Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Rafael Barbosa Correia

Construção de uma plataforma de e-commerce: uma abordagem baseada numa arquitetura de microsserviços

Dezembro 2021

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Rafael Barbosa Correia

Construção de uma plataforma de e-commerce: uma abordagem baseada numa arquitetura de microsserviços

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Dissertação supervisionada por
António Manuel Nestor Ribeiro

Dezembro 2021

AGRADECIMENTOS

A presente dissertação conclui a minha etapa universitária. Foram cinco anos de aprendizagem, esforço e dedicação que me fizeram crescer como estudante e como pessoa, mas que não fiz sozinho. Esta caminhada é o resultado de todas as pessoas que me acompanharam e que de alguma forma contribuíram para isso.

Ao meu orientador, Professor António Manuel Nestor Ribeiro, pela orientação, interesse e disponibilidade demonstrada. Este trabalho não poderia ter sido feito sem as suas revisões, sugestões e críticas que melhoraram em muito esta dissertação.

Aos meus pais e irmã, pela compreensão e apoio nos bons e maus momentos, pelas palavras de incentivo, confiança depositada e investimento na minha educação. Nunca deixaram de acreditar em mim e sem eles esta caminhada não era possível.

À Rita, pelo apoio incondicional, pela força dada para continuar esta etapa e por estar sempre presente.

Aos meus amigos, em especial ao Zé e ao Pedro, pelos momentos passados nestes cinco anos. Foram anos de amizade, aprendizagem mútua e muitas histórias.

À Kodly, por me fornecerem este tema de dissertação, acreditarem e investirem em mim, fazendo-me crescer como profissional e pessoa.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico.

Confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

RESUMO

As arquiteturas monolíticas estão, em grande parte, presentes na maioria das plataformas de e-commerce, o que leva a um processo de modificação mais complicado e entregas demoradas ao cliente, uma vez que não está preparada para trabalho em paralelo.

A arquitetura de microsserviços veio proporcionar outra forma de desenvolvimento destas plataformas, permitindo o trabalho em simultâneo por diferentes equipas, produzindo novas entregas para o cliente de forma mais acelerada e segura. Todavia, esta possui alguns desafios e complexidades, o que leva muitas vezes à escolha de uma arquitetura monolítica para o desenvolvimento da aplicação.

A maioria das aplicações não são imutáveis, pois mesmo estando entregues ao cliente são sujeitas a modificações. Esta necessidade de modificar a aplicação leva a preocupações acerca da rapidez com que as novas funcionalidades são entregues ao cliente. É preciso tomar decisões no início do desenvolvimento sobre que arquitetura seguir, de modo a tomar a decisão mais vantajosa. No caso de aplicações monolíticas a mudança para uma arquitetura de microsserviços facilita este aspeto, bem como muitos outros. Contudo, esta separação pode-se tornar quase impossível se o monolítico não for bem preparado para uma eventual futura mudança.

Uma das maiores dificuldades numa migração de um monolítico para microsserviços, relaciona-se com a definição do que deve ser cada microsserviço e na comunicação entre estes. A migração deve partir da identificação de partes do código que possam ser isoladas sem ter muito impacto no resto do código. Com o desenho de um diagrama de *packages* é possível obter uma visão sobre a estrutura do sistema, percebendo que componentes são mais fáceis e mais difíceis de extrair. Deve-se começar por extrair aqueles que contém menos dependências, adquirindo as vantagens de uma migração incremental que permite que sejam reduzidos os erros efetuados, porém, pode haver situações em que se queira extrair um componente com mais dependências.

É necessário compreender o porquê da migração para uma arquitetura de microsserviços. Esta decisão não deve ser tomada apenas porque a arquitetura de microsserviços está em voga, mas sim por razões fundamentadas. Dentro destas razões encontra-se a rapidez com que as mudanças são efetuadas e colocadas em produção, pois é mais fácil realizar modificações e voltar a instalar os microsserviços sem que toda a aplicação tenha que reiniciar. Isto permite uma melhor estruturação da equipa, possibilitando que várias equipas possam trabalhar em simultâneo para a mesma aplicação, não prejudicando em nada outros microsserviço. Outra razão é a necessidade de escalar os microsserviços independentemente, providenciando maior robustez, pois a falha de um serviço não leva à falha de toda a aplicação ou então pela escolha de tecnologia, podendo-se implementar os microsserviço com a tecnologia que seja mais adequada e eficiente.

PALAVRAS-CHAVE Arquitetura de Microsserviços, Arquitetura Monolítica, Arquiteturas de Software, Padrões de Microsserviços

ABSTRACT

Monolithic architectures are largely present in most e-commerce platforms, which leads to a more complicated modification process and time-consuming deliveries to the customer as it is not prepared for work in parallel.

Microservices architecture has provided another way of developing these platforms, allowing different teams to work simultaneously, producing new deliveries for the client in a faster and safer way. However, this has some challenges and complexities, which often leads to the adoption of a monolithic architecture for the development of the application.

Most applications are not immutable, as they are subject to modification even when delivered to the client. This need to modify the application leads to concerns about how quickly new functionality is delivered to the customer. It is necessary to make decisions at the beginning of the development about what type of architecture to choose, in order to make the most advantageous decision. In the case of monolithic applications switching to a microservices architecture facilitates this aspect, as well as many others. However, this separation can become almost impossible if the monolith is not well prepared for possible future change. One of the greatest difficulties in a migration from a monolithic to microservices is related to the definition of what each microservice should be and the change of communication between them. With this, the migration should start from the identification of parts of the code that can be isolated without having much impact on the rest of the code. By drawing a package diagram it is possible to gain insight into the structure of the system, realising which components are easier and more difficult to extract. You should start by extracting those modules which contain fewer dependencies, acquiring the advantages of an incremental migration that allows the reduction of errors made and their size, however, there may be situations in which you want to extract a component with more dependencies. It is necessary to understand why migration to a microservices architecture is necessary. This decision should not be made just because microservices architecture is in vogue, but for well-founded reasons. One of these reasons is the speed with which changes are made and put into production, as it is easier to make modifications and reinstall the microservices without having to restart the whole application. This allows a better structuring of the team, enabling several teams to work simultaneously for the same application, without harming other microservices. Another reason is the need to scale microservices independently, providing greater robustness, as the failure of one microservice does not lead to the failure of the whole application, or by the choice of technology, being able to implement microservices with the technology that is most appropriate and efficient.

KEYWORDS Microservice Architecture, Microservice Patterns, Monolithic Architecture, Software Architectures

CONTEÚDO

LISTA DE FIGURAS

LISTA DE EXCERTOS DE CÓDIGO

1

INTRODUÇÃO

Nos dias que correm, a maioria das plataformas de *e-commerce* são desenvolvidas segundo uma arquitetura monolítica, o que torna a sua modificação um processo mais complicado e difícil, dado que toda a aplicação é construída como uma unidade só.

Este tipo de arquiteturas não estão preparadas para suportar alterações por diferentes equipas em simultâneo, levando a que o processo de adição e modificação de novas funcionalidades para o consumidor sejam demoradas.

Com o mercado *online* cada vez mais competitivo, onde a rapidez com que se lançam novas funcionalidades para o consumidor é determinante, aliado ao crescimento de sistemas alojados na *Cloud* devido ao seu menor custo, maior escalabilidade, resiliência e disponibilidade, torna-se necessária uma remodelação deste tipo de plataformas.

A arquitetura de microsserviços veio revolucionar a forma como se desenvolve *software* atualmente. Esta arquitetura, juntamente com processos de integração contínua (CI/CD) e alojada em *containers* na *Cloud*, permite desacoplar as funcionalidades por diferentes equipas independentes, o que permite um ritmo acelerado de introdução de novas funcionalidades para o consumidor de forma segura e controlada.

É preciso analisar as características de uma arquitetura monolítica, comparando-a com uma de microsserviços. Cada uma tem as suas vantagens como desvantagens e existem vários fatores a ter em conta na escolha de uma destas arquiteturas para o desenvolvimento de uma aplicação.

Neste tema de dissertação, vai-se desenvolver uma aplicação de *e-commerce* seguindo uma arquitetura monolítica, migrando-a posteriormente para uma arquitetura de microsserviços, de forma a perceber o impacto da migração de uma arquitetura monolítica para microserviços. Para isso, introduziu-se a necessidade de preparar a aplicação quando se for efetivamente migrar para uma arquitetura de microsserviços. Isto é conseguido através da organização do código, estando este agrupado à volta de domínios. Pretende-se que a alteração de código nestes domínios tenha o menor impacto possível no resto do código. Tal como este, também o código relativo à base de dados deve ser devidamente agrupado com os domínios a que pertencem. Com esta organização, temos uma única aplicação instalada conjuntamente, mas composta por vários módulos que são bem visíveis e podem tornar-se em futuros serviços quando se proceder à migração para microsserviços.

O processo de migração de código pode ser facilitado, recorrendo a padrões de migração que fornecem diretrizes para se remover código que esteja menos e mais acoplado na aplicação, bem como garantias de que a funcionalidade continua a se comportar da mesma forma que antes da migração. O mesmo acontece para a base de dados. Existem padrões de migração para lidar com a necessidade de migrar dados e de aceder a estes. Um destes casos é quando se remove funcionalidade do monolítico, mas se continua a necessitar de dados que

se encontram no monolítico. Outro é ter que mudar a posse dos dados, tendo que migrar dados do monolítico para uma base de dados única ao serviço.

Antes de mais, é preciso saber por onde começar a migração. É necessário ter uma visão geral da aplicação, da estrutura desta e das dependências existentes entre componentes. Para isso, o desenho de um diagrama de *packages* é fundamental para se perceber que componentes são mais fáceis e mais difíceis de extrair. De modo a minimizar os erros a acontecer e a magnitude destes, aconselha-se uma migração incremental, com mudanças pequenas e incrementais, para se poder retroceder para versões anteriores caso algum problema aconteça.

1.1 MOTIVAÇÃO

Numa arquitetura monolítica uma aplicação é construída como uma unidade só, estando num único processo [Newman (2019)]. A maioria das aplicações são constituídas por três camadas: a camada de apresentação, a camada de negócio e a camada de dados.

A camada da apresentação está responsável por apresentar as funcionalidades da aplicação ao utilizador, de reencaminhar as ações do utilizador para a camada de negócio e de receber os efeitos dessas ações da camada de lógica. A camada de negócio coordena a aplicação, recebe/envia dados, toma decisões de acordo com os pedidos que recebe, entre outros. A camada de dados está encarregue de armazenar todos os dados que se pretendam persistir, recebendo pedidos da camada de negócio.

Um monolítico pode ser escalado horizontalmente, isto é, possuir várias réplicas iguais que, em caso de falha da que está ativa, tomam lugar para que o sistema não fique inoperacional. Um dos maiores problemas das aplicações monolíticas prende-se com o facto de ser preciso que toda a aplicação seja novamente compilada e instalada quando existe uma mudança a alguma parte da aplicação, por mais pequena que essa mudança seja [Newman (2019)].

Para colmatar estes problemas, existe o modelo arquitetural de microsserviços. Nesta arquitetura uma aplicação é composta por um conjunto de microsserviços, cada um a correr no seu próprio processo e comunicando através de mecanismos de entrega de mensagens, permitindo que sejam independentes um dos outros, que possam escalar de formas/quantidades distintas e sejam escritos em tecnologias diferentes [Newman (2015)].

Muitas aplicações são inicialmente desenvolvidas segundo uma arquitetura monolítica, porém por motivos de escalabilidade, resiliência e desempenho é decidido muitas vezes a mudança para uma arquitetura de microsserviços. O maior problema em transformar um monolítico em microsserviços prende-se na mudança da forma de comunicação entre os microsserviços e na separação do modelo de dados.

A arquitetura de microsserviços pode ser definida como um conjunto de microsserviços autónomos que trabalham conjuntamente entre si para o funcionamento de um produto de *software*. Estes focam-se em ser independentemente instaláveis, significando que é possível efetuar modificações, proceder à sua instalação e não ter impacto noutros microsserviços. Esta independência permite que possam ser desenvolvidos em diferentes tecnologias e existe apenas se estes forem bem definidos, sendo importante que sejam definidos à volta de um domínio, de modo que quando se efetuem alterações relativas a este domínio, estas sejam só efetuadas num e não em vários. Isto implica que cada microsserviço tenha a posse dos seus próprios dados, não existindo bases

de dados partilhadas, porque se um microsserviço necessitar de dados de outro, não deve aceder diretamente à base de dados, mas sim efetuar uma chamada ao microsserviço que possui os dados que este pretende. Os microsserviços devem ocultar o máximo de informação possível, permitindo uma maior facilidade na gestão de cada um.

1.2 OBJETIVOS

Esta dissertação pretende o desenvolvimento de uma plataforma *e-commerce* que seja escalável, resiliente e responsiva. Os requisitos foram previamente levantados.

Com os requisitos previamente levantados, proceder-se-á em primeiro lugar desenvolver a aplicação seguindo uma arquitetura monolítica, realizando depois a conversão de alguns módulos para seguir uma arquitetura de microsserviços.

Para isto, os objetivos passam por:

- Investigar o estado da arte na área de plataformas de *e-commerce*.
- Identificar e analisar as diferentes características de sistemas baseados numa arquitetura de microsserviços em contraste com uma arquitetura monolítica.
- Identificar e analisar padrões de migração de uma arquitetura monolítica para uma arquitetura de microsserviços.
- Desenvolver um protótipo funcional, seguindo uma arquitetura monolítica, que implemente um site de *e-commerce* com as funcionalidades básicas (autenticação, listar produtos, adicionar ao carrinho) que implemente a solução escolhida.
- Migrar a plataforma de *e-commerce* para uma arquitetura de microsserviços, atendendo a fatores como escalabilidade, resiliência e disponibilidade.
- Comparar a arquitetura monolítica com a arquitetura de microsserviços.

1.3 ESTRUTURA DO DOCUMENTO

No capítulo 2 aborda-se sobre o estado da arte na área de plataformas de *e-commerce*.

Neste apresenta-se a arquitetura monolítica, o que a define, as suas vantagens e desvantagens e quando deve ser considerada para desenvolvimento de uma aplicação.

A seguir aborda-se a arquitetura de microsserviços, apresentando o que representa um microsserviço nesta arquitetura e como o definir, falando-se de como pode ser efetuada a comunicação entre estes e outros temas em volta dos microsserviços. Após isto abordam-se as vantagens que esta arquitetura possui, bem como as suas desvantagens. Por último apresentam-se alguns desafios que se encontram ao adotar uma arquitetura de microsserviços que não se encontrariam com uma arquitetura monolítica e quando esta é a decisão acertada.

Menciona-se ainda a arquitetura de *Service-Oriented Architecture* (SOA), uma arquitetura que se encontra associada à de microsserviços, tendo algumas semelhanças, mas também diferenças.

No capítulo 3 aborda-se a parte teórica de como efetuar uma migração de uma aplicação que segue uma arquitetura monolítica para uma arquitetura de microsserviços, sugerindo-se uma migração incremental, para minimizar os riscos desta.

Apresentam-se as alterações necessárias a efetuar no código e na base de dados, bem como padrões de migração que ajudam nesta tarefa. Com a separação da camada de dados, é necessário abordar transações e para complementar este tema, apresentam-se sagas, um mecanismo de gerir uma transação que envolve vários microsserviços.

Concluindo este capítulo, abordam-se as dependências entre componentes da aplicação e como estas ajudam a perceber que componentes são mais fáceis e difíceis de extrair do monolítico.

Posto isto, passa-se para o capítulo 4 onde se efetua uma descrição da plataforma a desenvolver, tecnologia a utilizar, modulação desta e instalação.

