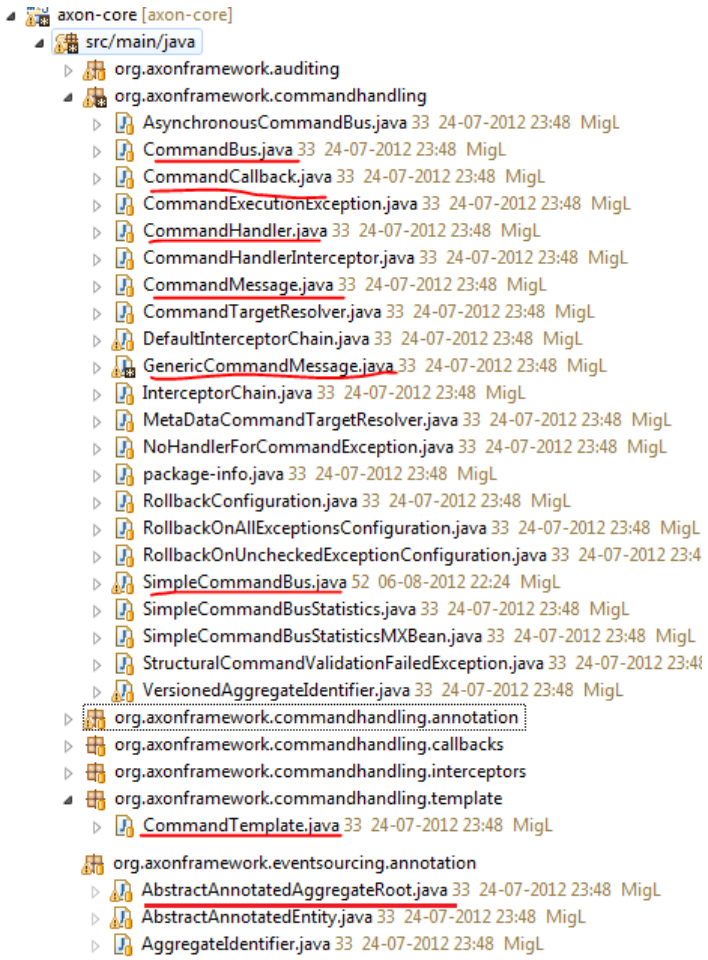

Imagem 55 – Projeto base da axon framework

Pela Imagem 55 podemos ver (sublinhado a vermelho) no nível mais baixo, a interface como "*CommandBus*" encarregue da gestão de comandos contendo os principais métodos "*dispatch*", responsável por executar tudo o que é relativo a um comando subscrito, retornando *void* tal como é suposto em CQRS. Outro método é o "*subscribe*" que é responsável por subscrever um comando ao seu respetivo tipo de *command handler*. A implementação para esta interface é feita pelo "*SimpleCommandBus*" em que, por exemplo, se pode ver a implementação do método *dispatch* (Imagem 56):

```
public void dispatch(CommandMessage<?> command) {
        CommandHandler commandHandler = findCommandHandlerFor(command);
        try {doDispatch(command, commandHandler); ( . . .)
private Object doDispatch(CommandMessage<?> command, CommandHandler
commandHandler) throws Throwable {
        UnitOfWork unitOfWork = unitOfWorkFactory.createUnitOfWork();
        InterceptorChain chain = new DefaultInterceptorChain(command,
unitOfWork, commandHandler, interceptors);
```

Imagem 56 – Implementação do método dispatch

No método público da Imagem 56 podemos ver que é pesquisado um *handler* para o comando em questão de forma a proceder ao seu "*dispatch*" de acordo com a sua própria implementação. De seguida é delegado num método interno que o trata através das unidades de trabalho e de métodos internos à *framework*.

Para a definição e implementação de objetos de domínio, como por exemplo, "Livro", é estendida a classe "*AbstractAnnotatedAggregateRoot*" que nos permite usufruir de mecanismos já implementados de eventos e de identificação do próprio objeto. Com isto, é possível enviar diretamente um evento aquando da criação dos objetos de domínio para que fiquem registados todos os dados da sua criação. Esta classe tem como interface "*AggregateRoot*".

Existe outro projeto mais utilizado pelo padrão de eventos, que permite a sua integração no sistema.

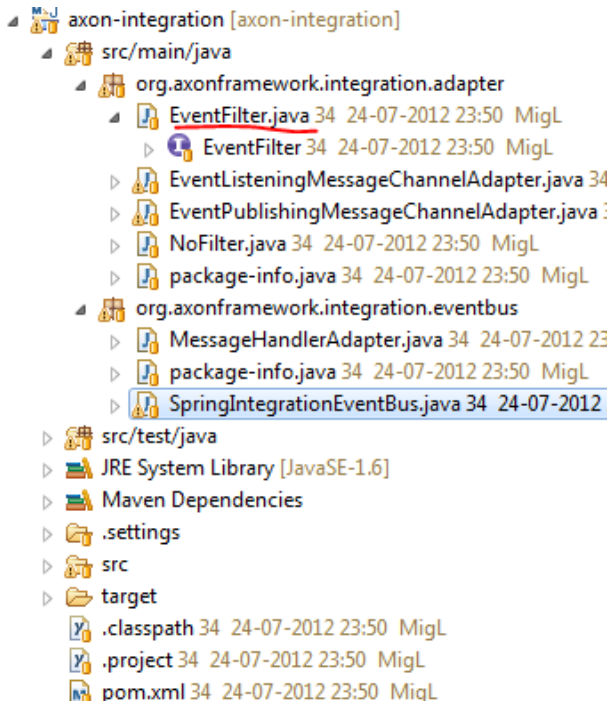

Imagem 57 – Projeto integração eventos na AXON

Pela Imagem 57 é possível verificar que este projeto tem o objetivo de oferecer uma interface comum para os eventos. Essa interface comum, permite decidir quais os eventos que são aceites e reencaminhados pelos "*adapters*" e quais é que são rejeitados. Assim, ao criar um evento, se mais tarde ele necessitar de ser modificado, ou até, se existir uma nova versão e for necessário descontinuar a antiga, a melhor forma será, atualizar este filtro para que o antigo não seja registado.

No caso da AXON existe um exemplo denominado "*EventListeningMessageChannelAdapter*", que está à espera de um filtro *filter*:

```
public class EventListeningMessageChannelAdapter implements EventListener,
InitializingBean {

    private final MessageChannel channel;
    private final EventFilter filter;
    private final EventBus eventBus;

    (...)
    @Override
    public void afterPropertiesSet() {
        eventBus.subscribe(this);
    }
    @Override
    public void handle(EventMessage event) {
        if (filter.accept(event.getPayload().getClass())) {
            channel.send(new GenericMessage<Object>(event.getPayload()));
        }
```

Imagem 58 – Código filtro de eventos

Pela Imagem 58 é possível verificar que existe o atributo da classe "*filter*" baseada na interface falada anteriormente, em que seguidamente é utilizado no respetivo *handler*. Isto permite averiguar se irá aceitar aquele evento ou não mediante a respetiva classe. Caso seja aceite, o evento é enviado e o processamento da sua mensagem será alvo de tratamento da classe que trata da sua publicação no sistema, denominada "*EventPublishingMessageChannelAdapter*":

```
public class EventPublishingMessageChannelAdapter implements MessageHandler
{
    private final EventFilter filter;
    private final EventBus eventBus;
( . . .)
    public void handleMessage(Message<?> message) {
        Class<?> eventType = message.getPayload().getClass();
        if (filter.accept(eventType)) {
            eventBus.publish(new GenericEventMessage(message.getPayload(),
message.getHeaders()));
        }
    }
}
```

Imagem 59 – Publicar e registar o evento

Com a Imagem 59 facilmente se conclui que existe também o atributo com o filtro, mas neste caso o objetivo não passa apenas por verificar se irá ser aceite ou não, mas sim por publicar e registar no sistema a mensagem contida no evento, ou seja, a mensagem em si contém o evento a publicar com todos os detalhes, como os dados. Caso assim não aconteça, é enviada uma exceção dizendo que aquele tipo de eventos está bloqueado pelo sistema. Com esta lógica o sistema permite para além de garantir que apenas os eventos necessários são registados e enviados, também há a garantia de alguma segurança pois não permite que

qualquer sistema que queira utilizar os eventos consiga enviar eventos, e portanto, também não permite que sejam todos aceites. No nosso caso especifico da prova de conceito de CQRS a implementação destes filtros não foi necessária e por isso foi utilizada a classe "NoFilter" que tal como o nome indica, não é aplicada a nenhum filtro processando os eventos sem restrições.

6.4.2 Projetos adicionais

Os projetos demonstrados de seguida são baseados em alguns samples da *framework* retirados de http://www.axon*framework*.org/samples/.

Pela Imagem 60 é possível ver o projeto *xopBooksInfoAPI* que é caracterizado por conter o objeto *Book*, que contém informação acerca dos livros da loja e *Information* informações acerca da loja de forma genérica (neste caso exemplo, tem os contadores que são necessários mostrar ao utilizador).

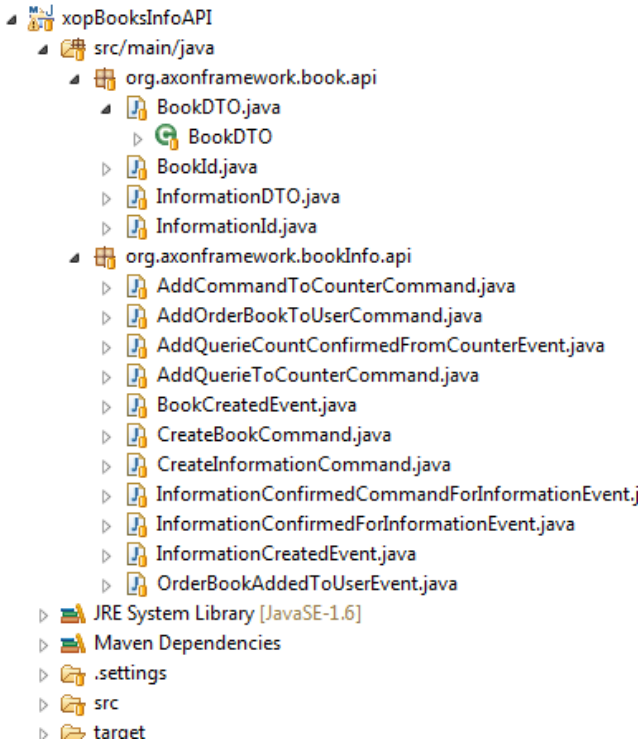

Imagem 60 – Projeto xopBooksInfoAPI

Esta Imagem 60, ajuda a perceber o que aparecerá ao utilizador e o que é passado nos comandos e *queries.* É conveniente conhecer o detalhe destes dois objetos, visto serem importantes para todo o sistema.

Detalhe de *InformationDTO* e *BookDTO* (Imagem 61):

```
public class InformationDTO {
  private InformationId informationId;
  private long queriesExecuted;
  private long commandsExecuted;
  }
public class BookDTO {
  private BookId bookId;
  private UserId userId;
  private String title;
  private Double price;
  private String shDescription;
  private String fullDescription;
  private String author;
  private String publisher;
  private Date releaseDate;
  private String isbn;
  private int pages;
  }
```

Imagem 61 – Detalhes dos objetos Information e Book

Neste projeto estão alguns comandos que são enviados para o sistema, como a inserção de livros e de informações da loja, denominados "*CreateBookCommand*" e "*CreateInformationCommand*" respetivamente. Cada um deles tem como atributo o seu respetivo DTO (Imagem 61) que permite passar assim para o comando a informação que queremos ver persistida no sistema.

Além disso, também existem alguns eventos que foram criados para que fique registado e novamente reprocessado caso aconteça algum erro. Um exemplo é o *BookCreatedEvent* ou o *OrderBookAddedToUserEvent*, em que no primeiro caso tal como no comando importa guardar o DTO respetivo do livro e no segundo caso guarda a compra de um livro por um determinado utilizador, em que para ser processado novamente interessa o ID do livro e o ID da compra.

Outros comandos, nomeadamente o *AddCommandToCounterCommand* é responsável por adicionar a execução de um ou vários comandos no contador enquanto o *AddQuerieToCounterCommand* adiciona ao contador das *queries*, contendo estes dois também os seus respetivos eventos *AddQuerieCountConfirmedFromCounterEvent* e *AddCommandCountConfirmedFromCounterEvent*.

