IPV - ESTGV |

Instituto Politécnico de Viseu

Escola Superior de Tecnologia e Gestão de Viseu

Instituto Politécnico de Viseu

Escola Superior de Tecnologia e Gestão de Viseu

Dedicatória

Dedico esta dissertação à minha mãe, Maria Conceição Alves Costa e ao Silvério Varandas, por terem insistido comigo para me inscrever neste curso de Mestrado. Dedico ao meu pai, José Manuel Rodrigues Ribeiro, por me ter ajudado muito desde sempre, mas especialmente durante a minha vida académica e por último à minha tia Maria Teresa Ribeiro, que é a pessoa que eu mais amo no mundo e, sempre foi para mim, mais que uma mãe. Por último, mas não menos importante, à minha namorada, Filipa, a mulher da minha vida e o meu grande amor.

Agradecimentos

Quero agradecer em primeiro lugar à minha mãe e ao Silvério, por terem insistido durante várias semanas para eu me inscrever no mestrado, insistindo que seria uma mais valia quer em termos pessoais, quer em termos profissionais, agora passado este tempo todo reconheço que vocês estavam certos. Á minha tia Teresa, pelo me ter criado e ter sido pai e mãe e ter estado para mim nos bons e maus momentos, foste, és e serás a pessoa mais importante para mim. Ao meu pai, por me ter ajudado em tantas situações e em especial durante o tempo que estudei em Viseu. E á minha namorada, Filipa Trigo, que é a mulher da minha vida e com quem partilho os meus dias.

Queria também deixar uma palavra de agradecimento e gratidão ao meu orientador, o Prof. Doutor Carlos Quental, por ter sido incansável, sempre disponível e por todas as suas ideias e sugestões que me foi dando ao longo deste tempo, mais uma vez, obrigado professor.

Queria deixar também uma palavra de agradecimento aos meus colegas da Estamos Juntos, em especial à Carina Penedo por ter sido uma grande amiga e "irmã mais velha" que me ajudou a integrar na minha nova etapa em Lisboa, ao Marcos André por me ter integrado no projeto e me ter ensinado uns truques de programação, ao Carlos Sampaio, Gonçalo Rijo Hugo Cristina e Rui Pinheiro pela vossa liderança e espírito empreendedor nesta empresa, Ao Sérgio Machado, que recentemente entrou na equipa do projeto, bem como a todos os outros colegas.

Um bem-haja a todos!

Resumo

Nas equipas de software modernas geralmente há uma pessoa responsável por tomar decisões relativas à estrutura e arquitetura da aplicação a implementar, o Arquiteto de Software. Este elemento é responsável por conceber a estrutura, ou arquitetura da aplicação a partir dos requisitos atuais e futuros do software. Esta decisão acarreta um compromisso de médio-longo prazo entre a equipa de desenvolvimento e a organização. Esta dissertação visa ajudar o arquiteto de software a tomar essa decisão e enquadra duas das principais arquiteturas de software usadas na atualidade: Arquiteturas Monolíticas e Arquiteturas de Micro serviços.

Foi feita uma introdução à temática, bem como a escolha de uma metodologia de estudo e contextualização com o caso de estudo. De seguida foi efetuada uma revisão teórica de conceitos como arquitetura de software, escalabilidade, arquiteturas monolíticas, micro serviços, bem como outras arquiteturas de software para depois colocar esses conceitos em prática no caso de estudo.

No caso de estudo foi feito um enquadramento ao software em estudo, bem como apresentadas todas as ferramentas e tecnologias que foram usadas e por fim foi feita uma comparação entre a arquitetura monolítica que o software tinha, com uma alteração que foi feita à arquitetura do mesmo software para micro serviços e retiradas as devidas conclusões.

Palavras-chave: Arquiteturas de Software, Arquitetura de micro serviços, Arquitetura monolítica

Abstract

In modern software teams there is usually one person responsible for making decisions regarding the structure and architecture of the application to be implemented, the Software Architect. This element is responsible for designing the structure, or architecture of the application from the software current and future requirements. This decision entails a medium-long-term commitment between the development team and the organization. This dissertation aims to help the software architect to make this decision and focuses on two of the main software architectures used today: monolithic architectures and microservices.

An introduction was made to the subject matter, as well as the choice of a study methodology and contextualization with the case study. Then a theoretical revision of concepts such as software architecture, scalability, monolithic architectures, microservices, as well as other software architectures was carried out to later put these concepts into practice in the case study.

In the case study, a contextualization was made for the software under study, as well as all the tools and technologies that were used, and finally a comparison was made between the monolithic architecture that the software had, to the microservices-based architecture that was later implemented.

Índice

DEDICATÓRIA V

AGRADECIMENTOS VII

RESUMO IX

ABSTRACT XI

ÍNDICE XIII

ÍNDICE DE FIGURAS XVII

ÍNDICE DE TABELAS XIX

GLOSSÁRIO XXI

ABREVIATURAS XXIII

INTRODUÇÃO 1

1.1 MOTIVAÇÃO E OBJETIVOS 4

1.2 ORGANIZAÇÃO DO DOCUMENTO 5

2 METODOLOGIA 7

2.1 OBJETIVOS DO CASO DE ESTUDO 8

2.2 TIPOS DE CASOS DE ESTUDO 8

2.3 CONCLUSÕES 9

3 ARQUITETURAS DE SOFTWARE 11

3.1 ARQUITETURA DE SOFTWARE 11

3.2 ESCALABILIDADE 13

3.2.1 Diferença entre escalabilidade horizontal e escalabilidade vertical 14

3.3 ARQUITETURAS MONOLÍTICAS 15

3.4 ARQUITETURAS DE MICRO SERVIÇOS 18

3.5 OUTRAS ARQUITETURAS DE SOFTWARE 21

3.5.1 Arquitetura em N camadas *22*

3.5.2 Arquitetura orientada a eventos *24*

3.5.3 Arquitetura em microkernel *26*

3.5.4 Arquitetura baseada em espaço *29*

3.5.5 Arquitetura orientada a serviços *31*

4 CASO DE ESTUDO 36

4.1 PONTO DE PARTIDA 36

4.2 SOFTWARE DE DESENVOLVIMENTO 36

4.2.1 Visual Studio 2017 .... *36*

4.2.2 SQL Server Management Studio *38*

4.2.3 Notepad++ *38*

4.2.4 Tortoise Git *39*

4.2.5 Azure Storage Emulator *39*

4.2.6 Azure Compute Emulator *39*

4.2.7 SQL Azure MW *40*

4.3 TECNOLOGIAS UTILIZADAS NA APLICAÇÃO 41

4.3.1 ASP.NET MVC *41*

4.3.2 LINQ *42*

4.3.3 Entity Framework *42*

4.3.4 jQuery e KendoUI *43*

4.4 RECOLHA DE DADOS 44

4.5 ARQUITETURA MONOLÍTICA ORIGINAL DA APLICAÇÃO 45

4.5.1 Esquema da Arquitetura *45*

4.5.2 *Desempenho* 48

4.5.3 *Vantagens* 52

4.5.4 *Desvantagens* 53

4.5.5 *Conclusões* 53

4.6 APLICAÇÃO DA ARQUITETURA DE MICRO SERVIÇOS 54

4.6.1 *Enquadramento* 54

4.6.2 *Esquema da arquitetura* 55

4.6.3 *Alocação de recursos por serviço* 57

4.6.4 *Desempenho* 58

4.6.5 *Vantagens e Desvantagens* 62

4.6.6 *Conclusões* 63

5 CONCLUSÕES E TRABALHO FUTURO 65

5.1.1 *Respostas aos objetivos estabelecidos* 66

5.1.2 *Trabalho futuro* 67

6 REFERÊNCIAS 69

ANEXO A: DIAGRAMA GANTT - CRONOGRAMA ORIGINAL 75

ANEXO B: DIAGRAMA GANTT – CRONOGRAMA REVISTO 77

ANEXO C: ARQUITETURA MONOLÍTICA 79

ANEXO D: ARQUITETURA DE MICRO SERVIÇOS 81

Índice de Figuras

Figura 1 - Arquitetura Monolítica de um sistema de ERP 2
Figura 2 – Arquitetura de micro serviços numa aplicação de e-commerce 3
Figura 3 - Escalabilidade Horizontal e Vertical (Mejia, 2016) 14
Figura 4 – Arquitetura Monolítica 15
Figura 5 – Balanceador de carga a distribuir os acessos pelas diferentes instâncias 16
Figura 6 - Exemplo de rede social baseada em micro serviços 19
Figura 7 – Tipologias de bases de dados em arquiteturas baseadas em micro serviços 20
Figura 8 – Comparação entre a escalabilidade de micro serviços e arquiteturas monolíticas. [Fonte: Fowler, Martin (2015)] 21
Figura 9 - Esquema da arquitetura de software em camadas 23
Figura 10 - Arquitetura orientada a eventos com tipologia de mediador [Adaptado de (Richards, 2015)] 25
Figura 11 - Arquitetura orientada a eventos, tipologia de intermediário [Adaptado de: (Richards, 2015)] 26
Figura 12 – Arquitetura em microkernel 27
Figura 13 - Arquitetura de *microkernel* em sistemas ERP 28
Figura 14 - Arquitetura baseada em espaço [Adaptado de: (Richards, 2015)] 29
Figura 15 - Unidade de processamento 30
Figura 16 – *Webservice* usando o protocolo SOAP [Fonte (D. Barry, 2017)] 33
Figura 17 - *Webservice* a trocar mensagens em JSON [Fonte: (D. Barry, 2017)] 34
Figura 18 - Arquitetura Orientada a Serviços usando JSON 34
Figura 19 - Visual Studio 2017 - Ecrâ tradicional 37
Figura 20 - SQL Server Management Studio - operação de consulta 38
Figura 21 - Edição de ficheiros XML a partir do Notepad++ 39
Figura 22 - Azure compute emulator 40
Figura 23 – Interface do software SQL Azure MW 40
Figura 24 - Padrão de arquitetura de software MVC 41
Figura 25 - Consulta efetuada com LINQ [Fonte: (Koirala, 2009)] 42
Figura 26 - Consulta efetuada através da Entity Framework 43
Figura 27 - Código de kendoWindow usando jQuery e KendoUI 44
Figura 28 - Kendo UI – Indicador de visitantes mensais da página web 44
Figura 29 - Projetos da aplicação no Solution Explorer doVisual Studio 2017 46
Figura 30 - Arquitetura da aplicação 47
Figura 31 – Gráfico com nº de utilizadores na instância app.Website.ClientPortal 49
Figura 32 - Percentagem de utilização de CPU na instância app.Website.ClientPortal 50
Figura 33 - Leitura e escrita em disco na instância app.Website.ClientPortal 51
Figura 34 - Tráfego médio de entrada e saída na app.Website.ClientPortal 52
Figura 35 - Projetos no Microsoft Visual Studio com arquitetura de micro serviços 55

Figura 36 - Esquema da arquitetura de micro serviços. ... 57
Figura 37 - Número de utilizadores por *website* numa arquitetura de micro serviços ... 59
Figura 38 - Serviço ExcelGeneration - Leitura e escrita em disco ... 61
Figura 39 - Serviço FileImports - Leitura e escrita em disco ... 61
Figura 40 - Tráfego da entrada e saída por hora na REST API ... 62

Índice de Tabelas

Tabela 1 - Pontos fortes e fracos das arquiteturas monolíticas ... 17
Tabela 2 - Vantagens e desvantagens dos micro serviços ... 21
Tabela 3 – Responsabilidades das camadas ... 22
Tabela 4 - Elementos de Informação REST [Fonte: (Fielding, 2000)] ... 33
Tabela 5 - Recursos disponíveis para instâncias de classe A no Microsoft Azure – [Adaptado de (A. George, Lepow, Squillace, McKenna, & Kabrt, 2017)] ... 48
Tabela 6 - Serviços, requisitos de negócio e recursos alocados ás instâncias ... 57
Tabela 7 - Tabela da utilização de CPU nas diferentes instâncias em Azure ... 60

Glossário

Ad-Blocker: Software responsável pelo bloqueio de publicidade, geralmente é utilizado nos navegadores da internet ou em aplicações para dispositivos móveis.

***Blob storage*:** É um tipo de armazenamento normalmente encontrado em serviços de *cloud computing* que permite o armazenamento de dados não estruturados como imagens, vídeos, ficheiros, entre outros.

***Browser**:* navegador de internet.

Cloud Computing: Refere-se à pratica de contratar e utilizar recursos como CPU, memória RAM, discos, rede, bases de dados, através da internet, a "*cloud*".

Customer Relationship Management: É software em que a abordagem coloca o cliente como principal foco dos processos de negócio, com o intuito de perceber e antecipar as suas necessidades e preocupações, para proporcionar o melhor atendimento possível (Luck & Lancaster, 2003).

Deploy: disponibilizar um sistema para uso, seja num ambiente de desenvolvimento, testes ou produção.

Enterprise Resource Planning: Aplicação empresarial personalizável que inclui soluções de negócio integradas para o ramo de negócio da empresa bem como as principais funções administrativas (Klaus, Rosemann, & Gable, 2000).

Framework é um conjunto de classes cooperantes que constituem um elemento de design reutilizável para uma classe específica de software (Gamma, Helm, Johnson, & Vlissides, 2002).

Kernel: Núcleo do sistema operativo. O núcleo é a parte principal do sistema operativo do computador. A função do núcleo do sistema é ligar o software ao hardware, estabelecendo uma comunicação eficaz entre os recursos do sistema.

REST: *Representational State Transfer* (REST), em português Transferência de Estado Representacional, é uma abstração da arquitetura da *World Wide Web*, mais precisamente, é um

estilo arquitetural que consiste num conjunto coordenado de restrições arquiteturais aplicadas a componentes, conectores e elementos de dados dentro de um sistema de hipermédia distribuído. O REST ignora os detalhes da implementação de componente e a sintaxe de protocolo com o objetivo de focar nos papéis dos componentes, nas restrições sobre sua interação com outros componentes e na sua interpretação de elementos de dados significantes (Fielding, 2000).

Round-Robin: Neste algoritmo, a cada processo é atribuído um intervalo de tempo, o quantum, no qual ele é permitido executar; Se no final do quantum o processo não terminou, o processo é interrompido e outro processo entra para executar; quando um processo termina o seu quantum, ele é colocado no final da fila.

Stakeholders: Público estratégico; pessoa ou grupo de pessoas que tem interesse em uma empresa, um projeto, um negócio, etc.

Stand-alone*:* Componente ou software que funciona de forma independente.

Web service: É uma solução utilizada na integração de sistemas e na comunicação entre aplicações diferentes, de forma normalizada, normalmente utilizando as tecnologias da internet, como o HTTP, XML, JSON.

URI: *Uniform Resource Identifier* ou Identificador Uniforme de Recurso é uma *string* compacta usada para identificar ou denominar um recurso da internet.

XML: *Extensible Markup Language* é uma linguagem de notação usada para descrever informação. O Standart XML é uma forma flexível de criar formatos de informação e partilhar eletronicamente a mesma através da internet. (Bray, Paoli, Sperberg-McQueen, Maler, & Yergeau, 2008)

Abreviaturas

API: Application Programming Interface

DAL: Data Access Layer

DTO: Data Transfer Object

EVD: Event Driven Architecture

HTTP: Hypertext Transfer Protocol

IDE: Integrated Development Environment

LA: Layered Architecture

MKA: Microkernel Architecture

MLA: Monolithic Architecture

MSA: Micro Service Architecture

MVC: Model View Controller

SBA: Space Based Architecture

SOA: Service Oriented Architecture

Introdução

A arquitetura de software emergiu como uma importante subdisciplina da engenharia de software. Arquitetura é, grosso modo, uma divisão de um todo em partes, com relações específicas entre as partes. Esta divisão permite que grupos de pessoas trabalhem em colaboração para resolverem um problema de complexidade muito superior do que qualquer uma dessas pessoas poderia resolver individualmente. (Bass, Clements, & Kazman, 2012)

Uma arquitetura de software é uma representação do sistema que ajuda a compreender como este se irá comportar (SEI, 2015) e serve como modelo para o sistema e o projeto que o implementa, atribuindo tarefas que devem ser seguidas por equipas de design e desenvolvimento. A arquitetura é o principal motor de qualidades como o desempenho, modificabilidade e segurança. Nenhuma destas pode ser alcançada sem uma visão de arquitetura unificadora. É uma ferramenta inicial de análise para avaliar se uma abordagem de design irá obter um resultado aceitável. Ao construir uma boa arquitetura, é possível identificar riscos de design e mitigá-los mais cedo no ciclo de desenvolvimento do projeto.

No âmbito desta dissertação irão ser abordadas com maior enfoque duas arquiteturas de software amplamente usadas nos projetos de software em meio empresarial: arquiteturas monolíticas e arquiteturas de micro serviços.

Nos sistemas com arquiteturas monolíticas, a aplicação é concentrada numa única base de código, que se torna extensa á medida que a aplicação cresce. Este tipo de arquitetura também tem a grande desvantagem de ter um ponto único de falha. Ou seja, se uma funcionalidade falhar, esta pode comprometer o acesso a outras áreas da aplicação. A sua base de código pode ser intimidante para os programadores que entrem a meio do ciclo de desenvolvimento do projeto, já que a complexidade do código é bem maior. Por outro lado, temos um sistema muito fácil de implementar, já que a base de dados irá evoluir uniformemente para todas as funcionalidades da aplicação e existe apenas um único ponto onde a implementação é feita.

Além disso, não há duplicidade de código e de classes necessárias nos diferentes módulos, já que todas elas fazem parte de um único bloco de código, como podemos ver na Figura 1.

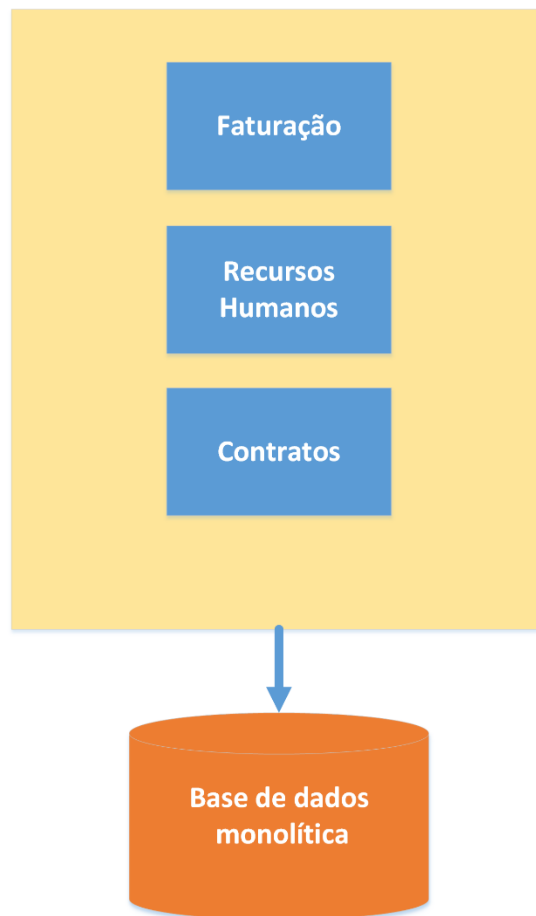


Figura 1 - Arquitetura Monolítica de um sistema de ERP

Nos sistemas com arquiteturas de micro serviços, a aplicação está particionada em módulos ou serviços que podem ser implementados de forma independente. Cada equipa de desenvolvimento trabalha no seu micro serviço e de forma autónoma em relação às outras equipas. Este tipo de abordagem facilita a integração dos programadores nas equipas de desenvolvimento porque a base de código de um serviço é relativamente simples, o sistema deixa de ter um ponto único de falha, tendo um por cada micro serviço. Tem algumas desvantagens, como a duplicidade de código comum a vários módulos, o facto de haver a necessidade de fazer um *deploy* por cada serviço (pode não ser uma desvantagem porque permite que pequena alteração ao código não implique ter de fazer *deploy* à aplicação por inteiro). Estes serviços, por norma, utilizam uma API em HTTP para comunicar entre si.

Outra questão relevante é a base de dados. Em algumas abordagens de arquiteturas de micro serviços esta é única, centralizada e usada por todos os serviços e em outro tipo de abordagens, existem várias bases de dados, que são acedidas por serviços específicos.

Na Figura 2 apresenta-se um esquema de uma arquitetura de micro serviços numa plataforma de *e-commerce*. Os diversos serviços interagem com o serviço central, a loja *online*.

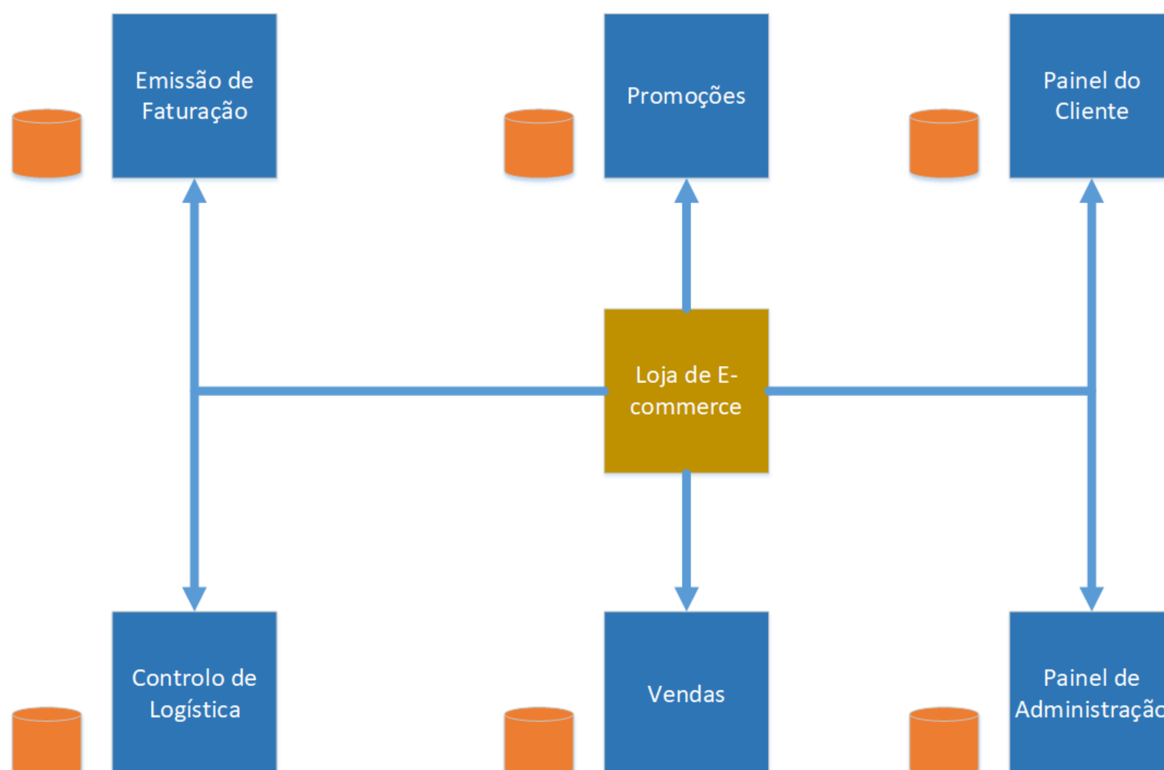


Figura 2 – Arquitetura de micro serviços numa aplicação de e-commerce

Os serviços são independentes entre si, como o controlo de logística, emissão de notas fiscais, promoções, disponibilização de *e-books*. Se colocarmos como cenário a falha do serviço de promoções, todos os outros continuam a funcionar sem problemas.

No âmbito desta dissertação, irá ser realizado um estudo comparativo entre estes dois tipos de arquitetura de software, que visa responder a uma questão principal: Qual é a melhor arquitetura em cada situação? Quais são as vantagens e desvantagens de cada uma das duas arquiteturas? De forma a responder a estas questões, e a outras que possam surgir, irá ser

feita uma extensa revisão da literatura e estado da arte, de forma a esclarecer e melhor compreender os conceitos apresentados. Numa fase posterior irão ser realizados inquéritos a vários especialistas na área da arquitetura de software, de forma a perceber que tipo de arquiteturas são mais usadas no meio empresarial em Portugal, as razões para isso acontecer, e o porquê de acontecerem.

De seguida, irão ser aplicadas ambas as arquiteturas num estudo de caso, de forma a fazer uma comparação objetiva e obter respostas às perguntas anteriormente referidas. Após esta fase, existirá um período reservado à escrita e melhoramento do documento da dissertação, e respetiva preparação para a defesa pública da mesma.