Com a plataforma apresentada, introduz-se o capítulo 5 onde se começa por definir o processo de migração para o caso prático, diretrizes a seguir, escolha de padrões de migração a usar, tipo de separação da base de dados, entre outros. A seguir, mencionam-se as dependências entre componentes, demonstrando quais componentes seriam mais fáceis a extrair e os mais difíceis. Com isto, identificaram-se três componentes a extrair, cada um com tipos diferentes de dependências e funcionalidades. Para terminar o capítulo, discutiu-se esta migração, mencionando as melhorias que trouxe à aplicação, as vantagens e desvantagens dela.

Por último, efetua-se uma conclusão e apresenta-se o trabalho futuro.

2

ESTADO DA ARTE

No presente capítulo será apresentado um estudo das arquiteturas monolítica e de microsserviços. Irá ser feita uma comparação destas duas arquiteturas, as vantagens e desvantagens de cada uma e quando se deve escolher uma ao invés da outra.

2.1 ARQUITETURA MONOLÍTICA

> "Any organization that designs a system... will inevitably produce a design whose structure is a copy of the organization's communication structure."
>
> (Conway's law)

A arquitetura monolítica é uma arquitetura onde um produto de *software* é concebido para funcionar como uma unidade só, estando num único processo. Os componentes neste tipo de arquitetura encontram-se conectados e dependentes uns dos outros, resultando num código mais acoplado, o que leva a que toda a aplicação tenha que ser instalada juntamente [Newman (2019)].

Uma aplicação monolítica é constituída maioritariamente por três camadas, a camada de apresentação, camada de negócio e camada de dados. Na Figura 1 temos um diagrama representativo de uma arquitetura monolítica, onde é possível observar a presença destas três camadas. Toda a funcionalidade, código, componentes e lógica formam uma única entidade [Javed (2019)].

Esta arquitetura é bastante usada porque é universal e bem conhecida. Outra razão é devido à forma como as equipas a desenvolver uma aplicação estão organizadas [Newman (2019)].

Nesta arquitetura as equipas estão normalmente organizadas conforme as competências de cada programador, sejam elas administração de base de dados, programação *backend* ou *frontend*. Desta forma, o desenvolvimento de uma aplicação, por parte de equipas que se organizam desta forma será organizada de acordo com a organização das equipas [Conway (1968)].

Normalmente numa aplicação monolítica existe apenas uma base de dados a servir toda a aplicação [Javed (2019)], sendo que depois por motivos de desempenho e de tolerância a faltas, se possa replicar a base de dados, para prevenir a perda de dados ou para acelerar os acessos a esta. As aplicações que seguem esta arquitetura possuem um código bastante extenso, muito acoplado e por vezes pouco modular.

É possível ter múltiplas instâncias de uma mesma aplicação monolítica por razões de robustez e de escalonamento, mas a aplicação continua basicamente a consistir numa única unidade [Newman (2019)].

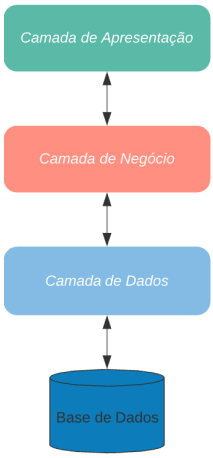


Figura 1: Arquitetura monolítica.

2.1.1 *Tipos de monolítico*

Monolítico num único processo

O exemplo mais comum de um monolítico é um sistema onde todo o código está instalado num único processo. É possível ter várias instâncias deste monolítico por questões de robustez e resiliência, mas fundamentalmente todo o código se encontra num único processo [Newman (2019)].

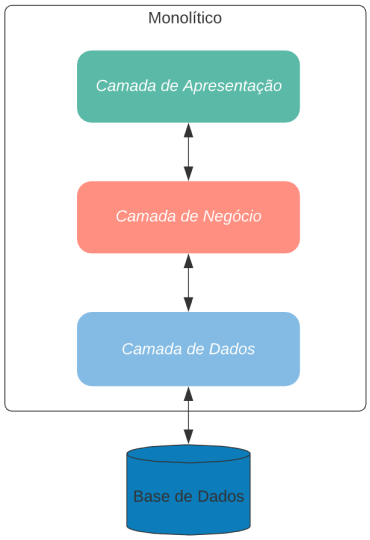


Figura 2: Monolítico num único processo.

Monolítico Modular

O monolítico modular é uma variação do monolítico num único processo, exceto que o sistema se encontra separado por módulos, que podem ser desenvolvidos independentemente, mas que precisam de estar juntos para a aplicação poder ser instalada [Newman (2019)].

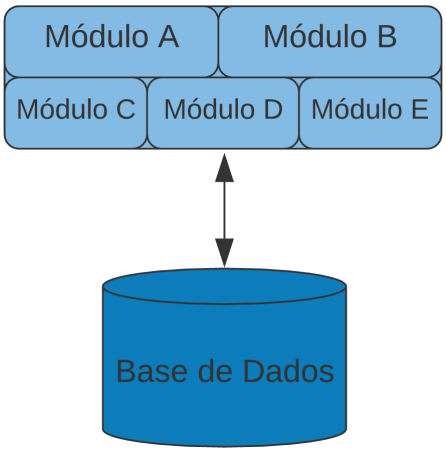


Figura 3: Monolítico Modular.

Esta forma de organização do código permite o trabalho em paralelo por várias equipas nos diferentes módulos, não necessitando de haver preocupação com os problemas de sistemas distribuídos adquiridos da adoção de microsserviços [Newman (2019)].

Um dos problemas deste tipo de monolítico é a base de dados tender a não ter o mesmo nível de decomposição que o código, o que pode dificultar um eventual processo de separação do monolítico [Newman (2019)].

Monolítico Distribuído

Um monolítico distribuído é um sistema composto por vários serviços, mas que por alguma razão necessita de ser instalado conjuntamente, isto é, os serviços não podem ser instalados independentemente [Newman (2019)].

2.1.2 *Vantagens*

A arquitetura monolítica possui vantagens como as apresentadas:

- **Fácil desenvolvimento**

  O desenvolvimento de uma aplicação monolítica é mais fácil, no sentido em que as ligações entre componentes não tem de ser estabelecidas sobre a rede, não havendo necessidade de preocupações com a perda de mensagens, nem com problemas adotados dos sistemas distribuídos [Richardson (2018)].

  Além disso, numa fase inicial do desenvolvimento de *software* não existe a necessidade de ter que pensar como deve ser a separação dos microsserviços [Newman (2019)].

- **Simples de resolver problemas**

  Um problema que cause a falha de uma funcionalidade ou da aplicação é mais facilmente encontrado e resolvido visto toda a funcionalidade se encontrar num único lugar.

- **Simples de testar**

  Um produto de *software* deve contemplar testes às suas funcionalidades, como testes unitários ou de integração, para que se determine a qualidade do código e que as funcionalidades desenvolvidas comportam-se como efetivamente pretendido.

  Num único artefacto é muito mais fácil de testar a aplicação na totalidade e a integração com cada componente [Richardson (2018)].

- **Instalação mais fácil**

  Uma aplicação monolítica é um único artefacto sendo só necessário copiar o artefacto para um servidor e pôr a correr [Richardson (2018)].

  Se a aplicação fosse composta por vários componentes que tivessem que ser instalados separadamente era necessário a configuração destes para a aplicação funcionar na totalidade.

  Com uma aplicação monolítica não é necessário ter a preocupação de configurar todos os componentes da aplicação, visto que eles já se encontram conectados. Evita-se também ter problemas normalmente associados aos sistemas distribuídos.

- **Simples de escalar**

  Um único artefacto é fácil de escalar horizontalmente, tendo várias máquinas a correr a mesma aplicação atrás de um *load balancer* [Richardson (2018)], tal como apresentado na Figura 4.

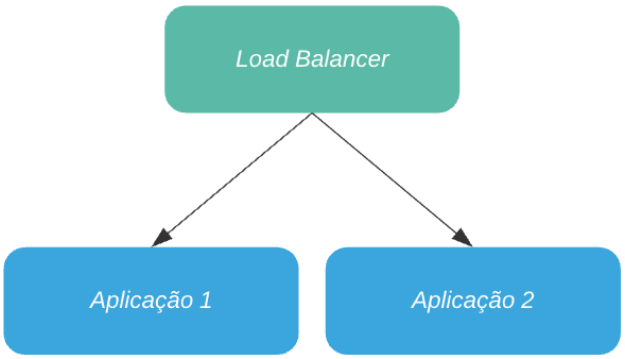


Figura 4: Escalonamento de uma Arquitetura Monolítica.

- **Desempenho**

  Todos os componentes encontram-se no mesmo processo, sendo a comunicação entre estes efetuada através de abstrações da tecnologia utilizada e recorrendo a memória partilhada. Isto é mais rápido e eficiente do que fazer pedidos a outros componentes que não residam no mesmo processo. Neste caso

os pedidos iriam ser feitos pela rede que pode ter inconsistências, tal como atrasos, perda de mensagens, entre outros, que leva a um aumento do tempo de resposta [Newman (2019)].

2.1.3 *Desvantagens*

Apesar de vantagens, uma aplicação monolítica possui também desvantagens, como as apresentadas:

- Uma aplicação monolítica acaba por ter um código bastante extenso e de difícil compreensão, o que leva a uma maior dificuldade e demora em eventuais alterações ou adições às funcionalidades da aplicação. A introdução de um novo elemento na equipa de desenvolvimento também será feita com algum custo. Este terá mais dificuldade em compreender o código, dado que o código estará muito acoplado, complexo e extenso [Richardson (2018)].
- Uma modificação a um componente da aplicação implica que toda a aplicação tenha que compilar de novo e voltar a ser instalada [Richardson (2018)].
- Não está adaptada para várias equipas trabalharem em simultâneo. Várias pessoas a trabalhar no mesmo produto de *software* pode levar a que umas se atrapalhem às outras, acabando por modificar partes que outros estavam a modificar, levando a conflitos no código [Newman (2019)].

  Leva a um processo de adição de novas funcionalidades para o utilizador mais demorado, podendo haver atrasos em várias fases da entrega do produto.
- A falha de num componente da aplicação leva à falha de toda a aplicação.

  A falha de um componente da aplicação tem impacto na disponibilidade da aplicação, pois mesmo que a aplicação seja escalada horizontalmente se existir um problema num componente este estará em todas as outras máquinas [Javed (2019)].
- Complexidade na adoção de novas tecnologias.

  A arquitetura monolítica dificulta a adoção de novas linguagens ou *frameworks*.

  A adoção de uma nova tecnologia implica a reescrita de grande parte do código da aplicação, o que pode ser problemático em termos de tempo, custo e eventuais problemas que possam surgir [Newman (2019)].
- Escalabilidade

  Quando se quer escalar a aplicação, tem que se escalar na totalidade. Não é possível escalar apenas componentes da aplicação, que necessitem de uma maior eficácia e desempenho [Richardson (2018)].

2.1.4 *Decisão arquitetural*

Desenvolver segundo uma arquitetura monolítica permite uma maior facilidade no início do desenvolvimento. Não existe a necessidade de pensar na divisão dos microsserviços, o que leva a uma maior rapidez no desenvolvimento do produto de *software* ou do MVP (minimum viable product) [Fowler (2015)].

A decisão de se desenvolver uma aplicação seguindo uma arquitetura monolítica deve ser tida em conta quando:

- A equipa de desenvolvimento é pequena.

  Uma equipa pequena ao se focar no desenvolvimento de uma aplicação que siga uma arquitetura de microsserviços acabaria por perder bastante tempo e recursos a lidar com as complexidades desta arquitetura. Uma aplicação monolítica acaba por ser a solução mais rápida e eficiente de se desenvolver a aplicação.

- A aplicação a desenvolver é simples.

  Uma arquitetura monolítica é simples de entender e desenvolver uma aplicação com poucas funcionalidades [Fowler (2015)].

- Falta de conhecimento com a arquitetura de microsserviços.

  Começar a desenvolver uma aplicação seguindo uma arquitetura de microsserviços sem ter conhecimento na área torna-se um processo dispendioso e frustrante. É necessário gerir os vários serviços, as ligações entre estes, conflitos e problemas que surjam, o que dificulta a resolução destes, dado que os microsserviços se encontram em processos diferentes [Fowler (2015)].

- A aplicação deve ser instalada rapidamente.

  Uma aplicação monolítica é mais fácil de ser instalada, visto que toda a aplicação se encontra num único artefacto, sendo apenas necessário colocar este artefacto num servidor e pôr a correr.

2.2 ARQUITETURA DE MICROSSERVIÇOS

A arquitetura de microsserviços é uma abordagem ao desenvolvimento de *software* definida como um conjunto de microsserviços autónomos que trabalham conjuntamente e comunicam entre si para o funcionamento de um produto de *software* [Fowler and Lewis (2014)].

Ao contrário do que se encontra na Figura 1 apresentada na secção 2.1, na Figura 5 podemos observar que a camada de negócio foi separada em vários microsserviços, independentes, que comunicam entre si e possuem a sua própria base de dados.

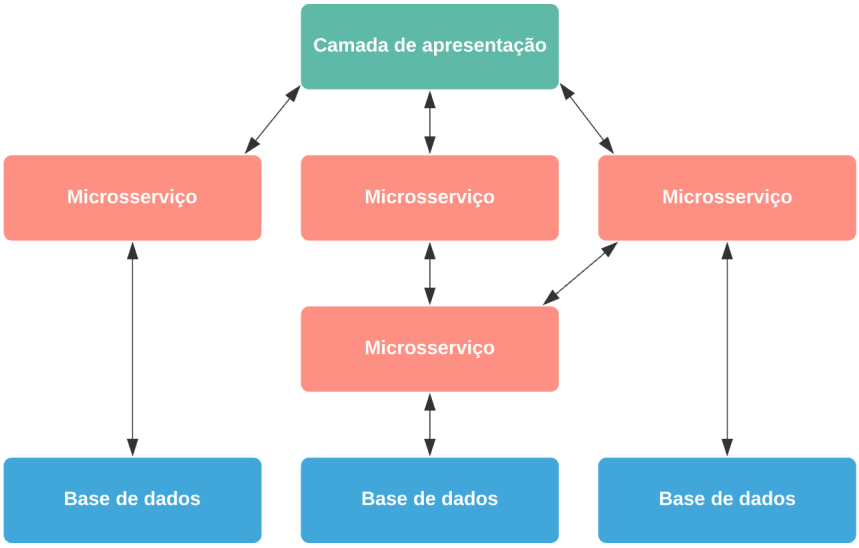


Figura 5: Arquitetura de Microsserviços.

2.2.1 *Definição de um microsserviço*

Numa arquitetura de microsserviços um microsserviço é uma unidade de *software* que é independente de todos os outros, capaz de ser substituído e atualizado sem que os outros microsserviços tenham que ter qualquer tipo de ação perante esta mudança [Newman (2019)].

Cada microsserviço corre num processo exclusivamente seu e de forma autónoma, encapsulando funcionalidade e comunicando com outros através da rede [Newman (2021)]. Como é possível observar pela Figura 5 cada microsserviço tem a sua própria base de dados, sendo este responsável por gerir os seus dados. Caso um microsserviço necessite de informação que outro possua, este realiza um pedido a esse microsserviço para que este lhe devolva a informação. Do exterior, um microsserviço é tratado como uma caixa negra e nenhum acede diretamente à base de dados de outro. No entanto, pode existir um microsserviço que não necessite de base de dados [Newman (2021)].