Na Imagem 62 é possível ver o detalhe do projeto *xopBooksInfo.*

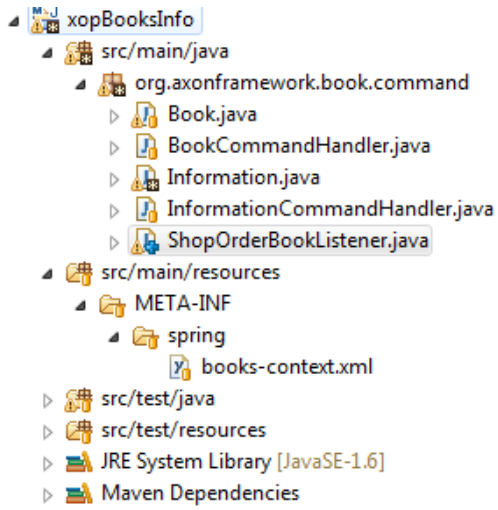

Imagem 62 – Projecto xopBooksInfo

O projeto *xopBooksInfo* (Imagem 62) tem a implementação dos *command handlers* das entidades *Book* e *Information*, que se caracterizam por conterem o seu próprio repositório para guardarem numa espécie de fila os pedidos que lhes vão chegando, a qual vai sendo consumida ao longo do tempo. Estes pedidos no caso do *BookCommandHandler* podem ser para criar um novo Livro (objeto de domínio), ou seja, adicionar ao sistema um novo livro (*handleCreateBook*), ou adicionar uma encomenda de um determinado livro (*handleAddOrderBook*). Em termos práticos temos o seguinte código (Imagem 63):

```
public class BookCommandHandler {
 private Repository<Book> repository;
 @CommandHandler
 public void handleCreateBook(CreateBookCommand command) {
        Book book = new Book(command.getBookDTO());
        repository.add(book);
 }
 @CommandHandler
 public void handleAddOrderBook(AddOrderBookToUserCommand command) {
        Book company = repository.load(command.getBookId());
        company.addOrderBook(command.getOrderBookId());
 }
```

Imagem 63 – Código do Handler dos livros

Para o caso das informações da loja também existe o respetivo *command handler* denominado *InformationCommandHandler*, que possui os métodos *handleAddQuerieToCounterCommand* e o *handleAddCommandToCounterCommand* responsáveis, respetivamente pela atualização do número de *queries* do sistema (o primeiro) e pelos comandos (o segundo).

Este projeto ainda utiliza as vantagens do Spring através do ficheiro books-context.xml que contém os detalhes da instanciação dos dois repositórios utilizados, *bookRepository* e *informationRepository* através de *tags* específicas da *framework* AXON, exemplo (Imagem 64):

```
<beans xmlns=( … )">
    <bean class="org.axonframework.book.command.BookCommandHandler">
        <property name="repository" ref="bookRepository"/>
    </bean>
    <axon:event-sourcing-repository id="bookRepository"
                                    aggregate-
type="org.axonframework.book.command.Book"
                                    cache-ref="ehcache"
                                    event-bus="eventBus"
                                    event-store="eventStore">
        <axon:snapshotter-trigger event-count-threshold="50" snapshotter-
ref="snapshotter"/>
    </axon:event-sourcing-repository>
</beans>
```

Imagem 64 – Instanciação via spring

Em relação à compra e venda de livros, é necessário existir um "motor" bem definido e isolado que consiga gerir este tipo de transações, visto poder ser utilizado com especificidades de negócio. Para isso foram criados dois projetos separadamente. Assim, define-se claramente quais são os comandos, objetos de transferência de dados, e respetivos *handlers* que levam ao seu processamento.

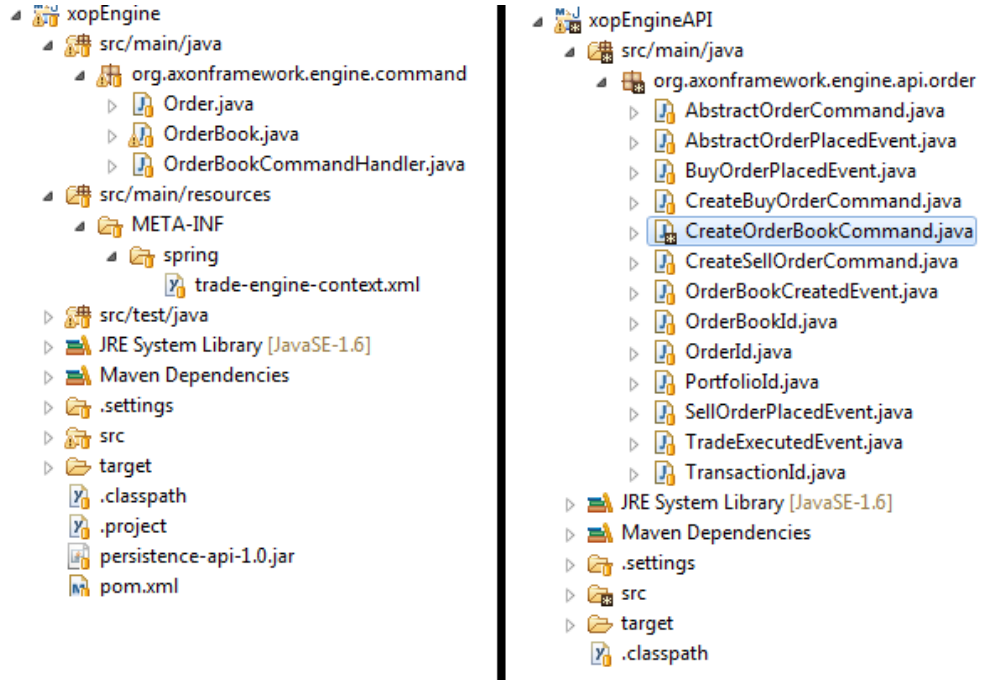

Imagem 65 – Projetos xopEngine e xopEngineAPI

Pela Imagem 65 é possível ver dois projetos responsáveis pelo registo e lógica de troca de livros (não implementado a nível interface). O objetivo destes projetos é providenciarem uma forma de ter entre vários utilizadores uma opção de troca e não apenas de venda. No *xopEngine* vemos uma entidade denominada "*Order*" que tem como atributos os detalhes de uma compra, como a identificação única no sistema de troca, qual o valor da compra, e a respetiva quantidade, ou seja (Imagem 66):

```
class Order extends AbstractAnnotatedEntity {
    private OrderId orderId;
    private TransactionId transactionId;
    private final Double itemPrice;
    private final long tradeCount;
    private final PortfolioId portfolioId;
    private long itemsRemaining;
 }
```

Imagem 66 – Classe Order

À semelhança dos outros *command handlers*, existe a classe *OrderBookCommandHandler* que é a que procede à gestão do comando recebido e neste caso poderá ser uma ordem de troca, em que uma parte quer vender e outra comprar, ou seja, temos os seguintes *handlers* (Imagem 67):

```
@CommandHandler
    public void handleBuyOrder(CreateBuyOrderCommand command) {
        OrderBook orderBook = repository.load(command.getOrderBookId(),
null);
        orderBook.addBuyOrder(command.getOrderId(),
                command.getTransactionId(),
                command.getTradeCount(),
                command.getItemPrice(),
                command.getPortfolioId());
    }

    @CommandHandler
    public void handleSellOrder(CreateSellOrderCommand command) {
        OrderBook orderBook = repository.load(command.getOrderBookId(),
null);
        orderBook.addSellOrder(command.getOrderId(),
                command.getTransactionId(),
                command.getTradeCount(),
                command.getItemPrice(),
                command.getPortfolioId());
    }
```

Imagem 67 – Classe OrderBookCommandHandler

O que é feito no *handleBuyOrder* e *handleSellOrder* é ler os detalhes do comando que é recebido e adicionar ao registo daquela troca, mais informação, como a quantidade por exemplo.

Do lado do *xopEngineAPI*, temos a definição dos comandos e respetivos eventos, que podem ser enviados para o registo. Pensemos no comando *CreateBuyOrderCommand,* que dará origem depois de processado, a um *BuyOrderPlacedEvent*, que tal como todos os eventos contém os mesmos dados que o objeto "*Order*".

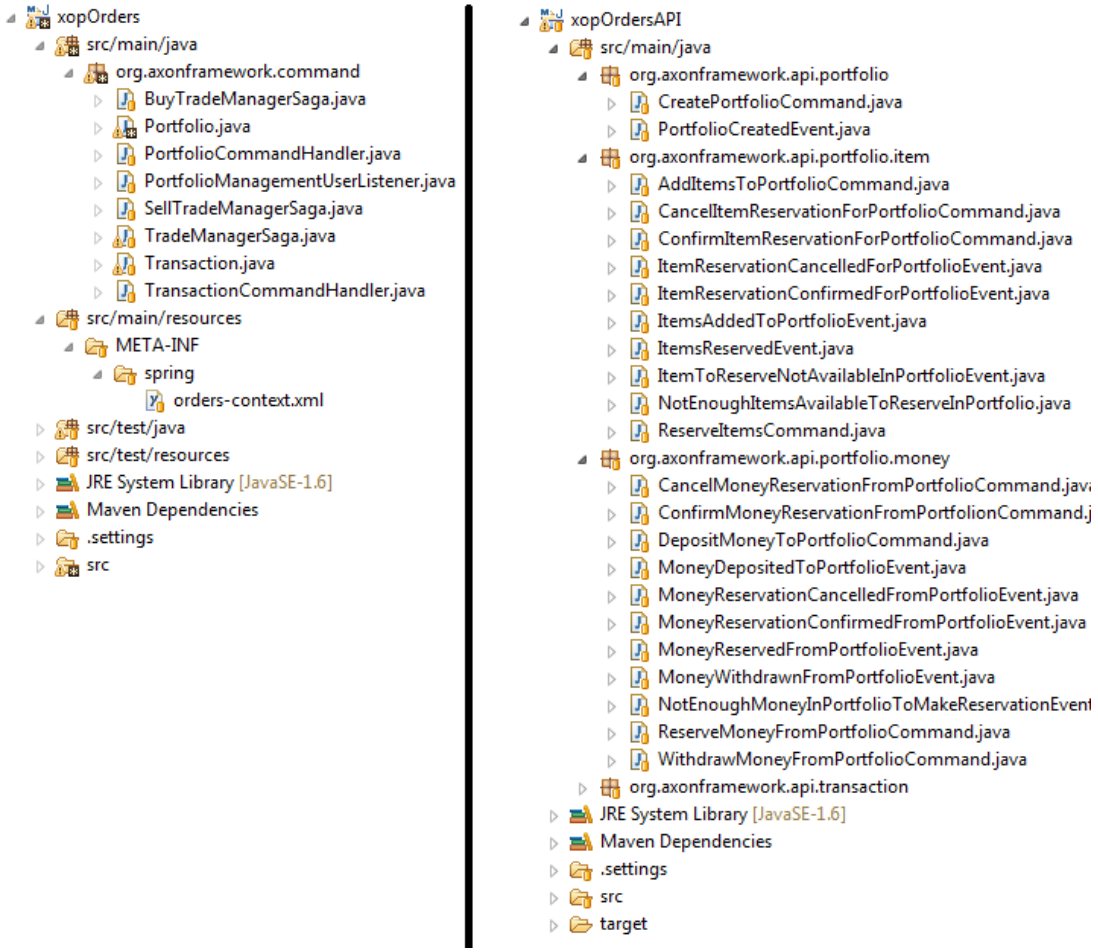

Imagem 68 – Projeto xopOrders e xopOrdersAPI

Para a compra de livros existem os dois projetos da Imagem 68. Aí, mostra-se que na *xopOrders*, se pretende ter todo o mecanismo de compra através de dois comandos essenciais, o *TransactionCommandHandler* e o *PortfolioCommandHandler*.