1.1 Motivação e Objetivos

A arquitetura de software é um ramo da engenharia de software, com cada vez mais importância nos dias de hoje. Com a evolução dos processos de desenvolvimento de aplicações surgem desafios novos todos os dias. Um desses principais desafios é a escalabilidade, e é esse atributo o principal foco desta dissertação.

Propomos um estudo sobre duas arquiteturas: arquiteturas monolíticas e arquiteturas em micro serviços; além da escalabilidade, já referida, irão ser analisados outros atributos destas duas arquiteturas, destacando-se:

- Facilidade de manutenção de código
- Complexidade
- Separação de conceitos
- Nível de abstração
- Segurança
- Fiabilidade

O tema desta dissertação é bastante interessante, do ponto de vista do mestrando, porque aborda temáticas atuais e também visa responder a questões que as pessoas que trabalham ou estudam arquiteturas de software têm diariamente no seio da sua atividade profissional. É desafiante, porque embora existam alguns artigos e dissertações a abordar a temática da escalabilidade de

aplicações informáticas, há poucas que tenham o foco direcionado para arquiteturas de micro serviços ou monolíticas. Esta temática é interessante do ponto de vista do mestrando, porque além de ter liberdade, no âmbito da sua atividade profissional, de tomar decisões sobre arquitetura do software desenvolvido, o conhecimento que irá ser adquirido no âmbito desta dissertação irá ser uma mais-valia, tanto no plano pessoal como profissional. Neste último, atualmente, o mestrando trabalha com uma arquitetura monolítica. Este estudo irá também servir para melhor compreender esta arquitetura, e para definir, em situações futuras, que tipo de abordagem é a mais adequada para cada caso.

Um outro objetivo, não menos importante deste estudo, é a sua publicação numa conferência científica da área da Arquitetura de Software, após a conclusão do mesmo ou, eventualmente, a publicação das conclusões deste estudo numa revista científica da área do estudo.

1.2 Organização do Documento

Esta dissertação está dividida em 5 capítulos, que se descrevem a seguir:

- **Capítulo 1: Introdução** – Introdução ao tema e definição de objetivos a atingir
- **Capítulo 2: Metodologia:**
- **Capítulo 3: Estado da Arte** – Enquadramento teórico das arquiteturas de software em estudo.
- **Capítulo 4: Caso de Estudo** – Enquadramento prático das arquiteturas de software em estudo.
- **Capitulo 5: Conclusão** - Conclusões obtidas e trabalho futuro a desenvolver.

2 Metodologia

A metodologia é um ponto essencial em qualquer trabalho de investigação, e como tal, é crucial entender o seu significado. De acordo com Gil (2008)

> *"Metodologia é um conjunto de abordagens técnicas e processos utilizados pela ciência para formular e resolver problemas de aquisição objetiva do conhecimento, de uma maneira sistemática."*

O papel de qualquer trabalho de investigação científica é produzir novo conhecimento, de forma consistente, lógica e criteriosa. Para isso é necessário tomar como ponto de partida teorias estabelecidas, conhecimento produzido no passado, mas revisto e atualizado de forma a produzir novo conhecimento com validade científica.

Uma das formas de produzir o novo conhecimento é o estudo de caso, metodologia de pesquisa que teve origem nas ciências sociais. Em meados do século XX, Schramm (1955) propôs a seguinte definição de estudo de caso:

> *A essência de um estudo de caso, a principal tendência em todos os tipos de estudo de caso, é que ela tenta esclarecer uma decisão ou um conjunto de decisões: o motivo pelo qual foram tomadas, como foram implementadas e com quais resultados.*

Ao longo do tempo a definição foi explorada por vários investigadores das ciências sociais, mas sem obterem êxito na tentativa de encarar o estudo de caso como uma metodologia formal de pesquisa. Um erro comum era confundir estudo de caso com estudos etnográficos, (Fetterman, 1989), ou com observação participante (Jorgensen, 1989). Em Kidder & Judd (1981) o "trabalho de campo" ainda é tratado como uma técnica de recolha de dados e omitem qualquer discussão do estudo de caso. A definição mais amplamente aceite pela comunidade científica neste tema é atribuída a Robert Yin., Segundo (Yin, 1989) um estudo de caso é uma investigação empírica que investiga um fenómeno contemporâneo dentro de seu contexto da

vida real, especialmente quando os limites entre o fenómeno e o contexto não estão claramente definidos.

2.1 Objetivos do Caso de estudo

Para (Yin, 1989) o caso de estudo pode ser do tipo exploratório, descritivo ou explanatório.

O caso de estudo exploratório tem como objetivo a descoberta, explicação ou constatação de fenómenos ou assuntos com os quais há pouco ou nenhum conhecimento prévio.

O caso de estudo descritivo tem como objetivo observar, analisar, registar e interpretar fenómenos sem intervenção do investigador, dentro do seu contexto.

O caso de estudo explanatório tem como objetivo explicar relações de causa-efeito a partir de uma teoria ou hipótese formulada à priori.

O estudo de caso presente nesta dissertação assume uma perspetiva descritiva, centrando os seus objetivos no entendimento da forma como as diferentes arquiteturas de software são aplicadas a um software comum e quais os impactos das mesmas no software em causa, ou seja, os processos são mais valorizados que os resultados. Dentro desta lógica foram identificados dois objetivos:

- Perceber se existem aumentos de desempenho passando de uma arquitetura monolítica para uma de micro serviços
- Compreender as vantagens e desvantagens de ambas as arquiteturas e em que caso devemos optar por uma arquitetura monolítica ou micro serviços.

2.2 Tipos de Casos de Estudo

Ao longo dos anos a comunidade científica apresentou vários critérios de tipificação dos casos de estudo. Em 1989 Robert Yin classificou os casos de estudo de modo quantitativo (Yin, 1989). Segundo Yin, existem dois tipos de casos de estudo: único, em que o objeto de estudo é uma

pessoa, um grupo de pessoas, uma instituição, um software; e múltiplo, no qual, vários estudos são conduzidos em simultâneo.

Além do modo quantitativo, em 2000, Robert E. Stake classificou os casos de estudo quanto à sua finalidade: intrínseco, instrumental e coletivo (Stake, 2000).

No caso de estudo intrínseco o objeto de estudo é de interesse para o investigador, e este pretende obter uma melhor compreensão sobre o caso apenas pelo interesse para com o mesmo. No estudo de caso instrumental, o interesse do caso não assenta exclusivamente em compreender o caso em si, mas provar algo mais amplo, como uma teoria ou fenómeno ou até mesmo refutar uma teoria provando que o caso não se enquadra na mesma. Por fim temos o estudo de caso coletivo em que o investigador estuda em conjunto vários casos para investigar algo através da comparação entre eles.

De acordo com o critério quantitativo, estamos perante um tipo de caso de estudo único, visto que o objeto do estudo é apenas uma única aplicação.

Em relação à sua finalidade, estamos claramente perante um estudo de caso intrínseco visto que o objeto de estudo é do interesse do investigador e pretende responder a questões colocadas por este.

2.3 Conclusões

O estudo de caso permite que o investigador observe, entenda, analise e descreva uma determinada situação, adquirindo conhecimento e experiência que podem ser úteis em tomadas de decisão futuras.

O método de investigação dá liberdade ao mestrando para conduzir o estudo de uma forma dinâmica e progressiva tendo contacto prático sobre as arquiteturas de software que irá estudar ao longo do estudo do caso.

A expectativa é que o mestrando adquira conhecimentos e experiência para tomar decisões e identificar à priori, dada uma determinada especificação de software, qual a arquitetura mais apropriada ao desenvolvimento do mesmo.

3 Arquiteturas de Software

Neste capítulo apresentam-se os conceitos teóricos que vão ser usados no decorrer desta dissertação, e que foram a base de desenvolvimento do trabalho. Vamos tentar definir uma arquitetura de software, explicar a origem do conceito e, de seguida, abordaremos a temática da escalabilidade onde tentaremos definir o conceito e apresentar tipos de escalabilidade.

As principais características das arquiteturas monolíticas e de micro serviços serão explicadas, assim como outras arquiteturas alternativas complementares que, embora não sendo o alvo do estudo do caso deste trabalho, nos permitem completar a contextualização teórica.

3.1 Arquitetura de Software

Em 1995, David Garlan e Dewayne Perry definiram arquitetura de software da seguinte forma:

> "*The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time*". (Garlan & Perry, 1995)

Para complementar esta definição, Nord, Hofmeister, & Soni (1999) definem quatro grandes áreas:

- A arquitetura conceptual descreve o sistema em termos dos seus principais elementos de design e das relações entre eles.
- A arquitetura de interconexão entre os módulos engloba duas estruturas ortogonais: decomposição funcional e por camadas.
- A arquitetura de execução descreve a estrutura dinâmica de um sistema.
- A arquitetura do código descreve como é que o código fonte, ficheiros binários e bibliotecas são organizadas no ambiente de desenvolvimento.

No livro "*Patterns of Enterprise Application Architecture*", Fowler (2002) identifica dois elementos comuns enquanto tenta definir arquiteturas de software:

One is the highest-level breakdown of a system into its parts; the other, decisions that are hard to change. It's also increasingly realized that there isn't just one way to state a system's architecture; rather, there are multiple architectures in a system, and the view of what is architecturally significant is one that can change over a system's lifetime.

Bass, Clements, & Kazman (2012), no livro "*Software Architecture in Practice",* apresentam uma definição mais consensual*:*

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

Em 1968 Edsger Dijkstra defendeu que seria importante entender como o software era particionado e estruturado, em oposição a apenas programar de forma a obter um resultado correto (Dijkstra, 1968). Dijkstra estava a escrever sobre um sistema operativo, e foi o primeiro a apresentar a noção de uma estrutura em camadas, em que os programas eram agrupados em camadas, e os programas de uma camada poderiam apenas comunicar com programas de camadas adjacentes. Dijkstra mostrou que a integridade conceptualmente elegante exibida por esta organização iria produzir resultados na facilidade de desenvolvimento e manutenção.

David Parnas continuou com esta linha de observação com as suas contribuições sobre módulos de ocultação de informação (Parnas, 1972), estruturas de software (Parnas, 1974) e famílias de programas (Parnas, 1976).

Uma família de programas é um conjunto de programas (que nem todos necessariamente foram ou serão construídos) em que é lucrativo considerá-los como um grupo. Este termo evita conceitos ambíguos como "funcionalidade semelhante" que por vezes surgem ao descrever domínios. Por exemplo, *websites* e videojogos não são normalmente considerados no mesmo domínio, embora possam ser considerados membros de uma mesma família de programas em discussões sobre ferramentas para o auxílio de interfaces gráficas, que ambos usam. [1]

[1] Este exemplo ilustra que os membros de uma mesma família de programas podem incluir elementos que normalmente são considerados de domínios diferentes.

> *Todo o trabalho desenvolvido no ramo da arquitetura de software pode ser visto como evoluindo para um paradigma de desenvolvimento de software baseado nos princípios da arquitetura, pelas mesmas razões dadas por Dijkstra e Parnas: A estrutura é importante e ter uma estrutura boa traz benefícios.* (Clements & Northrop, 1996).

Outro ponto que é importante esclarecer é o propósito da arquitetura de software. A arquitetura de software usualmente refere-se a uma combinação de um conjunto de pontos de vista estrutural de um sistema, sendo que cada ponto de vista é uma abstração legítima do sistema em relação a um critério concreto. O utilizador, por exemplo, pode estar preocupado com o ponto de vista da usabilidade e fiabilidade, enquanto o cliente pode estar preocupado se o desenvolvimento de software cabe dentro do orçamento para o projeto, ou é feito em tempo útil.

Este conceito, relativamente simples, foi amplamente adotado por uma larga variedade de *stakeholders* e participantes no desenvolvimento do software.

3.2 Escalabilidade

Segundo Luke (1993) "*The concept of scalability in parallel systems is a simple one: given a reasonable performance on a sample problem, a problem of increased workload can be solved with reasonable performance given a commensurate increase in computational resources.*" Luke define a escalabilidade como algo que pode ser resolvido apenas com o aumento de recursos computacionais, e, provavelmente, em 1993, era uma explicação aceitável, mas nos dias de hoje é difícil concordar com esta definição.

Segundo Technopedia (2016) *"Scalability is an attribute that describes the ability of a process, network, software or organization to grow and manage increased demand. A system, business or software that is described as scalable has an advantage because it is more adaptable to the changing needs or demands of its users or clients."*. Este conceito, mais atual, e também mais adaptado à realidade dos dias de hoje, descreve a escalabilidade como a forma de um processo, rede, software ou organização crescer e gerir os seus serviços em alturas de maior carga, sendo adaptável á carga exercida pelos seus utilizadores ou clientes.

3.2.1 Diferença entre escalabilidade horizontal e escalabilidade vertical

Escalabilidade horizontal (*scale in/out*) significa que iremos adicionar mais nós (ou remover nós) ao sistema, como por exemplo passar de um nó com 1 CPU / 1 GB RAM para três nós de características iguais (Figura 3). Este tipo de escalabilidade tem a vantagem de normalmente ter custos mais baixos e a sua principal desvantagem é o facto de necessitar de um balanceador de carga para distribuir o fluxo de carga pelos vários nós.

Escalabilidade vertical (*scale up/down*) significa que vamos adicionar recursos a (ou remover recursos de) um sistema, normalmente CPU, memória RAM, discos. Este tipo de escalabilidade tem a vantagem de ser mais favorável à virtualização de recursos e tem a desvantagem de normalmente ter custos mais elevados e de o nó único ser um ponto único de falha.

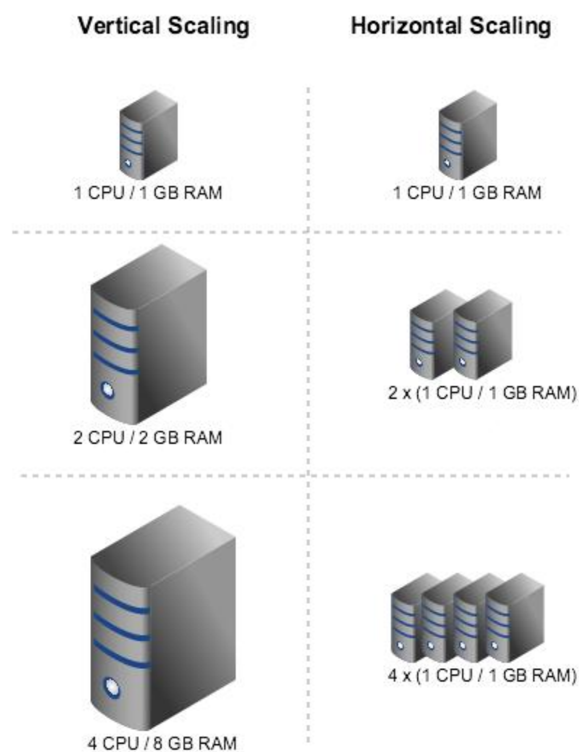


Figura 3 - Escalabilidade Horizontal e Vertical (Mejia, 2016)

3.3 Arquiteturas monolíticas

Uma arquitetura monolítica, de acordo com Richardson (2014b) é fácil de desenvolver, implementar e de escalar, tendo também algumas desvantagens inerentes. A principal é a sua complexidade, que pode, por vezes, assustar programadores menos experientes; por outro lado, quanto maior a quantidade de código numa só aplicação, mais difícil é fazer a sua manutenção.

Em 2014 Ketan Parmar, no seu artigo sobre arquitetura de software, apresentou uma definição para aplicação monolítica como "*Monolithic application has single code base with multiple modules. Modules are divided as either for business features or technical features. It has single build system which build entire application and/or dependency. It also has single executable or deployable binary*" (Parmar, 2014). Um esquema representativo de uma arquitetura monolítica é mostrado na Figura 4.

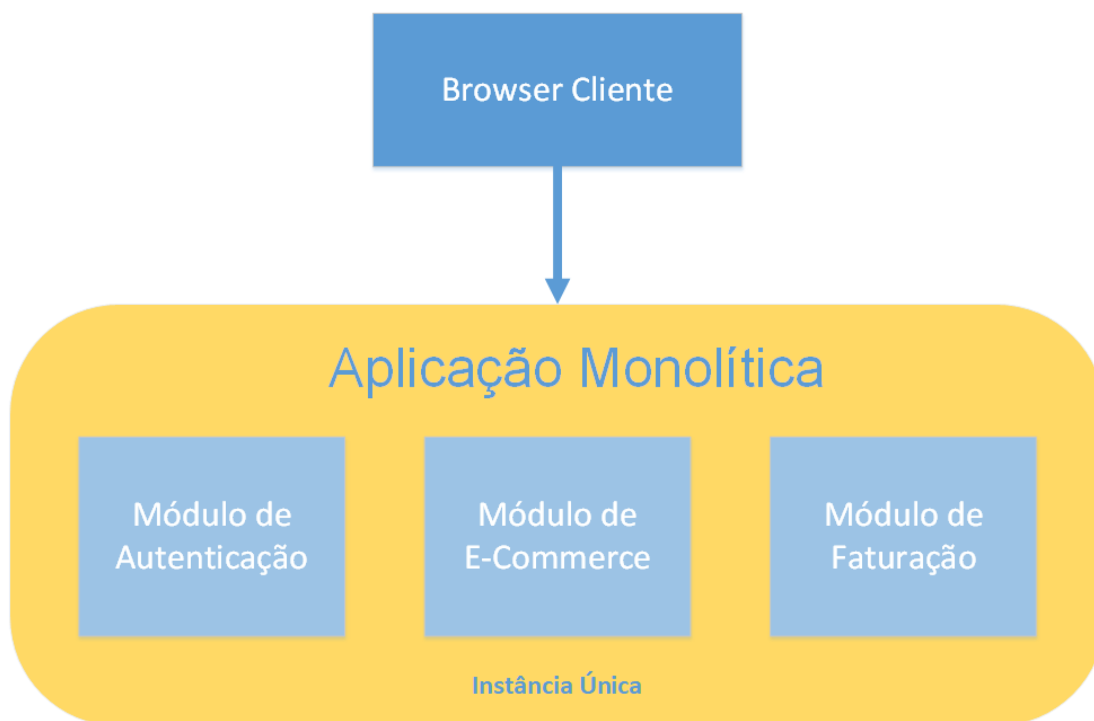


Figura 4 – Arquitetura Monolítica

As arquiteturas de micro serviços e monolíticas são escaláveis, no entanto as arquiteturas monolíticas geralmente são apenas escaladas horizontalmente, ou seja, são adicionadas novas instâncias de forma a que os pedidos sejam repartidos entre elas.

O mecanismo que permite repartir os pedidos feitos pelos *browsers* clientes pelas várias instâncias é o balanceador de carga, tal como se mostra na Figura 5.

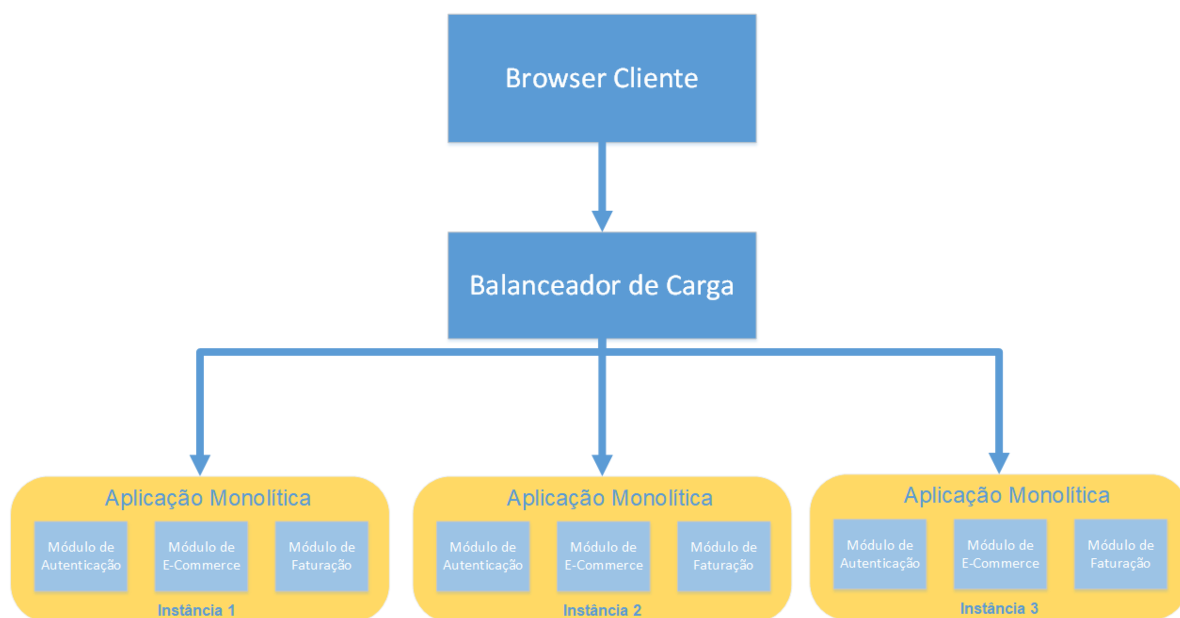


Figura 5 – Balanceador de carga a distribuir os acessos pelas diferentes instâncias

De acordo com Kotkondawar, Khaire, Akewar, & Patil (2014), um balanceador de carga é um mecanismo que distribui dinamicamente a carga de forma uniforme entre todas as instâncias evitando uma situação em que algumas instâncias estão a ser sobrecarregadas enquanto outras estão com pouca ou nenhuma carga.

Para escalar uma aplicação monolítica é necessário fazer a sua replicação em várias instâncias, o que nos leva a um problema de ineficiência. Vejamos a aplicação de *e-commerce* da Figura 4 no cenário seguinte:

- 10.000 utilizadores a usar o módulo de autenticação
- 600 utilizadores a usar o módulo de *e-commerce*
- 50 utilizadores a usar o módulo de faturação

Assumindo que o balanceador de carga distribui os utilizadores uniformemente pelas várias instâncias, continua a existir aqui um problema: apenas necessitamos de escalar o módulo de autenticação, os restantes módulos não têm utilizadores suficientes que justifique adicionar novas instâncias. Este problema acaba por se acentuar ainda mais se o número de utilizadores envolvidos for de centenas de milhares, ou até mesmo milhões.

Apesar desta clara desvantagem no que concerne à escalabilidade, uma arquitetura monolítica oferece também vantagens. O desenvolvimento é mais rápido numa fase embrionária do software devido à não preocupação em separar os componentes em serviços distintos. É mais fácil de testar e de depurar porque todos os módulos se encontram no seio de uma única aplicação. Como o código se encontra no mesmo local, é mais simples de navegar por todo o código da aplicação usando uma ferramenta como um IDE. Como todos os módulos da aplicação também se encontram juntos, é mais fácil de implementar a aplicação em servidores remotos.

Tabela 1 - Pontos fortes e fracos das arquiteturas monolíticas

Pontos Fortes	**Pontos Fracos**
Facilidade de desenvolvimento	Ponto único de falha
Facilidade de *deploy*	Base de código extensa
Suporte de ambientes de desenvolvimento integrados (IDE)	Ineficiência de escalabilidade horizontal
Suporte da comunidade de programadores	Utilização de uma única linguagem de programação para toda a aplicação (pode também ser um ponto forte)
Maioria das aplicações tem por base esta arquitetura	Falta de flexibilidade para adicionar novos módulos
Integração entre os componentes	Alterações a um componente implicam o *deploy* da aplicação inteira

3.4 Arquiteturas de micro serviços

Em 2014, Martin Fowler e James Lewis definiam micro serviços como uma abordagem para desenvolver uma única aplicação como um conjunto de serviços, cada um executado de forma independente e comunicando entre si através da rede local ou da internet, geralmente através de uma REST API. Estes serviços estão divididos de acordo com as necessidades de negócio e podem ser *deployed* de forma independente uns dos outros (Fowler & Lewis, 2014). De acordo com Richardson (2014c), algumas empresas[2] migraram, ou consideram migrar, as suas aplicações atuais para micro serviços.

Uma arquitetura de micro serviços tem alguns benefícios em relação às arquiteturas monolíticas: 1) cada micro serviço é relativamente pequeno, e o código é de fácil compreensão por parte dos programadores (Richardson, 2014a); 2) o código pode ser implementado em separado, pois cada micro serviço é independente dos restantes; 3) , cada micro serviço pode ser escalado horizontalmente (aumentando o número de instâncias e balanceando de carga); 4) a arquitetura de software elimina um compromisso de longo prazo com uma tecnologia ou *stack* de software específica.