Um microsserviço pode representar o inventário, outro a gestão das encomendas e outro o envio das encomendas, mas juntos trabalham conjuntamente para constituir uma aplicação.

Ter microsserviços que possuem fronteiras claras e bem definidas que não são modificadas quando existe uma mudança numa implementação interna, resulta num sistema que possui pouco acoplamento e bastante coesão [Newman (2021)], algo a mencionar mais à frente na secção 2.2.3.

Agregados

Uma das dificuldades encontrada quando se desenvolve uma aplicação seguindo uma arquitetura de microsserviços é definir o que deve ser um microsserviço, com que funcionalidades este deve ficar encarregue e quais as suas fronteiras.

Um conceito apresentado por Eric Evans no seu livro *Domain-Driven-Design* [Evans (2003)] é o de *agregado*. Um agregado é um conjunto de objetos de domínio que podem ser tratados como uma unidade só. São representações de domínios reais. Um exemplo de um agregado pode ser uma encomenda e os produtos desta encomenda. Estes são objetos diferentes e separados, mas pode ser vantajoso tratá-los como um agregado só [Fowler (2013)].

Um agregado normalmente possui um ciclo de vida, o que permite que seja implementado como uma máquina de estados. O que se pretende é tratar agregados como unidades independentes, que todo o código que trata das transações de estado do agregado seja agrupado conjuntamente, bem como o próprio estado. Agregados podem ter relações com outros agregados [Newman (2019)].

Mapeando o conceito de agregados para microsserviços, um único microsserviço pode lidar com o ciclo de vida e armazenamento de dados de um ou mais agregados. Caso outro microsserviço queira alterar o estado de um agregado de outro, deve efetuar um pedido a este para efetuar a transferência de estado. Esta possibilidade de alteração de estado deve ser definida pelo microsserviço que gere o agregado, evitando mudanças de estado ilegais [Newman (2019)].

Bounded Context

> "Cells can exist because their membranes define what is in and out and determine what can pass."
>
> (Eric Evans)

Bounded Context [Evans (2003)] é outro conceito introduzido no livro *Domain-Driven Design* que define fronteiras numa aplicação, indicando vários sub-domínios que possuem funções e características diferentes. A ideia passa por indicar que qualquer domínio consiste num conjunto de *Bounded Context*, sendo que dentro de cada *Bounded Context* existem modelos que podem ou não ser partilhados com o exterior [Evans (2003)].

Eric Evans faz a analogia com as células, dizendo que estas existem porque as suas membranas definem o que se encontra dentro e fora da célula e o que entra e o que sai. *Bounded Context* encontra-se como uma boa definição de o que deve ser um microsserviço. Deve estar responsável pelos seus próprios dados e funcionalidades, receber pedidos e fazer pedidos a outros microsserviços, controlando o que se encontra dentro de si próprio [Newman (2015)].

Cada microsserviço possui uma *interface*, onde expõe dados para que o exterior possa aceder. A base de dados de um microsserviço encontra-se encapsulada dentro das suas fronteiras, só podendo ser acedida por este. Cada um deve decidir que informação partilhar com outros *Bounded Context* [Newman (2015)].

Do ponto de vista da implementação de um microsserviço, um *Bounded Context* contém um ou mais agregados. Alguns agregados podem ser expostos para fora do contexto e outros serem escondidos internamente. Tal como agregados, *Bounded Context* podem ter relações para outros *Bounded Context* [Newman (2019)].

2.2.2 *Características de um microsserviço*

Independentemente instalado

Cada microsserviço deve poder ser modificado e instalado sem afetar os outros. Desta forma não existe necessidade de modificar os outros microsserviços por causa deste, o que diminui a quantidade de instalações necessárias. Para se conseguir tal, é necessário que os microsserviços sejam pouco acoplados, para se poder efetuar alterações sem ter obrigatoriamente que mudar outros [Newman (2019)].

Modulado à volta de um domínio

A realização de uma adição ou mudança que envolva vários processos é complicada e dispendiosa. Se for necessário a mudança a uma funcionalidade que envolva dois microsserviços, será necessária a compilação e instalação destes, acabando por se ter o dobro do trabalho que se teria se apenas fossem necessárias alterações a um microsserviço. Desenvolver uma funcionalidade que envolva a mudança em mais que um microsserviço é dispendioso, pois é necessário coordenar o trabalho entre estes, potencialmente entre equipas e gerir cuidadosamente a ordem com que devem ser instaladas as novas versões destes, o que leva a maior trabalho, do que desenvolver apenas a funcionalidade num único microsserviço [Newman (2021)].

Possui os seus próprios dados

Cada microsserviço deve ter a sua própria base de dados e não devem existir bases de dados partilhadas. Se um microsserviço necessita de dados de outro, não deve aceder diretamente à base de dados desse, mas sim pedir-lhe por esses dados [Newman (2019)].

Desta forma permite-se que cada microsserviço defina quais dados pretende partilhar e quais esconder do exterior. Além disso, também se reduz a acoplamento entre microsserviços.

Ocultar informação

Com microsserviços é possível ocultar informação, que significa ocultar o máximo de informação possível no microsserviço e expor o mínimo para o exterior. Permite que exista uma separação clara entre o que pode ser facilmente mudado e aquilo que é mais complicado e se deve ter mais cuidado. Implementações que se encontrem ocultas do exterior podem ser modificadas sem preocupação, dado que não devem afetar o exterior, pois estes não tem acesso a estas [Newman (2021)].

Agnóstico à tecnologia

Cada microsserviço pode ser implementado na tecnologia pretendida, podendo escolher a melhor tecnologia para cada situação [Newman (2019)].

2.2.3 *Acoplamento e Coesão*

> "A structure is stable if cohesion is high, and coupling is low."
>
> (Larry Constantine)

É importante quando se desenvolve microsserviços que se tenha em conta o acoplamento e coesão entre microsserviços aquando da definição das fronteiras de cada um [Newman (2019)].

Acoplamento é a medida de dependência que existe entre sistemas. Quanto mais os microsserviços se encontram acoplados, maior mudanças implicará nesses quando uma alteração tiver que ser realizada [Newman (2019)].

Uma boa definição de coesão é que o código, que precisa de ser modificado quando ocorre uma alteração, deve estar agrupado [Newman (2019)].

Com microsserviços o que se pretende é que quando seja necessário efetuar uma alteração não se tenha que modificar vários microsserviços, mas sim apenas num. Desta forma o que se pretende é que se tenha o código agrupado de certa forma a que quando for necessário efetuar alterações, estas sejam realizadas no menor número possível de microsserviços, alcançado através de microsserviços muito coesos e pouco acoplados [Newman (2019)].

A existência de microsserviços que sejam pouco coesos e muito acoplados leva a que seja muito dispendioso lidar com mudanças, visto que é necessário modificar microsserviços que residem em processos separados, levando à necessidade de ter que instalar estes microsserviços que são independentemente instaláveis [Newman (2019)].

O que se pretende com uma arquitetura de microsserviços é que se tenham microsserviços bastante coesos e estáveis, para se alcançar o conceito de cada microsserviço poder ser independentemente instalável. Para isto, cada um deve providenciar uma *interface* estável, que não seja constantemente mudada [Newman (2019)].

2.2.4 *Comunicação entre microsserviços*

James Lewis e Martin Fowler introduziram um conceito que explica como devem ser as comunicações entre microsserviços. Este conceito é originalmente conhecido por *"smart endpoints and dumb pipes"* [Fowler and Lewis (2014)], em tradução livre: "endpoints inteligentes e comunicação simples", que significa que não se deve colocar muita complexidade no transporte das mensagens, tal como roteamento, transformação de mensagens, entre outros. A comunicação entre microsserviços deve ser o mais simples possível, enquanto toda a lógica de tratamento das mensagens se deve encontrar dentro de cada um.

Os microsserviços podem comunicar sobre vários protocolos de comunicação, de entre os quais se destacam:

REST

REST (REpresentational State Transfer) [Fielding and Taylor (2000)] define-se como um estilo arquitetural que consiste num conjunto de normas aplicadas a componentes e dados de um sistema, disponibilizando funcionalidades por parte deste.

Em REST todos os componentes apresentam a mesma *interface*, com um conjunto de operações fixas e universais, tais como GET, PUT, POST, DELETE, entre outras, que possibilitam a interação com outro sistema, possibilitando que os sistemas possam ser implementados independentemente uns dos outros, sem ser necessário estar a estabelecer protocolos entre eles [Doyle et al. (2021)].

REST possui a característica de não necessitar de estado indicando que cada sistema não necessita de saber nada sobre o outro que lhe faz o pedido, nem vice-versa [Fielding and Taylor (2000)].

Cada sistema a utilizar REST é constituído por um conjunto de recursos. Estes recursos representam qualquer objeto, documento ou informação que cada sistema armazena e que pode ser acedido por outros. Cada recurso tem um identificador único, podendo estes ser unicamente acedidos através da *interface* REST disponibilizada por cada sistema. Normalmente estes recursos nunca são disponibilizados, mas sim representações destes que podem ser devolvidas em vários formatos. A mais popular é JSON (Javascript Object Notation) devido à sua fácil compreensão e por ser agnóstico em termos de linguagem.

Uma das desvantagens do REST prende-se pela sua comunicação ser síncrona [Doyle et al. (2021)], o que faz com que o sistema que invoca o *endpoint* tenha que ficar à espera, bloqueado até receber a resposta. Pode ser problemático caso no sistema invocado tenha ocorrido uma falha ou se a comunicação através da rede estiver mais lenta.

RPC

RPC (Remote Procedure Call) é uma técnica que permite realizar uma chamada local a uma função que na verdade é executada num outro sistema remoto. Em vez de se aceder remotamente a um sistema, faz-se a chamada localmente, sendo que esta esconde os detalhes da comunicação remota [Matturro (2020)].

Durante uma chamada remota acontecem os seguintes passos:

- O sistema invoca uma função local que é tratada por um *stub*, uma *interface* que permita a comunicação com o sistema remoto, que coexiste localmente com o sistema.
- Este *stub* transforma os parâmetros da função para uma mensagem compreendida pelo sistema remoto.
- Esta mensagem é passada através da rede para o sistema remoto.
- No sistema remoto esta mensagem é entregue ao *skeleton* do sistema que está encarregue de extrair a informação da mensagem e chamar no sistema remoto a função pretendida com os parâmetros que vieram na mensagem.
- Completada a execução é retornado o resultado ao *skeleton* que o transforma numa mensagem e envia através da rede para o *stub* do sistema que lhe fez o pedido.

- O *stub* do sistema recebe o resultado e extrai o conteúdo da mensagem, devolvendo o resultado ao sistema.

Este processo está apresentado na Figura 6.

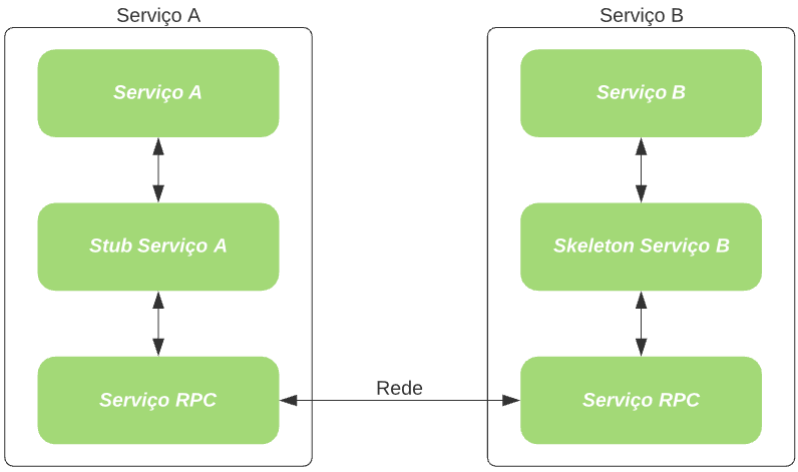


Figura 6: Comunicação RPC.

Como alguns exemplos de RPC temos Java RMI e Protocol Buffers .

Message Brokers

Um *message broker* é um *middleware* que permite aplicações comunicarem entre si e trocarem informação. Este transforma uma mensagem que recebe do remetente para uma mensagem que o destinatário perceba, possibilitando que ambas as partes possam ser desenvolvidas em linguagens e tecnologias diferentes, atuando como intermediário entre aplicações. Este valida as mensagens, armazena-as, faz roteamento destas e entrega-as ao destino devido [IBM (2020b)].

Uma das vantagens dos *message brokers* deve-se ao facto de permitir uma comunicação assíncrona, o que faz com que o sistema a enviar uma mensagem não tenha que ficar bloqueado à espera da resposta. Estes permitem que o remetente envie uma mensagem para um destinatário sem saber a localização deste, se estão ou não ativos, ou quantos outros sistemas existem. O sistema apenas tem conhecimento do *message broker* sendo de lá que recebe e envia mensagens, permitindo um maior desacoplamento entre sistemas, podendo até ser substituído por uma nova versão e os sistemas que interagem com este continuam operacionais, não necessitando de nenhumas modificações [IBM (2020b)].

A um *message broker* está muitas vezes associada uma ou várias *message queues* que permitem o armazenamento e ordenação de mensagens até que os destinatários processem estas mensagens.

Existem alguns modelos de *message broker*, de entre os quais temos:

- **Ponto a Ponto:** Este modelo é utilizado em situações em que existe uma relação um para um entre o remetente e o destinatário. Cada mensagem na *queue* é enviada unicamente para um destinatário e consumida uma única vez.

- **Publicação/Subscrição:**

  Neste modelo, um remetente publica uma mensagem para um tópico, sendo que vários destinatários podem subscrever a esse tópico e consumir as mensagens para lá enviadas. Neste caso existe uma relação de um para muitos.

  Um sistema pode-se subscrever a vários tópicos e publicar para vários tópicos.

Comunicação baseada em Eventos

Outra forma de comunicação entre aplicações é uma comunicação baseada em eventos. Neste padrão arquitetural as aplicações atuam como produtoras ou consumidoras de eventos, podendo até desempenhar os dois papeis [IBM (2020a)].

Um evento é um registo de algo que aconteceu, de uma mudança de estado. Os eventos são imutáveis, não podem ser modificados nem eliminados e encontram-se por ordem de criação [Jansen and Saladas (2020)].

Quando uma aplicação executa localmente uma ação ou uma mudança de estado que pretende que outra aplicação tenha conhecimento, esta produz um evento, representando essa ação ou mudança, que a outra deva tomar conhecimento. Essa outra aplicação deve estar subscrita a estes eventos para os consumir e posteriormente processar esses eventos, executando uma ou mais ações [Jansen and Saladas (2020)]. As aplicações possuem um maior desacoplamento, visto que permite que comuniquem entre si de forma assíncrona sem que nenhuma delas necessite de qualquer conhecimento sobre a outra.

Existem duas formas de transmitir eventos:

- **Event Messaging (Publicação/Subscrição):**

  Este modelo é o apresentado na secção 2.2.4.

- **Event Streaming:**

  Neste modelo, existe uma *stream* onde os eventos são publicados. Um produtor gera um evento e coloca-o na *stream*. Um consumidor subscreve uma ou mais *streams*, contudo em vez de receber e consumir todos os eventos publicados na *stream*, este consegue apenas consumir os eventos que pretende [IBM (2020a)].

  A diferença desta forma para a anterior é que os eventos permanecem na *stream* mesmo após serem consumidos. Os consumidores podem subscrever-se a qualquer *stream* em qualquer altura do tempo, podendo consumir eventos que tenham ocorrido antes da sua subscrição.