O *TransactionCommandHandler* tem a função de iniciar o processo de compra através de um comando recebido que contém os dados iniciais, como o preço do livro e a quantidade que deseja comprar, como se pode ver pelo código seguinte (Imagem 69):

```
    @CommandHandler
    public void handleStartBuyTransactionCommand(StartBuyTransactionCommand
command) {
Transaction transaction = new
Transaction(command.getTransactionIdentifier(),
TransactionType.BUY, command.getOrderbookIdentifier(),
command.getPortfolioIdentifier(),command.getTradeCount(),
command.getItemPrice());repository.add(transaction); }
```

Imagem 69 – Implementação do TransactionCommandHandler

Depois de criado, este novo objeto "*Transaction*", fará com que seja adicionado ao repositório mas também seja enviado um evento, com essa nova transação, ou seja compra. Verificando o construtor deste objeto e o respetivo *event handler* (Imagem 70) conclui-se que:

```
public Transaction(TransactionId transactionId,
                   TransactionType type,
                   OrderBookId orderbookIdentifier,
                   PortfolioId portfolioIdentifier,
                   long amountOfItems,
                   Double pricePerItem) {
        switch (type) {
            case BUY:
                apply(new BuyTransactionStartedEvent(transactionId,
                        orderbookIdentifier,
                        portfolioIdentifier,
                        amountOfItems,
                        pricePerItem));
                break;
@EventHandler
    public void onBuyTransactionStarted(BuyTransactionStartedEvent event) {
        this.transactionId = event.getTransactionIdentifier();
        this.amountOfItems = event.getTotalItems();
        this.type = TransactionType.BUY;
    }
```

Imagem 70 – Construtor e event handler das transações

Com isto temos o registo da compra, faltando agora realizar a dedução do preço dos livros comprados, ao saldo que o utilizador tem. Isso é feito através do envio do comando "*WithdrawMoneyFromPortfolioCommand*" do projeto *xopOrdersAPI* que contém a que utilizador é que se fará a dedução e qual o valor da mesma dedução.

Aquando da chegada ao *handler* denominado *PortfolioCommandHandler* temos o seguinte código:

```
  @CommandHandler
    public void
handleMakePaymentFromPortfolioCommand(WithdrawMoneyFromPortfolioCommand
command) {
        portfolio.makePayment(command.getAmountToPayInCents());    }
public void makePayment(Double amountToPayInCents) {
        apply(new MoneyWithdrawnFromPortfolioEvent(portfolioId,
                    amountToPayInCents));      }
@EventHandler
 public void onPaymentMadeFromPortfolio(
              MoneyWithdrawnFromPortfolioEvent event) {
        amountOfMoney -= event.getAmountPaidInCents();
```

Imagem 71 – Implementação do PortfolioCommandHandler e eventos

Pela Imagem 71 é possível verificar que ao passar os dados respetivos para a entidade "Portfolio" torna-se necessário tal como na *Transaction* enviar um evento com os respetivos dados e no *handler* colocar a lógica do evento.

O projeto *xopOrdersAPI* está preparado para mais funcionalidades como a venda, confirmação ou cancelamento de compras, no entanto, não está implementado a nível de interface porque não traria grande diferença a nível estrutural da compra genérica de livros.

Neste projeto da Imagem 72 estão localizadas as principais classes que permitem fazer *queries* a todo o sistema e entidades.

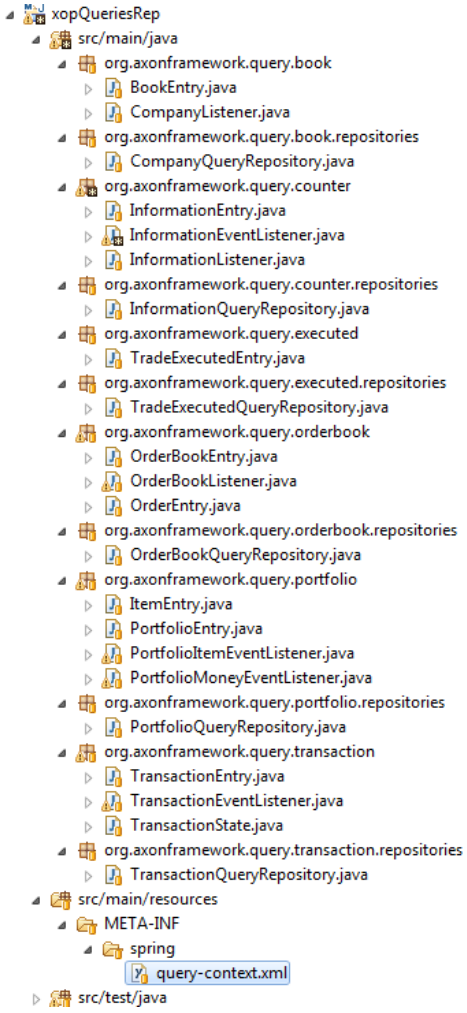

Imagem 72 – Projeto xopQueriesRep

Este projeto é principalmente usado pela interface e está separado dos outros projetos para que possa ser utilizado independente da implementação dos comandos. Aqui contém classes com a terminação "*Entry*", isto significa que representa uma entrada/registo da entidade em questão, como quando queremos o detalhe de um livro (Imagem 73):

```
public class BookEntry {
  private BookDTO bookDTO;
  public BookDTO getBookDTO() {
         return bookDTO;
  }
  public void setBookDTO(BookDTO bookDTO) {
         this.bookDTO = bookDTO;
  }
}
```

Imagem 73 – Implementação do registo de um livro (BookEntry)

Na sua área pessoal, convém que o utilizador consiga visualizar uma lista de livros que já comprou, por que preço e em que quantidade, e para isso, o objeto utilizado é o "*TransactionEntry*" com os seguintes atributos (Imagem 74):

```
public class TransactionEntry {
    @Id
    private String identifier;
    private String orderbookIdentifier;
    private String portfolioIdentifier;
    private String bookTitle;
    private long amountOfItems;
    private long amountOfExecutedItems;
    private Double pricePerItem;
    private TransactionState state;
    private TransactionType type;
}
```

Imagem 74 – Implementação de um registo de transação

Com o código acima é possível, identificar a transação através do "*identifier*", o nome do livro através do "*bookTitle*", quantos foram comprados pelo "*amoutOfItems*", a que preço pelo "*pricePerItem*" e por fim o tipo de transação (na prova de conceito será sempre compra – "BUY"). Para abastecer este objeto sobre cada transação, a forma que se arranjou foi o aproveitamento do evento que já existe, também para leitura.

Para que exista acesso aos repositórios da base de dados é necessário através do Spring instanciar tal como nos outros repositórios.

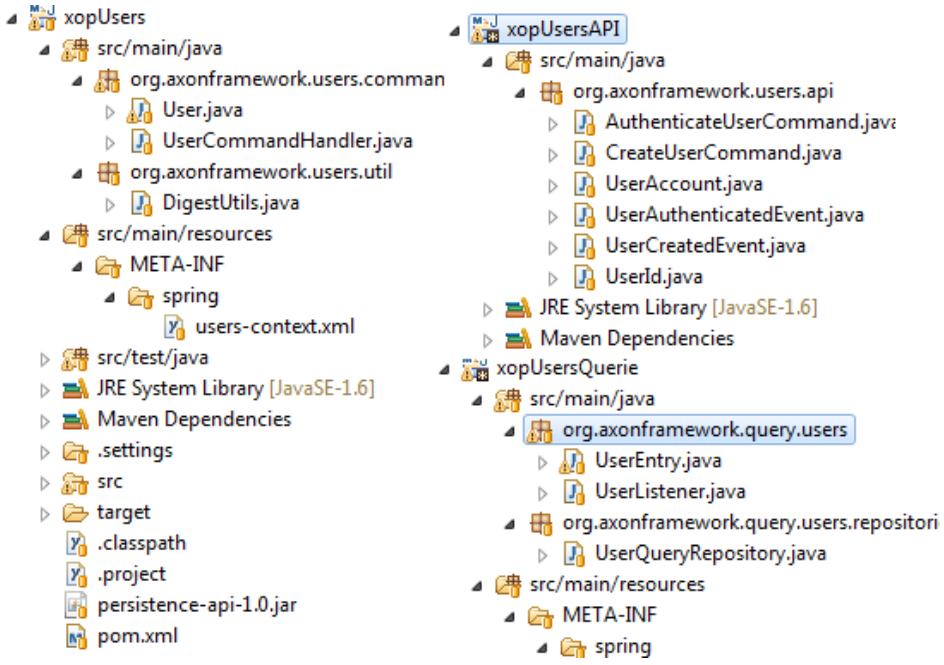

Imagem 75 – Projetos xopUsers, xopUsersQuery, xopUsersQueryAPI

Por fim existe o projeto para gestão de utilizadores, que segue a mesma linha do padrão CQRS.

Pela Imagem 75, é possível ver que o *xopUsers* contém a caracterização do objeto de domínio "*User*" que tem o respetivo *command handler*.

Este *command handler* é responsável por autenticar e registar utilizadores no sistema, e a entidade contém o *Event Handler* que regista os eventos que chegam através de comandos.

No projeto *xopUsersAPI* existem os comandos "*CreateUserCommand*" e "*AuthenticateUserCommand*", que servem para criar e autenticar respetivamente e são comunicados ao *xopUsers* para o respetivo *listener*.

A nível de *queries* que se poderá fazer ao sistema sobre utilizadores, estas estão guardadas no projeto *xopUsersQuerie* através das já denominadas *entries*,e neste caso, temos (Imagem 76):

```
public class UserEntry implements UserAccount, Serializable{

  private String identifier;
  private String name;
  private String username;
  private String password;
}
```

Imagem 76 – Implementação do registo de utilizador

6.4.3 Interface com o utilizador

Neste subcapítulo é explicado como é que foi desenhada a parte da interface e como é que ela comunica entre si e com os outros componentes. Pretende-se dar a conhecer uma forma de o utilizador ver um sistema onde poderá comprar livros e para o interesse desta tese verificar que se pode utilizar os conceitos de CQRS a qualquer *site* em que se poderá fazer todas as funcionalidades sem restrições.

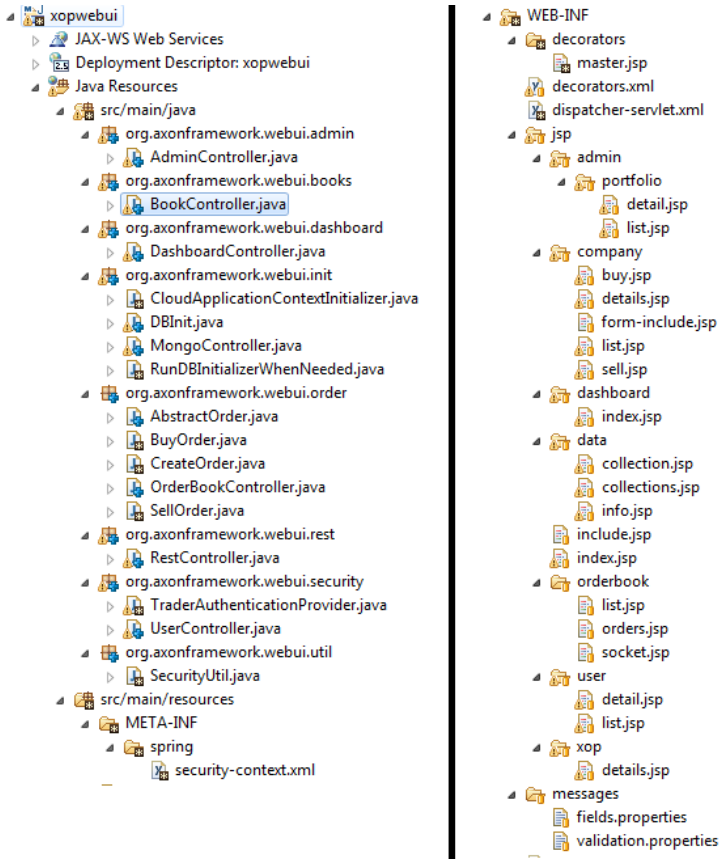

Imagem 77 – Classes do projeto web xopwebui

Neste projeto web temos a separação por funcionalidade a disponibilizar ao utilizador, ou seja, pela Imagem 77 podemos ver que do lado esquerdo temos a parte das classes que contém a lógica da apresentação e onde são invocados os comandos e *queries* e do lado direito a parte gráfica onde se podem ver as JSPs (*JavaServer Pages*). Tanto uma parte como outra estão divididas por funcionalidades, por exemplo, onde se pode ler "*admin*", cabe ao administrador de sistema conseguir gerir todo o sistema a partir deste ponto. Foi desenhado para poder verificar todos os dados do sistema, como se fosse um resumo, para assim ter entre outras visões, uma visão de gestão de *stocks*. Para a gestão da área pessoal de um utilizador temos a chamada "*dashboard*", onde o utilizador poderá ver o resumo da sua conta pessoal, como as compras que fez, quanto dinheiro ainda tem, quanto é que já gastou, etc. Nesta área e em termos mais técnicos isto é acedido através do respetivo "*find*" denominado "*findByPortfolioIdentifier* "na classe "*DashboardController*".