Isto deve-se ao facto de comunicarem de acordo com protocolos standard e, no desenvolvimento de um micro serviço novo, os programadores terem a liberdade de utilizar qualquer linguagem ou *framework* mais ajustada ao serviço em causa.

Este tipo de design poliglota também acaba por ser benéfico para as empresas, pela possibilidade de aproveitarem ao máximo a capacidade de programação dos programadores em várias linguagens e atribuindo os programadores especialistas em uma dada linguagem de programação para um micro serviço desenvolvido nessa linguagem. Sam Newman refere-se a este conceito como *Technology Heterogeneity* e apresenta a sua definição como:

> *With a system composed of multiple, collaborating services, we can decide to use different technologies inside each one. This allows us to pick the right tool for each job,*

---

[2] Chris Richardson dá alguns exemplos de empresas que fizeram a migração de arquiteturas monolíticas para micro serviços no seu artigo como a Amazon, Ebay, Uber e Netflix

rather than having to select a more standardized, one-size-fits-all approach that often ends up being the lowest common denominator." (Newman, 2015)

Newman explora este conceito mostrando que até as bases de dados podem ser de tecnologias diferentes consoante o tipo de dados que armazenam. Por exemplo, para uma rede social, podemos guardar as interações entre utilizadores numa base de dados orientada a grafos, as publicações numa base de dados orientada a documentos e as imagens e vídeos publicados num armazenamento em *blob storage*, tal como mostra a Figura 6.

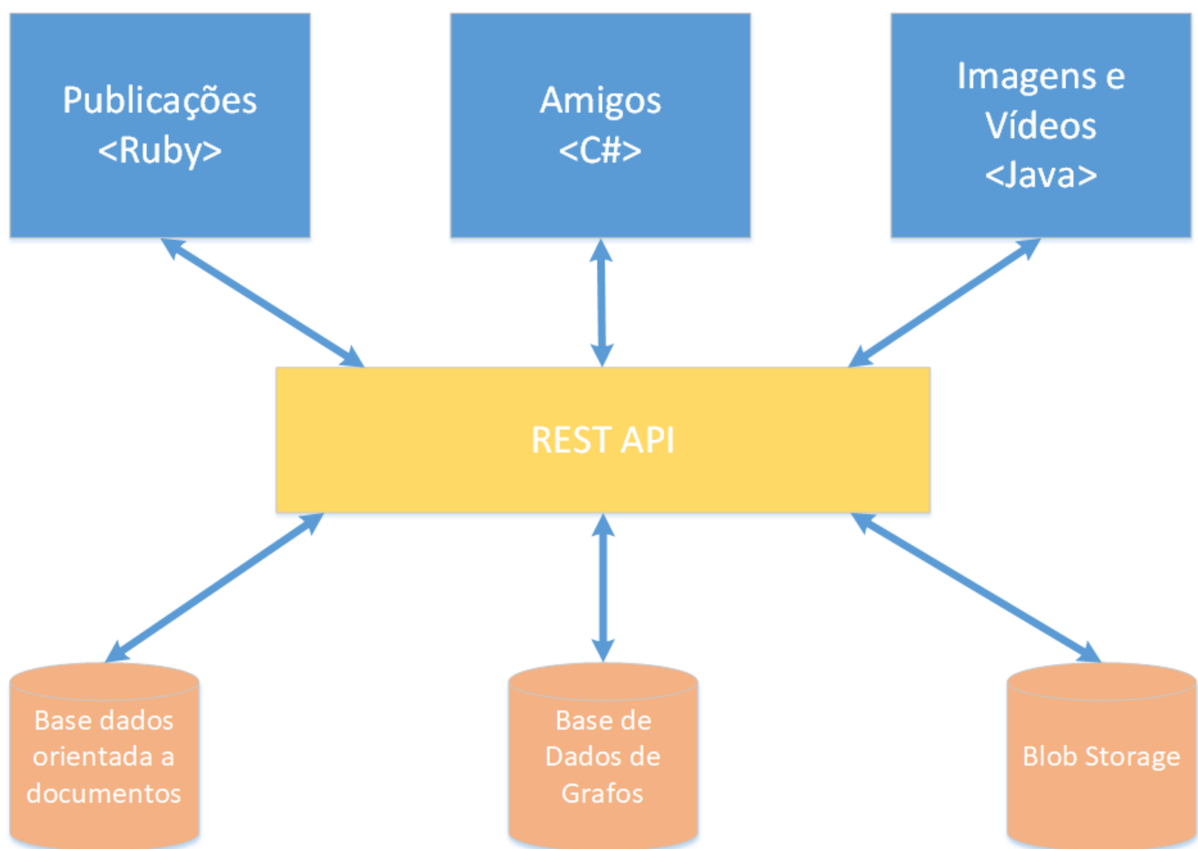
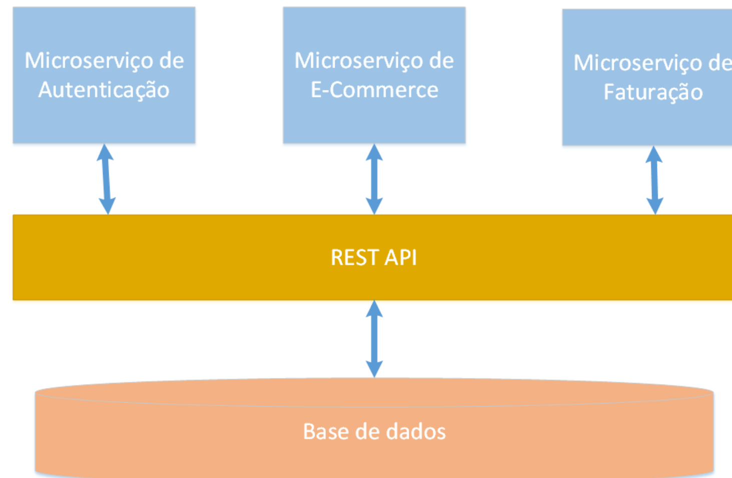


Figura 6 - Exemplo de rede social baseada em micro serviços

Outra vantagem é a possibilidade de descentralização da base de dados. Nas aplicações que usam arquitetura de micro serviços pode ser usada uma única base de dados ou podemos optar por múltiplas bases de dados, sendo que neste último modelo, um serviço pode comunicar com apenas uma ou várias bases de dados, através da REST API, tal como se mostra na Figura 7.



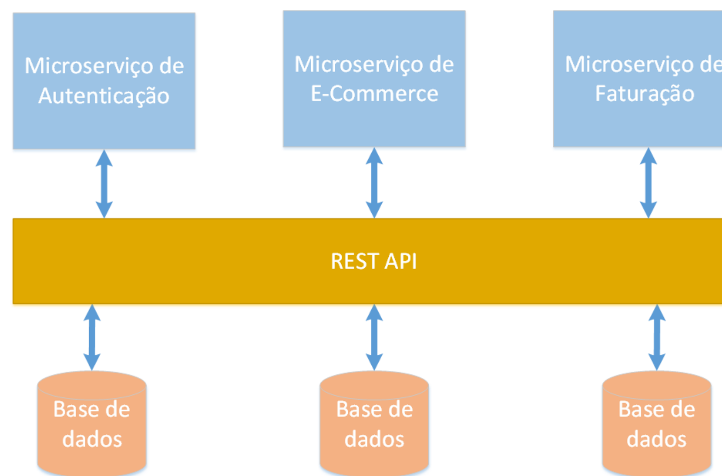


Figura 7 – Tipologias de bases de dados em arquiteturas baseadas em micro serviços

A escalabilidade das arquiteturas de micro serviços tem a vantagem de poder ser feita horizontalmente, adicionando novas instâncias dos serviços, mas também verticalmente, particionando a aplicação em módulos pequenos para as principais funcionalidades de negócio. Se for necessário criar uma funcionalidade de negócio posterior, é fácil adaptar-se à arquitetura. É suficiente adicionar um novo serviço para comunicar com a API já existente, tal como se mostra na Figura 8.

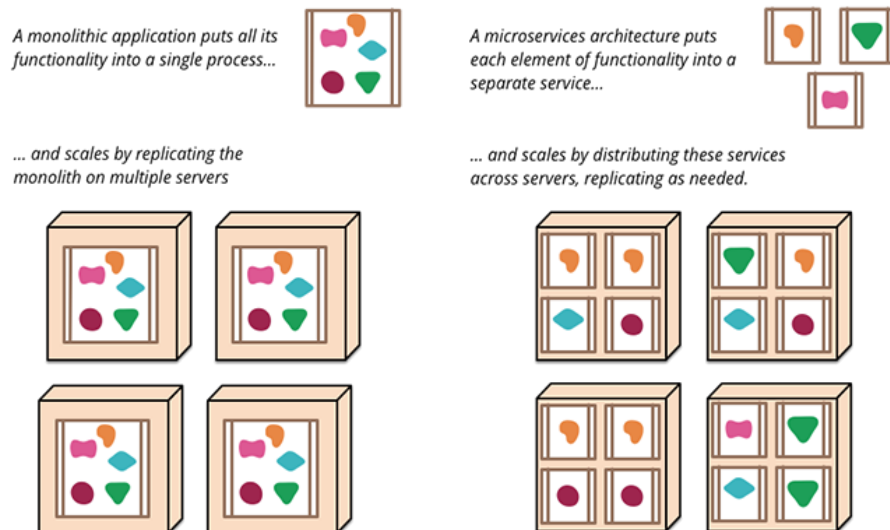


Figura 8 – Comparação entre a escalabilidade de micro serviços e arquiteturas monolíticas. [Fonte: Fowler, Martin (2015)]

É importante realçar que numa arquitetura de micro serviços não existe um ponto único de falha, isto é, se um destes serviços falhar, o acesso ao resto da aplicação não é afetado. Podemos, também, corrigir bugs de forma independente em cada serviço. Na Tabela 2 podemos ver algumas vantagens e desvantagens da arquitetura de micro serviços em síntese.

Tabela 2 - Vantagens e desvantagens dos micro serviços

Vantagens	**Desvantagens**
Independência dos serviços	Complexidade
Simplicidade em adicionar novas funcionalidades	Necessidade comunicar através de redes locais ou da internet
Tolerância a falhas	Fraco suporte dos IDE para sistemas distribuídos
Heterogenia em tecnologias	Necessidade de possuir mais unidades de processamento pode levar a custos superiores.

3.5 Outras arquiteturas de software

Esta secção tem o propósito de apresentar outras arquiteturas de software amplamente usadas, tanto como alternativa, como em conjunto com as duas arquiteturas que são usadas no caso de estudo desta dissertação. Estas arquiteturas são usadas pelas comunidades de programadores um pouco por todo o mundo e compõem uma grande fatia das aplicações usadas todos os dias:

- Arquitetura em camadas
- Arquitetura orientada a eventos
- Arquitetura em *microkernel*
- Arquitetura baseada em espaço
- Arquitetura orientada a serviços

3.5.1 Arquitetura em N camadas

A arquitetura de software mais comum é a arquitetura em camadas, também conhecida como *layered architecture* ou arquitetura em N camadas, sendo N o número de camadas.

Os componentes usados na arquitetura são organizados em camadas horizontais, com cada camada desempenhando uma tarefa específica no seio da aplicação. A definição da arquitetura em camadas não especifica um número exato de camadas, mas a maioria das arquiteturas por camadas consistem em quatro camadas essenciais: apresentação, negócio, acesso aos dados e base de dados (Richards, 2015).

Cada camada desta arquitetura tem uma função específica e uma responsabilidade dentro da aplicação. A camada de apresentação é responsável pela interface de utilizador e lógica de comunicação com o browser. A camada de negócio é responsável por executar código específico às regras de negócio da aplicação. A camada de persistência permite fazer a ligação entre a camada de negócio e a camada de base de dados, enquanto que a camada de base de dados permite armazenar a informação ao longo do tempo (Richards, 2015). Na Tabela 3 podemos observar as responsabilidades inerentes a cada uma das camadas.

Tabela 3 – Responsabilidades das camadas

Camada	**Responsabilidade**
Apresentação	Apresentação e formatação da informação
Negócio	Calcular valores; agregar e validar informação; aplicar regras e políticas de negócio.
Acesso aos dados	Isolar as camadas superiores do SGBD que a aplicação usa.
Base de dados	Guardar a informação de forma persistente

Estas camadas são independentes. A camada de negócio, por exemplo, não necessita de saber como é que a aplicação apresenta a informação, apenas necessita de aplicar a lógica do negócio (e.g. calcular valores, validar condições), e passar essa informação para a camada de

apresentação, que por sua vez é responsável pela representação dos dados nas interfaces utilizadas pelos utilizadores.

A principal vantagem do modelo em camadas é a separação de conceitos entre os componentes. Os componentes de uma camada apenas têm de se preocupar com a lógica dessa camada. Este tipo de classificação dos componentes faz com que seja fácil definir responsabilidades e funções diferentes para os componentes de uma camada dentro da arquitetura. Também acaba por se tornar mais fácil desenvolver, testar e manter uma aplicação usando esta arquitetura devido aos componentes e camadas bem definidos e com raio de ação limitado (Bushmann, Meunier, & Rohnert, 1996).

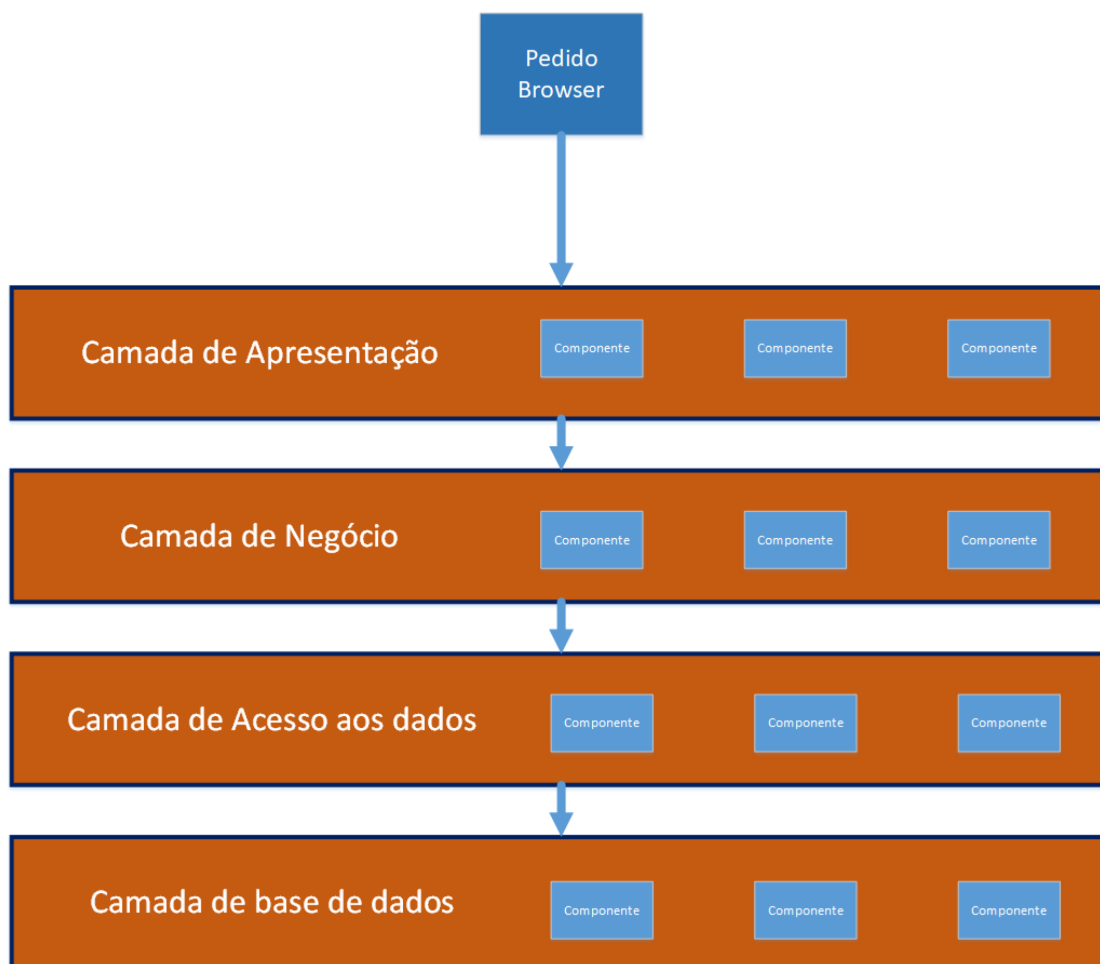


Figura 9 - Esquema da arquitetura de software em camadas

A arquitetura em camadas é uma arquitetura sólida e com créditos firmados, que pode ser um bom ponto de partida para a maioria das aplicações, especialmente se não foi tomada uma decisão sobre a arquitetura a utilizar na aplicação.

3.5.2 Arquitetura orientada a eventos

A arquitetura orientada a eventos (EOA), é um padrão de desenvolvimento popular em sistemas distribuídos de forma assíncrona. É flexível e pode ser usado tanto em aplicações simples como em aplicações mais complexas. Esta arquitetura é feita de componentes altamente desacoplados e com funções individuais que recebem e processam eventos de forma assíncrona (Richards, 2015).

Este tipo de arquitetura consiste em duas principais topologias: "*mediator*" e "*broker*", ou em português respetivamente, mediador e intermediário.

A topologia de mediador é usada quando é necessário orquestrar múltiplas operações através de um mediador central. Nesta tipologia existem quatro tipos de componentes principais:

- Mediador de eventos
- Canais de eventos
- Processadores de eventos

O fluxo do evento começa com o cliente a enviar um evento para uma fila de eventos, que consiste em enviar vários pedidos assíncronos para os canais de eventos para executar cada passo do processo. Os processadores de eventos recebem o evento através do mediador e executam a lógica do negócio para processar o evento. A Figura 10 ilustra a tipologia do mediador numa arquitetura orientada a eventos.

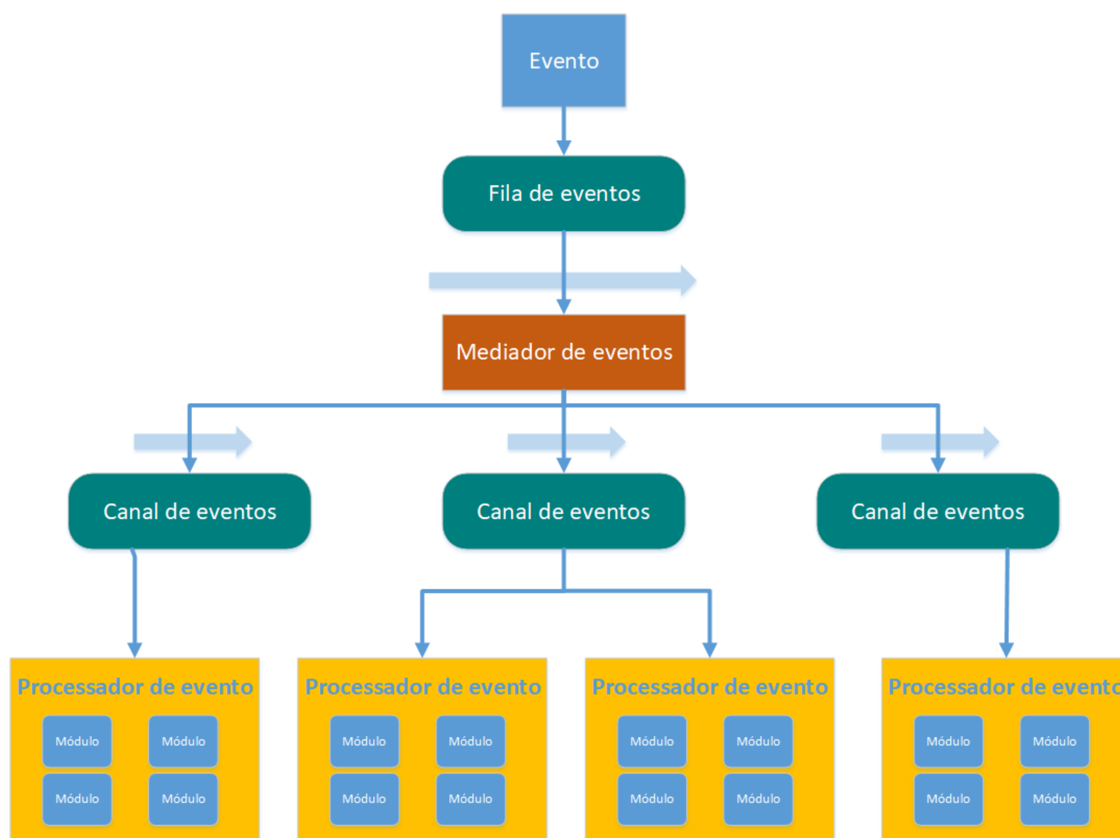


Figura 10 - Arquitetura orientada a eventos com tipologia de mediador [Adaptado de (Richards, 2015)]

Neste tipo de tipologia existem dois tipos de eventos, os eventos iniciais e os eventos processados. Os eventos iniciais são os eventos recebidos pelo mediador, e os eventos processados são os eventos que são gerados pelo mediador e recebidos pelos componentes subsequentes.

A tipologia em *broker* difere da tipologia de mediador na medida que não existe um mediador central, em vez disso o fluxo é distribuído entre os canais de eventos em cadeia através de um intermediário. Esta topologia é útil em aplicações em que o fluxo de eventos é simples e não é desejável ou necessário um mediador central.

Nesta topologia temos dois tipos principais de componentes: *Broker* (Intermediário) e Processador de eventos. O intermediário contém todos os canais de eventos que são usados no fluxo. Os canais de evento que estão contidos no intermediário podem ser filas de mensagens, tópicos de mensagens ou uma combinação dos dois.

Como podemos ver na Figura 11, não existe um mediador de eventos central a orquestrar o evento inicial, em vez disso cada processador de eventos é responsável por processar o evento.

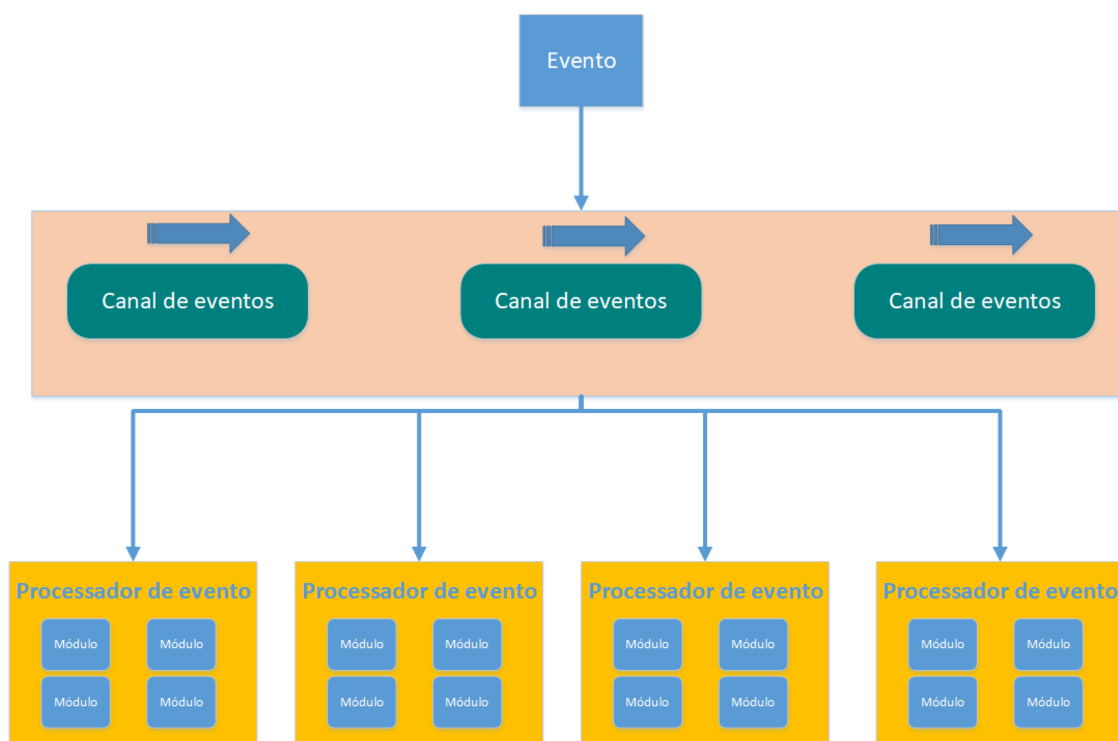


Figura 11 - Arquitetura orientada a eventos, tipologia de intermediário [Adaptado de: (Richards, 2015)]

A tipologia de intermediário funciona com base numa cadeia de eventos para realizar um requisito de negócio. Para ilustrar melhor este conceito vamos usar uma analogia com corridas com passagem de testemunho. Numa corrida com passagem de testemunho os atletas correm uma determinada distância com o testemunho e passam-no ao próximo corredor, até o último atleta passar a meta e acabar a corrida. Neste tipo de corridas a partir do momento que o atleta entrega o testemunho, a corrida acabou para ele. Isto também é verdade na tipologia de intermediário, a partir do momento que o processador de eventos passa o evento para o intermediário, deixa de estar envolvido no processamento desse evento específico.