2.2.5 *Vantagens sobre uma arquitetura monolítica*

A arquitetura de microsserviços possui várias vantagens, sendo muitas delas adquiridas de sistemas distribuídos. No entanto, muitas destas vantagens atingem um maior grau de benefício devido à definição dos microsserviços e das suas fronteiras. Combinando os conceitos de ocultação de informação, abordado na secção 2.2.1 e de *domain-driven design*, abordado na secção 2.2.1 e 2.2.1 com os de sistemas distribuídos, a arquitetura de

microsserviços consegue possuir mais vantagens do que um simples sistema distribuído [Newman (2021)], apresentando-se a seguir em comparação com uma arquitetura monolítica.

- **Resiliência**:

  Numa aplicação monolítica se um componente falhar toda a aplicação falha.

  Uma forma de melhorar este aspeto é escalar a aplicação horizontalmente, isto é, adicionar mais máquinas onde a aplicação corre para que no caso de falha da máquina primária uma destas assuma o cargo e continue com a aplicação disponível. O problema é que caso a máquina primária falhe devido a um erro de um componente, a aplicação irá sempre falhar mesmo que outras máquinas tomem o seu lugar.

  Com microsserviços temos unidades que são independentes uns dos outros. A falha de um microsserviço não leva à falha de toda a aplicação. A aplicação continua a correr, porém, o microsserviço que falhou estará temporariamente indisponível para os outros da aplicação. Na situação desse microsserviço ter um erro que faça com que este falhe repetidamente é muito mais fácil de o corrigir e não terá tanto impacto na aplicação, visto que apenas esse será compilado e instalado novamente [Newman (2015)].

  É importante mencionar que a adoção de microsserviços não implica que tenhamos necessariamente maior robustez. Separar as funcionalidades da aplicação em microsserviços mais pequenos e colocar estes em processos separados não aumenta por si só a robustez da aplicação. Para a alcançar, deve-se adotar mecanismos que diminuam ao máximo a falha destes microsserviços ou perda de mensagens. É possível colocar os vários microsserviços a correr atrás de um balanceador de carga e também adotar um mecanismo de comunicação entre microsserviços que permita que mensagens não sejam perdidas caso algum falhe [Newman (2019)].

- **Escalonamento**

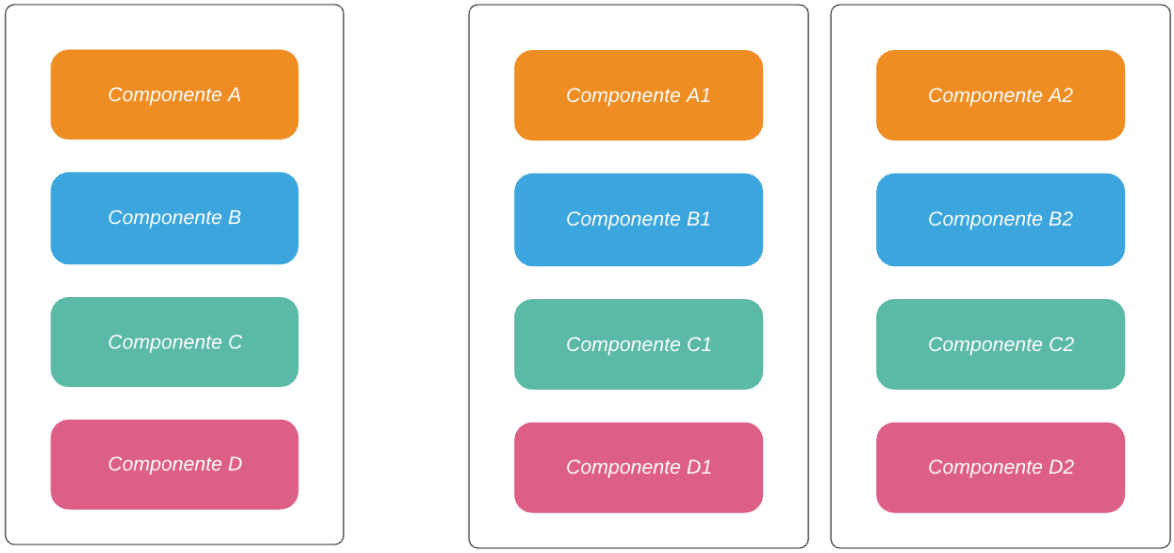
  Na Figura 7a consegue-se observar uma aplicação monolítica composta por quatro componentes que não está escalada. Visto que a única forma de escalar um monolítico é escalar a aplicação na totalidade, temos na Figura 7b um exemplo de escalonamento de uma aplicação monolítica. Neste exemplo colocou-se mais uma instância da aplicação.

  Numa aplicação em microsserviços é possível escalar independentemente [Bruce and Pereira (2018)].

  Imagine-se uma aplicação composta pelos quatro microsserviços apresentados na Figura 8 que comunicam entre si. Alguns destes podem estar a ter um tempo de resposta maior do que esperado, sendo necessário escalar.

  Como se pode observar pela Figura 9 cada microsserviço foi escalado conforme as suas necessidades. Nesta consegue-se observar que o microsserviço B foi escalado três vezes porque recebe muitos pedidos, ao invés do microsserviço D que só foi escalado uma vez, não recebendo provavelmente muitos pedidos ou que não possui muita carga computacional na execução destes.

- **Instalação**:



(a) Sem escalonamento (b) Com escalonamento

Figura 7: Escalonamento de uma arquitetura monolítica.

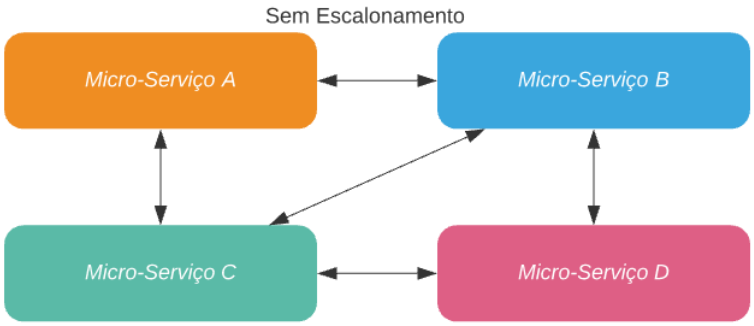


Figura 8: Arquitetura de microsserviços sem escalonamento.

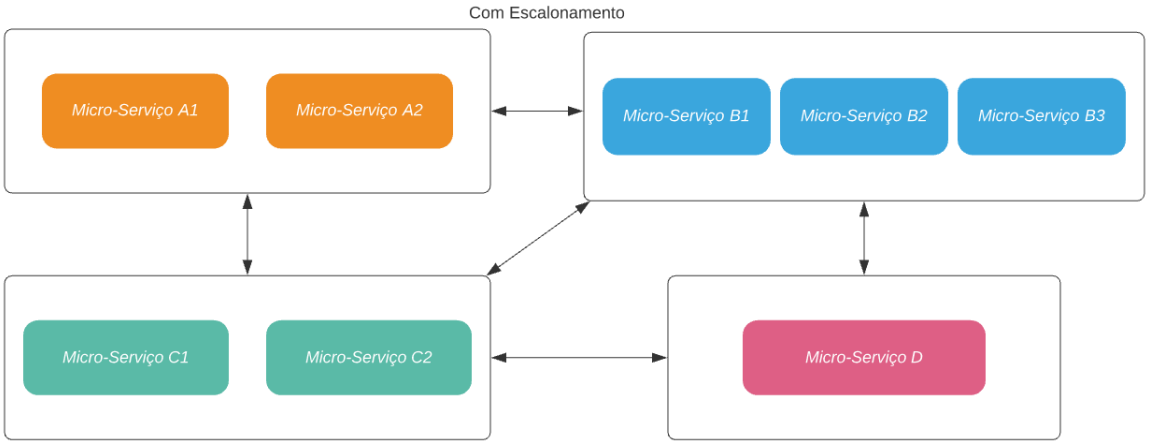


Figura 9: Arquitetura de microsserviços com escalonamento.

Numa aplicação monolítica a mudança de uma linha de código leva a que toda a aplicação precise de ser novamente compilada e instalada para que as mudanças sejam efetuadas, o que leva a uma instalação com um impacto enorme e também de alto risco, sendo que na eventualidade de a nova mudança ter um erro que tenha passado despercebido, será necessário despender mais tempo para corrigir o problema ou retroceder nas mudanças, levando a um maior tempo de inatividade da aplicação [Richardson (2018)].

Na arquitetura de microsserviços, a mudança num microsserviço não implica a instalação de toda a aplicação. Pode-se efetuar mudanças a um microsserviço e instalar sem ter impacto nos restantes, levando a uma instalação mais rápida e permitindo que novas funcionalidades sejam entregues ao cliente mais rapidamente [Newman (2015)].

Caso ocorra algum problema com a nova instalação, acontece apenas a um microsserviço isolado de todos os outros, sendo facilmente revertida caso necessário.

- **Substituível**:

  Numa arquitetura monolítica toda a funcionalidade encontra-se num mesmo lugar, levando a uma maior dificuldade e risco aquando da necessidade de querer modificar alguma parte do código [Richardson (2018)].

  Na arquitetura de microsserviços, o custo de trocar este microsserviço por uma implementação nova ou até apaga-lo é muito menor e mais fácil de gerir, não levando a conflitos com os outros [Newman (2015)].

- **Adoção de novas tecnologias**

  Como referido anteriormente cada microsserviço pode ser escrito numa tecnologia diferente da dos restantes, conseguindo assim tirar o melhor partido que uma tecnologia oferece para a sua implementação, de forma a ter o melhor desempenho possível [Nadareishvili et al. (2016)].

  Numa aplicação monolítica isto não acontece. Todos os componentes da aplicação devem ser escritos na mesma tecnologia [Richardson (2018)].

- **Estrutura da Equipa**

  A existência de microsserviços que são independentes e que estão em processos diferentes permite que várias equipas possam trabalhar em simultâneo para a mesma aplicação [Richardson (2018)].

  Para além disto, sendo as equipas mais pequenas e focadas num único propósito, torna-as mais autónomas, pois sendo pequenas em número aumenta a facilidade de se comunicarem e de não se atrapalharem em termos de escrita de código [Newman (2019)].

- **Entregas mais rápidas**

  A utilização de microsserviços permite maior rapidez na mudança de funcionalidades e na entrega de novas funcionalidades ao cliente [Newman (2019)].

  Se existirem muitas mudanças necessárias a efetuar à aplicação, tendo os microsserviços independentes entre si, é mais fácil fazer modificações e voltar a instalar sem que toda a aplicação tenha que reiniciar [Newman (2015)].

2.2.6 *Desvantagens*

A arquitetura de microsserviços traz várias vantagens, como se pode constatar na secção anterior. Contudo, possui também desvantagens, apresentadas a seguir.

- **Complexidade**

  Uma arquitetura de microsserviços é um sistema distribuído, composto por vários microsserviços em processos diferentes, que comunicam através da rede que pode ter atrasos e até falhas. É por isso mais complexo o desenvolvimento de aplicações onde é necessário ter preocupações com a entrega de mensagens, com as ligações entre microsserviços e a gestão de todas as base de dados [Richardson (2018)].

- **Difícil de Testar**

  Uma funcionalidade de uma aplicação em microsserviços pode implicar a interação entre vários microsserviços, o que pode dificultar o teste destas funcionalidades na totalidade, pois estes encontram-se separados.

- **Experiência de desenvolvimento**

  À medida que se têm mais microsserviços, a experiência de desenvolvimento começa a diminuir. A maioria dos ambientes de desenvolvimento, como JVM (Java Virtual Machine), limitam o número de processos que podem estar a correr numa única máquina. É possível correr talvez quatro ou cinco processos no mesmo computador sem problemas, mas dez ou vinte começa a ser mais complicado [Newman (2021)].

  Uma solução passa por começar a desenvolver na *cloud*, onde os programadores deixam de poder desenvolver localmente. Outra solução passa por limitar o número de microsserviços que cada programador necessita de trabalhar [Newman (2021)].

2.2.7 *Desafios*

A adoção de uma arquitetura de microsserviços para o desenvolvimento de um produto de *software* leva a alguns desafios que não apareceriam no caso de se ter adotado uma arquitetura monolítica.

Alguns desses desafios são:

- **Definição de um microsserviço**

  Quando se começa a desenvolver uma aplicação em microsserviços é necessário pensar que microsserviços vão existir, o que cada um deve ter, quais as suas fronteiras e o que expor destes [Bruce and Pereira (2018)].

  Esta dificuldade pode ser ultrapassada recorrendo à definição de *Bounded Context* como referido na secção 2.2.1. Contudo, mesmo com esta definição, a limitação das fronteiras de um microsserviço é sempre uma tarefa mais complicada e que se não for bem definida pode levar a complicações futuras.

- **Comunicação entre microsserviços**

  Sendo a arquitetura de microsserviços composta por microsserviços que se encontram instalados em processos diferentes é necessário que estes comuniquem entre si. Esta comunicação necessita de ser efetuada através da rede. A comunicação através da rede não é instantânea, o que significa que é necessário a preocupação com a latência e eventuais perdas de mensagens. Além disso, é necessário considerar que estas latências possam variar, o que leva a uma maior dificuldade na definição dos comportamentos dos microsserviços [Newman (2019)].

- **Falha de microsserviços**

  Um microsserviço pode falhar e desligar-se, deixando de responder a pedidos ou então começar a comportar-se de maneira diferente da que devia, podendo levar à indisponibilidade de algumas funcionalidades do sistema. É necessário prevenir a falha de um microsserviço, podendo-se escalar horizontalmente, onde se adicionam mais máquinas que disponibilizam o mesmo microsserviço [Newman (2019)].

- **Escalonamento dinâmico**

  Pode haver momentos do ciclo de vida de um produto de *software* em que este pode ter uma grande adesão e outros em que não. Por isso, um dos microsserviços pode ter a necessidade de em alguma situação ter mais máquinas, de escalar horizontalmente para fazer face ao crescimento de utilizadores. Mais tarde essa adesão pode diminuir não havendo necessidade de ter tantas máquinas, podendo assim libertar algumas.

  A complexidade aparece quando se acrescentam novas máquinas para um dos microsserviços e se quer que estas sejam efetivamente usadas, isto é, que estejam também a ser usadas pelo *load balancer*, de modo a que a carga seja distribuída igualmente por todas.

- **Visibilidade**

  A ocorrência de um problema na aplicação pode ser mais difícil de identificar, visto que temos vários microsserviços em processos separados [Bruce and Pereira (2018)].

  Um pedido à aplicação pode envolver vários microsserviços, por isso na ocorrência de uma eventual falha é necessário identificar qual o microsserviço que originou o erro e porquê, para que o problema possa ser corrigido.

  É necessária por isso a existência de um *log* centralizado onde se possa verificar qual a origem de um problema.

  Também pode ser necessário monitorizar todos os microsserviços de forma centralizada, para verificar se algum deles está em baixo e também o desempenho de cada um.

- **Microsserviço muito utilizado**

  Pode existir um microsserviço que seja muito utilizado por todos os outros, estando todos esses dependentes deste. É necessário prevenir que este microsserviço esteja inoperacional, porque pode levar à indisponibilidade de quase toda a aplicação.

É necessário dotar a aplicação de mecanismos de tolerância a faltas e também escalar estes microsserviços muito consumidos para aumentar a disponibilidade e tempo de resposta da aplicação.

Concluindo, percebe-se que a utilização de microsserviços implica ter alguns desafios no desenvolvimento, instalação e manutenção da aplicação, alguns deles adquiridos dos sistemas distribuídos.

Para conseguir as vantagens apresentadas é necessário saber como automatizar todo o processo de instalação, teste e monitorização.