Com isto conseguimos obter todas as compras de um determinado utilizador ao aceder ao repositório, e ao adicionar ao "*model*" (sendo este a nossa interface) podemos passar para a JSP /dashboard/index.jsp que lista toda esta informação.

Para o catálogo de livros existe o "*BookController*", que tal como o nome indica, tem as funções de gerir a parte do catálogo de livros e respetivas compras. Esta classe invoca os comandos de compra de um livro e de diminuição de saldo do utilizador, como podemos ver no seguinte código (Imagem 78):

```
StartBuyTransactionCommand command = new StartBuyTransactionCommand(
                            new TransactionId(), new OrderBookId(
                                        bookEntry.getIdentifier()), new
PortfolioId(
                                        portfolioEntry.getIdentifier()),
                            order.getTradeCount(), order.getBookPrice());
                commandBus
                            .dispatch(new
GenericCommandMessage<StartBuyTransactionCommand>(
                                        command));
                WithdrawMoneyFromPortfolioCommand commandAmount = new
WithdrawMoneyFromPortfolioCommand(
                            new
PortfolioId(portfolioEntry.getIdentifier()),
                            order.getTradeCount() * order.getBookPrice());
                commandBus
                            .dispatch(new
GenericCommandMessage<WithdrawMoneyFromPortfolioCommand>(
                                        commandAmount));
```

Imagem 78 – Invocação da compra do livro

Estes comandos são invocados a partir da interface /company/buy.jsp, que após a visualização do detalhe do livro, reencaminha para um formulário onde o utilizador irá escolher a quantidade que quer mediante o saldo que tem. Neste formulário irá encontrar-se um ponto falado em CQRS que é a pré-validação, ou seja, antes de submetermos um comando diretamente, verifica-se primeiro se o utilizador tem saldo suficiente para o fazer logo na interface, de forma a não submeter comandos que irão ser rejeitados posteriormente.

Para esta tese, também se tornou importante facultar no rodapé do *website* um contador que permitisse ao utilizador verificar quantas *queries* e comandos são executados em cada ação e no total. Com tal objetivo em mente, foi implementado um mecanismo que se baseia também no padrão CQRS em comandos e *queries* para obter e registar essa informação. Assim, atualizamos o contador através de comandos e, visualizamos o contador através das *queries*. Para o preenchimento na interface temos o seguinte código (Imagem 79):

```
Iterable<InformationEntry> informationEntrys =
informationQueryRepository
                    .findAll();
        InformationEntry informationEntry = new InformationEntry();
        for (InformationEntry entry : informationEntrys) {
              (…)

        }
        model.addAttribute("countQ", String.valueOf(informationEntry
                    .getInformationDTO().getQueriesExecuted()));
        model.addAttribute("countC", String.valueOf(informationEntry
                    .getInformationDTO().getCommandsExecuted()));
```

Imagem 79 – Implementação do contador de queries

Podemos desta forma, aceder a todas as informações, através de uma *querie* e neste caso ao encontrar a informação do contador preenchemos na interface. Pela Imagem 80 podemos verificar a atualização do contador em que é enviado um comando da seguinte forma (Exemplo para atualização do numero de *queries*):

```
        AddQuerieToCounterCommand addQuerieToCounterCommand = new
AddQuerieToCounterCommand(
                    1,
informationEntry.getInformationDTO().getInformationId());

        commandBus
                    .dispatch(new
GenericCommandMessage<AddQuerieToCounterCommand>(
                                    addQuerieToCounterCommand));
```

Imagem 80 – Atualização do contador

Por fim, na parte da implementação é possível aceder à base de dados de uma forma mais simples do que aceder diretamente, já que a base de dados está em MongoDB e por isso guarda os dados em BSON, o que não torna os dados legíveis para humanos. Essa forma é a JSP /dta/collection.jsp em que se pode ver por exemplo um utilizador criado (Imagem 81):

_id	50561909008555449c092428
_class	org.axonframework.query.users.UserEntry
identifier	97f0e621-0e0c-4193-b439-e66130195625
name	Buyer One
username	buyer1
password	ec5c9b91242f18e1ac487661d8b3ea4206701e0e

Imagem 81 – Interface utilizador criado

Ou por exemplo um evento de criação de utilizador (Imagem 82):

Number of pages : 22

_id	50561909008555449c092426
aggregateIdentifier	97f0e621-0e0c-4193-b439-e66130195625
sequenceNumber	0
serializedPayload	<org.axonframework.users.api.UserCreatedEvent><userId><identifier>97f0e621-0e0c-4193-b439-e66130195625</identifier></userId><username>buyer1</username><name>Buyer One</name><password>ec5c9b91242f18e1ac487661d8b3ea4206701e0e</password></org.axonframework.users.api.UserCreatedEvent>
timeStamp	2012-09-16T19:23:05.070+01:00
type	User
payloadType	org.axonframework.users.api.UserCreatedEvent
payloadRevision	
serializedMetaData	<meta-data/>

Imagem 82 – Interface evento de utilizador criado

6.4.4 Exemplo de utilização

Neste subcapítulo irá ser demonstrado partes da interface faladas nos capítulos anteriores.

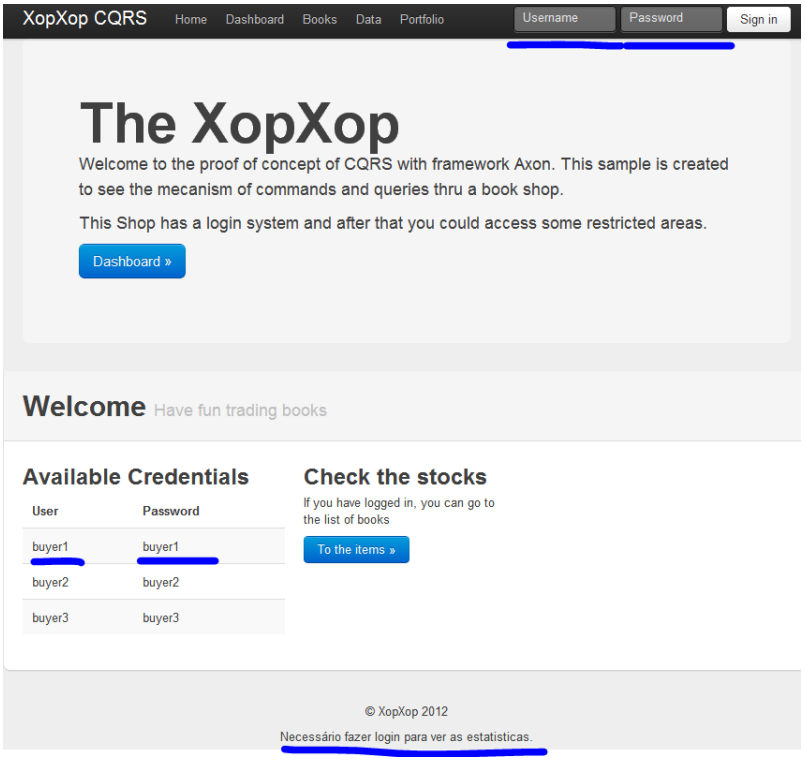

Imagem 83 – Página de login da aplicação exemplo

Tal como referido anteriormente, o primeiro passo para se começar a usar a aplicação web, será efetuar o login. Utilizando a Imagem 83 podemos ver que temos três utilizadores para fazer login (as credenciais serão apenas apresentadas na aplicação exemplo).

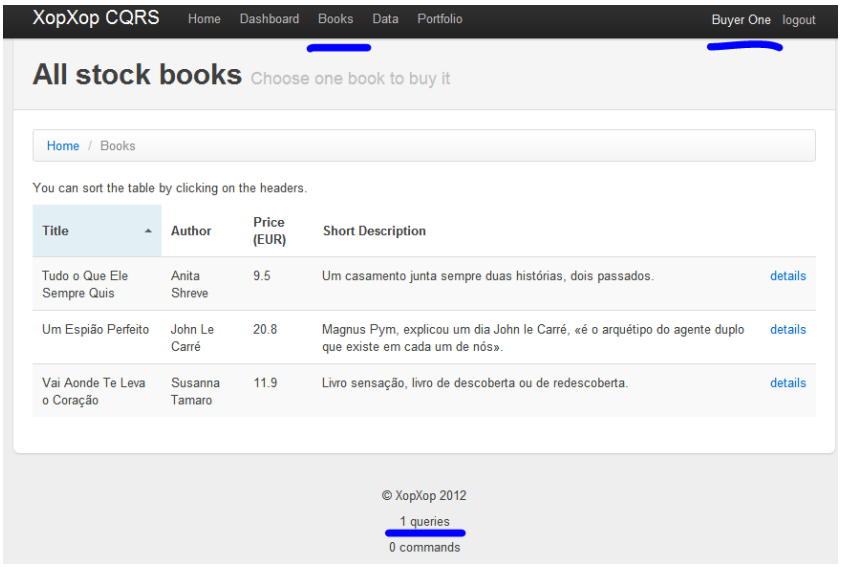

Imagem 84 – Catálogo de livros

Depois de efetuado o login, e ter clicado em *Books*, é apresentado um catálogo de livros no qual ao clicar em "*Details*" poderá ver os detalhe do livro escolhido. Note-se que o contador já apresenta a execução de uma *querie* executada no sistema (Imagem 84).

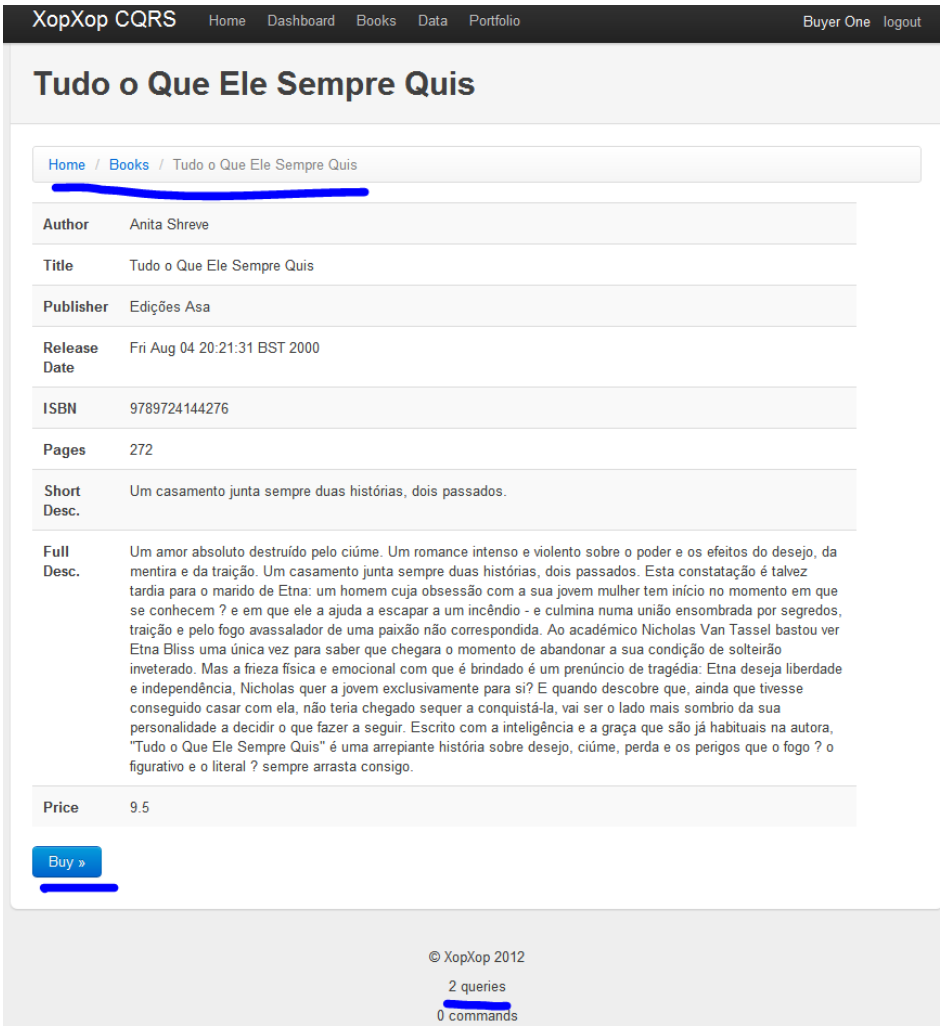

Imagem 85 – Detalhes de um livro

Ao clicar em "*Details*" é apresentado uma interface como a Imagem 85, em que temos todo o detalhe de um determinado livro e temos a opção de o comprar, sendo que neste momento já temos duas *queries* executadas no sistema.