3.5.3 Arquitetura em *microkernel*

Este tipo de arquitetura, também conhecida como *plug-in architecture* é um tipo natural de arquitetura para implementação de aplicações baseadas em produtos. Este tipo de aplicação

pressupõe a existência de um produto-base, mas que é possível adicionar extensões e *plug-ins*, quer feitos pela equipa de desenvolvimento ou por terceiros, que visam aumentar as capacidades da aplicação para além do seu produto base (Richards, 2015).

Os *browsers* modernos, por exemplo, como o Google Chrome ou Mozilla Firefox, permitem aos utilizadores instalarem extensões que dotam a aplicação de funcionalidades não presentes na aplicação base, como é o caso dos *ad-blockers,* que permitem evitar que o utilizador veja publicidade enquanto navega no *browser.*

A arquitetura em *microkernel* consiste em dois tipos de componentes: o núcleo de sistema e os módulos de *plug-in.* A lógica da aplicação está dividida entre ambos, fornecendo extensibilidade, flexibilidade e isolamento de funcionalidades da aplicação e lógica de processamento personalizada. A Figura 12 ilustra uma arquitetura de *microkernel*.

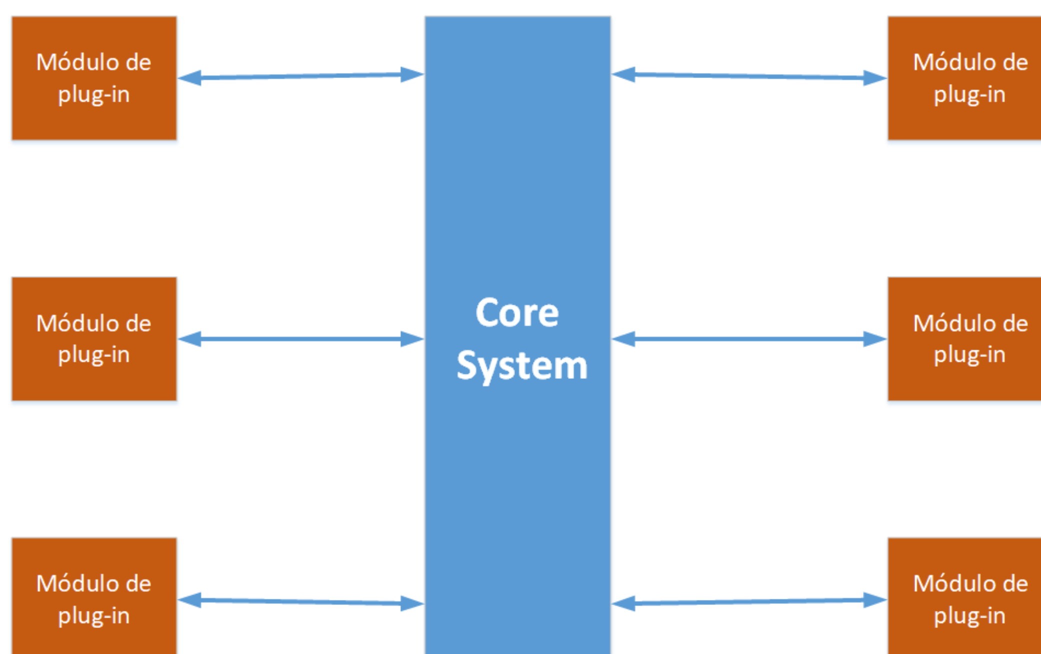


Figura 12 – Arquitetura em microkernel

O núcleo de sistema de uma arquitetura em *microkernel* tradicionalmente contém apenas funcionalidades essenciais de forma a tornar a aplicação operacional; um exemplo disto são os sistemas operativos, daí a origem do nome da arquitetura. Dum ponto de vista do negócio, o núcleo de sistema pode ser visto como a lógica de negócio com a adição de algum código extra para casos especiais, regras ou processamento complexo.

Os módulos de *plug-in* são *stand-alone* ou independentes que contém processamento especializado, funcionalidades extra e código personalizado que serve para melhorar ou aumentar a capacidade do núcleo de sistema.

O núcleo de sistema necessita de saber que módulos de *plug-in* existem e como interagir com eles. A forma mais usada de atingir esse fim é o registo de *plug-ins*. Este registo contém informação sobre cada *plug-in*, como o seu nome, permissões ou protocolo de comunicação.

Estes *plug-ins* ligam-se ao núcleo da aplicação de maneiras variadas, mas a mais comum é através de *webservices.*

Um exemplo de tipo de aplicação que usa esta arquitetura são os sistemas de *Enterprise Resource Planning* ou ERP. A Figura 13 ilustra os diferentes *plug-ins* a ligarem-se com o núcleo do sistema ERP.

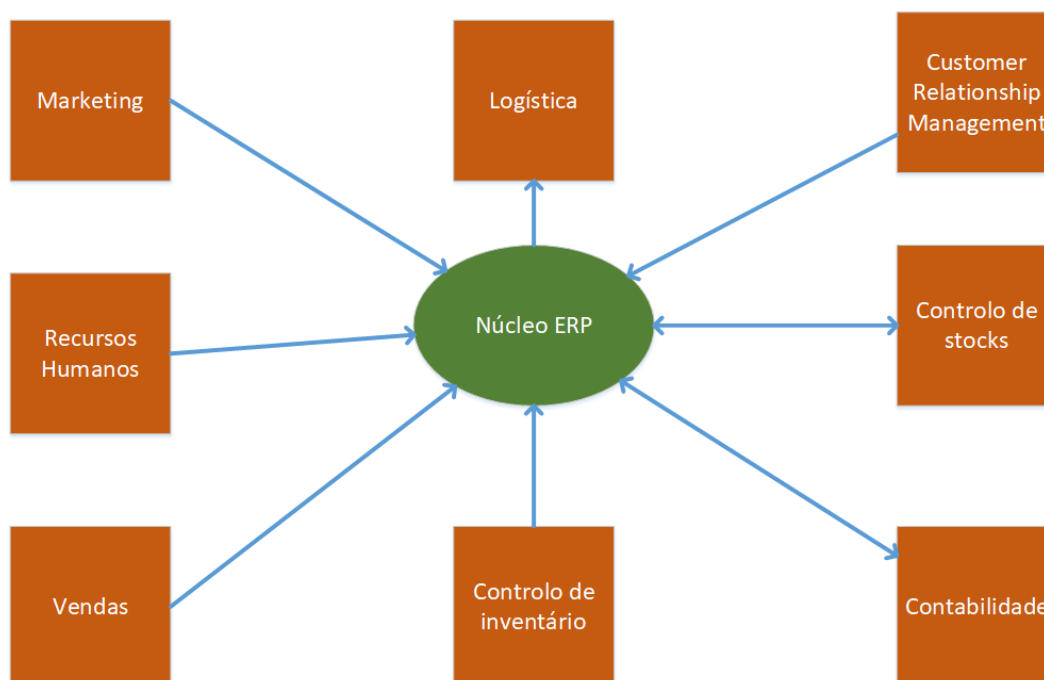


Figura 13 - Arquitetura de *microkernel* em sistemas ERP

Uma consideração importante em relação a esta arquitetura é que pode ser usada em conjunto ou como parte de outra arquitetura existente; por exemplo uma aplicação pode inicialmente ser desenvolvida numa arquitetura em N camadas, e quando estiver estável e for necessária mais personalização, pode-se aplicar a arquitetura de *microkernel*, em que o núcleo continua

a ter por base um padrão de N camadas, enquanto que a aplicação como um todo permite a adição de novos módulos de *plug-in*.

3.5.4 Arquitetura baseada em espaço

A arquitetura baseada em espaço, também conhecida por *Space based architecture, Cloud based Architecture ou* SBA foi desenhada especificamente para resolver os problemas de escalabilidade e concorrência para aplicações que têm volumes de utilizadores concorrentes imprevisíveis. Este problema pode ser resolvido com escalabilidade horizontal ou vertical, mas a arquitetura baseada em espaço é uma tentativa de resolver este problema pela raiz, a arquitetura (Taylor et al., 1996).

Esta ideia de escalabilidade alta é alcançada com a eliminação da base de dados, guardando a informação em memória que é replicada por todas as unidades de processamento. As unidades de processamento podem ser ligadas e desligadas consoante o volume de utilizadores aumenta ou diminui. Como não existe uma base de dados central, essa limitação é removida, criando uma escalabilidade quase infinita no seio da aplicação. Existem dois tipos principais de componentes usados nesta arquitetura: as unidades de processamento e o *middleware* virtualizado. A Figura 14 ilustra o funcionamento desta arquitetura, bem como os seus componentes principais.

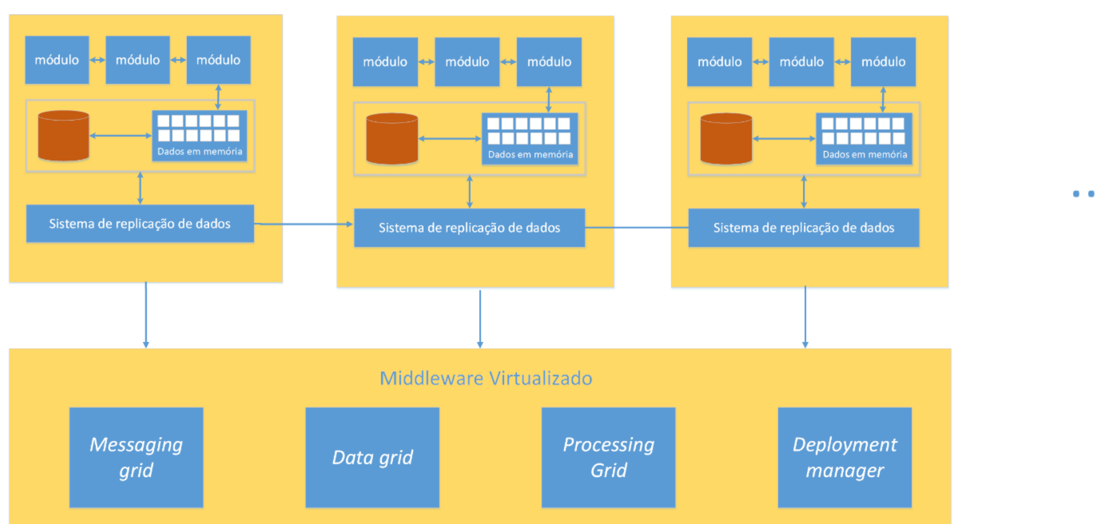


Figura 14 - Arquitetura baseada em espaço [Adaptado de: (Richards, 2015)]

A unidade de processamento contém os componentes da aplicação (ou partes destes), isto inclui componentes web, mas também a lógica de negócio. Os componentes da unidade de

processamento variam com o tipo de aplicação. Aplicações mais pequenas podem ser implementadas com uma unidade de processamento único, enquanto que aplicações maiores podem dividir a lógica de processamento em várias unidades de processamento de acordo com as áreas funcionais da aplicação. Além destes componentes, a unidade de processamento contém um módulo de memória para armazenamento de informação, além de um mecanismo de replicação de dados, tal como mostra a Figura 15 .

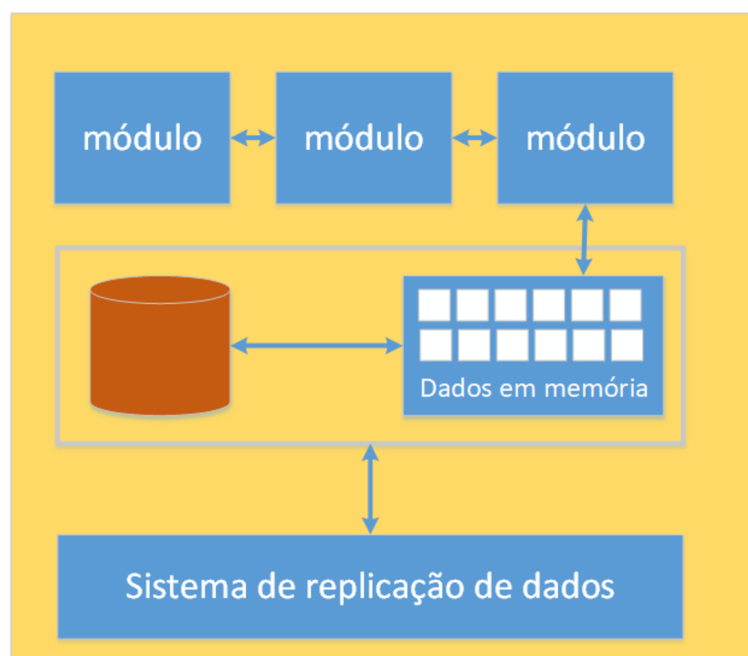


Figura 15 - Unidade de processamento

O *middleware* virtualizado é essencialmente o controlador para a arquitetura e gere os pedidos, sessões, replicação de dados, distribuição de pedidos e gestão de unidades de processamento (Lenk, Klems, Nimis, Tai, & Sandholm, 2009). Há quatro componentes principais do *middleware* virtualizado:

- *Messaging grid*
- *Data grid*
- *Processing grid*
- *Deployment manager*

O *messaging grid* é responsável pela gestão dos pedidos e das informações de sessão. Quando um pedido entra no *middleware* virtualizado, o *messaging grid* determina que unidades de processamento estão ativas e reencaminha o *request* para uma dessas unidades de

processamento. A complexidade deste componente pode variar de um simples algoritmo de *round-robin* para outros algoritmos mais complexos.

O *Data grid* é provavelmente o componente mais importante e crucial desta arquitetura. Este componente interage diretamente com o mecanismo de replicação de dados em cada unidade de processamento sempre que atualizações aos dados ocorrem, para efetuar a replicação dos dados entre unidades de processamento.

O *message grid* pode enviar um pedido para qualquer uma das unidades de processamento disponíveis, por isso é essencial que estas contenham exatamente os mesmos dados em memória.

O *processing grid* é um componente opcional que funciona quando existem múltiplas unidades de processamento, cada uma com uma parte da aplicação, e é responsável por encaminhar o pedido para a unidade de processamento correta.

Por fim, *o Deployment manager* é o componente que faz a gestão dinâmica da inicialização ou encerramento de unidades de processamento de acordo com a carga ou volume de utilizadores da aplicação em modo concorrencial. Aumenta o número de unidades de processamento se a carga aumentar, e diminui o número de unidades de processamento se a carga diminuir (A. D. George, Squillace, Nottingham, & Pratt, 2017).

Embora uma base de dados central não esteja contemplada na arquitetura, esta é normalmente incluída para carregar inicialmente os dados em memória e para ser possível o armazenamento persistente dos dados, o que é feito pelo *data grid* e pelos sistemas de replicação de dados, geralmente de forma assíncrona (Shalom, 2014).

3.5.5 Arquitetura orientada a serviços

A arquitetura orientada a serviços (SOA), tem como imagem de marca o facto de ser uma arquitetura orientada para lidar com os problemas de escalabilidade e disponibilidade em grandes ambientes empresariais. Esta arquitetura é na sua essência um conjunto de serviços que comunicam entre si através de algum mecanismo, seja ele internet ou rede local.

Barry (2003) define SOA como *"A service is a function that is well-defined, self-contained, and does not depend on the context or state of other services." e argumenta que* os serviços são ligados geralmente através de *WebServices*. Endrei et al. (2004) recordam uma definição da autoria do grupo de arquitetos de software da W3C (Booth et al., 2004):

> *"A Web service is a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols."*

Os *webservices* comunicam através de protocolos. Os mais conhecidos são *Simple Object Access Protocol (*SOAP), *Representational State Transfer* (REST) e *Javascript Object Notation* (JSON).

O SOAP é um protocolo de mensagens que permite a transmissão de informação através da internet. O protocolo SOAP utiliza o formato XML e tipicamente (mas não exclusivamente) enviado através do protocolo HTTP. Cada mensagem SOAP está contida num "envelope" que inclui cabeçalho e corpo da mensagem.

Como todas as mensagens SOAP utilizam o mesmo formato, este é capaz de funcionar em diferentes sistemas operativos e protocolos. Na Figura 16 podemos observar como um *webservice* utiliza o protocolo SOAP para enviar mensagens através da internet.

Fielding (Fielding, 2000) define REST da seguinte forma:

> *The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements. It encompasses the fundamental constraints upon components, connectors, and data that define the basis of the Web architecture, and thus the essence of its behavior as a network-based application.* (Fielding, 2000).

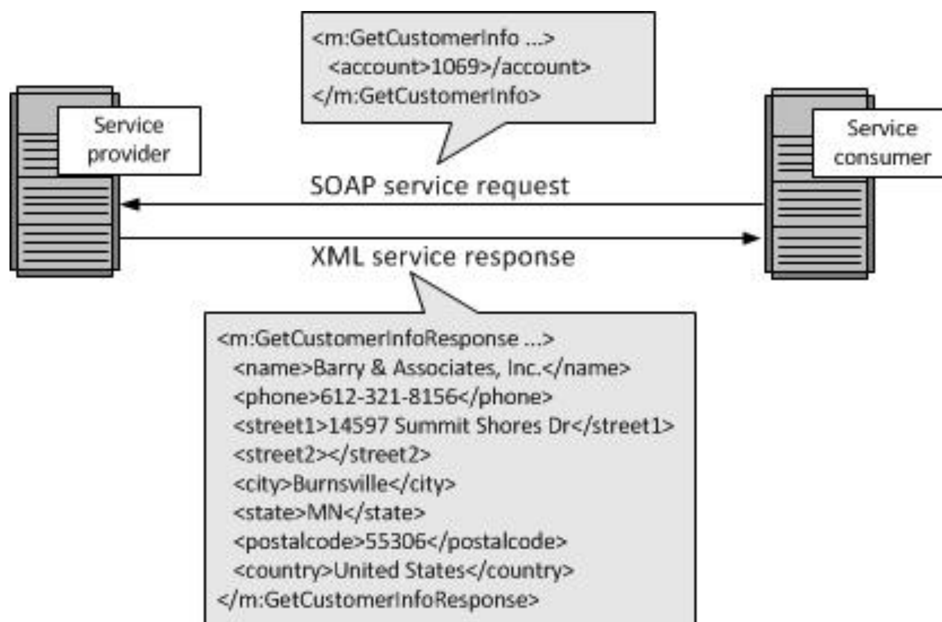


Figura 16 – *Webservice* usando o protocolo SOAP [Fonte (D. Barry, 2017)]

Na Tabela 4 podemos ver os elementos do REST de forma sumarizada.

Tabela 4 - Elementos de Informação REST [Fonte: (Fielding, 2000)]

Data Element	**Modern Web Examples**
Resource	the intended conceptual target of a hypertext reference
resource identifier	URL, URN
Representation	HTML document, JPEG image
representation metadata	media type, last-modified time
resource metadata	source link, alternates, vary
control data	if-modified-since, cache-control

Por fim também existem *webservices* que usam a notação JSON derivada da especificação da linguagem ECMAScript (ECMA-262, 1999) publicada em 1999 pela ECMA. Embora o JSON tenha derivado do ECMAScript é um formato de texto independente de linguagem.

O JSON tem por base dois pontos essenciais, uma coleção de pares de chave/valor e uma lista ordenada de valores, que estão presentes nas linguagens de programação modernas (Bulgarian et al., 2009). Na Figura 17 podemos observar um *webservice* a trocar mensagens através de JSON.

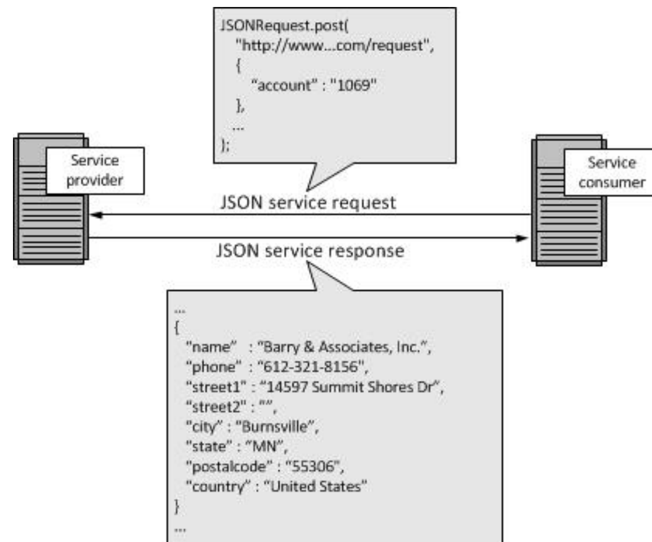


Figura 17 - *Webservice* a trocar mensagens em JSON [Fonte: (D. Barry, 2017)]

A Figura 18 apresenta-nos uma visão geral da arquitetura orientada a serviços, neste caso utilizando o protocolo JSON.

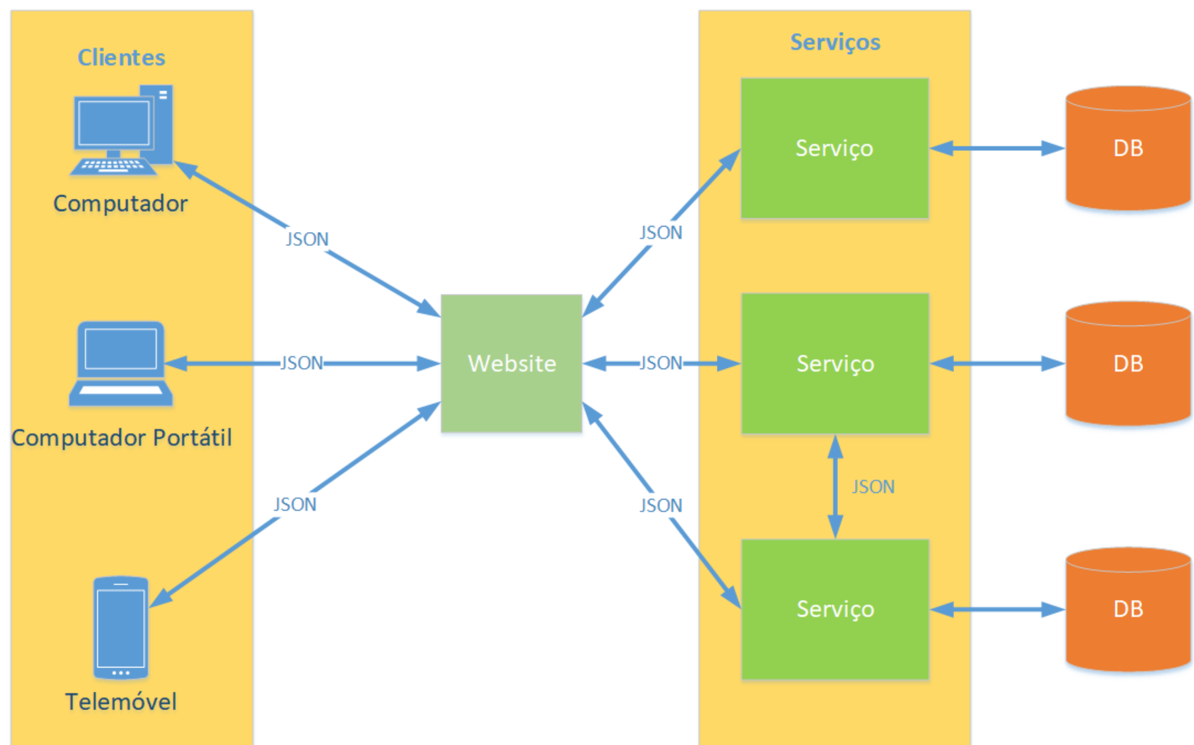


Figura 18 - Arquitetura Orientada a Serviços usando JSON

Podemos então dizer que numa arquitetura orientada a serviços temos serviços que comunicam entre si e com o exterior (Aplicações, *websites*, bases de dados …) através de *webservices* utilizando protocolos conhecidos, como o SOAP, XML ou JSON.