2.2.8 *Decisão arquitetural*

Em algumas situações, escolher uma arquitetura monolítica para o desenvolvimento de uma aplicação não é a escolha mais acertada.

O início do desenvolvimento de um produto de *software* é o momento exato para se começar a pensar nos diferentes microsserviços necessários e na separação destes [Newman (2015)].

Quando se constrói um monolítico a pensar no futuro de separar para microsserviços é errado pensar que existem componentes dentro deste monolítico prontos a ser facilmente separados e colocados em microsserviços. Até pode ser verdade que os limites dos componentes estejam bem definidos e se saibam quais se devem separar, porém, estes estão normalmente muito acoplados e cheios de ligações entre eles. Estes componentes estão todos a comunicar através de abstrações da mesma tecnologia que usam, partilham objetos, modelos e fazem parte do mesmo modelo de persistência, partilhando a mesma base de dados [Tilkov (2015)].

A escolha de uma arquitetura de microsserviços deve ter em conta:

- A rapidez com que se pretende a aplicação desenvolvida.

  Com a arquitetura de microsserviços consegue-se um desenvolvimento mais rápido, visto que a equipa de desenvolvimento pode dividir-se em subgrupos e desenvolver os microsserviços de forma autónoma e independente dos outros [Bruce and Pereira (2018)].

  Todavia, esta vantagem só acontece se se contemplar o próximo ponto.

- A familiarização com microsserviços.

  Se se pretende um desenvolvimento rápido com esta arquitetura é necessário conhecimento e experiência em microsserviços. Se a equipa de desenvolvimento não possui conhecimentos nesta área, torna-se difícil e custoso o desenvolvimento da aplicação. O tempo de entrega da aplicação para o cliente é muito maior e a resolução de eventuais problemas que apareçam serão muito mais difíceis de alcançar [Bruce and Pereira (2018)].

- A necessidade de alguma parte da aplicação ser mais eficiente.

  Pode existir algum componente que se prevê que seja mais consumido pelos utilizadores e tenha assim um maior volume de acesso. Uma aplicação onde a funcionalidade principal seja a procura por algo que a aplicação oferece, pode ser problemático não ter esse componente de procura o mais eficiente possível.

A utilização de microsserviços neste caso é útil, dado que esse componente mais utilizado pode ser um microsserviço que pode ser escalado independentemente dos outros e assim aumentar o desempenho e a eficiência deste [Indrasiri and Siriwardena (2018)].

2.3 SERVICE-ORIENTED ARCHITECTURE (SOA)

Ao falar de microsserviços é necessário falar também sobre Service-Oriented Architecture (SOA), porque microsserviços pode ser considerado um tipo de *service-oriented architecture* [(Newman, 2021, p. 5)].

SOA é uma arquitetura de *software* que define o desenvolvimento de uma aplicação como uma composição de componentes de *software* que se encontram separados [IBM (2019b)]. Com esta abordagem pretende-se que se construa uma aplicação utilizando serviços já desenvolvidos, totalmente independentes que se encontram disponíveis a fornecer as suas funcionalidades, de modo a construir a nossa aplicação de forma mais desacoplada, sem necessidade de voltar a desenvolver outra vez um componente já existente [Josuttis (2007)]. O objetivo é promover a reutilização destes serviços por outras aplicações [Javed (2019)].

Imagine-se que já existe um serviço que está encarregue de fazer autenticação de um utilizador através de APIs externas, como Google Authenticator. Em SOA, é utilizado este serviço já implementado ao invés de reescrever um componente que faça essa funcionalidade.

Nesta arquitetura um serviço é um processo completamente separado encarregue por uma tarefa que comunica com outros serviços através da rede usando protocolos de comunicação como REST, SOAP, entre outros. Existem dois tipos de serviços, os que fornecem e os que consomem. Um serviço pode no entanto agir das duas formas [IBM (2019b)]. A comunicação entre estes serviços é feita através de *Enterprise Service Bus* (ESB).

Enterprise Service Bus é um *middleware* que permite a integração de vários sistemas de forma a estes poderem comunicar entre si [Josuttis (2007). ESB permite aos serviços conectarem-se ao *bus*, podendo subscrever para receber um tipo de mensagens e publicar mensagens neste, sendo que o ESB fica encarregue de fazer o roteamento das mensagens [Schmidt et al. (2005)]. Consegue-se o desacoplamento por parte dos serviços, permitindo que estes tenham uma comunicação entre eles sem necessitarem de conhecimento acerca do outro serviço [IBM (2019a)].

SOA surgiu como uma evolução dos sistemas distribuídos e uma alternativa às aplicações monolíticas que acabavam por ser difíceis de manter, promovendo a reutilização de serviços e funcionalidades [IBM (2020c)].

Tendo como objectivo facilitar a manutenção e escrita de *software*, é possível substituir um serviço por outro que disponibilize as mesmas funcionalidades sem que outros serviços o saibam, desde que a semântica do serviço não mude muito.

2.3.1 *Diferença entre SOA e Microsserviços*

A maior diferença entre SOA e microsserviços encontra-se na forma como as ligações entre serviços são feitas, o âmbito da sua utilização e o propósito de cada uma.

SOA é usado em contexto empresarial, permitindo que aplicações/serviços desenvolvidos em contexto empresarial possam ser expostas, cada uma correspondendo a um conjunto de funcionalidades [Erl (2007)]. Desta maneira permitimos que outras aplicações possam reutilizar as funcionalidades de outras aplicações/serviços.

Microsserviços é usado em contexto aplicacional, permitindo que uma aplicação possa ser particionada em microsserviços menores que são independentes de modo a atingir vantagens mencionadas anteriormente. Não define como as aplicações devem comunicar entre elas [Richards (2015)].

SOA está focado em tentar maximizar a reutilização de funcionalidades já implementadas noutros serviços, enquanto que microsserviços foca-se mais no desacoplamento entre componentes [IBM (2020c)].

Em SOA todos os serviços podem partilhar a mesma base de dados, sendo que a decisão é de quem constrói a aplicação, enquanto que em microsserviços cada microsserviço possui a sua base de dados [IBM (2020c)].

3

MIGRAÇÃO PARA MICROSSERVIÇOS

A maioria das aplicações criadas acabam por nunca serem imutáveis. No processo de desenvolvimento de uma aplicação é necessário considerar, que mesmo quando a aplicação é entregue ao cliente, que esta precisará de ser mudada e adaptada [Newman (2015)].

Numa aplicação é quase impossível prever todas as funcionalidades e requisitos da aplicação. Por esta razão é necessário preparar a aplicação para que esta seja modificável sem grande impacto.

Um monolítico cresce ao longo do tempo, adquirindo novas funcionalidades e cada vez mais ligações entre componentes, o que torna o código mais acoplado. Quando houver a necessidade de mudar uma linha de código, esta modificação terá grande impacto na aplicação, pois leva à instalação de toda a aplicação [Fowler and Lewis (2014)]. A adoção de uma arquitetura de microsserviços pode facilitar na mudança e adição de novas funcionalidades, bem como uma entrega mais rápida de funcionalidades para o cliente. O maior problema em migrar de uma arquitetura monolítica para uma de microsserviços, prende-se na definição do que deve ser cada microsserviço e na mudança da forma de comunicação entre estes [Newman (2015)].

A adoção de uma arquitetura monolítica deve ser uma decisão consciente. Uma decisão que parte do princípio de se pretender algo que não se consegue com o sistema arquitetural que já se possui [Newman (2021)].

3.1 MIGRAÇÃO INCREMENTAL

> "If you do a big-bang rewrite, the only thing you're guaranteed of is a big bang."
>
> (Martin Fowler)

A migração para microsserviços não deve ser realizada abruptamente. É aconselhado realizar mudanças incrementais, extraindo funcionalidades uma a uma. Esta migração de forma incremental ajuda a que se aprenda mais sobre microsserviços à medida que se efetua esta migração e limita o impacto de se acabar por realizar algo mal [Newman (2019)].

> "Think of our monolith as a block of marble. We could blow the whole thing up, but that rarely ends well. It makes much more sense to just chip away at it incrementally."[1]
>
> (Sam Newman)

1 Sam Newman. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2nd edition, August 2021

Transformar um monolítico numa arquitetura de microsserviços pode ser muito dispendioso, principalmente se este estiver bastante acoplado. Efetuar todas as mudanças em simultâneo, leva a que seja difícil de perceber se a migração está a ser concretizada de forma correta ou não. É muito mais fácil particionar esta mudança em pequenas etapas, sendo cada uma focada numa única ação. É preciso perceber que vão acontecer erros e que algo pode ser efetuado incorretamente. A adoção de uma migração incremental permite que sejam reduzidos os erros a efetuar e a dimensão destes [Newman (2019)].

É aconselhado começar por separar apenas um ou dois componentes da aplicação em microsserviços, proceder à sua instalação e avaliar se tudo correu bem e o impacto que esta mudança trouxe à aplicação [Newman (2019)].

O custo de modificar código e movê-lo de um lado para o outro é relativamente pequeno. Existem variadas ferramentas e editores de texto que nos ajudam nesta tarefa e que possibilitam retroceder nas ações realizadas. Contudo, modificar uma base de dados é muito mais trabalhoso e reverter estas ações ainda mais. O mesmo se aplica caso se tenha que reescrever uma API que é consumida por muitos outros microsserviços, visto que é necessário modificar todos os outros para ficarem atualizados com esta mudança [Newman (2021)].

Uma boa forma de começar a pensar numa migração parte pelo desenho da arquitetura de como se pretende a aplicação. Desta forma conseguimos visualizar os microsserviços que se pretende que existam e que ligações existem entre eles. É preciso imaginar alguns exemplos de funcionalidades, tal como o registo de utilizadores e perceber se existem dependências cíclicas. Caso existam é porque algo não está correto e deve ser pensado como remover estes ciclos. O mesmo se aplica no caso de se perceber que existam dois microsserviços que comunicam muito entre si, indicando talvez que estes devam formar um só [Newman (2019)].

3.2 SEPARAÇÃO DO CÓDIGO

O primeiro passo na transformação de um monolítico para microsserviços passa pela organização do código.

Na maioria das vezes a principal barreira em migrar de um monolítico para microsserviços é porque o código se encontra pouco coeso e não está organizado à volta de domínios [Newman (2019)].

Seam, um conceito introduzido no livro *Working Effectively with Legacy Code* [Feathers (2004)], define-se como uma parte do código que pode ser isolada do resto, sem ter impacto no resto do código. É uma parte onde se podem efetuar mudanças sem se terem que mudar outras partes devido a estas mudanças efetuadas [Feathers (2004)]. Estas *seams* podem ser vistas como alternativa para uma boa organização do código, consistindo na organização do código por *namespaces* ou *packages*, dependendo da linguagem utilizada, onde dentro de cada um se encontra código direcionado para um mesmo propósito [Newman (2019).

No monolítico o que se pretende é que se encontrem *seams*, conseguindo-se obter uma melhor organização do código, para que posteriormente se consiga definir mais facilmente os microsserviços [Newman (2015)]. Uma boa identificação destas *seams* pode ser efetuado através da identificação de *Bounded Context* como falado na secção 2.2.1.

Com estes *seams* identificados, o próximo passo é transformá-las em módulos, tornando o monolítico num monolítico modular, como apresentado na secção 2.1.1. Continua-se a ter uma única aplicação que tem que ser

instalada conjuntamente, porém esta unidade é composta por vários módulos que são bem visíveis e podem ser extraídos para microsserviços. A natureza destes módulos depende da tecnologia a ser utilizada. Para uma aplicação em Java o monolítico modular pode consistir em múltiplos ficheiros JAR [Newman (2019)].

No entanto, mesmo com estes *Bounded Context* definidos, o código continua acoplado. Existem *Bounded Context* que acedem a funcionalidades ou objetos de outro. A separação destes para microsserviços deve ser realizada de forma incremental, começando por aquele que possui menos dependências de outros *Bounded Context*.

3.3 PADRÕES DE MIGRAÇÃO - CÓDIGO

Existem várias formas para efetuar a separação do código. Nesta secção serão apresentados alguns padrões que facilitam esta tarefa.

3.3.1 *Strangler Fig Application*

Uma técnica frequentemente usada quando se pretende efetuar uma reescrita de um sistema é a de *Strangler Fig Application* [Fowler (2004)]. Martin Fowler inspirou-se para este padrão num tipo de figueira (*fig*), denominada *strangler fig*. Estas nascem nos ramos mais altos de uma árvore e à medida que vão crescendo, descem pela árvore até que chegam ao solo e se enraízam. Neste processo elas envolvem a árvore que se torna na estrutura de suporte desta figueira. A árvore vai gradualmente morrendo e por último apodrecendo, deixando apenas a nova figueira, que já possui o seu próprio suporte e independente da árvore antiga [Fowler (2004)].

Esta metáfora é uma boa forma de explicar uma técnica de efetuar uma reescrita a um sistema. Cria-se um novo sistema à volta do sistema antigo, deixando este ir crescendo gradualmente à volta do antigo até este ficar obsoleto e deixar de ser necessário [Fowler (2004)].

A ideia é que o sistema antigo e o novo podem coexistir em simultâneo, dando tempo ao novo sistema de crescer e de substituir por completo o antigo. Com isto, procede-se a uma migração incremental para o novo sistema, como referido na secção 3.1, adquirindo todas as vantagens desta. É possível retroceder alguma ação que se tenha feito [Newman (2021)].

A execução do padrão de *strangler fig* consiste em três passos, como se pode observar pela Figura 10. Primeiro é necessário identificar as partes do sistema que se pretende migrar. A seguir é preciso implementar estar partes em microsserviços. Por último, estando este implementado, é necessário redirecionar as chamadas a estas partes no monolítico para serem agora efetuadas ao novo microsserviço criado [Newman (2021)].

Caso esta parte extraída seja também utilizada por outras partes no monolítico é preciso também reescrever estas partes, visto que agora as chamadas não serão locais. Por outro lado, se este novo microsserviço necessitar de alguma funcionalidade que se encontre no monolítico, é preciso realizar mudanças a este para se expor esta funcionalidade ao novo microsserviço criado [Newman (2021)], algo mencionado na secção 3.8.1.

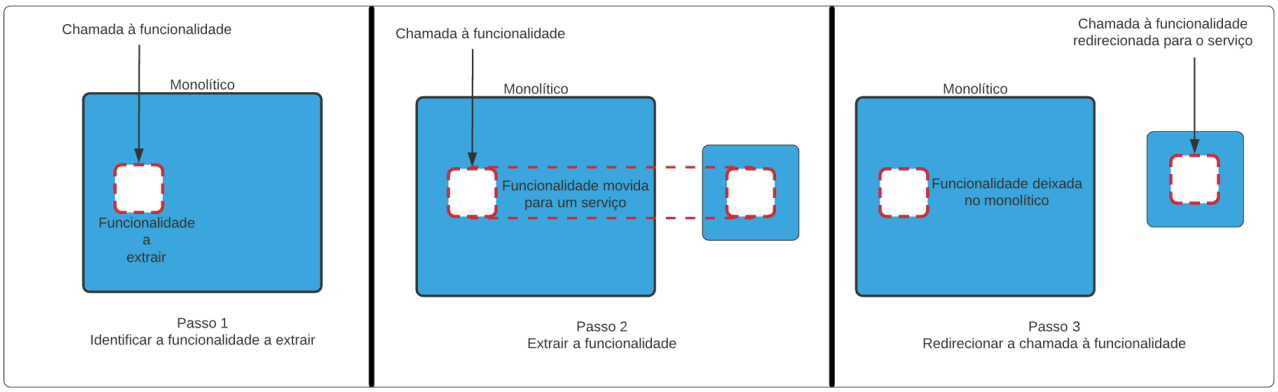


Figura 10: Strangler Fig Application.