Depois de clicar em "*Buy*" é apresentada a Imagem 86.

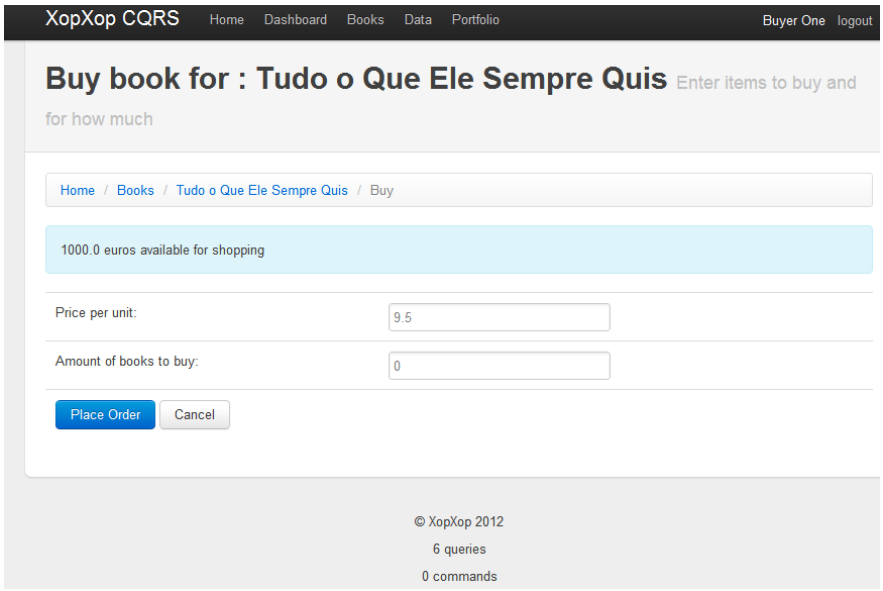

Imagem 86 – Comprar um livro

Com a ajuda da Imagem 86 podemos ver que já temos os detalhes da conta do utilizador, ou seja, o seu saldo, e é possível agora escolher a quantidade de livros que se deseja comprar. De forma a confirmar a compra é necessário clicar em "*Place Order*". Para obter todos os detalhes para comprar foi necessário realizar mais *queries* o que fez com o que o contador também aumentasse.

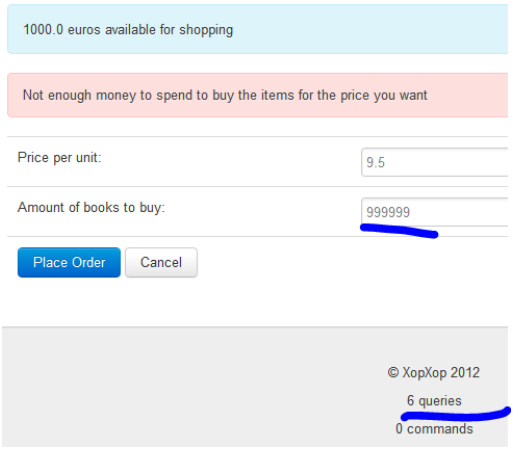

Imagem 87 – Pré validação de dados

Tal como indica o padrão CQRS, a Imagem 87 demonstra que no caso de o utilizador não possuir saldo suficiente para a compra, isto é pré-validado a nível de interface de forma a não enviar nenhum comando nem *querie* desnecessariamente.

Depois de corrigido os problemas com a quantidade, é então invocada a operação de compra.

Seguidamente na Imagem 88 existe a área pessoal do utilizador.

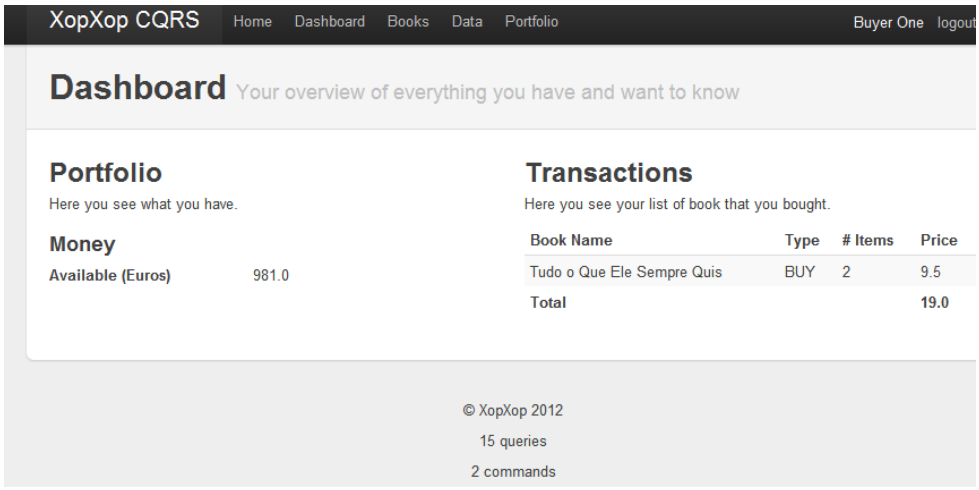

Imagem 88 – Área pessoal do utilizador

Na área pessoal do utilizador, é possível ver (na Imagem 88) que ficamos com menos saldo que o inicial e quais os livros que já comprámos. Neste caso foram compradas duas unidades, de valor unitário de 9.5 euros e que no total foram gastos 19 euros. Além disso no contador, já podemos verificar que no total foram executadas 15 *queries* para obtenção de todos os dados e 2 comandos para a realização da compra.

6.5 Conclusão

Para se conseguir exemplificar a implementação do padrão CQRS, foi necessário elaborar o exemplo deste capítulo 6, de forma a conseguir dar algumas diretivas e formas práticas de interligação dos componentes em CQRS.

Com isto foi conseguido responder a todos os requisitos funcionais que eram pretendidos, destacando-se os mais importantes como: segurança através do login, área pessoal do utilizador, aceder a um catálogo de livros, comprar um livro, visualizar o detalhe de um livro e fazer o checkout desse livro. Em termos práticos foi possível implementar comandos, *queries* e eventos, de forma a existir uma perceção do local de onde os devemos criar e, mais importante, saber exatamente o que cada um deve conter e fazer. Isto, torna-se assim uma parte fundamental desta tese, visto que, permite esclarecer de forma prática a realização desde o início, da criação de uma aplicação passível de corresponder ao padrão estudado.

Para além dos requisitos funcionais, foi possível executar os requisitos não funcionais: todas as funcionalidades quer de utilizador quer de administração e objetos foram criados de forma a respeitarem o padrão; para cada funcionalidade foram implementados eventos, incluindo o contador de *queries* e comandos; e por fim foi utilizado o MongoDB sendo este um motor de base de dados *NoSQL*.

Em termos de interface foi possível executar algumas validações que deveriam ocorrer em CQRS, como a validação do saldo do cliente aquando de uma compra, evitando assim o envio desnecessário de um comando para o sistema quando já se sabia à partida que iria ser rejeitado. Além disso e de considerável importância para esta tese foi a demonstração da quantidade de comandos e *queries* executadas no sistema, de forma a dar logo a perceção ao utilizador do que acontece a cada ação que ele tem no sistema.

O desenvolvimento desta prova de conceito pretende também que seja possível de continuar no futuro, conseguindo adicionar novas funcionalidades e conseguir demonstrar alguns pontos de sistemas de larga escala que não foram descritos, como a interligação com componentes de outras tecnologias. Pretende-se também que possa servir de inicialização para os demais interessados em construir um sistema deste tipo e que forneça todos os conceitos e cuidados a ter aquando na sua elaboração.

No próximo capítulo serão detalhadas todas as conclusões, limitações e trabalhos futuros desta tese.

7 Conclusões

7.1 Objetivos cumpridos

Com o trabalho efetuado nesta dissertação foi possível demonstrar em que se baseia o padrão CQRS e que problemas é que são possíveis de colmatar nas empresas atuais e nas que serão criadas, respondendo assim a todos os objetivos propostos.

Os dois objetivos iniciais baseiam-se no ***estudo de condições empresariais de larga escala*** e ***estudo das tendências que se avizinham*** sendo que para isso foi realizada uma análise de como é que serão as empresas do futuro para determinar quais serão os seus objetivos, se pretendem crescer em larga escala e quais os problemas que irão ter com esse facto. Assim pretende-se conseguir responder na melhor forma a um problema que já começa a existir atualmente: um único modelo para leitura e escrita.

Em termos de objetivos que as empresas pretendem: como a replicação de dados, balanceamento de carga e distribuição geográfica, serão conseguidos de modo a manter-se o máximo de tempo disponível sem quaisquer quebras nos seus serviços e assim, ganharem confiança tanto no seu próprio sistema como nos clientes. Com isto nascem novas preocupações para estas empresas que passam pelo facto de poderem aumentar os custos, tanto a nível de construção como a nível de manutenção, coerência imediata de todos os dados e toda a heterogeneidade dos sistemas de larga escala, uma vez que se pretende resolver ou atenuar estes problemas/limitações com o estudo detalhado do padrão CQRS.

Conseguiu-se concluir também que em 2020 os sistemas terão a tendência para estarem em ambiente *cloud* o que irá potenciar a orientação a determinados serviços como *infrastructure-as-a-service*, *platform-as-a-service*, e *software-as-a-service*. Além disso, ***nem todas as empresas necessitarão da utilização deste padrão*** e por isso, supõe-se que irão existir quatro tipos de empresas: *Cognizant Enterprise*, baseadas na obtenção de conhecimento da área em que trabalham, necessitando que os padrões tenham algum tipo de gestão sobre grande volume de dados; *Community-oriented Enterprise*, que serão empresas que se irão preocupar em fornecer valores às pessoas, tendo em conta a legislação e ética utilizada no seu dia-a-dia, e para isso a implementação de padrões nestas empresas tem que ter em linha de conta

algum tipo de mecanismo de inteligência artificial que possa fornecer dados quanto às operações efetuadas no sistema; as *Green Enterprise*, tal como o nome indica terão muito interesse na poupança de recursos e com isso o estudo do padrão terá de ter a possibilidade de ser dinâmico quanto à alocação de recursos; por fim as *Glocal Enterprise* serão empresas que trabalharão a nível local mas também a nível global de forma a conseguirem chegar a uma grande parte do seu público-alvo. Cada vez mais este tipo de empresas necessita de ter um sistema que aplique determinados padrões que possam ser distribuídos e aplicados em *cloud computing*. Exemplo: uma sede da empresa que espalha pelo mundo pequenas agências que trabalham todas em conjunto para o sucesso.

Com este estudo do estado da arte sobre as condições que as empresas apresentam atualmente e quais são as suas pretensões para o futuro podemos concluir que é necessário aplicar determinados padrões que possivelmente não irão dar resposta a todos os problemas nem serão uma "*silver-bullet*" mas que o facto de ajudarem a diminuir os problemas ou fornecerem boas práticas para que estes sejam diminuídos desde início, torna o padrão CQRS numa mais-valia estudada.

De seguida é detalhado o ***estudo do padrão CQRS para utilização em larga escala e que alterações representam nas arquiteturas a implementação do CQRS.*** O CQRS trata-se de um padrão com o objetivo de refinar as arquiteturas. No CQRS pretende-se separar a arquitetura baseada na natureza das suas operações, ou seja, em dois modelos, um de leitura e outro de escrita, dando ênfase ao facto de poderem estar separados totalmente. Desta forma onde anteriormente existia apenas um objeto agora passam a existir dois, com objetivos bem definidos e otimizados, em que o modelo de leitura é denominado de ***querie*** e o de manipulação de dados de ***command (comando)***.