4 Caso de Estudo

Este caso de estudo insere-se num contexto de trabalho do mestrando na empresa Estamos Juntos, LDA no desenvolvimento de um software para uma empresa cliente do sector energético, em particular na compra e venda de energia elétrica. Este software é composto por duas áreas cruciais, o ***Front-office***, que é um site público acessível pelos clientes e público geral, e o ***Back-office***, que é um site privado acessível apenas pelos colaboradores da empresa.

4.1 Ponto de partida

A aplicação em causa teve o início da sua construção no ano de 2013, pelo que não foi desenvolvido inicialmente pelo mestrando, no entanto trata-se de um projeto em constantes alterações promovidas, quer pelo setor regulador como pelas necessidades de inovação da empresa. Tendo isso em conta, o software que existe, nos dias de hoje, é bem mais moderno e robusto do que aquele que foi criado há 5 anos atrás. Com o aumento de clientes da empresa neste universo de cinco anos tornou-se também necessário reformular a arquitetura do software de forma a aumentar o desempenho da aplicação e também fazer uma escalabilidade de forma mais eficiente. O mestrando teve a sua entrada para o projeto no ano de 2016 e é no período compreendido entre setembro de 2016 e outubro de 2017 que o caso de estudo vai incidir.

4.2 Software de Desenvolvimento

Nesta secção irá ser descrito o software utilizado no ambiente de desenvolvimento da aplicação, bem como mostrados, através de exemplos, o impacto prático do uso dos mesmos.

4.2.1 Visual Studio 2017

O software usado para todo o trabalho de desenvolvimento é o Microsoft Visual Studio Enterprise 2017, versão 15.3.5, que tem pré-instalada a Microsoft .NET Framework 4.7.02046

bem como algumas extensões. Este software possui *intellisense*[3], que é bastante útil no processo de desenvolvimento devido às suas funcionalidades de *auto-complete*. A Figura 19 mostra um típico ecrã deste software.

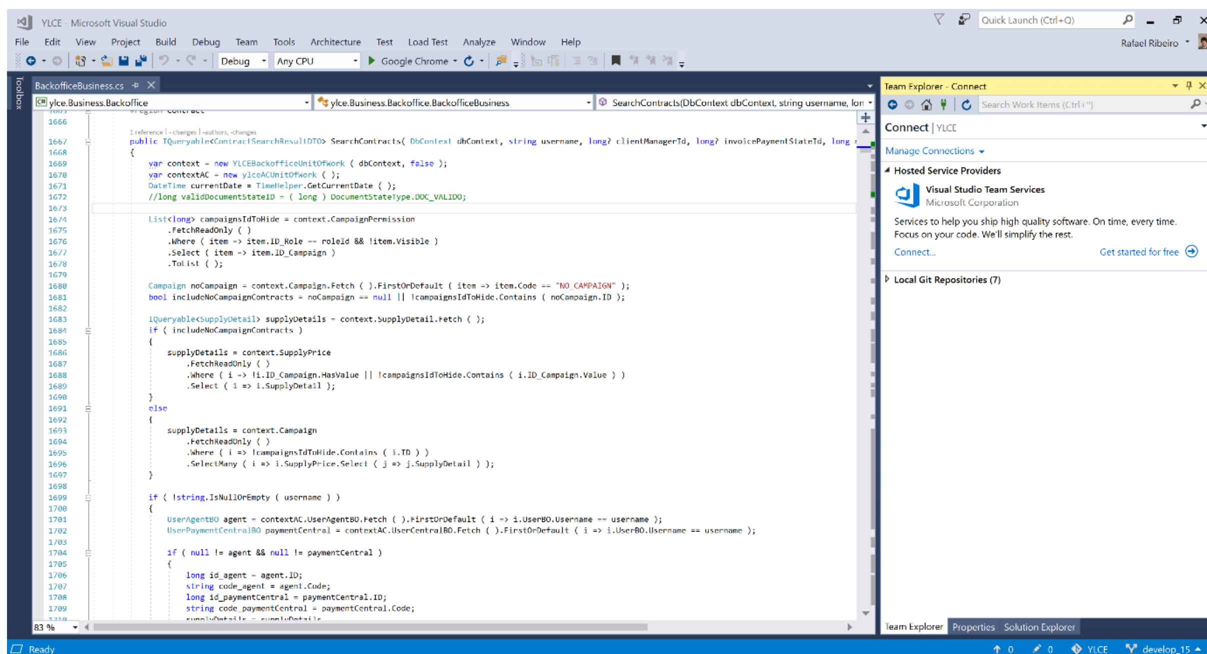


Figura 19 - Visual Studio 2017 - Ecrã tradicional

O Visual Studio permite também instalar uma série de *packages,* através do Nuget package manager, que são funcionalidades que são acrescentadas, não ao nível do Visual Studio, mas sim ao nível da aplicação que estamos a desenvolver.

Este IDE também permite algumas opções interessantes de depuração, como o mecanismo de passo-a-passo em que podemos ir executando o código iteração a interação, linha a linha.

Além disso este IDE é o único que trabalha de forma integrada com a .NET Framework, não só por ambos serem produtos Microsoft, mas também porque outros IDE necessitam de pacotes e extensibilidade para trabalharem nativamente com a plataforma .NET.

[3] Intellisense é um termo geralmente usado para descrever uma variedade de funcionalidades de edição de código como: auto-complete do código, informação dos parâmetros, informação rápida, e listas ligadas, Também é conhecido como "code completion", "content assist" e "code-hinting" [Fonte: (Microsoft Corporation, 2017)]

4.2.2 SQL Server Management Studio

O software SQL Server Management Studio é o escolhido para efetuar manipulação e consultas na base de dados. Este software é frequentemente ligado tanto ao ambiente de desenvolvimento local, como aos servidores de base de dados remotos, que se encontram na plataforma Azure. A versão utilizada é a 17.3 com o número de compilação 14.0.17199.0.

Este software também possui suporte ao *intellisense*, tal como o Visual Studio o que facilita imenso as operações de consulta. A Figura 20 mostra como é a interface do SQL Server Management Studio.

```
USE [app]
GO

/****** Object:  View [app_business].[EstornosView]    Script Date: 18/10/2017 23:01:45 ******/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

create view [app_business].[EstornosView] as
    select
        C.ID, C.Year, I.InvoiceName, C.Comment, I.ID IdInvoice
    from
        [app_business].[Credit] C
        inner join [app_business].[Invoice] I on C.ID_OriginInvoice = I.ID
        inner join [app_business].[Credit_Type] CT on CT.ID = C.ID_CreditType
        INNER JOIN [app_business].[CreditState_Type] CST on CST.ID = C.ID_CreditState
    where
        CT.Token = 'TOTAL_CREDIT'
        and CST.Token = 'CREDIT_NOT_LIQUIDATED'
        and C.Active = 1 and C.EndDate is null;


GO
```

Figura 20 - SQL Server Management Studio - operação de consulta

4.2.3 Notepad++

Como é prática comum em todos os ambientes de desenvolvimento não podia faltar o editor de texto, útil para edição de ficheiros para importação, bem como para pequenos ajustes a ficheiros de desenvolvimento, este editor foi escolhido, não só pela sua compatibilidade, bem como pela sua facilidade de utilização. Este editor também é o usado quando é necessário criar scripts de BD, antes de serem executados a partir do SQL Server Management Studio. Uma das suas principais armas em relação aos demais editores de texto é o chamado *syntax highlighting*, ou seja, o facto de atribuir cores diferentes ao código dependendo da sua sintaxe. A versão usada é a 7.4.2 (64 bit). A figura mostra o aspeto do Notepad++ a editar um ficheiro XML.

Figura 21 - Edição de ficheiros XML a partir do Notepad++

4.2.4 Tortoise Git

O Tortoise Git é a ferramenta usada para o controlo de versões. É necessário instalar o Git para Windows para que funcione corretamente e permite efetuar diversas operações destacando-se o *commit, pull, push, revert, switch/checkout* e *merge*.

4.2.5 Azure Storage Emulator

Para simular localmente o sistema de armazenamento de ficheiros do Microsoft Azure, existe uma aplicação que é executada sempre que é colocado o projeto a executar. O Azure Storage Emulator é capaz de emular armazenamento de ficheiros *blob*, filas e tabelas de ficheiros (Myers & Pratt, 2017). Este emulador não possui uma interface gráfica, e qualquer comando necessita de ser executado na linha de comandos.

4.2.6 Azure Compute Emulator

Além do emulador do sistema de ficheiros também é necessário um emulador de outros processos de computação. Entre outras coisas, este emulador tem um balanceador de carga interno e permite emular múltiplas instâncias, funcionando de forma semelhante ao serviço na *cloud*. Este emulador tem uma interface gráfica onde é possível ver e configurar endereços IP atribuídos ás instâncias que emula. A Figura 22 mostra a interface do Azure Compute Emulator.

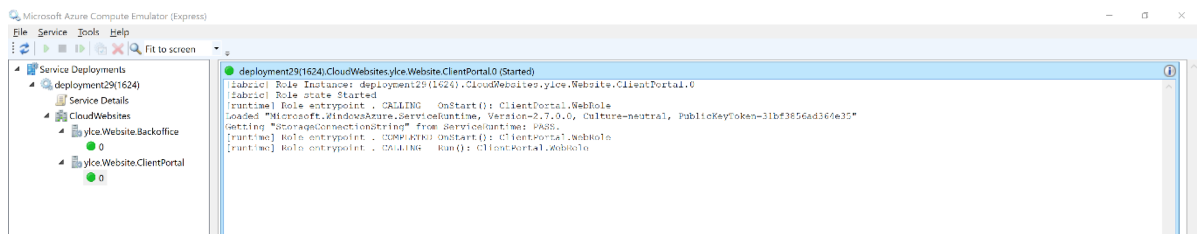


Figura 22 - Azure compute emulator

4.2.7 SQL Azure MW

Este software é usado para efetuar importação e exportação de bases de dados de diferentes formatos. No contexto desta aplicação é usado para copiar a base de dados de produção para o ambiente de desenvolvimento. Esse processo consiste na exportação da base de dados de produção e, posteriormente, importação da BD exportada para o ambiente de desenvolvimento. A Figura 23 apresenta a interface gráfica do software SQL Azure MW.

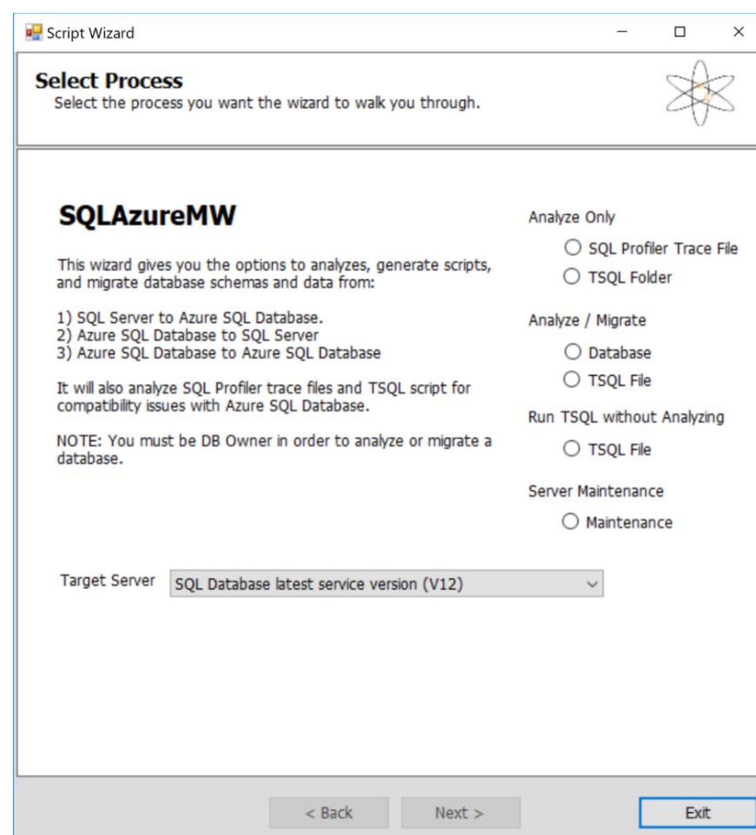


Figura 23 – Interface do software SQL Azure MW

4.3 Tecnologias utilizadas na aplicação

Nesta secção irão ser descritas as principais tecnologias presentes na aplicação, tais como: *frameworks*, padrões de desenvolvimento, dependências.

4.3.1 ASP.NET MVC

O software foi desenvolvido sob a plataforma da Microsoft ASP.NET Framework 4.5.1, utilizando o padrão de software MVC. Este padrão define 3 componentes essenciais:

1. ***Model*** - Informação sobre o estado da aplicação ou os seus componentes, pode ser simples como uma *string* ou até conter objetos derivados de classes mais complexas.
2. ***View*** – É uma representação dos dados contidos no *model* visível ao utilizador; os dados do *model* podem ser representados sob a forma de texto, áudio, vídeo, gráficos e outros.
3. ***Controller*** - O *controller* é responsável por gerir as interações exteriores, invocar alterações no *model* e na *view* mas também pode ser completamente independente de ambos, por exemplo ao receber *callbacks* que não interagem com a *view* ou *model*.

A Figura 24 mostra uma representação gráfica do padrão MVC.

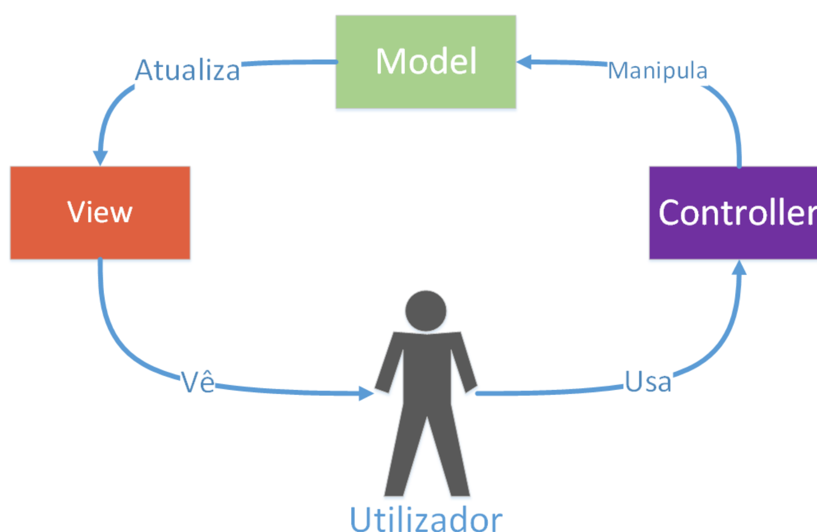


Figura 24 - Padrão de arquitetura de software MVC

4.3.2 LINQ

No caso concreto da aplicação em causa são usadas também outras tecnologias como o *Language Integrated Query* (LINQ) que adiciona componentes de consulta ao C#. A sintaxe do LINQ foi inspirada na *Structured Query Language* (SQL). Uma consulta em LINQ, quando executada em base de dados, é sempre traduzida para SQL pelo sistema de gestão de base de dados. A Figura 25 ilustra esse conceito.

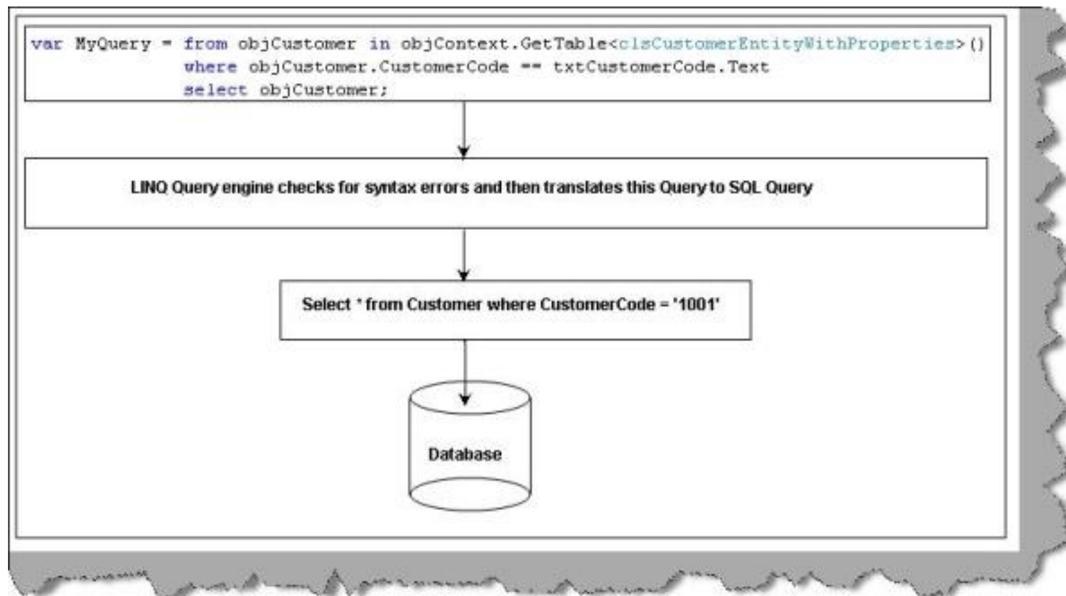


Figura 25 - Consulta efetuada com LINQ [Fonte: (Koirala, 2009)]

4.3.3 Entity Framework

Na camada de acesso aos dados, que iremos falar adiante, é usada também a Entity Framework que é um mecanismo de mapeamento objeto-relacional que permite realizar qualquer tipo de operação á base de dados, através do LINQ, independentemente do sistema de gestão de base de dados que é usado. Isto é especialmente relevante visto que neste projeto são usados motores de base de dados diferentes nos diferentes ambientes em que a aplicação é executada, tal como mostra a Figura 26.

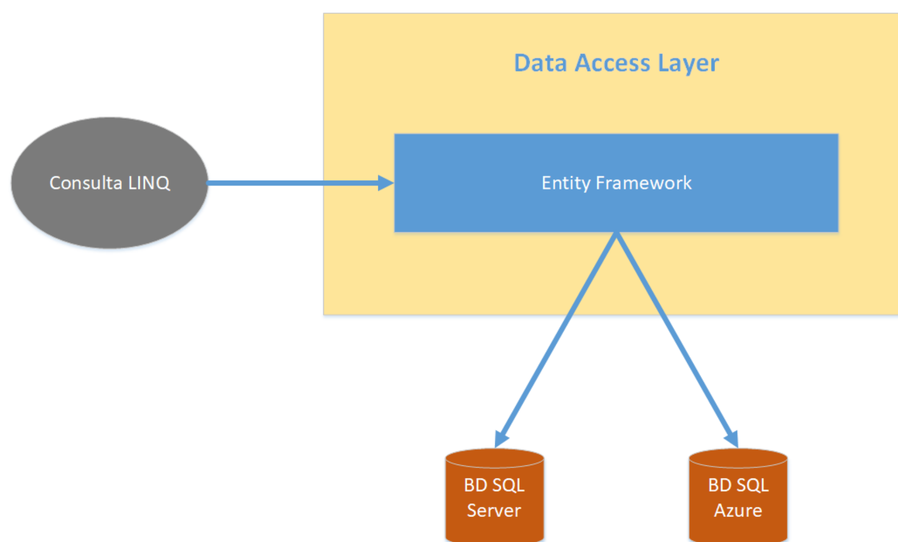


Figura 26 - Consulta efetuada através da Entity Framework

4.3.4 jQuery e KendoUI

O jQuery é uma biblioteca feita na linguagem de *scripting* Javascript que é usada em praticamente todos os *websites* dos dias de hoje. Segundo o *website* oficial (jQuery, n.d.)

> *"jQuery UI is a curated set of user interface interactions, effects, widgets, and themes built on top of the jQuery JavaScript Library. Whether you're building highly interactive web applications or you just need to add a date picker to a form control, jQuery UI is the perfect choice."*

O jQuery pode funcionar de forma independente, no entanto o Kendo UI necessita do jQuery para funcionar. Na Figura 27 podemos ver o código de uma *kendoWindow*, que é uma *popup* comum do KendoUI que utiliza alguns elementos de jQuery como é o caso dos *selectors*.

```
<script>
    $(document).ready(function () {
        $('#export-button').click(function (e) {
            e.preventDefault();
            var kWindow = $('#export-window');

            kWindow.empty();

            $("#export-window").kendoWindow({
                title: '@GlobalResources.MENU_ACCOUNTING',
                actions: ["Close"],
                modal: true,
                content: '/@ContextControllerName/ExportBilling', // + $( "#ExportAllInfo" ).is( ':checked' ),
                    draggable: true,
                    resizable: true,
                    width: '540px'
            });


            var win = kWindow.data("kendoWindow");
            win.center()
            win.open();
        });
    });
```

Figura 27 - Código de kendoWindow usando jQuery e KendoUI

Na Figura 28 podemos ver um gráfico gerado pelo KendoUI, através de um pedido AJAX efetuado de forma assíncrona ao servidor.

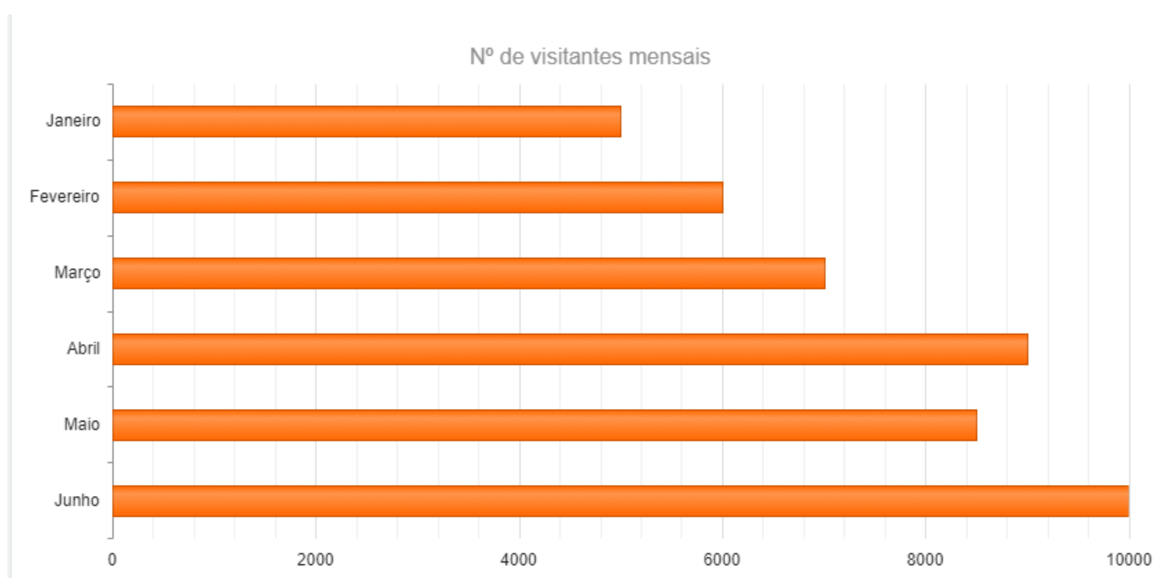


Figura 28 - Kendo UI – Indicador de visitantes mensais da página web

4.4 Recolha de dados

O processo de recolha de dados é muito importante para a análise de desempenho que será realizada em ambas as arquiteturas. Esta análise é fundamental para conhecermos as vantagens e desvantagens da implementação de uma arquitetura sobre a outra.

Em primeiro lugar foi instalado o serviço Application Insights da plataforma Azure. A documentação da Microsoft (Stephens, Crider, Wheeler, Willis, & Freemanwa, 2017) diz o seguinte:

> *"Application Insights is an extensible Application Performance Management (APM) service for web developers on multiple platforms. Use it to monitor your live web application. It will automatically detect performance anomalies. It includes powerful analytics tools to help you diagnose issues and to understand what users actually do with your app. It's designed to help you continuously improve performance and usability."*

Na prática, é instalado um *package* na aplicação, que é responsável pela monitorização dos dados e envia informação telemétrica para o portal do Azure. Essa informação pode depois ser consultada em *dashboards*.

4.5 Arquitetura monolítica original da aplicação

Nesta secção vai ser descrita a arquitetura original da aplicação, que é baseada na arquitetura monolítica com elementos da arquitetura em camadas. Após ser apresentado o esquema da arquitetura, vão ser mostrados os resultados efetuados em alguns parâmetros de desempenho, e de seguida iremos analisar as vantagens e desvantagens desta arquitetura no seio da aplicação e tentar obter algumas conclusões.