Este padrão permite que seja movido e extraído funcionalidade do monolítico para microsserviços sem ter que o modificar nem alterar, sendo uma vantagem quando o próprio monolítico continua a ser desenvolvido e a ser adicionado com novas funcionalidades [Newman (2021)].

3.3.2 *Branch by Abstraction*

Para que o padrão *strangler fig application* possa ser aplicado e bem-sucedido, é preciso que a funcionalidade a extrair não esteja muito acoplada no monolítico, ou seja, devem ser partes que não são tão utilizadas por outras no monolítico [Newman (2019)].

Para extrair estas partes bastante utilizadas é preciso que modificações sejam feitas ao monolítico. Estas mudanças podem ser significantes e disruptivas para outras pessoas a trabalhar no mesmo código em simultâneo [Newman (2019)].

Muitas vezes, para mitigar esta disrupção são usadas *branchs* separadas para realizar estas mudanças. O desafio está quando o trabalho de migração a ser feito nesta *branch* é terminado e necessita de ser fundida com a *branch* principal, o que pode levar a conflitos e ainda mais trabalho. Quanto mais tempo a *branch* existir, maior serão os problemas a enfrentar. O que se pretende é realizar estas mudanças ao código com o menor impacto possível no trabalho que outros programadores estejam a realizar nele [Newman (2019)].

Para estes casos em que se pretende justamente começar por extrair aquelas partes mais enraizadas no monolítico, existe o padrão *branch by abstraction* [(Newman, 2019, p. 104)], que não necessita de recorrer à utilização de *branches* para a realização destas mudanças. Este padrão baseia-se em realizar mudanças ao código já existente, permitindo que várias implementações da mesma parte a modificar/extrair possam coexistir no mesmo código em simultâneo, sem se causar muita disrupção [Newman (2019)].

Este padrão consiste em cinco passos:

1. **Criar uma abstração para a funcionalidade a ser substituída**

   O primeiro passo é criar uma abstração que represente as interações entre o código a ser extraído e as partes que o chamam [Newman (2019)]. Como se pode observar pela Figura 11, a funcionalidade

que se pretende extrair é a representada pela letra C. É criado então uma abstração de C que a própria implementa.

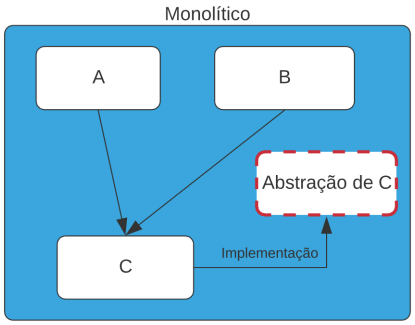


Figura 11: Criar uma abstração.

2. **Mudar as chamadas à funcionalidade para usarem a nova abstração**

   Com a nova abstração criada, é preciso agora redirecionar as chamadas que eram feitas a C para usarem agora a nova abstração, como se pode observar pela Figura 12.

   Este processo envolve uma análise ao código, para encontrar as chamadas efetuadas a esta funcionalidade. Neste ponto nenhuma mudança deve ter sido efetuada ao comportamento e código do sistema [Newman (2019)].

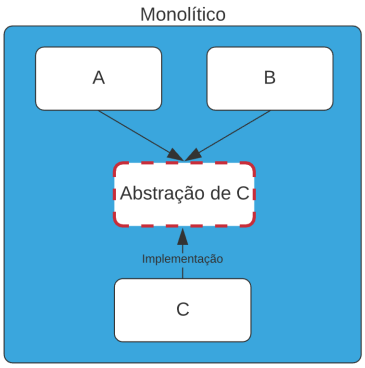


Figura 12: Redirecionar chamadas para a nova abstração.

3. **Criar uma nova implementação da abstração**

   Tendo agora a nova abstração criada é possível começar a desenvolver uma nova implementação desta [Newman (2019)]. No nosso caso, como se pretende uma migração para microsserviços, esta nova implementação realizará chamadas ao novo microsserviço que se desenvolverá, como se pode observar na Figura 13.

   É importante perceber que neste ponto apenas uma das implementações é usada pelo sistema, mesmo existindo duas implementações em simultâneo, da mesma abstração, o que permite ir-se desenvolvendo o novo microsserviço, sendo que este só é ativo e utilizado quando estiver completo [Newman (2019)].

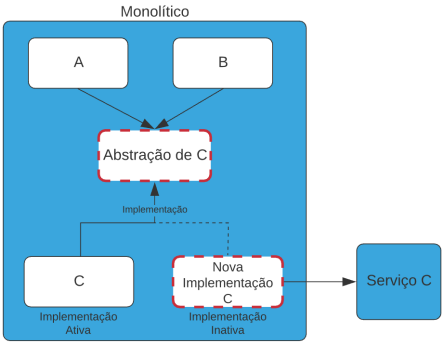


Figura 13: Nova implementação da abstração.

4. **Mudar a abstração para usar a nova implementação**

   Com o novo microsserviço implementado pode-se alterar a abstração para utilizar a nova implementação, ao invés da antiga [Newman (2019)], tal como mostra a Figura 14.

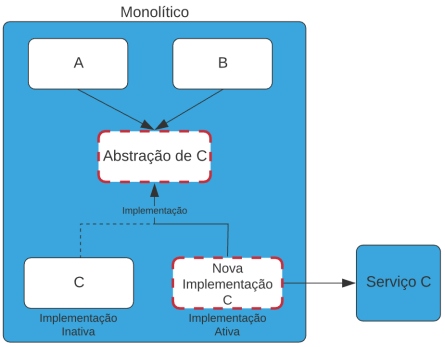


Figura 14: Mudar para usar a nova implementação.

   Idealmente, pretende-se ter um mecanismo que nos permita facilmente mudar de uma implementação para outra. Desta forma pode-se facilmente mudar para a implementação antiga caso algum problema aconteça com a nova [Newman (2019)].

5. **Remover a implementação antiga**

   Com o novo microsserviço a fornecer toda a funcionalidade que a antiga implementação fornecia, pode-se proceder à remoção desta. Visto que esta já não é usada e tem-se a certeza que o novo microsserviço funciona como esperado é possível eliminar esta implementação antiga [Newman (2019), como se pode observar pela Figura 15.

   Por último, com a implementação antiga removida, também temos a possibilidade de remover a abstração criada para este processo, como se pode observar na Figura 16.

   É possível, no entanto, que esta nova abstração tenha melhorado a estrutura e organização do código ao ponto de não se querer remover.

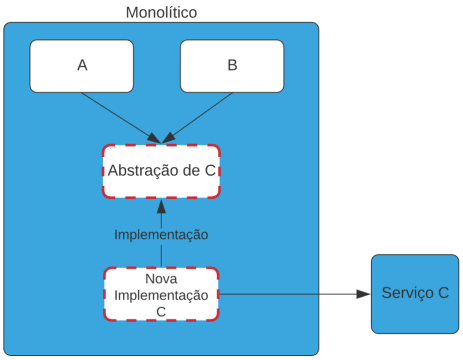


Figura 15: Remover implementação antiga.

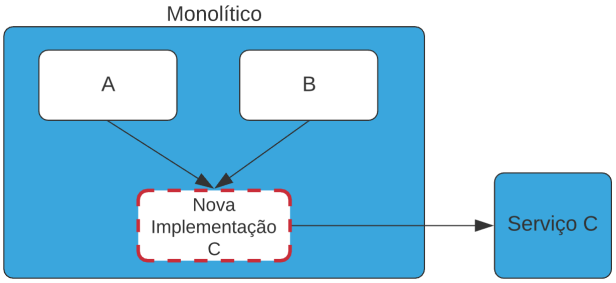


Figura 16: Remover a abstração.

Com este padrão consegue-se progredir e ir adicionando novas funcionalidades à aplicação sem haver disrupções entre a equipa de desenvolvimento, sendo possível instalar as modificações para produção em qualquer altura, dado que a aplicação se encontra sempre funcional. Também não existe a pressão de terminar rapidamente esta extração para um microsserviço, pois é possível modificar a funcionalidade a ser extraída em simultâneo com a migração [Hammant (2007)].

3.3.3 *Verify Branch by Abstraction*

A habilidade de se ter um mecanismo que nos permita escolher a implementação que se pretende, seja ela a antiga ou a nova implementação, é muito útil [Newman (2019)]. Contudo, existe algum risco de haver alguma falha da aplicação quando se muda para a nova implementação, que pode não ter sido detetada antes de se ter efetivamente mudado para esta e eliminado a antiga [Smith (2013)].

O padrão *Verify Branch by Abstraction* é uma variante do padrão *Branch by Abstraction* apresentado na secção 3.3.2. Este é igual, à exceção que adiciona uma fase de verificação por detrás da abstração criada. Desta forma, é possível que no caso da falha da nova implementação se reverta as chamadas para a antiga de forma dinâmica. Além disso, sabemos que a abstração obriga a cada implementação a possuir o mesmo comportamento, porém não consegue garantir a mesma implementação do comportamento em ambas. Esta fase de verificação chama ambas as implementações com os mesmos parâmetros para as várias funcionalidades que expõe e verifica que o resultado é o mesmo. Desta forma, pretende-se mitigar as incompatibilidades entre implementações e saber na

totalidade quando a nova implementação se encontra terminada [Smith (2013)]. Este padrão encontra-se na Figura 17.

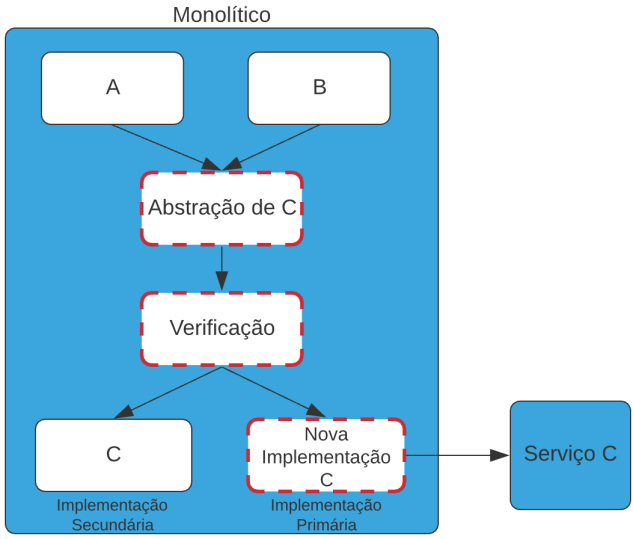


Figura 17: Verify Branch by Abstraction.

Com este mecanismo, diminui-se o custo de falha aquando da mudança para a utilização da nova implementação, visto que quando ocorre a falha de uma chamada à nova implementação é usada a antiga ao invés desta [Newman (2019)].

3.3.4 *Parallel Run*

Os padrões de migração *Strangler Fig Application* 3.3.1 e *Branch by Abstraction* 3.3.2 permitem que tanto a implementação antiga como a nova coexistam, permitindo que se execute ou a antiga do monolítico, ou a nova que já se encontra num microsserviço. Estas também permitem que se possa reverter para a implementação antiga em caso de falha da nova implementação, diminuindo assim o risco desta extração [Newman (2021)].

Com o padrão *parallel run*, em vez de se usar apenas uma implementação, usam-se as duas, permitindo que se compare os resultados para se garantir que ambas as implementações são equivalentes. Apesar de ambas as implementações estarem a receber chamadas às suas *interfaces*, apenas uma delas é considerada como primária num determinado momento e é desta a resposta a enviar da chamada. Normalmente, a implementação antiga é considerada como primária até que a nova implementação se verifique ser de confiança [Newman (2021)]. Os resultados das respostas dos pedidos feitos a ambas as implementações são guardados, para serem analisados e se verificar que são idênticos. Este processo pode ser observado pela Figura 18.

Com este padrão, não se garante apenas que a nova implementação retorna a mesma resposta que a antiga, mas que também apresenta o desempenho pretendido, tal como se demora muito a responder ou se não devolve resposta. Este padrão é tipicamente utilizado nos casos onde a funcionalidade a ser mudada é considerada de risco [Newman (2019)].

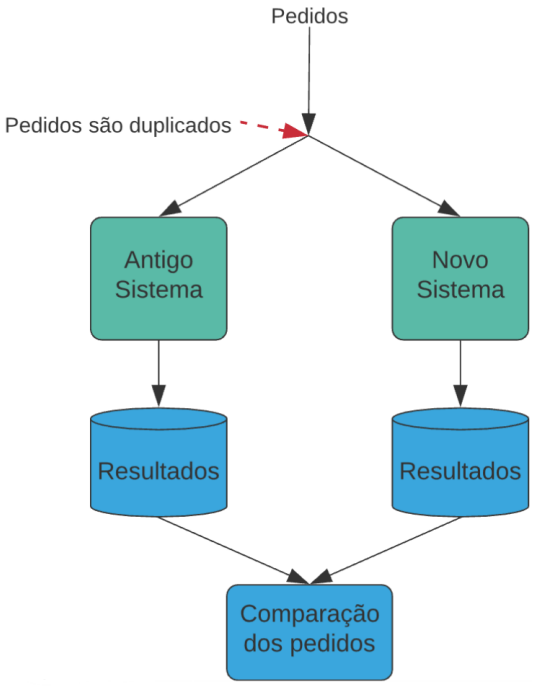


Figura 18: Parallel Run.

Existem situações em que uma ação desencadeia outra, tal como o envio de um correio eletrónico de boas-vindas quando um utilizador se regista numa aplicação. Neste caso, não se quer que o utilizador receba dois correios eletrónicos. Para este caso é preciso que se simule a ação no sistema que não é o primário [Newman (2019)].

Este padrão é uma forma de implementar uma técnica denominada *dark launching*. Com esta técnica, a nova implementação é instalada e testada, sem estar visível nem acessível aos utilizadores [Fowler (2020)].

É importante referir que este padrão não é o mesmo que *canary release* [(Newman, 2019, p. 118)]. Uma *canary release* é uma técnica para reduzir o risco de introduzir uma nova implementação do sistema em produção. Nesta, apenas alguns utilizadores têm acesso ao novo sistema antes de ficar disponível para todos os utilizadores [Sato (2014)]. A ideia é que se a nova implementação tiver um problema, apenas um pequeno conjunto de utilizadores sofre o impacto destes problemas [Newman (2019)].

3.4 BASE DE DADOS PARTILHADA

Antes de se começar a pensar na separação da base de dados, vão ser abordados alguns padrões arquiteturais onde microsserviços partilham o mesmo esquema de base de dados. Apesar de não ser uma boa prática na arquitetura de microsserviços, é uma opção que pode ser temporária.

3.4.1 *Esquema de base de dados partilhado*

Aquando da migração para uma arquitetura de microsserviços, pode-se optar por não se particionar o esquema da base de dados e ter um partilhado por todos os microsserviços, como se pode observar na Figura 19. Esta opção traz menor complexidade, dado que não se separa o esquema da base de dados nem se lida com as dificuldades que possam ocorrer.

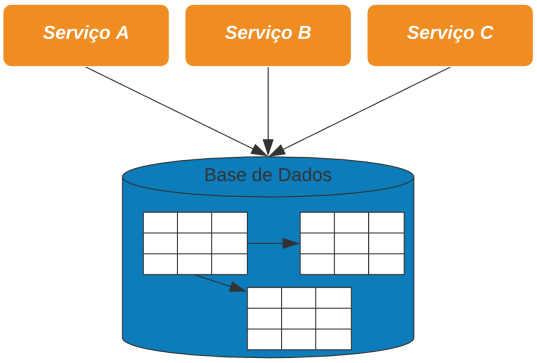


Figura 19: Esquema da base de dados partilhada por vários microsserviços.