Para além disso, a divisão de responsabilidades é muito importante neste padrão pois dessa maneira poderá ter-se duas bases de dados otimizadas para cada modelo, uma para leitura e outra para escrita ficando cada uma com os seus objetivos bem definidos e conseguindo colmatar separadamente o problema de cada uma. Em termos práticos, a base de dados utilizada pelas *queries* pode estar na 1FN de forma a ser de acesso rápido sem necessidade de relações entre tabelas, enquanto a de escrita utilizada pelos comandos pode estar na 3FN, pois necessita de uma estruturação mais cuidada. Geralmente a base de dados dos comandos é considerada a base de dados "principal" visto ser a utilizada para descrever em detalhe todas as entidades, transações, etc. e as suas relações no sistema. Sem esta separação não seria possível apenas com um modelo otimizar uma parte do sistema, visto que provavelmente se iria deteriorar o outro.

Outra forma de otimização existente é a de criar uma espécie de cache para as *queries* mais frequentes para que a repetição de pedidos para os mesmos dados se tornem mais rápidas, enquanto para o processamento de comandos é a criação de uma *queue* que é consumida ao longo do tempo, podendo assim balancear mediante a carga do sistema e garantindo que nenhum pedido é perdido. De salientar que o modelo de *queries* tem a necessidade de suportar muitos utilizadores ao mesmo tempo, porque além de poderem existir dados

dispersos geograficamente ainda poderão ser necessárias várias *queries* para construir uma página.

Esta abordagem não se trata de ser apenas uma boa prática e que deve ser utilizada cegamente em todos os sistemas, mas pelo contrário, ajudar quem está a desenhar sistemas, para permitir ver e pensar de outra forma, conseguindo ter sempre mais código sustentável, código de alto desempenho, simplicidade e escalabilidade. O facto de poder ser distribuída em *cloud*, torna o seu valor comercial muito mais elevado.

Para além deste padrão, torna-se ainda necessário nesta tese responder ao objetivo ***de que forma é que os componentes comunicam entre si***, concluindo que a melhor opção é utilizar sistemas de *messaging*, que se baseiam em trocas de mensagens e não em chamadas RPC. Para o padrão estudado, o *messaging* é o ideal para ser incorporado e aplicado à grande maioria dos sistemas, permitindo aumentar a tolerância a falhas, chamadas assíncronas, e o escalonamento, tornando-o mais simples. Este escalonamento é atingido através da distribuição por vários nós do modelo de leitura, da criação de uma ou várias *queues* para processamento de comandos e criando vários *workers* para o processamento dos comandos. A utilização de *messaging* ainda nos permite desacoplar o sistema visto que não existe dependência tecnológica para a troca de mensagens, o que leva a que as mensagens possam ser enviadas por entidades diferentes, consumidas de diferentes formas e sem complexidade de mudança de um local para outro. Por estes motivos ainda nos é possível concluir relativamente ao objetivo da ***facilidade de manutenção***, que se por exemplo, for necessário atualizar um determinado *software* num componente do sistema, podemos ter a garantia que ao ser desligado, as mensagens não se vão perder, ficando sempre registadas. Aquando do arranque desse mesmo nodo com o novo *software*, é possível começar novamente a consumir as mensagens permitindo assim que o sistema possa ser mantido em pequenas partes sem comprometer o sistema como um todo.

Para além destes conceitos, ainda nos é possibilitado o uso de eventos na nossa arquitetura, e para quem desenvolve ou dá suporte a um produto, isso será visto como uma mais-valia. Os eventos são baseados em mensagens que podem ser enviadas assincronamente, sendo processadas mais tarde, repartidas, ou até repetidas no sistema no caso de erros. De salientar novamente que o padrão CQRS *não* obriga à aplicação de eventos no sistema, no entanto é uma forma de registo de dados e manutenção que se interliga muito bem com este padrão. Com os eventos existe também a possibilidade de realizar auditorias mais simplificadas aos dados, de forma a garantir que todo o sistema está a funcionar como seria esperado e em casos mais graves, existe a possibilidade de realizar recuperações. As recuperações baseiam-se no facto de os eventos ao serem todos registados permitirem reconstruir todo um fluxo de dados que foi introduzido ou alterado no sistema.

Em termos de negócio, que é um dos pontos mais importantes para as empresas, permite ainda aumentar o seu valor com o uso de eventos. As empresas podem assim ter um caminho completo desde a criação da entidade e respetivas mudanças que teve ao longo da vida. Desta forma todos os dados interessantes para o negócio são mantidos por ordem e com todos os

detalhes sem grande esforço de implementação ao passo que as arquiteturas CRUD sofrem deste problema ou têm grandes custos. Além disso, isto permite cumprir o objetivo de ***facilitar a geração de reports com informação de negócio ou auditoria de dados***, pois a informação já aparece organizada em termos de negócio e é apenas preciso filtrar os dados relevantes. Estas auditorias podem ser baseadas em *logging*, que é responsável por guardar cada alteração realizada no sistema e assim restringir em termos temporais que ações é que foram feitas sobre o sistema. Outra das vantagens é permitir que a qualquer altura se possa escolher uma data e ter uma imagem de todo o sistema naquela precisa data, de forma a conseguir executar *queries* sobre o sistema e saber como e de que forma é que ele se encontrava naquele preciso momento. Para as empresas, os relatórios fazem parte do dia-a-dia, sendo que é aliado muitas vezes ao *data mining*. Este padrão, permite que essa tarefa fique mais facilitada devido a ser tudo registado com todos os passos, desde onde partiu até onde chegou, provocando uma maior confiança no sistema, quer internamente quer por quem o usa.

Outro objetivo respondido é o facto de ***ser necessário ter sempre a informação atualizada no cliente,*** sendo que nestes grandes sistemas distribuídos existem também problemas com a sincronia de dados: num sítio poderão ter sido atualizados com a última versão, mas devido à dimensão do sistema, existe uma parte desse mesmo sistema, que ainda possui dados desatualizados. Com o estudo deste padrão nesta tese, conseguimos concluir que não precisamos de ter todos os dados atualizados ao milissegundo, alguns deles poderão sem qualquer prejuízo, ser atualizados "mais tarde". Além disso obtém-se ainda a resposta ao objetivo de que os ***mecanismos utilizados para atualizar os dados posteriormente*** são realizados devido à progressiva atualização com os novos dados, das bases de dados para leitura as caches das próprias *queries* vão sendo atualizadas e a validade dos dados é ultrapassada. Desta forma é possível balancear os dados mais importantes, dando por exemplo, prazos de validade até que estes sejam dispersos por todos o sistema e deixar os menos importantes para serem processados mais tarde para que em empresas que usem estes sistemas em tomadas de decisão, não seja algo crítico e uma desvantagem relativamente aos antigos métodos CRUD.

Em termos de coerência dos dados, também existe a preocupação de dois clientes escreverem ao mesmo tempo os mesmos dados, e de que forma é que este padrão implementa este controlo respondendo assim ao objetivo ***como é possível garantir a coerência dos dados***. Esta coerência é obtida através do próprio controlo que os comandos fazem, ou seja, cada comando quando tenta atualizar determinados dados, estes dados tem a sua versão. Desta forma um comando escreve com um número de versão e o seguinte com outra, ficando assim salvaguardado que se por acaso os dados não forem os corretos são facilmente recuperáveis porque ficam os dois registados, mas com versões diferentes. Se a este controlo se aliar os eventos, torna-se ainda mais fácil a sua recuperação porque basta reprocessar esse evento com a versão seguinte e os dados tornam a ficar coerentes novamente. Existem mesmo assim, situações que levam a que isto não seja possível. Uma dessas situações pode ser quando uma versão mais atualizada tem conflitos com a anterior, se executadas "ao mesmo tempo".

Nesses casos a própria execução do comando no componente de validação dará o devido erro, exigindo que o emissor desse comando o volte a processar mediante os novos dados.

Posteriormente ao estudo deste padrão nesta tese, é importante realçar a existência de algumas entidades, que o aplicam como um todo, ou apenas uma parte, e têm sucesso na sua implementação, respondendo assim a dois objetivos inicialmente propostos: ***demonstrar que existem casos de sucesso no mercado*** e ***o estudo de uma nova arquitetura que possa ser aplicada a um sistema de larga escala ou a uma parte dele***. Os três casos estudados realçam o facto do padrão se conseguir dividir em vários modelos o que torna a sua implementação mais simples e objetiva. Em relação à sua expansão e manutenção tornou-se um trabalho mais facilitado. Além disso, o facto de haver casos em que é aplicado a uma parte do sistema e não a um todo, eleva ainda mais o seu valor visto que quando se pretender aplicar, poderá não se desejar a todo o sistema de uma vez mas apenas a uma parte, servindo muitas vezes de experiência para as empresas. Um dos casos estudados revelou que o padrão funciona essencialmente bem quando é utilizado desde início, o que permitiu que os problemas à medida que foram aparecendo, fossem colmatados com a aplicação deste padrão. Permitiu também modificar os objetos de domínio que estavam estruturados de uma forma, mas no final e devido a esta aplicação, foram melhorados. Os outros dois casos são aplicados a uma parte do sistema, visto ser um serviço que é disponibilizado, provando assim que é possível suportar muitos utilizadores e crescer com eles. O caso da aplicação deste padrão a uma base de dados permitiu concluir que mesmo com poucos recursos é possível desenvolver e manter um sistema deste tipo. Este sistema de base de dados é utilizado mundialmente e é um dos que é mais falado por ser facilmente escalável. Desta forma, é proporcionado um refinamento nas arquiteturas com custos bastante baixos comparativamente às mudanças efetuadas, mostrando assim com estes casos o objetivo do ***sucesso da aplicação deste padrão***.

Para a aplicação deste padrão na prova de conceito foram estudadas três *framework*s, e por isso pretende-se dar resposta aos objetivos de ***estudo de frameworks CQRS existentes e que ferramentas nos podem ajudar a implementar este padrão.*** A razão da escolha das três *framework*s foi o facto de serem as mais utilizadas em termos de mercado e serem as mais conhecidas na comunidade de implementação de CQRS. Além disso, o estudo teve em conta a diversidade tecnológica em que duas delas usam .NET, a NServiceBus e NCQRS Framework e uma JAVA, a AXON Framework. De salientar que a NServiceBus é limitada para desenvolvimento em que só é permitido duas instâncias do componente cliente, necessitando de uma licença comercial para o uso com mais clientes. Para cada uma delas foi estudada a sua arquitetura e a forma como é que implementavam o padrão, incluindo código exemplo e formas de comunicação. No final foi escolhida a AXON para a implementação, devido a ser *open source*, existirem bastantes *templates* e a uma fonte de documentação muito completa. Ao ser a AXON a mais usada nesta tese permitiu também verificar que para o início de uma arquitetura baseada em CQRS, esta *framework* torna o trabalho mais simplificado visto nos seus projetos base possuir alguns *templates* para comandos, *queries* e eventos, proporcionando assim um início mais simples, mais objetivo e mais delimitado conseguindo que quem o desenvolve consiga focar-se exatamente no que é importante.

Relativamente ***à influência que este padrão CQRS traz para a interface com o utilizador***, podemos concluir que existe uma nova base de construção de interfaces denominada de *Task Based UI,* que consiste em refinar a interface para que o utilizador não seja induzido a realizar ações do tipo CRUD. A interface deve conseguir determinar quais são as intenções do utilizador e estudar que possíveis ações poderá fazer, evitando assim por exemplo, ações descontextualizadas com o que o utilizador está a visualizar. Isto traz um aumento de valor para o negócio pois permite mais tarde estudar a forma como um utilizador se comporta no sistema, sabendo que ações executa mais, que dados visualiza mais e apresentar ao utilizador dados personalizados.

Outro fator importante é a interface ser um ponto fulcral na validação de comandos, ou seja, a interface antes de enviar a atualização de determinados dados (comando), poderá validar o máximo possível os dados que estão a ser enviados. Desta forma não são introduzidos no sistema comandos que já se sabe à partida que irão ser rejeitados. Consegue-se assim diminuir a latência das comunicações, visto que existem menos dados a circular, e da parte de quem utiliza o sistema, torna-o mais confiável porque garante que quando é enviado existe uma baixa probabilidade de ser rejeitado. Estas validações tal como o padrão, não devem ser aplicadas inconscientemente e em demasia, porque senão podem ter o efeito contrário, tornando os pedidos de validação demasiado complexos e lentos devido a terem de executar várias *queries*.