4.5.1 Esquema da Arquitetura

No inicio do ciclo de vida de uma aplicação de software uma das escolhas fundamentais é escolher o tipo de arquitetura que necessitamos para o software em causa, se necessitamos de uma aplicação compacta ou distribuída, com um *website* ou uma rede de *websites*, com uma base de dados ou várias. Estas decisões por vezes são difíceis de tomar porque as projeções feitas inicialmente podem ser diferentes do que iremos encontrar no futuro. Tendo isso em consideração, esta secção visa esclarecer todos os elementos que compõem a aplicação em estudo.

Como já foi referido na introdução do capítulo 4, a aplicação é composta por dois *websites*.

- ***Back-office* - BO** – Utilização privada e reservada a colaboradores da empresa
- ***Front-office* – FO** - Utilização pública através da internet destinado aos clientes e público em geral

Além dos dois *websites* esta aplicação está separada por projetos no Visual Studio, tal como podemos ver na Figura 29.

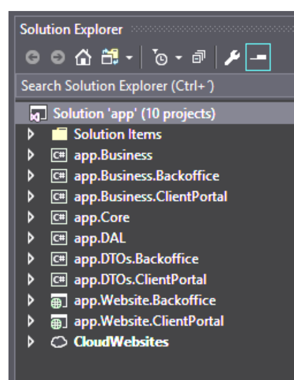


Figura 29 - Projetos da aplicação no Solution Explorer doVisual Studio 2017

O projeto app.Business contém uma camada de negócio partilhada por ambos os *websites*. Os projetos app.Business.Backoffice e app.Business.ClientPortal contém uma camada de negócio respetivamente, por *website*. O projeto app.Core contém bibliotecas e classes utilitárias, que, tal como o app.Business estão partilhadas por ambos os *websites*. A app.DAL (*Data Access Layer*) é a camada que contém os modelos e classes de dados usados pela Entity Framework e que permitem fazer a ligação com a base de dados. Ainda temos dois projetos app.DTOs.Backoffice e app.DTOs.ClientPortal com os DTO ou *Data Transfer Object*, que são classes que servem para transporte e mapeamento de dados entre os *websites* e as respetivas camadas de negócio. Por fim temos o CloudWebsites que não é mais que um projeto que serve para definir como é que o *deploy* da aplicação é feito na conta Microsoft Azure da empresa. A Figura 30 ilustra a arquitetura da aplicação.

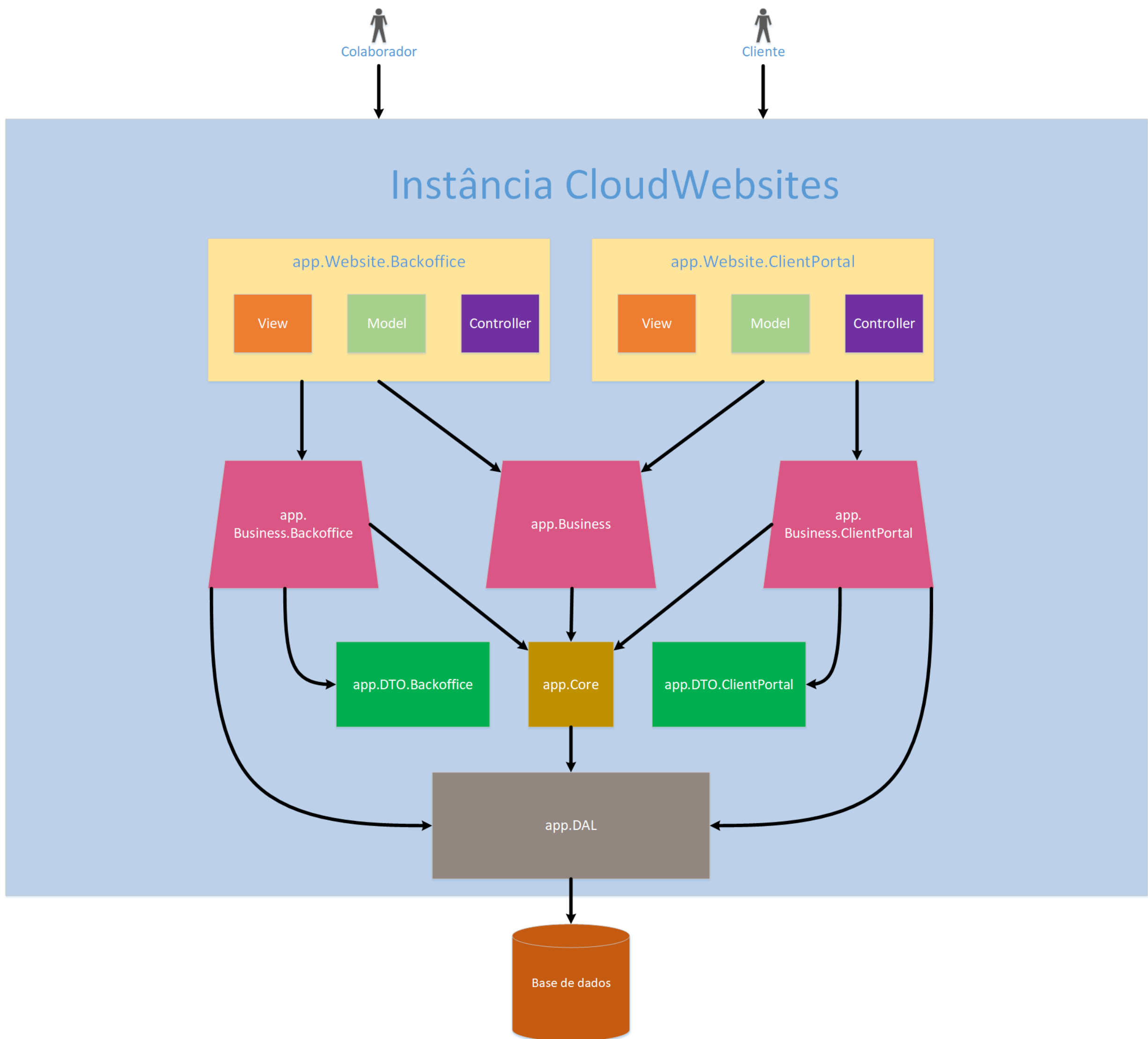


Figura 30 - Arquitetura da aplicação

Estamos perante uma arquitetura monolítica em camadas com uma única base de dados; as camadas são: camada de apresentação; camada de negócio; camada de acesso a dados; base de dados.

Esta aplicação, no entanto, tem uma nuance que é necessário realçar: a instância Azure contém tanto o *website* de FO como o de BO, como se pode verificar na Figura 30.

4.5.2 Desempenho

A arquitetura que atualmente está implementada no Microsoft Azure, tem apenas uma instância denominada app.Website.ClientPortal, com a app.Website.Backoffice contida "dentro" do *website* de FO através do uso de diretórios virtuais (*virtual directory*). A Tabela 5 mostra os valores de computações atribuídos a instâncias da serie A do Azure.

Tabela 5 - Recursos disponíveis para instâncias de classe A no Microsoft Azure – [Adaptado de (A. George, Lepow, Squillace, McKenna, & Kabrt, 2017)]

Tamanho	**Nº de Cores**	**Memória RAM (GiB)**	**Tamanho de Disco (GiB)**	**Nº de Placas de Rede / Largura de Banda de Rede**
ExtraSmall	1	0.768	20	1 / baixa
Small	1	1.75	225	1 / moderada
Medium	2	3.5	490	1 / moderada
Large	4	7	1000	2 / alta
ExtraLarge	8	14	2040	4 / alta
A5	2	14	490	1 / moderada
A6	4	28	1000	2 / alta
A7	8	56	2040	4 / alta

A instância app.Website.ClientPortal é uma instância A *Small* e está a suportar dois *websites*, o que a torna, por vezes, lenta para os utilizadores, principalmente em horas de pico, o que não é aceitável. Além deste problema, existe falta de redundância, o que é um problema grave, caso esta instância falhe, porque ambos os *websites* ficam offline. Tal situação não é aceitável nestas circunstâncias. Para compreender melhor o desempenho da aplicação, de seguida irão ser mostrados gráficos com valores recolhidos a partir da plataforma *cloud* Microsoft Azure, através dos Application Insights, referidos no ponto 4.4.

4.5.2.1 Utilização concorrencial

Um aspeto que é necessário ter em conta nestes casos é o nº de utilizadores na instância. O gráfico da Figura 31 mostra o número médio de utilizadores por hora durante o período compreendido entre os dias 8 e 12 de maio. Este intervalo de tempo é também igual em todas as restantes estatísticas desta secção.

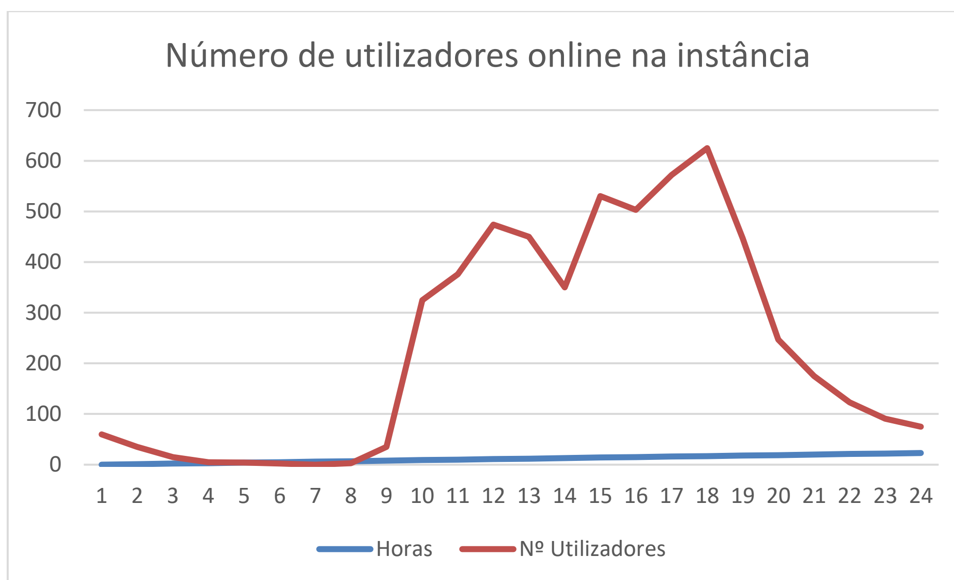


Figura 31 – Gráfico com nº de utilizadores na instância app.Website.ClientPortal

É de realçar que durante as horas de trabalho (período compreendido entre as 9h e as 18h) há, logicamente os maiores picos de utilização da aplicação. Isto deve-se tanto às visitas do site por parte dos clientes, como ao facto de os utilizadores de *Back-Office* estarem a trabalhar nesse período horário.

4.5.2.2 Utilização do CPU

Além do número de utilizadores em modo concorrencial, há outras métricas disponibilizadas pela plataforma Microsoft Azure que permitem mensurar a adequabilidade do comportamento da instância no serviço *cloud*, como as velocidades de leitura e escrita em disco, tráfego de rede (entrada e saída) ou a percentagem de utilização de CPU. Estes tipos de estatísticas são importantes para compreender se a instância é capaz de responder aos desafios colocados pelos utilizadores.

A Figura 32 representa a percentagem de utilização do CPU na instância app.Website.ClientPortal.

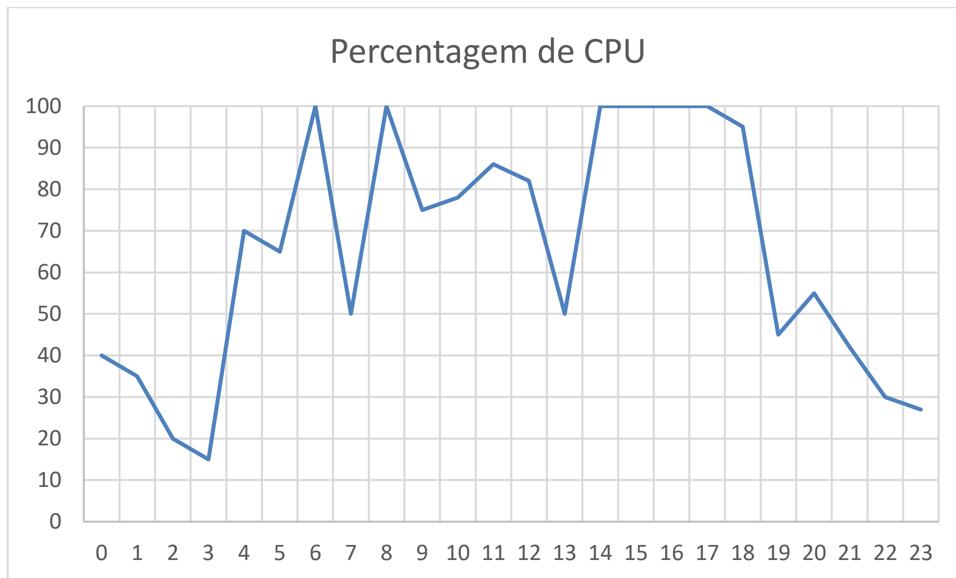


Figura 32 - Percentagem de utilização de CPU na instância app.Website.ClientPortal

É possível observar alguns picos de utilização de madrugada, isto é, devido a certos automatismos que a plataforma tem, que fazem processamento elevado. São agendados para executar em períodos horários em que há poucos utilizadores a utilizar a aplicação, de forma a garantir a celeridade dos mesmos para que o uso do CPU não afete a experiência de utilização dos *websites*.

4.5.2.3 Leitura e escrita em disco

Esta métrica não assume uma especial relevância no contexto desta aplicação porque toda a informação é guardada em base de dados ou serviços externos a esta instância, como é o caso dos ficheiros. No entanto a Figura 33 apresenta os dados recolhidos de velocidades de leitura e escrita no disco.

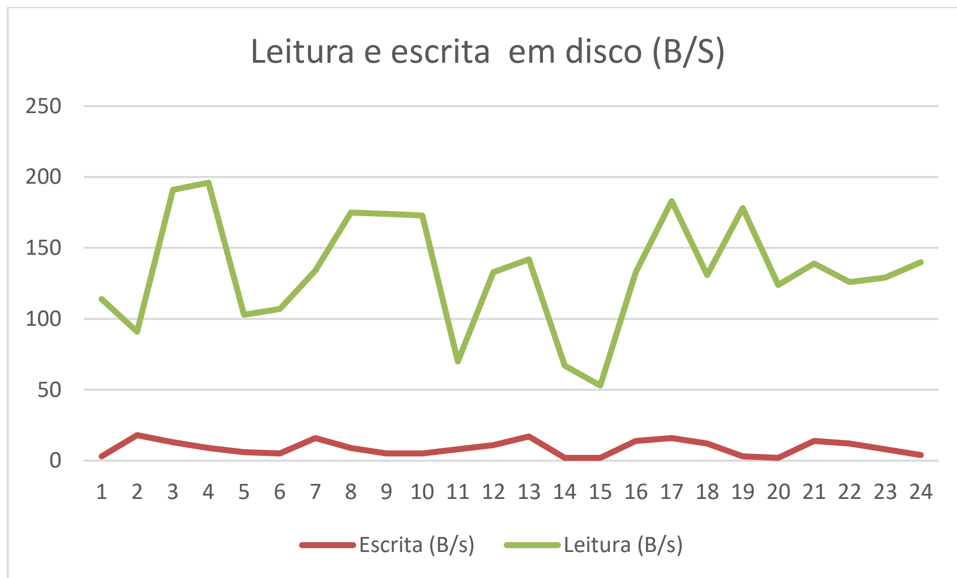


Figura 33 - Leitura e escrita em disco na instância app.Website.ClientPortal

Como podemos ver, estes valores são muito baixos e é seguro concluir que não seria pela leitura e escrita em disco que se encontra o problema de desempenho do sistema.

4.5.2.4 Tráfego de entrada e saída

O tráfego de entrada e saída é uma unidade de medida que permite avaliar se a instância tem a capacidade de resposta aos pedidos que lhe chegam através da internet.

A plataforma, no *Back-office,* tem muitos processos que envolvem importação, exportação e modificação de ficheiros. Com essa informação podemos observar alguns picos de atividade durante o horário de trabalho, tal como mostra a Figura 34.

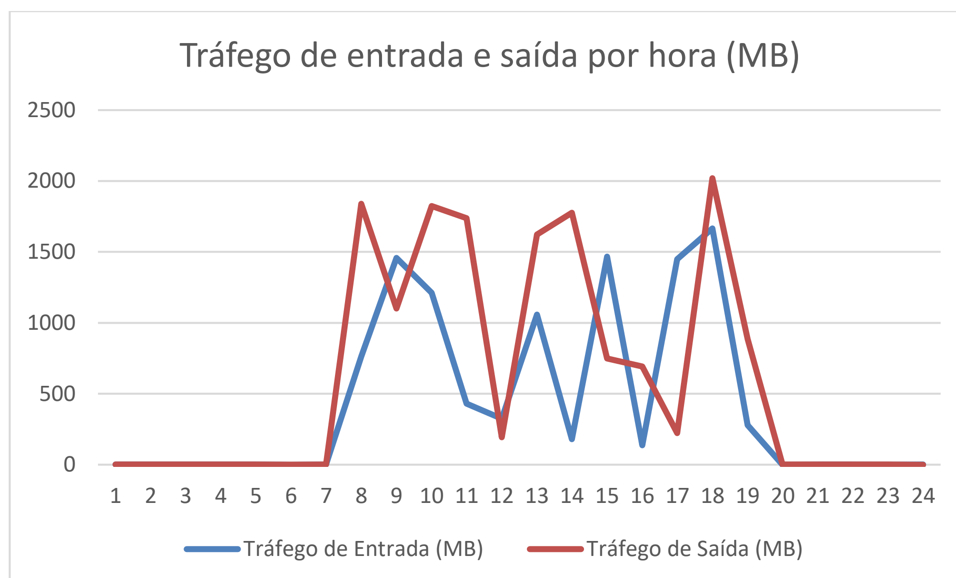


Figura 34 - Tráfego médio de entrada e saída na app.Website.ClientPortal

A Figura 34 mostra o valor em megabytes do tráfego de entrada e saída que ocorreu durante cada hora. É possível observar alguns picos de utilização de tráfego durante o horário de trabalho visto que é, geralmente, a altura do dia em que existem mais utilizadores a usar a aplicação. No entanto uma elevada percentagem destas operações é feita por serviços externos que comunicam com a aplicação e em que são enviados e recebidos ficheiros.

4.5.3 Vantagens

Uma arquitetura monolítica apresenta sempre algumas vantagens inerentes relativamente aos micro serviços, como a facilidade de *deploy*, o desenvolvimento mais rápido e a estrutura da arquitetura é disposta de uma forma mais natural do ponto de vista do programador. Nesta aplicação em concreto verificamos, no entanto, que com o uso de uma pasta virtual para encapsular o *Back-Office* na aplicação do *Front-Office*, acaba por ter vantagem no que diz respeito a custos, pois apenas vamos usar uma instância para acoplar dois *websites*. E nesse sentido além de apenas ser cobrado o custo-base da instância, também vão ser cobradas menos horas de computação. Isto pode ser interessante do ponto de vista económico, mas esse não é o foco deste caso de estudo.

4.5.4 Desvantagens

Além das claras vantagens do uso deste tipo de arquitetura monolítica, e como não há bela sem senão, há também algumas desvantagens. Em primeiro lugar não é possível escalar áreas específicas da aplicação; se for necessário escalar, a aplicação tem de o fazer como um todo. Isto é uma clara desvantagem face a arquiteturas distribuídas como é o caso dos micro serviços. Outra grande desvantagem é que qualquer alteração na aplicação implica o *deploy* de toda a aplicação, o que nem sempre é prático de fazer, visto que há processos que correm a determinadas horas e é imperativo que a aplicação esteja a funcionar.

Também é necessário realçar que a aplicação tem claras deficiências de desempenho em horas de utilização intensiva, como demonstraram os dados recolhidos através do Microsoft Azure.

Por fim e provavelmente a pior desvantagem desta arquitetura é a *codebase* [4] ser muito grande. Isto traz dificuldades no processo de desenvolvimento e acaba por levar ao problema de a médio-longo prazo, se tornar grande demais e que os desenvolvimentos numa área do código têm impactos inesperados em outras áreas do código.

4.5.5 Conclusões

Esta arquitetura é uma boa solução para aplicações de pequena e média dimensão. A aplicação em estudo apesar de se enquadrar nestes parâmetros, revela algumas deficiências em instâncias com poder de processamento mais baixo. Tal problema pode ser resolvido escalando horizontalmente ou verticalmente os recursos e recolhendo os dados novamente, para avaliar se resolveu o problema. Esta abordagem seria a mais correta de um ponto de vista dirigido aos resultados, no entanto o escopo deste estudo é avaliar as diferentes arquiteturas e se há diferenças de desempenho independentemente do hardware utilizado.

[4] O conceito de *codebase* é usado neste contexto para referir todo o código-fonte que é usado para compilar/executar a aplicação de software em causa.

4.6 Aplicação da Arquitetura de Micro serviços

Nesta secção vamos fazer um enquadramento de como surgiu a ideia de aplicar uma arquitetura de micro serviços da aplicação, apresentar a sua estrutura e esquema bem como falar um pouco em detalhe sobre a REST API que lhe dá suporte. Também serão feitos testes de desempenho, que visam fazer uma comparação da mesma aplicação, particionada em micro serviços. Serão apresentadas, no contexto desta aplicação, as vantagens e desvantagens da aplicação deste modelo de arquitetura e por fim tirar as devidas conclusões.

4.6.1 Enquadramento

A ideia de aplicar uma arquitetura de micro serviços surgiu após algumas queixas do cliente relativamente à performance da aplicação em determinadas horas do dia em que existiam mais utilizadores online. Tal como referido no ponto 4.4, foi instalada a ferramenta Application Insights, em que foi possível observar que alguns valores não estavam de acordo com o esperado. Foi então que o mestrando sugeriu a aplicação de uma arquitetura de micro serviços. No entanto esta implementação foi feita em duas fases: na primeira fase a aplicação foi dividida pelos *websites*, ficando o *website* de BO implementado numa instância à parte, e o *website* de FO mantendo-se na instância original, posteriormente.

Com esta primeira fase houve um aumento significativo da performance, visto que os processos ditos pesados, começaram a ser executados na instância que estivesse menos ocupada no momento. No entanto, o principal problema da escalabilidade mantinha-se, não sendo possível escalar a aplicação em granularidade suficiente para alcançar os objetivos; o máximo que se podia escalar era ao nível do *website*.

Numa segunda fase, a aplicação foi separada em vários micro serviços que comunicavam através de uma REST API, e aqui sim, já pode ser considerada uma arquitetura em micro serviços. Na secção seguinte será explicada em pormenor o esquema da arquitetura implementado.

4.6.2 Esquema da arquitetura

A estrutura de uma arquitetura é o ponto crucial para a sua definição. Apesar de terem sido referidas duas fases, a primeira foi apenas implementada em ambiente de testes, e não chegou a entrar em produção. Este caso de estudo será baseado apenas na arquitetura presente na segunda fase, que está atualmente em todos os ambientes de desenvolvimento.

A arquitetura em micro serviços foi implementada de acordo com as áreas que a aplicação anterior tinha, mas de uma forma em que seja possível fazer o *scale-up* ou *scale-down* de instâncias á medida que são mais ou menos utilizadas.

A Figura 35 representa os projetos criados no Visual Studio para dar suporte à nova arquitetura.

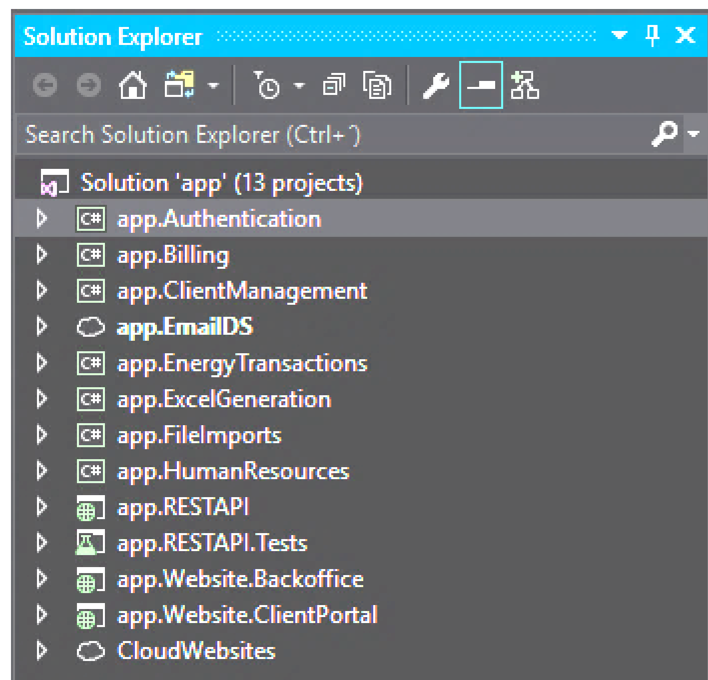


Figura 35 - Projetos no Microsoft Visual Studio com arquitetura de micro serviços

Como a aplicação teve de ser reestruturada, o Visual Studio força a apresentação dos projetos por ordem alfabética, algo que não acontecia anteriormente.