No entanto, existem alguns problemas de se usar um esquema de base de dados partilhado entre vários microsserviços. Permite-se que todos tenham acesso à informação que não lhes pertence, perdendo assim a oportunidade de decidir que dados devem ser partilhados e ocultos para outros microsserviços, o que significa que fica complicado de perceber que partes do esquema da base de dados podem ser modificadas de forma segura [Newman (2019)].

É problemático também no sentido que fica confuso de saber quem possui certos dados, implicando uma falta de coesão, pois se na Figura 19 os três microsserviços puderem fazer alguma operação sobre uma mesma tabela, pode acontecer de este comportamento estar diferente em cada um. Mesmo que não esteja diferente, a mudança de comportamento de uma mesma ação que estes façam sobre essa tabela, implica a alteração deste comportamento em mais do que um microsserviço [Newman (2019)].

O uso de um esquema de base de dados partilhados é apropriado para uma arquitetura de microsserviços quando se considera dados estáticos e apenas de leitura. Neste casos onde a estrutura de dados é bastante estável e provavelmente só é modificada por partes administrativas, uma base de dados partilhada pode ser uma boa opção [Richardson (2018)].

3.4.2 *Base de dados com vistas*

Ainda abordando a possibilidade da utilização de um mesmo esquema de base de dados, o uso de vistas pode ajudar a mitigar preocupações relacionadas com acoplamento. Com o uso de vistas, é apresentado a um microsserviço um esquema que é uma projeção restrita de um esquema base. Esta projeção, resultado de uma *query*, consegue limitar os dados visíveis para o microsserviço, ocultando informação que este não deve ter

acesso, permitindo que se possa modificar a base de dados desde que se consigam manter as vistas [Newman (2019)]. Esta opção é apresentada na Figura 20.

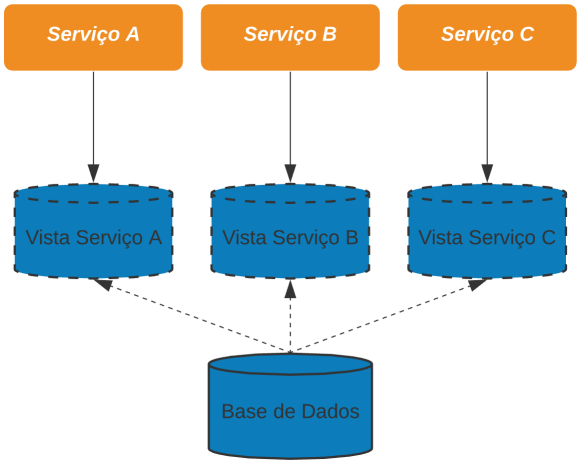


Figura 20: Utilização de vistas para cada microsserviço.

Dependendo da base de dados a utilizar, é possível a opção de criar vistas materializadas. Com uma vista materializada, a vista é previamente calculada, significando que uma leitura a este tipo de vistas não necessita de realizar uma leitura à base de dados, podendo assim ter mais desempenho. É necessário, que estas vistas materializadas estejam atualizadas com a base de dados, de modo a não serem lidos dados desatualizados [Newman (2019)].

Existem, no entanto, limitações ao uso de vistas. Dado que estas são o resultado de *queries* feitas à base de dados, estas permitem apenas leituras. Por outro lado, nem todos os motores de base de dados suportam vistas e ainda existem aqueles que não permitem vistas fora do mesmo esquema da base de dados, o que leva a um maior acoplamento aquando da instalação de algum microsserviço e também a um ponto único de falha [Newman (2019)].

3.4.3 *Encapsulamento da base de dados num microsserviço*

> "Sometimes, when something is too hard to deal with, hiding the mess can make sense."[2]
>
> (Sam Newman)

Pode acontecer de a base de dados ser bastante complicada de modificar, acabando por ser mais vantajoso de a esconder por detrás de um microsserviço [Newman (2019)].

Com este padrão, o que se pretende é encapsular a base de dados num microsserviço, movendo todos os acessos à base de dados para este [Newman (2019)].

Desta forma consegue-se controlar os dados que cada microsserviço pode aceder e quais estão ocultos. Posteriormente é preciso que todos os acessos à base de dados sejam agora redirecionados para o microsserviço,

2 Sam Newman. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2nd edition, August 2021

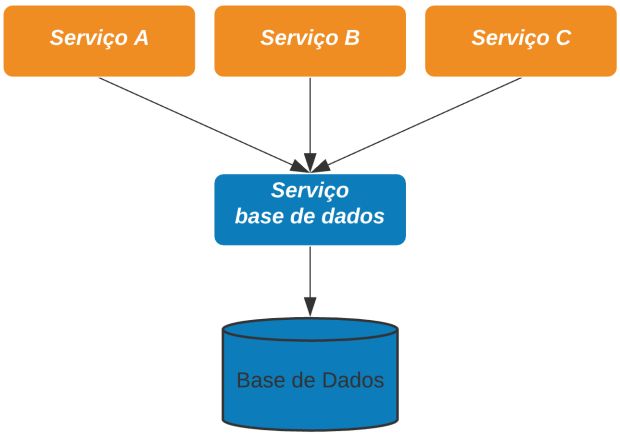


Figura 21: Encapsulamento da base de dados num microsserviço.

mudando de acessos diretos à base de dados para chamadas à API deste. Este padrão possui mais vantagens que o uso de vistas na base de dados, porque pode-se escrever código de forma a se apresentar projeções mais sofisticadas. Além disso, permite também escritas, algo que o uso de vistas não permitia [Newman (2019)].

3.4.4 *Base de dados como um microsserviço*

Em algumas situações, os microsserviços precisam apenas de realizar leituras à base de dados, seja para ir buscar enormes quantidades de dados ou apenas para ir buscar métricas sobre como se desempenha outro sistema. Nestas situações, permitir que outros microsserviços visualizem dados que outro gere na sua base de dados é vantajoso, desde que se separe a base de dados exposta e a gerida pelo próprio microsserviço [Newman (2019)].

Uma das abordagens é criar uma base de dados dedicada para ser exposta apenas para leituras, sendo esta populada quando existem mudanças na base de dados principal [Newman (2019)], como se apresenta na Figura 22.

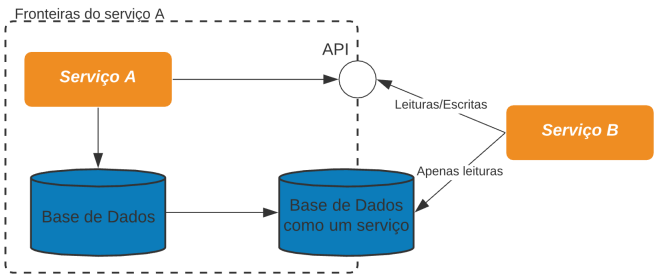


Figura 22: Base de dados como um microsserviço.

Como se pode ver, este padrão é vantajoso para microsserviços que necessitem apenas de realizar leituras.

Este padrão arquitetural tem algumas vantagens. É possível alterar a base de dados principal sem ter que se alterar a base de dados como um microsserviço. As leituras realizadas a esta base de dados como um

microsserviço não implica maior carga à base de dados principal e é possível ter várias para diferentes propósitos [Fowler (2014)]. Contudo, também possui desvantagens. Uma base de dados como um microsserviço necessita de estar atualizada com a base de dados principal [Fowler (2014)].

Em relação ao uso de vistas, este padrão é mais flexível, no sentido que não obriga a que seja usado a mesma tecnologia, podendo-se usar uma base de dados totalmente diferente da base de dados principal [Newman (2019)].

3.5 SEPARAÇÃO DA BASE DE DADOS

Assim como no código, na base de dados também é necessário encontrar partes que possam ser separadas [Newman (2015)]. É preciso perceber que partes efetuam leituras da base de dados e quais efetuam escritas.

Tal como ter o código separado por *namespaces* ou *packages*, o código que representa o acesso à base de dados deve também ser separado e estar junto do código a que pertence. Desta maneira consegue-se perceber que partes da base de dados são acedidas por certas partes de código [Newman (2015)].

A arquitetura de microsserviços funciona melhor quando cada microsserviço possui a sua informação, encapsulando-a e só permitindo certos acessos a esta. Apesar do que se apresentou na secção 3.4, é preciso ter em mente que quando se pretende migrar para uma arquitetura de microsserviços, é preciso separar a base de dados para que cada microsserviço tenha a sua própria, de modo a se obter as vantagens desta transição [Newman (2019)].

Um dos maiores problemas na separação de um monolítico é a base de dados, dado que é preciso considerar consistência entre os dados das várias bases de dados, transações, *joins*, latências, entre outros [Newman (2015)].

3.5.1 *Separação Física e Lógica*

Existem duas formas de realizar a separação do esquema da base de dados.

Existe a separação lógica, em que um único motor de base de dados possui mais do que um esquema de base de dados, como se apresenta na Figura 23.

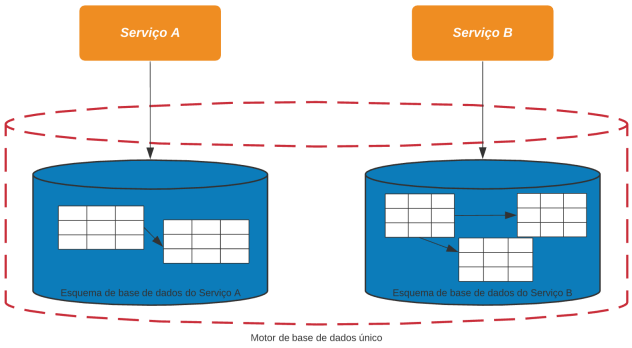


Figura 23: Separação lógica da base de dados.

O uso de uma separação lógica permite um processo mais simples e direto [Newman (2019)]. Contudo, fica-se sujeito a um ponto único de falha, dado que se o motor da base de dados falhar, ambos os microsserviços vão ser afetados.

Por outro lado, podemos ter separação física, tendo estes esquemas da base de dados em motores separados, como se observa na Figura 24.

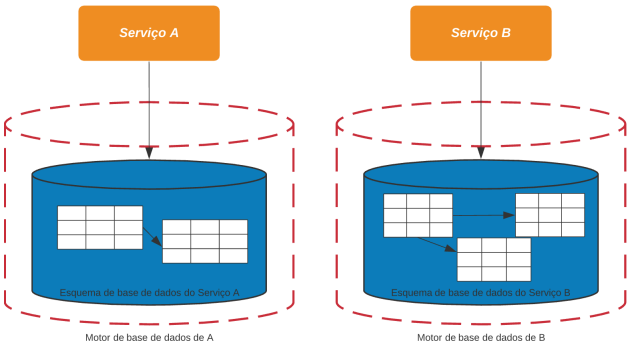


Figura 24: Separação física da base de dados.

O uso de uma separação física permite uma maior robustez ao sistema e ajuda a diminuir a contenção de recursos, melhorando o tempo de resposta e diminuindo a latência [Newman (2019)].

3.5.2 *Organização do código da base de dados*

Existem alguns desafios complexos quando se pensa em separar o esquema da base de dados.

Imagine-se que temos uma aplicação onde utilizadores podem fazer a gestão dos seus carros e realizar encomendas de componentes para os seus carros, tal como gerir informação sobre estes, datas em que algo é preciso ser feito ao carro, tal como uma mudança do óleo ou fazer uma encomenda de produtos para um certo carro, como um rádio, jantes, entre outros. Para isto existem vários componentes, dos quais se destacam um componente que trata de gerir os utilizadores e outro que trata de gerir os carros, como se pode observar na Figura 25.

Estando já o código organizado em componentes separados, pretende-se que o código de acesso à base de dados fique separado por repositórios como se apresenta na Figura 25.

Todavia, continuamos a ter componentes que acedem a tabelas que não lhes pertencem e existem relações entre tabelas, representadas por chaves estrangeiras.

Voltando ao exemplo mencionado anteriormente, consideremos que precisamos de saber quais são os carros de um certo utilizador. Para isso, dado que o componente do utilizador tem acesso à base de dados dos carros, pode ir diretamente à tabela dos carros buscar todos os que tenham como chave estrangeira o identificador do utilizador, como se pode observar pela Figura 26.

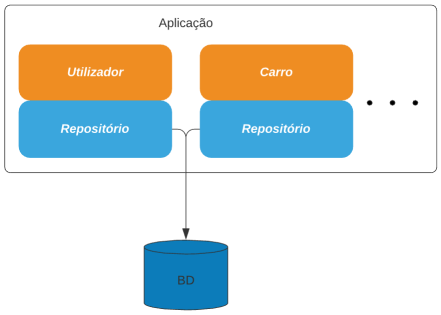


Figura 25: Separação por acesso à base de dados.

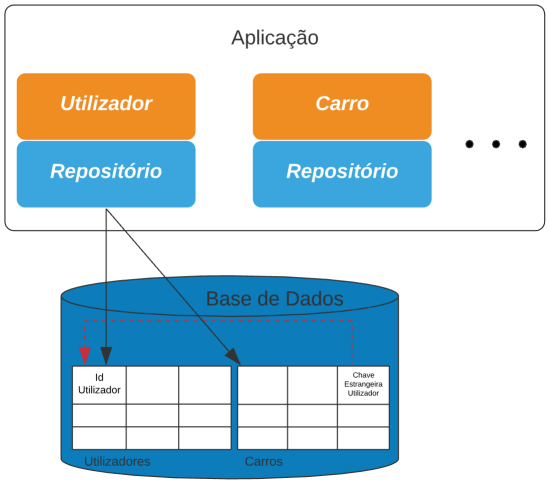


Figura 26: Chave estrangeira entre Utilizador e Carro.

3.5.3 *Remover Chaves Estrangeiras*

Antes de começar a pensar na separação do esquema de base de dados, seja esta separação física ou lógica, é preciso realizar mudanças no próprio esquema. É preciso não permitir que um componente tenha acesso a uma tabela da base de dados que não lhe pertence [Newman (2015)].

Caso se queira saber quais são os carros de um certo utilizador, é possível juntar as duas tabelas através da chave estrangeira e retirar apenas aqueles que são do utilizador pretendido. Como se constata na Figura 26 o componente do utilizador tem acesso à tabela dos carros e existe uma ligação entre a chave estrangeira da tabela dos carros e a chave primária dos utilizadores. Com esta relação entre tabelas, o motor de base de dados assegura a consistência dos dados, porque se existir um registo na tabela dos carros que se refere a um utilizador, é garantido que esse utilizador existe [Newman (2019)].

Se se pretender separar estes dois componentes para microsserviços independentes, é preciso que cada componente tenha apenas acesso às tabelas que lhe pertencem da base de dados e expor os seus dados através de uma API para que outros componentes possam aceder a esses sem ser diretamente à tabela [Newman (2015)], como é demonstrado na Figura 27.

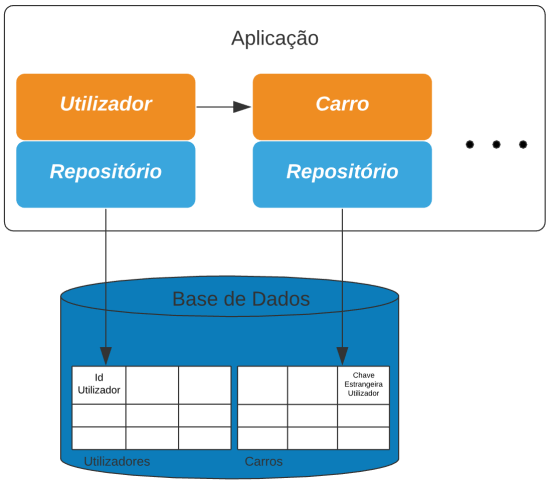


Figura 27: Remoção da chave estrangeira.