Além da parte teórica da tese, um dos objetivos inicialmente propostos seria de ***existir uma aplicação exemplo***, de modo a demonstrar a implementação de uma prova de conceito, em que se pretende que seja uma "*test-bed*" que possibilita e promove a investigação e desenvolvimento neste campo do padrão CQRS. Esta prova foi construída com ajuda a uma *framework* (AXON) que forneceu algumas diretrizes de como melhor implementar o padrão CQRS, conseguindo acompanhar com eventos e com todo o sistema de *Messaging.* Desta forma conseguiu-se visualizar como se implementa num cenário real. Esta implementação contém também uma interface para o utilizador para fornecer uma forma mais aproximada da realidade de utilização, em que são fornecidas ações no sistema, como visualização de catálogos de livros, compras de livros, área pessoal, etc. Para além das ações normais numa aplicação web, existe a visualização do número de *queries* e comandos executados no sistema para que desta forma seja possível, neste caso, provar a execução dos mesmos e dar uma perceção ao utilizador do que está a acontecer em background.

O estudo deste padrão juntamente com a aplicação exemplo, leva a concluir que este padrão, permite desde o início ajudar a construir um sistema de larga escala para além de também fornecer mecanismos para que seja possível migrar um já existente, o que o torna mais atraente e faz com que existam mais interessados em aplicar este padrão.

Por fim conclui-se que não é possível criar uma solução ótima para pesquisa, *report*ing, manipulação de dados e processamento de transações apenas com a utilização de um único modelo e é nisso que o CQRS nos consegue ajudar.

7.2 Limitações

Nesta tese existem algumas limitações, como o estudo detalhado da tolerância a falhas no padrão de CQRS e como este deveria ser implementado. Alguns dos pontos falados, como a facilidade de replicação, divisão de dados pelo sistema, implementação de eventos assíncronos, etc., já pretendem colmatar alguns tipos de falhas, no entanto deveria ser um ponto mais estudado e elaborado no futuro.

Um complemento interessante que poderia ser alvo de um estudo mais aprofundado seria o *Task Based UI*, que para além do que foi falado no capítulo 3.2.3 Comandos, deveria incidir mais sobre o conceito em CQRS mostrando também exemplos da sua aplicação, visto que mais tarde poderia ser implementada na aplicação exemplo.

Em relação à aplicação exemplo, não foi possível demonstrar a totalidade das funcionalidades pretendidas, como o facto de não ter sido testado num ambiente de larga escala de forma a ser possível comprovar que a aplicação teria a facilidade em crescer sem problemas. Esta aplicação apenas se encontra a utilizar um tipo de tecnologia, o JAVA, não sendo possível demonstrar a sua fácil interoperabilidade com outras, sendo este um dos fatores que o CQRS também favorece.

Em relação ás *framework*s estudadas, as que se baseavam em .NET não foram estudadas aprofundadamente visto que uma (NServiceBus) necessita de licença para mais de dois clientes e a outra (NCQRS) não possui boa documentação, o que limitou de alguma forma a sua escolha para a implementação da prova de conceito. No entanto a nível empresarial talvez fosse necessário um estudo melhorado sobre a NServiceBus visto ser utilizada por grandes empresas e realizar atualizações ao *software* regularmente.

7.3 Trabalho futuro

Em relação a trabalho futuro existem alguns pontos que poderiam ser mais desenvolvidos na aplicação exemplo.

Um ponto interessante seria testes de performance, ou seja, dividir em várias máquinas todo o sistema, incluindo base de dados, e verificar que tempos de resposta se obtêm de forma a poder comparar com outros sistemas CRUD do mercado.

Além disso não foi elaborado nenhum relatório sobre a interação e integração com outras tecnologias para que se pudesse provar a interoperabilidade entre tecnologias, sendo utilizado na sua maioria o Java como principal tecnologia (Amazon, 2012c).

Depois de estudados os sistemas de base de dados *NoSQL*, seria aliciante conseguir provar que a prova de conceito funcionaria sem qualquer problema com várias base de dados,

divididas pelo menos entre várias máquinas, de forma a provar que as alterações para o fazer não existiriam ou seriam mínimas.

8 Bibliografia

10gen, 2012a. *Replica Sets.* [Online]
Available at: http://www.mongodb.org/display/DOCS/Replica+Sets
[Acedido em 19 07 2012].

10gen, 2012b. *Replication.* [Online]
Available at: http://www.mongodb.org/display/DOCS/Replication
[Acedido em 19 07 2012].

10gen, 2012c. *Sharding.* [Online]
Available at: http://www.mongodb.org/display/DOCS/Sharding
[Acedido em 19 07 2012].

10gen, 2012d. *SQL to Mongo Mapping Chart.* [Online]
Available at: http://www.mongodb.org/display/DOCS/SQL+to+Mongo+Mapping+Chart
[Acedido em 19 07 2012].

10gen, 2012e. *The MongoDB Manual.* [Online]
Available at: http://docs.mongodb.org/manual/
[Acedido em 19 07 2012].

10gen, 2012f. *Tutorial - Running MongoDB.* [Online]
Available at: http://www.mongodb.org/display/DOCS/Tutorial#Tutorial-RunningMongoDB
[Acedido em 18 07 2012].

Abdullin, R., 2010a. [Online]
Available at: http://abdullin.com/journal/2010/9/23/command-handlers-without-2pc-and-

with-various-levels-of-reli.html
[Acedido em 16 06 2012].

Abdullin, R., 2010b. [Online]
Available at: http://abdullin.com/journal/2010/4/8/couchdb-document-persistence-powered-by-cqrs.html
[Acedido em 18 06 2012].

Abdullin, R., 2010c. [Online]
Available at: http://abdullin.com/journal/2010/4/14/cqrs-validation-and-business-rules.html
[Acedido em 28 05 2012].

Abdullin, R., 2010d. [Online]
Available at: http://abdullin.com/journal/2010/11/3/cqrs-architecture-and-definitions.html
[Acedido em 14 05 2012].

Abdullin, R., 2010e. [Online]
Available at: http://abdullin.com/journal/2010/3/23/dddd-cqrs-and-other-enterprise-development-buzz-words.html
[Acedido em 17 06 2012].

Abdullin, R., 2010f. [Online]
Available at: http://abdullin.com/journal/2010/12/5/messaging-server-for-cloud-cqrs.html
[Acedido em 10 06 2012].

Abdullin, R., 2010g. [Online]
Available at: http://abdullin.com/journal/2010/9/26/theory-of-cqrs-command-handlers-sagas-ars-and-event-subscrip.html
[Acedido em 14 06 2012].

Abdullin, R., 2010h. [Online]
Available at: http://abdullin.com/journal/2010/11/16/key-cqrs-ingredient.html
[Acedido em 27 05 2012].

Abdullin, R., 2011a. [Online]
Available at: http://abdullin.com/wiki/cloud-computing.html
[Acedido em 01 06 2012].

Abdullin, R., 2011b. [Online]
Available at: http://abdullin.com/wiki/infinitely-scalable-system.html
[Acedido em 30 05 2012].

Abdullin, R., 2011c. [Online]
Available at: http://abdullin.com/journal/2011/6/7/messages-and-distributed-systems.html
[Acedido em 10 06 2012].

Abdullin, R., 2011d. *Rinat Abdullin on efficient software development, cloud computing and CQRS.* [Online]
Available at: http://abdullin.com/journal/2011/1/19/scalable-and-simple-cqrs-views-in-the-cloud.html
[Acedido em 03 06 2012].

Abdullin, R., 2012. *DDD/CQRS Challenge - Integrating Distributed Systems.* [Online]
Available at: http://abdullin.com/journal/2012/5/22/dddcqrs-challenge-integrating-distributed-systems.html
[Acedido em 18 07 2012].

Abdullin, R., s.d. [Online]
Available at: http://abdullin.com/cqrs/
[Acedido em 17 06 2012].

Amazon, 2012a. [Online]
Available at: http://aws.amazon.com/pt/elasticmapreduce/
[Acedido em 01 06 2012].

Amazon, 2012b. [Online]
Available at: http://aws.amazon.com/pt/s3//177-9200638-6959247/
[Acedido em 2 1 2012].

Amazon, 2012c. [Online]
Available at: http://aws.amazon.com
[Acedido em 01 06 2012].

Amazon, 2012d. [Online]
Available at: http://aws.amazon.com/pt/ec2/
[Acedido em 01 06 2012].

Amazon, 2012e. [Online]
Available at: http://aws.amazon.com/pt/sqs/
[Acedido em 05 07 2012].

AntiPhish, 2009. [Online]
Available at: http://www.antiphishresearch.org/home.html
[Acedido em 19 3 2012].

Ashton, R., 2011. [Online]
Available at: HTTP://CODEOFROB.COM/ARCHIVE/2011/09/28/CQRS-IS-TOO-COMPLICATED.ASPX
[Acedido em 7 1 2012].

BenoitC, 2012. [Online]
Available at: http://wiki.apache.org/couchdb/CouchDB_in_the_wild
[Acedido em 18 06 2012].

Bridge, 2012. [Online]
Available at: HTTP://WWW.BRIDGE-PROJECT.EU/
[Acedido em 19 3 2012].

Buijze, A., 2012. [Online]
Available at: http://blog.orange11.nl/2010/11/12/tutorial-getting-started-with-cqrs-and-axon-framework/
[Acedido em 27 05 2012].

Buijze, A. & Coenradie, J., 2012a. *Reference Guide Axon Framework 1.3.3.* [Online]
Available at: http://www.axonframework.org/docs/1.3/introduction.html
[Acedido em 8 5 2012].

Buijze, A. & Coenradie, J., 2012b. *Reference Guide.* [Online]
Available at: http://www.axonframework.org/docs/1.3/architecture-overview.html
[Acedido em 8 5 2012].

Carnegie Mellon University, 2006. *Ultra-Large-Scale Systems - The Software Challenge of the Future.* Pittsburgh: s.n.

Charlton, J., 2009. *Aggregates and Aggregate Roots.* [Online]
Available at: http://thinkddd.com/blog/2009/02/14/aggregates-and-aggregate-roots
[Acedido em 08 10 2012].

Corporation, Microsoft, 2011. [Online]
Available at: http://msdn.microsoft.com/en-us/library/ms997506.aspx
[Acedido em 4 12 2011].

CUTELOOP, 2008. [Online]
Available at: HTTP://WWW.CUTELOOP.EU/
[Acedido em 19 3 2012].

Dahan, U., 2009. [Online]
Available at: http://www.udidahan.com/2009/12/09/clarified-cqrs/
[Acedido em 20 10 2011].

Dahan, U., 2011. [Online]
Available at: http://www.udidahan.com/2011/10/02/why-you-should-be-using-cqrs-almost-everywhere%E2%80%A6/
[Acedido em 27 05 2012].

Dahan, U., 2012a. [Online]
Available at: http://www.nservicebus.com/ArchitecturalPrinciples.aspx
[Acedido em 1 4 2012].

DAHAN, U., 2012b. [Online]
Available at: http://www.nservicebus.com/Overview.aspx
[Acedido em 31 3 2012].

Dahan, U., 2012c. *Using NServiceBus in a Web Application.* [Online]
Available at: http://nservicebus.com/docs/Samples/AsyncPages.aspx
[Acedido em 18 09 2012].

DAHN, U., 2008. [Online]
Available at: HTTP://WWW.UDIDAHAN.COM/2008/08/11/COMMAND-QUERY-SEPARATION-AND-SOA/
[Acedido em 10 10 2011].

Eeles, P., 2008. [Online]
Available at:
http://www.websphereusergroup.org.uk/wug/files/presentations/25/25_19_ArchitectLargeSystems.pdf
[Acedido em 7 1 2012].

Ferreira, E., 2010a. [Online]
Available at: HTTP://ESCALABILIDADE.COM/2010/04/06/INTRODUCAO-AO-NOSQL-PARTE-II/
[Acedido em 25 3 2012].

Ferreira, E., 2010b. [Online]
Available at: http://escalabilidade.com/2010/03/08/introducao-ao-nosql-parte-i/
[Acedido em 25 3 2012].

FInES, 2010. Future Internet Enterprise Systems (FInES). In: *Research Roadmap.* European Communities: s.n., pp. 23-50.

Fossmo, P., 2009. [Online]
Available at: http://blog.fossmo.net/post/Command-and-Query-Responsibility-Segregation-(CQRS).aspx
[Acedido em 14 05 2012].

Fowler, M., 2005a. [Online]
Available at: http://martinfowler.com/bliki/CommandQuerySeparation.html
[Acedido em 4 5 2012].