Esta arquitetura baseia-se na comunicação entre os *websites* e a REST API, que por sua vez faz pedidos aos micro serviços. Estes encaminham as operações que envolvem obtenção de dados

para a BD, enviam os mesmos para a REST API e depois esta devolve a informação aos *websites*. O esquema da aplicação é ilustrado na Figura 36.

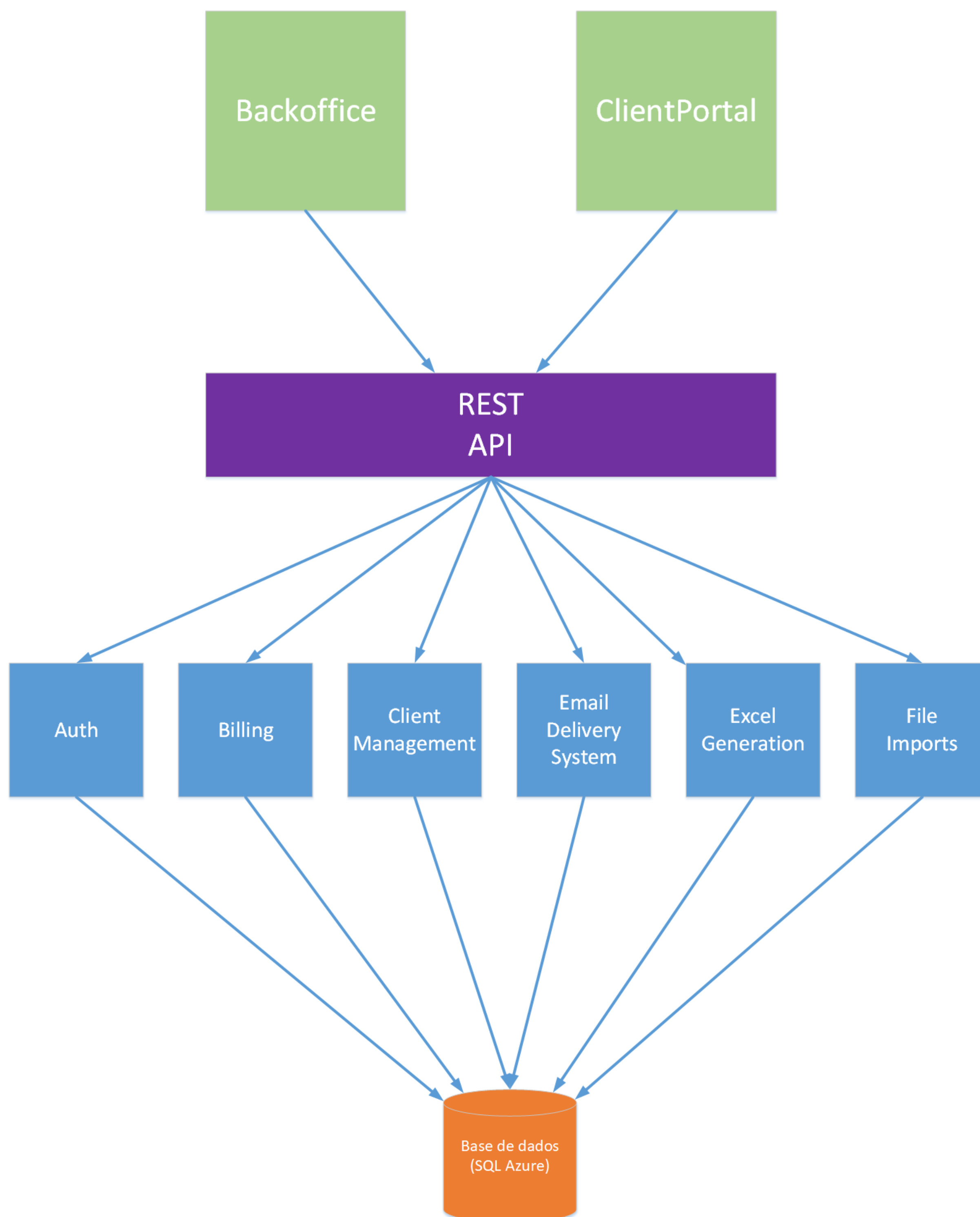


Figura 36 - Esquema da arquitetura de micro serviços.

4.6.3 Alocação de recursos por serviço

Estes micro serviços, de momento estão *deployed* no Microsoft Azure em instâncias de tamanhos diferentes de acordo com as necessidades do momento. Além disto todas estão configuradas com o mecanismo de *auto-scaling*, ou seja, assim que os recursos atingem a capacidade máxima, a instância automaticamente aumenta os recursos computacionais da mesma de forma para garantir uma boa resposta. A Tabela 6 ilustra os recursos alocados das instâncias (que já foram apresentados na Tabela 5), e o *scaling* máximo configurado na plataforma Azure relativamente a cada uma.

Tabela 6 - Serviços, requisitos de negócio e recursos alocados ás instâncias

Micro serviço /*Website*	Requisito de Negócio	Recursos da instância (padrão)	Recursos da instância (min)	Recursos da instância (max)
Website.Backoffice	Interface do Backoffice Interno	Small	ExtraSmall	Medium
Website.ClientPortal	Website do portal de cliente	Small	ExtraSmall	Medium
Billing	Faturação	ExtraSmall	ExtraSmall	Large
ClientManagement	Gestão de clientes e contratos	Small	ExtraSmall	Large
EmailDS	Envio de emails	Small	ExtraSmall	Small
EnergyTransactions	Compra e venda de energia	Medium	ExtraSmall	Large
ExcelGeneration	Exportação de informação em CSV / XLS	ExtraSmall	ExtraSmall	ExtraSmall
FileImports	Importação de ficheiros	Small	ExtraSmall	Large
HumanResources	Recursos Humanos	Small	ExtraSmall	Small
RESTAPI	API REST	Medium	Medium	Large

Mas na prática, o que acontece? Vamos supor que durante a semana ocorrem duas emissões de faturação, ou seja, o micro serviço de faturação no restante tempo, apenas tem que fazer pequenas operações de leitura e escrita na base de dados, sem efetuar processamento pesado. A instância de faturação durante praticamente toda a semana fica numa instância Extra Small, poupando recursos computacionais e tendo um custo mais baixo. A partir do momento em que

é iniciada uma emissão de faturação (processo pesado), a instância chega rapidamente ao seu limite e escala para Small. Caso atinja novamente o limite, escala para Medium, e por fim para Large (se necessário). A partir do momento que o processo acaba, a instância vai automaticamente fazer o *scale-down* até voltar a atingir o ExtraSmall. Ou seja, durante o tempo em que a necessidade do processamento era baixa, a instância manteve-se no patamar mais baixo do MS Azure, com menos recursos alocados. A partir do momento em que foi necessário mais "poder de fogo" a instância automaticamente foi subindo de escalão até atingir o limite superior definido (Large).

4.6.4 Desempenho

Ao longo desta secção será feita uma análise dos resultados obtidos através do serviço *Application Insights* no período compreendido entre 10 e 13 de setembro de 2017. Os valores apresentados são resultados de uma média por dia, ao longo dos três dias.

4.6.4.1 Utilização concorrencial

Este critério apenas pode ser quantificado nos *websites* e não nos serviços, na medida em que os utilizadores apenas se podem autenticar ou no *website* de B*ack-Office* ou no *website* de portal de cliente. Como podemos ver na Figura 37, o número de utilizadores de BO mantém-se constante ao longo de todo o dia (à parte das horas de almoço) visto que é operado por um conjunto definido de utilizadores. O portal de cliente, por outro lado, obtém alguns picos de utilização também durante as horas do dia. Fora do horário de trabalho, não há utilizadores de BO a trabalhar, no entanto os clientes podem continuar a aceder ao portal, embora se note uma quebra de utilizadores a partir das 18h.

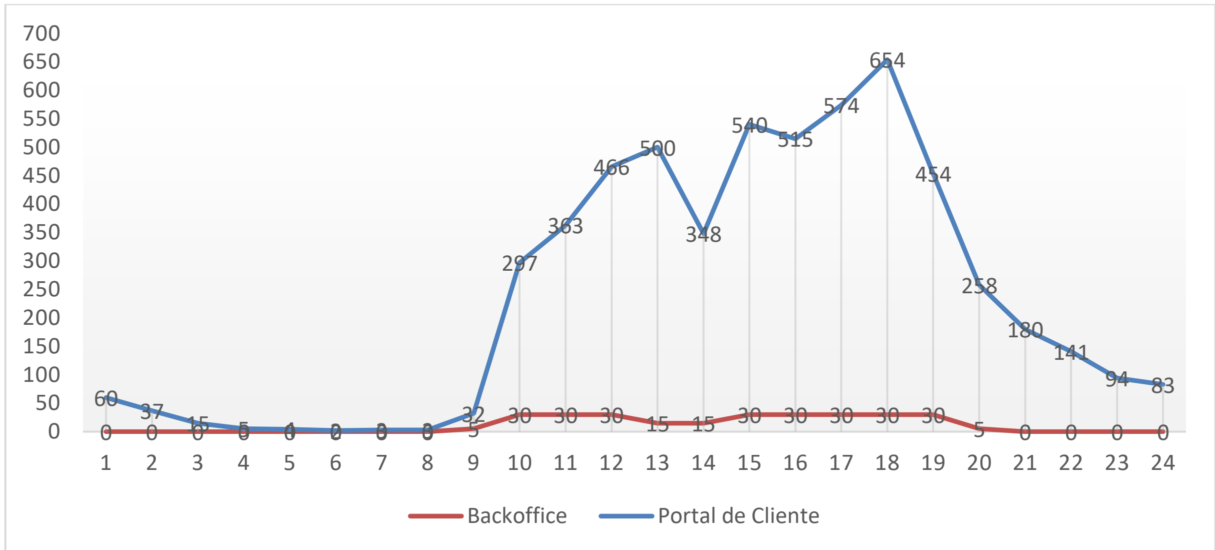


Figura 37 - Número de utilizadores por *website* numa arquitetura de micro serviços

Embora os números de utilizadores totais sejam semelhantes aos observados na arquitetura monolítica, é de notar que houve um substancial aumento da qualidade de serviço reportado pelos utilizadores de BO, visto que os fluxos de trabalho mais usados por estes, nomeadamente os recursos humanos e gestão de clientes e contratos, não têm diminuição de performance durante alturas em que outrora existia, por exemplo na exportação de faturação. Isso deve-se ao facto de a chamada a esse serviço estar toda feita num serviço à parte.

4.6.4.2 Utilização do CPU

A utilização do CPU é medida nas instâncias. Tendo em consideração que foram realizadas operações de *auto-scaling*, é natural que os valores apresentados não sejam possíveis de comparar intrinsecamente com os valores da arquitetura monolítica, porque a arquitetura monolítica não escalava automaticamente. Ou seja, estes valores acabam por ter pouco significado prático num exercício de pura comparação, número a número. Como um gráfico com tantas instâncias se ia tornar confuso, a Tabela 7 apresenta os valores de percentagem do CPU capturados na média dos três dias.

Tabela 7 - Tabela da utilização de CPU nas diferentes instâncias em Azure

Horas	Auth	Billing	ClientManagement	EmailDS	EnergyTransactions	ExcelGeneration	FileImports	HumanResources	RESTAPI	BO	FO
0	17	17	43	78	74	9	46	15	71	46	44
1	15	10	75	42	99	5	92	15	78	53	89
2	11	13	59	73	83	12	93	12	77	54	62
3	19	20	75	57	78	15	48	15	87	51	83
4	12	14	33	67	90	7	94	14	82	51	34
5	11	12	61	61	81	10	46	7	84	46	71
6	11	11	42	61	89	13	75	9	82	49	43
7	11	22	51	43	83	12	63	8	94	48	54
8	17	95	57	64	82	75	52	93	82	77	66
9	50	21	70	71	98	41	72	80	96	74	72
10	30	11	52	70	98	49	59	91	70	66	56
11	16	17	26	85	90	82	50	58	85	63	28
12	54	11	58	58	82	73	82	76	87	72	62
13	37	11	49	49	100	62	65	55	76	63	57
14	38	20	36	87	82	63	81	59	98	70	42
15	55	20	71	73	61	13	94	67	97	68	78
16	56	10	45	79	57	9	79	78	92	63	53
17	56	14	41	93	64	13	40	82	86	61	42
18	44	22	60	60	91	5	57	60	91	61	71
19	36	21	37	79	58	7	53	16	94	50	43
20	20	15	43	47	70	5	98	35	98	53	50
21	50	23	55	48	79	6	40	18	99	52	61
22	18	12	27	61	62	14	99	18	85	49	31
23	55	18	66	41	80	5	48	19	89	52	77

4.6.4.3 Leitura e escrita em disco

Este indicador só faz sentido ser apresentado para as instâncias de FileImports e ExcelGeneration, visto que para todas as outras, acaba por não ser significativo pois a informação é guardada em memória, transportada pela rede e guardada em BD.

O serviço ExcelGeneration é responsável por:

1. Receber o pedido da REST API
2. Ir buscar a informação necessária à base de dados
3. Gerar o ficheiro CSV ou XLSX localmente
4. Fazer o carregamento do ficheiro para o sistema de armazenamento *blob storage*
5. Retornar o URI para a REST API

Na Figura 38 podemos observar os valores em megabytes (MB) escritos e lidos em cada hora.

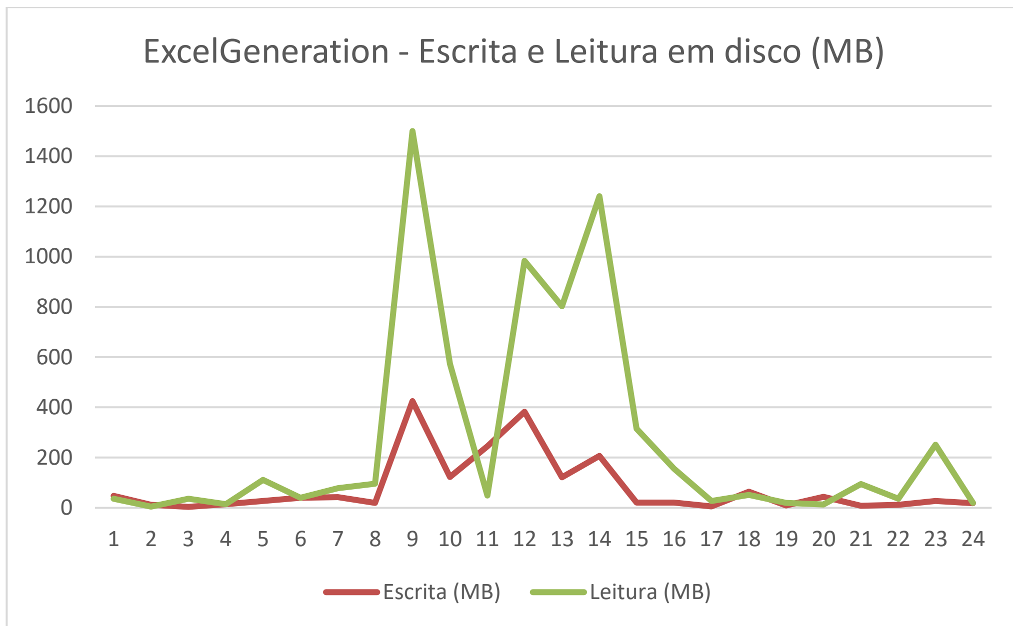


Figura 38 - Serviço ExcelGeneration - Leitura e escrita em disco

O serviço FileImports é responsável por toda a escrita de ficheiros importados e disponibilização de informação importada pelos mesmos. Na Figura 39 podemos ver os valores relativos à leitura e escrita em disco no serviço FileImports.

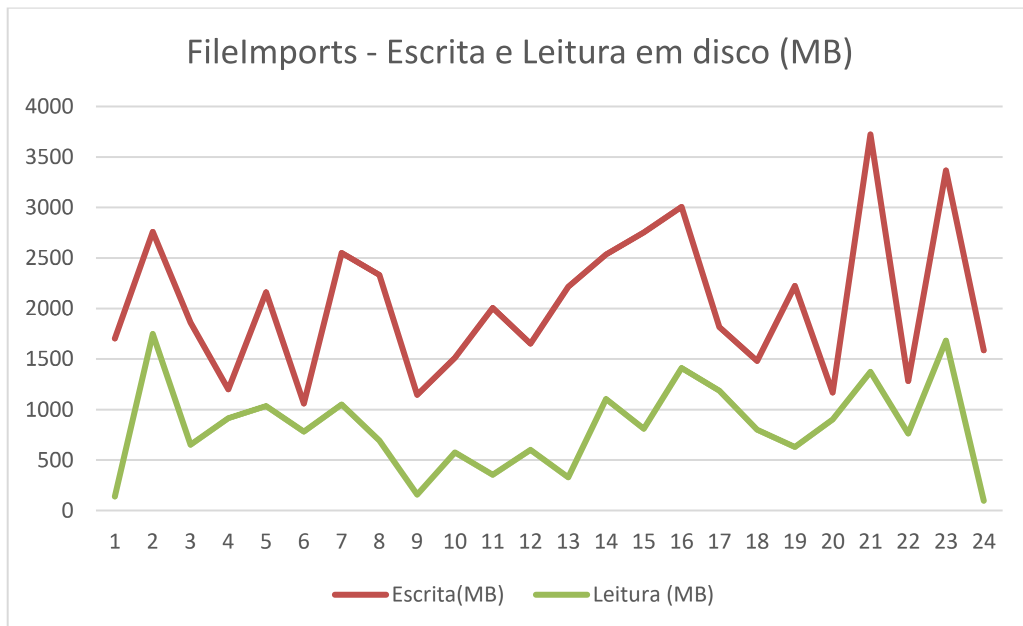


Figura 39 - Serviço FileImports - Leitura e escrita em disco

4.6.4.4 Tráfego de entrada e saída

Com a alteração da arquitetura para micro serviços, o maior impacto em termos de performance é no tráfego de entrada e saída, visto que numa arquitetura monolítica a comunicação entre módulos/áreas de negócio é feita inteiramente na mesma máquina, e numa arquitetura de micro serviços, geralmente, é feita através da intranet ou da internet com o auxílio do protocolo HTTP.

Como é lógico, a REST API será a instância com maior entrada e saída de dados, porque toda a comunicação passa por ela. Na Figura 40 podemos observar o tráfego de entrada e saída em Megabytes, ao longo de uma hora através da REST API.

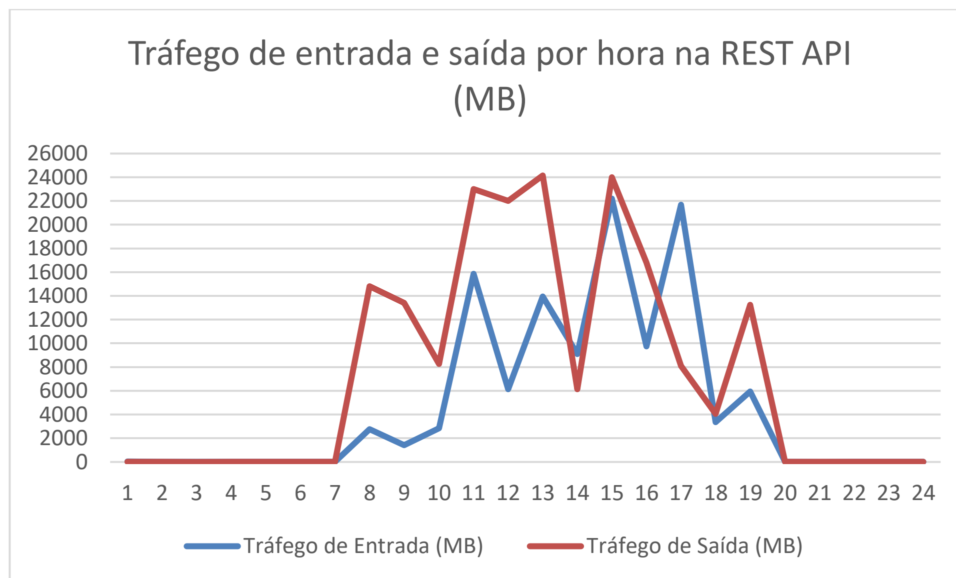


Figura 40 - Tráfego da entrada e saída por hora na REST API

Como mostra a Figura 40, o tráfego gerado em rede é substancialmente superior em arquiteturas de micro serviços. Isto é uma consequência natural de arquiteturas distribuídas.

4.6.5 Vantagens e Desvantagens

Com qualquer tipo de arquitetura há sempre vantagens e desvantagens associadas. A arquitetura de micro serviços tenta neste prisma desmistificar a complexidade da aplicação, ou seja, separa a aplicação em serviços categorizados pela área do negócio. Isto acaba por ser uma grande vantagem do ponto de vista do programador, porque este olha para o serviço e identifica

rapidamente as áreas do negócio. O mesmo não é verdade para a arquitetura monolítica, em que por vezes é necessário algum trabalho de "pesca".

A principal vantagem é a capacidade das instâncias (que contém micro serviços) poderem ser escaladas de forma individual. Isto torna-se ainda mais imperativo em processos pesados visto que como se encontram em micro serviços separados, não causam qualquer impacto aos utilizadores dos *websites*.

Uma desvantagem é o facto de a comunicação entre diferentes serviços ter de ser feita através de uma API, ao oposto de ser feita internamente pela aplicação. Sendo que a API fica exposta á internet, podem haver quebras de conectividade entre serviços.

Há também desvantagens relacionadas com os processos de *Deployment.* Fica mais complexo, visto que é necessário fazer *deploys* individuais de cada serviço, no entanto isto também pode ser considerado uma vantagem, porque não há necessidade de fazer o *deploy* de uma aplicação inteira quando só queremos mudar um parâmetro de inteiro para longo, por exemplo.

4.6.6 Conclusões

As conclusões desta mudança são de facto muito positivas, por vários fatores: 1). É possível fazer a escalabilidade com uma granularidade muito maior; 2) Ao partir a aplicação monolítica em micro serviços, conseguimos fazer *deploys* individualizados com maior facilidade; 3) Não há um ponto único de falha. A grande desvantagem é quando o serviço de autenticação falha, visto que, caso não existam utilizadores com a sessão iniciada, a aplicação fica reduzida aos *workflows* automáticos que a plataforma tem configurada.

Tendo em conta que o *website* de *BackOffice* é o principal responsável pelos valores altos de entrada e saída de tráfego da aplicação, uma conclusão que se pode tirar é que fazendo uma separação desta instância em vários serviços, é possível assim aferir que a possibilidade de particionar a arquitetura em micro serviços e fazer o seu dimensionamento em separado, trouxe, além de poupanças em termos monetários, aumentos significativos de performance.

Comparar as duas arquiteturas olhando única e exclusivamente para números, é um pouco injusto, visto que há vários fatores que podem desvirtuar este tipo de comparações. Em primeiro

lugar. foram comparados em datas diferentes, logo numa data pode ter tido mais utilizadores que na outra. Em segundo lugar há processos automáticos que não correm todos os dias, havendo a possibilidade de estes terem ocorrido em um intervalo de tempo, e não terem ocorrido noutro intervalo de tempo. Mas no final, o mais importante é que a experiência de utilização, tanto dos utilizadores dos *websites* como da equipa de programação, é bastante melhor. Este facto não se deve, única e exclusivamente, à arquitetura utilizada, mas a implementação do *auto-scaling* também ajuda.

O feedback obtido em torno desta arquitetura foi bastante positivo. As emissões de faturação ficaram substancialmente mais rápidas, outros processos pesados também e acima disso tudo, não provocaram lentidão nos *websites* como ocorria antes. No que toca á equipa de desenvolvimento, notam-se melhorias claras da eficiência. Como a equipa é pequena, cada elemento pode focar-se num conjunto de dois ou três micro serviços, e é agora também utilizada uma convenção de nomes dos métodos na REST API. De um modo geral irá acabar por ter benefícios a longo prazo também, pelo facto desta aplicação estar em constante mutação e com novas funcionalidades em cada versão, sendo que estas novas funcionalidades podem ser incorporadas em novos micro serviços, ou adicionadas a micro serviços existentes, dependendo do contexto

5 Conclusões e trabalho futuro

A evolução das organizações, nos dias de hoje, é muito diferente do que se verificava no passado. Há alguns anos era possível definir requisitos e manter aplicações praticamente inalteradas durante longos períodos de tempo. Um conjunto de fatores levou a que as aplicações fossem, na sua maioria, desenhadas sob um modelo monolítico.

O modelo monolítico continua a ser amplamente utilizado na maioria das organizações, mas à medida que as organizações vão alterando os seus princípios fundamentais no que toca aos requisitos do que o software deve representar, a engenharia de software tem dado resposta aos requisitos das organizações modernas.

Hoje em dia cada vez mais organizações estão a desenvolver aplicações sob uma base evolutiva, na nuvem e com a integração contínua entre a equipa de desenvolvimento e as necessidades específicas das organizações. Este novo paradigma originou mudanças importantes no ciclo de desenvolvimento de software, levando ao aparecimento das metodologias ágeis e do acompanhamento contínuo do software pela equipa de desenvolvimento durante o ciclo de vida do software. A Arquitetura de software não ficou para trás, apresentando soluções que visam dar resposta a este paradigma, tentando fazer um desacoplamento dos requisitos de negócio em unidades independentes de software.

Em resumo, a arquitetura de software tem evoluído à medida que os requisitos de negócio mudam, razão essa que leva muitas organizações a fazerem a transição de arquiteturas monolíticas em arquiteturas distribuídas, como é o caso da arquitetura orientada a serviços, ou mais recentemente dos micro serviços.

No contexto deste caso de estudo foi possível observar claras melhorias no desempenho da aplicação “partindo” o bloco de código monolítico em micro serviços, mas esta análise é claramente subjetiva ao tipo de aplicação e aos níveis de carga a que a mesma é sujeita. Não sendo possível classificar uma arquitetura melhor que outra em qualquer situação, é possível

apontar alguns critérios para que um arquiteto de software escolha uma arquitetura em detrimento de outra.

Em primeiro lugar há que ter em conta o contexto da aplicação, provavelmente não é a melhor ideia utilizar uma arquitetura monolítica para um serviço de *streaming* de vídeo como o YouTube, Twitch ou Netflix, visto que estes serviços requerem uma grande flexibilidade de alocação de recursos em função do número de utilizadores e da carga a que estão sujeitos. No entanto também não se justifica a aplicação de uma arquitetura de micro serviços para *websites* de empresas pequenas, ou até mesmo aplicações de consumo interno (da organização) que estão sujeitas a baixos níveis de carga.

Para responder à pergunta "Qual a melhor arquitetura em cada situação", pode referir-se um famoso tweet de um conhecido programador do Facebook, Kent Beck *"any decent answer to an interesting question begins, "it depends..."".* Como já explicado anteriormente, pode resumir-se, sucintamente, que depende da aplicação, mas de um modo geral, as aplicações monolíticas devem ser usadas em ambientes em que há pouca complexidade da aplicação, temos um número médio ou baixo de utilizadores e não é necessária uma escalabilidade flexível. A arquitetura de micro serviços deve ser utilizada em aplicações que denotam picos de utilização acentuados, níveis de carga de utilização imprevisíveis necessitando de ter uma escalabilidade muito flexível. No entanto, a principal conclusão que podemos tirar é que o arquiteto de software tem de analisar muito bem os requisitos e escolher a que melhor sirva os interesses da organização.

5.1.1 Respostas aos objetivos estabelecidos

O mestrando tem a convicção que esta dissertação contribuiu imenso para aprofundar o conhecimento deste tema tendo em consideração que foi feito um trabalho de escolha da metodologia, uma pesquisa mais aprofundada sobre as definições de arquitetura de software bem como as diferentes arquiteturas utilizadas. Além disso, o caso de estudo permitiu ao mestrando lidar com duas arquiteturas muito diferentes bem como analisar as vantagens e desvantagens da utilização de cada uma.

Foi também possível aferir qual a melhor arquitetura no parâmetro da escalabilidade, bem como na facilidade de manutenção de código, complexidade, separação de conceitos, fiabilidade e nível de abstração. No entanto não foi possível encontrar evidências quanto á segurança das arquiteturas.

Em suma, o principal objetivo deste trabalho era compreender em que situações é vantajoso usar uma arquitetura monolítica e em que situações é mais vantajoso usar uma arquitetura de micro serviços e esse objetivo foi alcançado.

Um dos objetivos a que o mestrando se propôs, mas não foi alcançado foi a realização de entrevistas a especialistas. Estas não foram realizadas devido à dificuldade de reunir arquitetos de software com disponibilidade para este estudo, mas com certeza que iriam apresentar uma mais valia do ponto de vista científico e até para complementar os resultados obtidos no caso de estudo.

5.1.2 Trabalho futuro

Embora a maioria dos objetivos tenham sido atingidos, seria interessante divulgar o resultado deste estudo numa conferência com a temática da arquitetura de software ou até mesmo publicar uma versão mais reduzida desta dissertação num jornal periódico de forma a poder contribuir para o conhecimento científico relativo ao tema visto que há uma certa dificuldade em encontrar artigos científicos e informação com rigor na internet. O mestrando tem a convicção que este trabalho poderá ajudar outros a obter respostas no que a esta temática diz respeito.

Um outro trabalho futuro seria poder aplicar o conceito de micro serviços a bases de dados, partindo de uma base de dados “monolítica” e particioná-la em unidades lógicas que façam sentido estar juntas. Por exemplo: criar uma base de dados para *logs*, em tecnologias NoSQL, visto que trazem benefícios para esse tipo de bases de dados.

6 Referências

Barry, D. (2017). Web Services Explained. Retrieved October 11, 2017, from https://www.service-architecture.com/articles/web-services/web_services_explained.html

Barry, D. K. (2003). *Web Services and Service-Oriented Architecture: The Savvy Manager's Guide*. (E. Science, Ed.) (2nd ed.). San Francisco: Morgan Kaufmann. Retrieved from https://www.e4net.net/tech/download/SOA/Morgan_Kaufmann-Web_Services_And_Service_Oriented_Architecture.pdf

Bass, L., Clements, P., & Kazman, R. (2012). Software Architecture in Practice. *Vasa*, *2nd*, 1–426. https://doi.org/10.1024/0301-1526.32.1.54

Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., & Orchard, D. (2004). Web Services Architecture, W3C Working Group Note 11 February 2004. *World Wide Web Consortium, Article Available from: Http://www. w3. org/TR/ws-Arch*, 13.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., & Yergeau, F. (2008). Extensible Markup Language (XML) 1.0. *W3C Recommendation 26 November 2008*, (5th). Retrieved from https://www.w3.org/TR/xml/

Bulgarian, A., Czech, C., English, D., German, F., Hebrew, G., Indonesian, H., … Turkish, S. (2009). Introducing JSON. *Security*, 1–6. Retrieved from http://www.json.org/

Bushmann, F., Meunier, R., & Rohnert, H. (1996). *Pattern-oriented software architecture: A system of patterns*. *John Wiley&Sons* (Vol. 1). Wiley. https://doi.org/10.1192/bjp.108.452.101

Clements, P. C., & Northrop, L. M. (1996). Software Architecture: An Executive Overview. Retrieved from https://resources.sei.cmu.edu/asset_files/TechnicalReport/1996_005_001_16457.pdf

Dijkstra, E. W. (1968). The Structure of the “ THE ” -Multiprogramming System. *Communications of the ACM*, *11*(5), 341–346. https://doi.org/10.1145/363095.363143

ECMA-262. (1999). Standard ECMA-262 ECMAScript Language Specification. Retrieved from https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262, 3rd edition, December 1999.pdf

Endrei, M., Ang, J., Arsanjani, A., Chua, S., Comte, P., Krogdahl, P., … Newling, T. (2004). Patterns: Service-Oriented Architecture and Web Services. *Contract*, *1*, 17–42. https://doi.org/10.1109/SOSE.2005.5

Fetterman, D. M. (1989). *Ethnography: Step-by-Step*. *Comparative and General Pharmacology* (Vol. 17). https://doi.org/10.1002/1521-3773(20010316)40:6<9823::AID-ANIE9823>3.3.CO;2-C

Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. Retrieved from https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. *I Can*.

Fowler, M., & Lewis, J. (2014). Microservice Architecture. *Martinfowler.Com*. https://doi.org/10.1007/978-1-4842-1275-2_3

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2002). Design Patterns – Elements of Reusable Object-Oriented Software. *A New Perspective on Object-Oriented Design*, 334. https://doi.org/10.1093/carcin/bgs084

Garlan, D., & Perry, D. (1995). Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, *21*(4), 269–274. Retrieved from http://dl.acm.org/citation.cfm?id=205314

George, A. D., Squillace, R., Nottingham, C., & Pratt, T. (2017). How to configure auto scaling for a Cloud Service in the classic portal. Retrieved October 8, 2017, from https://docs.microsoft.com/en-us/azure/cloud-services/cloud-services-how-to-scale

George, A., Lepow, D., Squillace, R., McKenna, S., & Kabrt, L. (2017). Virtual machine sizes for Azure Cloud services | Microsoft Docs. Retrieved October 15, 2017, from https://docs.microsoft.com/en-us/azure/cloud-services/cloud-services-sizes-specs

Gil, A. C. (2008). *Métodos e técnicas de pesquisa social*. *Journal Of The American Medical Association* (Vol. 264). https://doi.org/10.1590/S1517-97022003000100005

Jorgensen, D. L. (1989). Participant Observation: A Methodology for Human Studies. In *Participant Observation* (pp. 1–23). https://doi.org/10.4324/9780203846452

jQuery. (n.d.). jQuery User Interface. Retrieved October 19, 2017, from https://jqueryui.com/

Kidder, L. H., & Judd, C. M. (1981). *Research methods in social relations*. Holt, Rinehart, and Winston. Retrieved from https://books.google.pt/books/about/Research_Methods_in_Social_Relations.html?id=bOIkAQAAIAAJ&redir_esc=y

Klaus, H., Rosemann, M., & Gable, G. G. (2000). What is ERP? *Information Systems Frontiers*, *2*(2), 141–162. https://doi.org/10.1023/A:1026543906354

Koirala, S. (2009). How to improve your LINQ query performance by 5 X times? *Code Project*. Retrieved from https://www.codeproject.com/Articles/38174/How-to-improve-your-LINQ-query-performance-by-5-X

Kotkondawar, R. R., Khaire, P. A., Akewar, M. C., & Patil, Y. N. (2014). A Study of Effective Load Balancing Approaches in Cloud Computing. *International Journal of Computer Applications*, *87*(8), 975–8887. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.429.2935&rep=rep1&type=pdf

Lenk, A., Klems, M., Nimis, J., Tai, S., & Sandholm, T. (2009). What's inside the Cloud? An architectural map of the Cloud landscape. In *2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing* (pp. 23–31). IEEE. https://doi.org/10.1109/CLOUD.2009.5071529

Luck, D., & Lancaster, G. (2003). E-CRM: customer relationship marketing in the hotel industry. *Managerial Auditing Journal*, *18*(3), 213–231. https://doi.org/10.1108/02686900310469961

Luke, E. A. (1993). Defining and measuring scalability. *Proceedings of Scalable Parallel Libraries Conference*, 183–186. https://doi.org/10.1109/SPLC.1993.365568

Mejia, A. (2016). How to scale a Nodejs app based on number of users | Adrian Mejia Blog. Retrieved October 8, 2017, from http://adrianmejia.com/blog/2016/03/23/how-to-scale-a-nodejs-app-based-on-number-of-users/

Microsoft Corporation. (2017). IntelliSense. Retrieved October 18, 2017, from https://code.visualstudio.com/docs/editor/intellisense

Myers, T., & Pratt, T. (2017). Use the Azure storage emulator for development and testing | Microsoft Docs. Retrieved October 19, 2017, from https://docs.microsoft.com/en-us/azure/storage/common/storage-use-emulator

Newman, S. (2015). Microservices. In *Building Microservices* (pp. 1–11).

Nord, R. L., Hofmeister, C., & Soni, D. (1999). Preparing for Change in the Architec- ture Design of Large Software Systems, (February), 1–7. Retrieved from http://users.ece.utexas.edu/~perry/prof/wicsa1/final/nord.pdf

Parmar, K. (2014). Monolithic vs MicroService Architecture. Retrieved January 14, 2017, from https://www.linkedin.com/pulse/20141128054428-13516803-monolithic-vs-microservice-architecture

Parnas, D. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, *15*(12), 1053–1058. https://doi.org/10.1145/361598.361623

Parnas, D. (1974). On a 'Buzzword': Hierarchical Structure. In *74, Proceedings IFIP Congress* (pp. 336–3390). North Holland Publishing Company.

Parnas, D. (1976). On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, *SE-2*(1), 1–9. https://doi.org/10.1109/TSE.1976.233797

Richards, M. (2015). *Software Architecture Patterns*. (H. Scherer, Ed.) (O'Reilly M). Sebastopol, CA.

Richardson, C. (2014a). Microservices: Decomposing Applications for Deployability and Scalability. Retrieved April 15, 2017, from http://www.infoq.com/articles/microservices-intro

Richardson, C. (2014b). Monolithic Architecture pattern. Retrieved September 17, 2017, from http://microservices.io/patterns/monolithic.html

Richardson, C. (2014c). Who is using microservices? Retrieved October 6, 2017, from http://microservices.io/articles/whoisusingmicroservices.html

Schramm, W. (1955). The Nature of Communication between Humans. *The Process and Effects of Mass-Communications*, 43.

SEI. (2015). Defining Software Architecture: What Is Software Architecture? Retrieved September 17, 2017, from http://www.sei.cmu.edu/architecture/

Shalom, N. (2014). Space Based Architecture - Using SSD As A Foundation For New Generations Of Flash Databases. Retrieved October 8, 2017, from http://natishalom.typepad.com/nati_shaloms_blog/space-based-architecture/

Stake, R. E. (2000). Case Studies. *Handbook of Qualitative Research*. https://doi.org/10.1258/096214400320575624

Stephens, D., Crider, K., Wheeler, S., Willis, A. C., & Freemanwa, C. (2017). What is Azure Application Insights? Retrieved October 20, 2017, from https://docs.microsoft.com/en-us/azure/application-insights/app-insights-overview

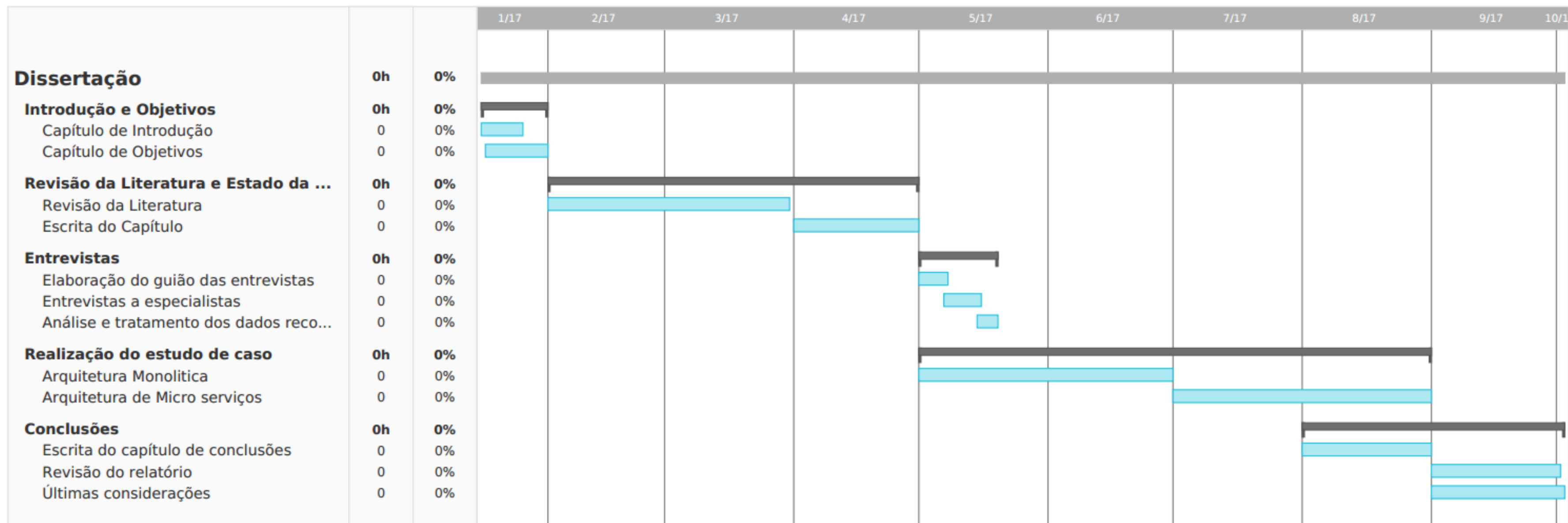
Taylor, R. N., Medvidovic, N., Kenneth, M., Anderson, James Whitehead, E., Bobbins, J. E., … Dubrow, D. L. (1996). A component- and message-based architectural style for gui software. *IEEE Transactions on Software Engineering*, *22*(6), 390–406. https://doi.org/10.1109/32.508313

Technopedia. (2016). What is Scalability? - Definition from Techopedia. Retrieved September

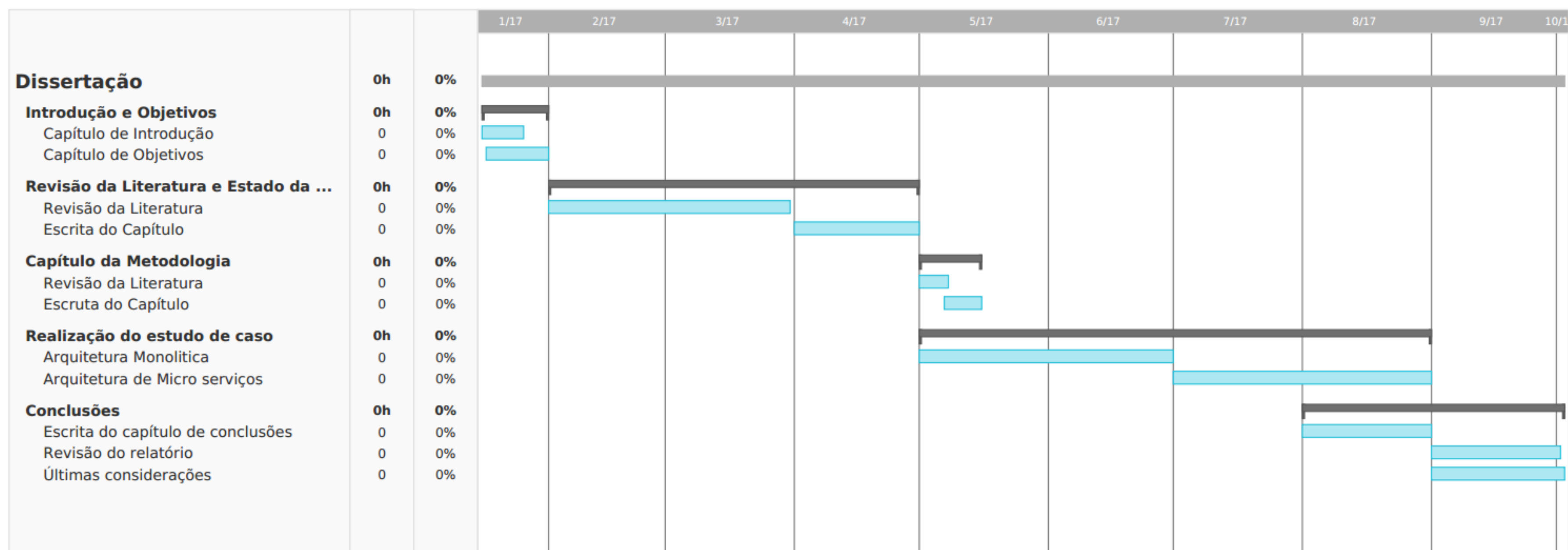
17, 2017, from https://www.techopedia.com/definition/9269/scalability

Yin, R. K. (1989). *Case study research : design and methods*. *Applied social research methods series Volume 5*. https://doi.org/10.1097/FCH.0b013e31822dda9e

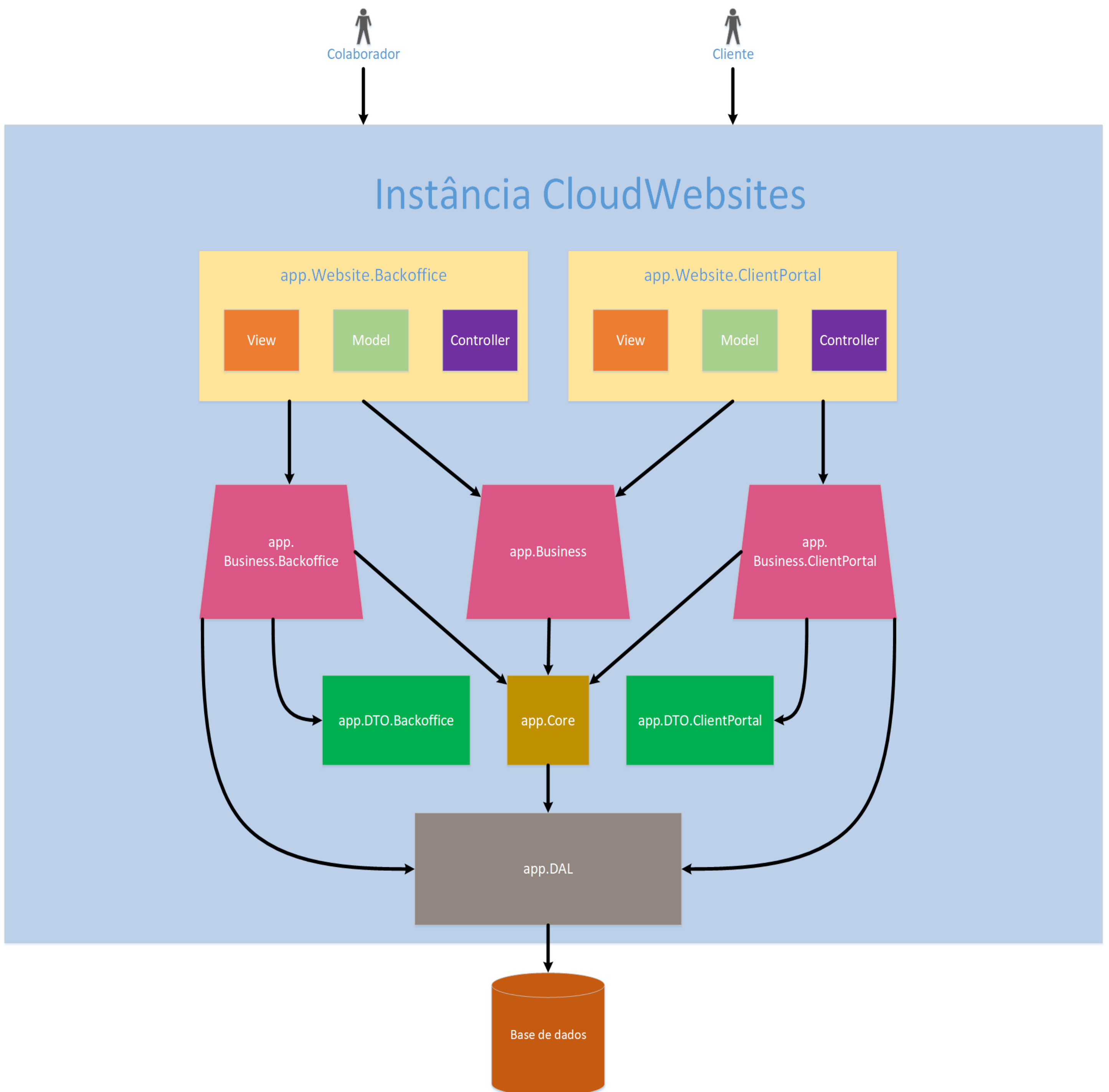
Anexo A: Diagrama Gantt - Cronograma original



Anexo B: Diagrama Gantt – Cronograma Revisto



Anexo C: Arquitetura monolítica



Anexo D: Arquitetura de micro serviços