Fowler, M., 2005b. [Online]
Available at: http://martinfowler.com/eaaDev/EventSourcing.html
[Acedido em 4 12 2011].

Fowler, M., 2009. [Online]
Available at: http://martinfowler.com/bliki/EagerReadDerivation.html
[Acedido em 4 12 2011].

Fowler, M., 2011. [Online]
Available at: http://martinfowler.com/bliki/CQRS.html
[Acedido em 3 12 2011].

Fowler, M., Parsons, R. & MacKenzie, J., 2012. *POJO.* [Online]
Available at: http://c2.com/cgi/wiki?PlainOldJavaObject &
http://martinfowler.com/bliki/POJO.html
[Acedido em 03 06 2012].

Google, 2012. [Online]
Available at: https://developers.google.com/appengine/
[Acedido em 01 06 2012].

GS1 System, 2012. [Online]
Available at: http://www.gs1.org/productssolutions
[Acedido em 19 3 2012].

Hat, R., 2012. [Online]
Available at: http://www.hibernate.org/
[Acedido em 03 06 2012].

HUMABIO, 2008. [Online]
Available at: HTTP://WWW.HUMABIO-EU.ORG/INDEX.HTML
[Acedido em 19 3 2012].

ISURF, 2011. [Online]
Available at:
http://www.srdc.com.tr/projects/isurf/index.php?option=com_content&view=frontpage&Itemid=1
[Acedido em 19 3 2012].

Janssens, T., 2011. [Online]
Available at: http://www.codeproject.com/Articles/291411/Continuous-thinking-CQRS-explained-to-a-10-year-ol
[Acedido em 4 1 2012].

Lennon, J., 2009. [Online]
Available at: http://www.ibm.com/developerworks/opensource/library/os-couchdb/index.html
[Acedido em 20 06 2012].

Maier, M. W., 2009. [Online]
Available at: http://www.infoed.com/Open/PAPERS/systems.htm
[Acedido em 4 3 2012].

Microsoft, 2012. [Online]
Available at: http://www.windowsazure.com/pt-br/
[Acedido em 2 1 2012].

Musil, S., 2011. [Online]
Available at: http://news.cnet.com/8301-1023_3-20037554-93.html
[Acedido em 2 1 2012].

Ncqrs, 2012. [Online]
Available at: http://ncqrs.org/reference/
[Acedido em 11 4 2012].

Newman, B. C., 1994. Scale in Distributed Systems. *Computing Systems. IEEE Computer Society Press*, pp. 1-22.

Nijhof, M., 2009. [Online]
Available at: http://elegantcode.com/2009/11/11/cqrs-la-greg-young/
[Acedido em 10 06 2012].

NOSQL, 2012. [Online]
Available at: HTTP://NOSQL-DATABASE.ORG/
[Acedido em 25 3 2012].

Oliver, J., 2011a. [Online]
Available at: http://blog.jonathanoliver.com/2011/03/cqrs-event-sourcing-and-immutable-data/
[Acedido em 2 1 2012].

Oliver, J., 2011b. *CQRS, Event Sourcing, Messaging.* [Online]
Available at: http://blog.jonathanoliver.com/2011/05/why-i-still-love-cqrs-and-messaging-and-event-sourcing/
[Acedido em 03 06 2012].

OpenTC, 2008. [Online]
Available at: HTTP://WWW.OPENTC.NET/
[Acedido em 19 3 2012].

PRIME, 2008. [Online]
Available at: HTTPS://WWW.PRIME-PROJECT.EU/
[Acedido em 19 3 2012].

Rackspace, 2012. [Online]
Available at: http://www.rackspace.com/cloud/
[Acedido em 01 06 2012].

S3MS, 2012. [Online]
Available at: HTTP://WWW.OMNYS.COM/ENGLISH/RESEARCH-INNOVATION/S3MS &

HTTP://WWW.S3MS.ORG/
[Acedido em 19 3 2012].

SECOQC, 2008. [Online]
Available at: HTTP://WWW.SECOQC.NET/
[Acedido em 19 3 2012].

Teague, J., 2012. *A year in review with CQRS.* [Online]
Available at: http://lostechies.com/johnteague/2012/02/05/a-year-in-review-with-cqrs/
[Acedido em 18 07 2012].

Tom, 2010. [Online]
Available at: http://www.corebvba.be/blog/post/My-CQRS-Cookbook.aspx
[Acedido em 5 1 2011].

Young, G., 2010. [Online]
Available at: http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/
[Acedido em 10 06 2012].

Young, G., 2011. [Online]
Available at: HTTP://CQRSINFO.COM/DOCUMENTS/CQRS-INTRODUCTION/
[Acedido em 13 9 2011].

Anexos

1. MongoDB na AXON

1.1. Introdução

Em relação a outras bases de dados tradicionais, o *MongoDB* contém na mesma coleção, índices, tendo a grande diferença de guardar tipos de documentos em *BSON* em vez dos tradicionais registos. Nestes documentos existem campos que podem ter vários tipos de dados e cada documento do mesmo tipo pode ter diferentes campos, daí ser caracterizada por ser uma base de dados livre de esquema (*Schema free*). Na maioria dos casos, o que acontece é que um tipo de coleção tem o mesmo tipo e número de campos, e acaba assim por se tornar homogéneo, no entanto não é nenhum requisito. Com isto é permitido um desenvolvimento muito mais dinâmico sem a necessidade de existirem "*alter tables*" para cada alteração que seja precisa na base de dados e facilita as migrações, deixando a base de dados de ser um problema (10gen, 2012f).

1.2. Mongo vs SQL syntax

Com as mudanças para um motor de base de dados *NoSQL* é necessário saber quais são as diferenças em termos de sintaxe e de conceitos que foram adicionados por ser baseado em documentos, e outros que foram retirados por deixarem de fazer sentido. Alguns exemplos de sintaxe e as suas diferenças estão esquematizadas na Tabela 2 abaixo (10gen, 2012d):

Tabela 2 – Comparação de sintaxe de SQL vs. Mongo (10gen, 2012d)

SQL	**Mongo**
table	collection
row	BSON document
column	BSON field
join	embedding and linking
primary key	_id field
group by	aggregation
CREATE TABLE USERS (a Number, b Number)	implicit; can also be done explicitly with db.createCollection("mycoll")
SELECT a,b FROM users	db.users.find({}, {a:1,b:1})
SELECT * FROM users	db.users.find()
SELECT * FROM users WHERE age=33	db.users.find({age:33})
INSERT INTO USERS VALUES(3,5)	db.users.insert({a:3,b:5})

1.3. Replication e Sharding

Em termos de replicação a mongo suporta replicação assíncrona entre servidores, para casos de catástrofe e existência de redundância. A forma é através de um servidor master que escreve num determinado momento e assim permite ter operações atómicas. Neste caso, e para aplicação do CQRS é importante garantir que possam existir na mesma leitura, mesmo com eventual consistência de dados nos servidores secundários, o que, é garantido pela MongoDB. Existem duas formas de replicação, uma denominada *Replica Sets* e outra *Master-Slave*, em que a *master-slave* é o típico *master* que replica tudo para os *slaves*, enquanto a réplica *sets* já é caracterizada por ser um código muito mais estável e funcional e permite ter vários membros com papéis dinâmicos, ou seja, não existe especificamente um elemento primário, podendo mudar a qualquer altura (10gen, 2012a). Exemplo:

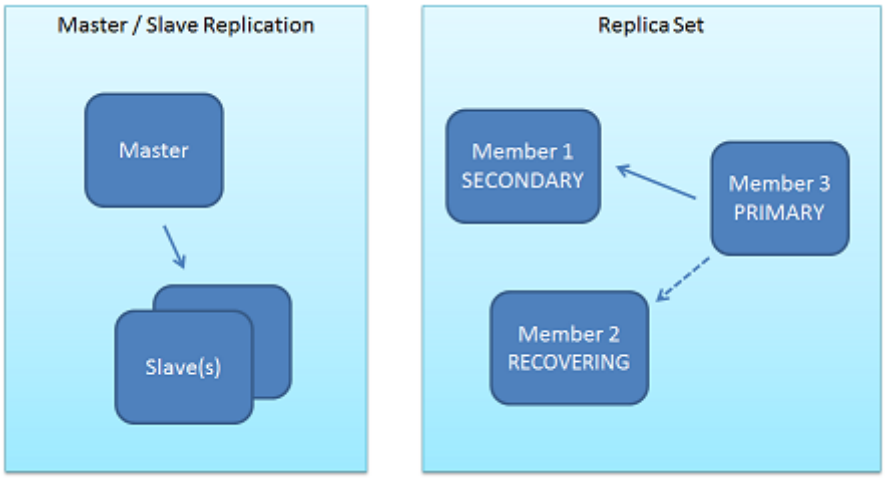

Imagem 89 – MongoDB e replicação (10gen, 2012b)

Em relação ao *sharding*, permite escalar horizontalmente através de particionamento. No *MongoDB* é denominado de *auto-sharding*, providenciando assim balanceamento automático em termos de carga e distribuição de carga, adição de novas máquinas sem necessidade de desligar o sistema, escalar para milhares de nodos, não ter um ponto de falha e por fim tolerância a falhas (10gen, 2012c).

1.4. Integração projecto Java/Eclipse

Na Imagem 90 é possível ver o projeto de ligação entre o JAVA e MongoDB.

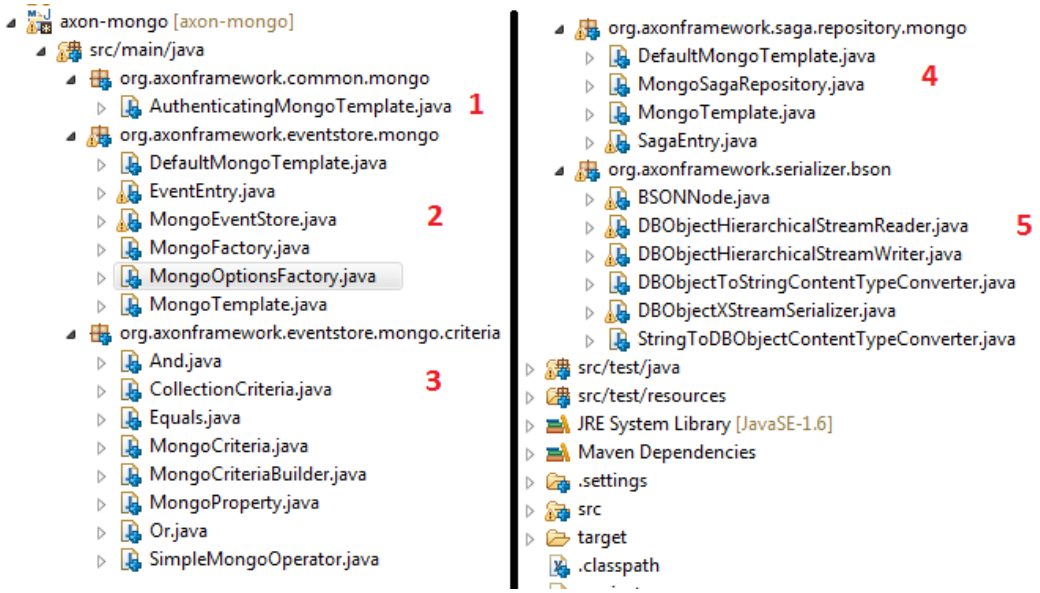

Imagem 90 – Projeto Mongo DB em Java

Em relação ao projeto (java) que interliga a *framework* com o sistema de base de dados, pode ser visto pela Imagem 90 que foi dividido em 5 áreas distintas: a **primeira** permite ter as definições quanto à autenticação no sistema mongo, a **segunda** tem as opções de ligação à base de dados, e também o tipo de fábricas disponíveis, etc., a **terceira** tem os critérios disponíveis para a construção de *queries* no sistema, em que é permitido passar por parâmetros, por exemplo 2 argumentos na classe "*Or*". Em relação à **quarta** secção tem definições das sagas que podem ser aplicadas em CQRS, a qual não é aplicada no nosso caso, e por fim temos a **quinta** secção que permite construir e ler todos os documentos da base de dados, como por exemplo, os respetivos atributos, a sua serialização em *BSON*, etc. Este projeto é usado na nossa *framework* para que possamos ter ligação com o motor de base de dados *MongoDB* com todas as opções disponíveis.