Voltando ao exemplo mencionado anteriormente, para saber quais são os carros de um certo utilizador, o componente dos utilizadores tem que encontrar o utilizador pretendido e quando o tiver, tem que ir buscar os carros deste. Nesta situação, em vez de ir diretamente à tabela dos carros, faz um pedido ao componente responsável por gerir os carros, fornecendo-lhe o identificador do utilizador. Este irá processar o pedido e devolver quais os carros relativos ao utilizador fornecido, como se observa na Figura 27.

Neste caso, a chave estrangeira que estaria presente na tabela dos carros relativa ao utilizador deixa de estar ligada à tabela dos utilizadores. Pode-se observar que a seta a vermelho na Figura 26 que representa uma ligação entre esta chave e a chave primária da tabela dos utilizadores, deixa de estar representada na Figura 27. Todavia, a coluna continua a existir, para se saber qual o utilizador de um certo carro, o que acrescenta alguma complexidade, visto que será necessário gerir estas colunas que não são chaves estrangeiras, mas que vão atuar como tal, para que a consistência entre microsserviços permaneça [Newman (2019)].

Voltando ao exemplo da aplicação mencionada anteriormente, caso um utilizador que possua carros seja removido, ficamos com linhas na tabela dos carros com informação inválida. Com um único esquema de base de dados, não seria possível eliminar um utilizador caso este tivesse carros associados, assegurando assim a consistência dos dados. Contudo, esta restrição já não existe, pois, separou-se estes dados para esquemas de bases de dados diferentes. Neste caso, existem algumas opções que dependem de situação para situação.

- Verificar antes de eliminar

  Uma das opções é quando se quer eliminar um utilizador, verificar se não existem referências na tabela dos carros. Um problema com esta opção, é que cria uma dependência para todos os microsserviços que possam ter referências para os utilizadores, sendo agora preciso verificar com todos os outros se estes possuem referências para um utilizador que se eliminará [Newman (2019)].

- Lidar com a eliminação

Outra opção é permitir que se possa eliminar utilizadores sem preocupações de este estar a ser utilizado por outros microsserviços. Neste caso, o microsserviço dos carros pode admitir que pode ter referências para utilizadores que já não existem.

Nesta situação, se se imaginar num contexto de *backoffice* em que se quer visualizar todos os carros na plataforma, pode haver carros que não tenham referência a utilizadores. Para esta situação pode-se indicar que o utilizador já existiu, mas que já não está mais presente [Newman (2019)].

- Eliminação em cascata

  Outra possibilidade é notificar os microsserviços aquando da eliminação de um utilizador, o que pode ser realizado através da subscrição de eventos ou outro mecanismo. Com isto, podem eliminar os registos que possuem referências a este utilizador eliminado [Newman (2019)].

- Não permitir a eliminação

  Uma última alternativa para assegurar que não se introduz inconsistência de dados no sistema é simplesmente não permitir que utilizadores possam ser eliminados. Uma possibilidade pode ser marcar o utilizador como eliminado, mas este permanecer na base de dados, podendo ser conseguido através de uma coluna que indica o estado do utilizador. Esta opção tem o problema de se poder ter vários registos deixados na base de dados que deviam estar eliminados [Newman (2019)].

Concluindo, é preciso sublinhar que nem todas as chaves estrangeiras devem ser removidas, pois podem ser tabelas que necessitam de estar juntas, fazendo parte de um mesmo agregado. Nestas situações visto que provavelmente irão pertencer ao mesmo microsserviço, não há necessidade de remover a restrição associada à chave estrangeira, dado que vão pertencer ao mesmo esquema de base de dados [Newman (2019)].

3.5.4 *Dividir uma tabela*

Podem existir situações em que se tem dados numa única tabela que necessitam de ser divididos entre dois ou mais microsserviços [Newman (2019)]. Na Figura 28 vemos do lado esquerdo uma tabela partilhada por dois contextos, o *A* e o *B*. O que se pretende é extrair *A* e *B* para novos microsserviços, contudo os dados destes estão numa única tabela. É preciso dividir estes dados em duas tabelas separadas, como se apresenta na Figura 28 no lado direito.

O ideal é separar primeiro as tabelas no esquema de base de dados do monolítico antes de os separar para bases de dados diferentes. Se estas tabelas existissem no mesmo esquema da base de dados, faria sentido declarar uma chave estrangeira numa das tabelas para a outra. Contudo, visto que estas vão estar em motores de bases de dados diferentes, não faz sentido esta chave estrangeira [Newman (2019)].

Este exemplo é bastante direto, separaram-se os dados para os microsserviços conforme os acessos que tinham às colunas, não tenho nenhum deles a aceder a colunas do outro. Contudo, podemos ter contextos a aceder à mesma coluna como se pode observar na Figura 29 do lado esquerdo.

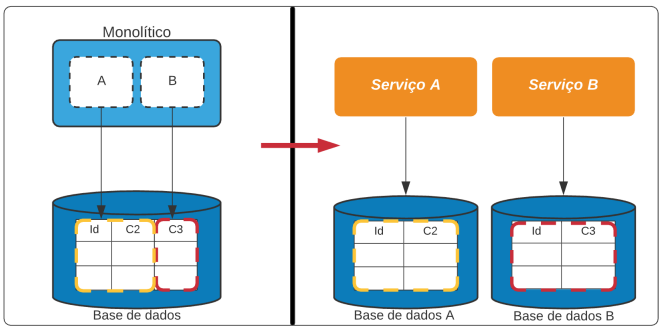


Figura 28: Dividir uma tabela.

Nesta situação apenas um dos microsserviços deve gerir esses dados, tendo o outro de aceder a estes para poder fazer leituras e escritas a esta coluna, como se demonstra na Figura 29 do lado direito. Nesta Figura o microsserviço *A* ficou encarregue dos dados, mas poderia ter sido ao contrário. Como referido na secção 2.2 o ideal é que cada microsserviço faça a gestão dos seus dados e que as mudanças de estado ocorram dentro destas, por isso nesta situação, tem que se perceber qual o microsserviço que de facto deve possuir estes dados.

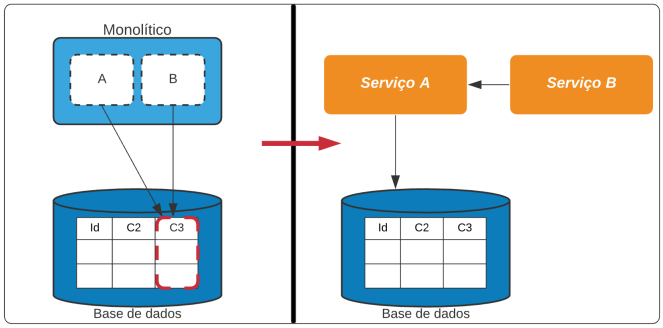


Figura 29: Dividir uma tabela com acessos à mesma coluna.

3.5.5 *Dados estáticos*

Existe em algumas aplicações conteúdo que é estático, conteúdo este que pode ser acedido por vários microsserviços. Muitas vezes este conteúdo estático é armazenado numa tabela na base de dados.

Imaginemos no exemplo da aplicação de gestão de carros. Existem várias marcas de carros e dentro das várias marcas existem vários modelos. Esta informação pode ser armazenada numa ou mais tabelas da base de dados.

Existem algumas opções a considerar.

- Duplicar os dados

  Uma das hipóteses é duplicar as tabelas em cada um dos microsserviços que a aplicação possui. Existem preocupações devido à duplicação e inconsistência de dados entre microsserviços, o que leva a que estes possam ter uma visão diferente destes dados [Newman (2019)].

Contudo, pode ser vantajoso se forem dados que muito dificilmente serão alterados [Newman (2015)]. Imagine-se ter que guardar os dados dos países que existem. Muito raramente existirá um novo país a ser criado e mesmo que aconteça vai provavelmente ser necessário apenas adicionar um novo registo a estes dados estáticos.

- Base de dados partilhada

  Outra hipótese seria ter uma base de dados partilhada responsável por estes dados estáticos.

  É, no entanto, necessário considerar todos os desafios de ter uma base de dados partilhada por vários microsserviços, algo abordado na secção 3.4.

  Esta opção tem a vantagem de evitar a duplicação de dados [Newman (2015)].

- Biblioteca estática

  Outra opção bastante mais simples seria guardar esta informação estática em ficheiros, tal como enumerações [Newman (2015)].

  De modo a não se ter que duplicar estas bibliotecas em todos os microsserviços, podia-se colocar estes dados numa biblioteca que possa ser acedida por todos. Contudo, o uso de uma biblioteca partilhada por todos os microsserviços, implicaria que não se pudesse desenvolver cada um em tecnologias diferentes e que se tivesse que voltar a instalar os microsserviços que a usam em caso de atualização desta biblioteca [Newman (2019)].

  Uma opção seria cada microsserviço ter a sua biblioteca estática, mas implica aceitar que nem todos precisam de ter a mesma versão desta biblioteca, porque caso precisem, voltamos outra vez à situação de terem todos que ser instalados aquando de uma alteração [Newman (2019)].

  Esta opção é vantajosa quando temos dados estáticos pequenos em volume e quando estes dados podem ter versões diferentes nos vários microsserviços.

- Microsserviço dedicado

  Uma última opção passa por colocar estes dados estáticos para um microsserviço independente e dedicado, sendo que os microsserviços que necessitem dessa informação a pedem a este [Newman (2015)].

  Com esta opção, está-se a adicionar mais um microsserviço que aumentará as chamadas feitas pela rede, levando a um aumento da latência. Uma solução seria não armazenar estes dados estáticos numa base de dados, mas sim em memória, o que depende do volume de dados estáticos, mas caso não seja muito grande, é uma solução viável e estando os dados em memória acaba por ser muito mais rápido do que ir buscar à base de dados [Newman (2019)].

  Por outro lado, caso estes dados não sejam facilmente mudados, cada microsserviço que necessite destes pode guardá-los na sua própria memória. Estes podem subscrever a eventos deste microsserviço dedicado para serem notificados quando existem novos dados ou alguma alteração [Newman (2019)].

Esta opção é vantajosa quando temos um valor enorme de conteúdo estático, onde ter um microsserviço que faça a gestão deste conteúdo valha realmente a pena. Se estes dados podem ser mudados com maior frequência é vantajoso ter apenas um local onde seja possível realizar estas alterações. É, no entanto, necessário ter em conta o custo associado com a criação de mais um microsserviço [Newman (2015)].

3.6 TRANSAÇÕES

Esta separação da base de dados leva a dificuldades aquando de operações que envolvam transações.

Normalmente depende-se muito da base de dados para assegurar a consistência e para poder executar várias operações em simultâneo, contando que estas são realizadas de forma atómica. Contudo, ao separar estes dados por vários microsserviços e bases de dados, perde-se a vantagem de realizar transações para realizar modificações de forma atómica [Newman (2019)]. Esta falta de atomicidade pode causar problemas especialmente se o sistema a migrar depender bastante desta propriedade.

Uma transação é um conjunto de operações que devem ser executadas juntas e que garante que ou todas elas foram executadas, ou que nenhuma delas foi executada, permitindo que o sistema continue num estado consistente, visto que caso alguma das operações falhe, as anteriores a elas são anuladas.

3.6.1 *Transações ACID*

ACID é um acrónimo que indica as propriedades-chave de transações da base de dados que levam a que um sistema seja confiável e que assegure a durabilidade e consistência dos dados [Newman (2019)]. ACID significa atomicidade, consistência, isolamento e durabilidade.

- Atomicidade

  Assegura que todas as operações de uma transação são realizadas na totalidade com sucesso ou nenhuma é. Caso alguma das mudanças a ser realizada falhar, toda a transação é abortada e revertem-se quaisquer mudanças realizadas durante a transação [Richardson (2018)].

- Consistência

  Quando alterações são realizadas à base de dados é assegurado que esta é deixada num estado válido e consistente. Os dados estão num estado consistente quando a transação começa e quando acaba [Richardson (2018)]. Por exemplo, se uma aplicação transferir dinheiro de uma conta para a outra, a propriedade de consistência assegura que o número total de dinheiro nas duas contas é o mesmo no início e no fim da transação.

- Isolamento

  Permite que várias transações sejam realizadas em simultâneo, sem que nenhuma interfira na outra. O estado intermediário de uma transação é invisível para outras transações. O resultado, de executar várias transações concorrentemente, é o mesmo caso elas fossem executas numa ordem arbitrária [Richardson

(2018)]. Por exemplo, na aplicação que transfere dinheiro de uma conta para outra, a propriedade de isolamento assegura que outra transação que esteja a ser efetuada, veja este dinheiro na primeira ou na segunda conta, nunca nas duas, nem em nenhuma.

- Durabilidade

  Assegura que após o término de uma transação, os dados permanecem na base de dados e não são perdidos, nem desfeitos, mesmo que exista alguma falha [Richardson (2018)].

É importante referir que nem todas as bases de dados asseguram transações ACID. As que não garantem as propriedades ACID são normalmente denominadas BASE que significa *Basically Available, Soft State and Eventual Consistency*, numa tradução livre "Basicamente disponível, estado delicado e eventual consistência"[Newman (2019)].

- Basicamente disponível

  Operações de escrita e leitura estão disponíveis tanto quanto possível, mas sem qualquer garantia de consistência, o que pode causar a que escritas não sejam persistidas devido a conflitos e que leituras não devolvam o último estado [Kleppmann (2017)].

- Estado delicado

  Sem a garantia de consistência, a probabilidade de saber o estado mais recente é diminuída, dado que as mudanças podem não ter sido convergidas.

  Indica que o estado do sistema pode mudar temporalmente, mesmo sem haver novas alterações realizadas, devido a não se garantir consistência [Kleppmann (2017)].

- Consistência Eventual

  Indica que o sistema ficará consistente eventualmente e se possam realizar leituras que representam o estado consistente do sistema, desde que, entretanto, não sejam efetuadas novas alterações Kleppmann (2017).

3.7 SAGAS

Uma saga é um algoritmo que consegue coordenar várias mudanças de estado, sem a necessidade de bloquear os recursos por um longo período de tempo, modulando os passos envolvidos nesta mudança de estado como atividades discretas que podem ser executadas independentemente [Newman (2019)].

Transações que são de longa duração, detém recursos da base de dados por períodos relativamente longos, acabando por atrasar outras transações mais curtas, o que causa problemas caso outros processos estejam a tentar aceder a estes recursos bloqueados para efetuar leituras ou escritas.

A ideia passa por partir estas transações numa sequência de transações mais pequenas, podendo ser cada uma tratada independentemente [Garcia-Molina and Salem (1987)]. Uma saga é definida como uma sequência

de transações locais, tendo cada transação local as propriedades ACID. A conclusão de uma transação local aciona a execução da próxima transação local [Richardson (2018)]. Com isto, a duração de cada transação será curta e vai apenas modificar uma parte dos dados. Como resultado, tem-se menos contenção de dados [Garcia-Molina and Salem (1987)].

Uma saga consiste em três tipos de transações locais.

- Transação compensável - Transações que podem ser potencialmente revertidas. Para isso, usa-se uma transação de compensação, algo a ser apresentado na secção 3.7.2.
- Transação pivô - Uma transação pivô é uma transação que define o sucesso ou não da saga. Se esta terminar com sucesso, é assegurado que a saga vai também terminar com sucesso. Esta pode ser a última transação compensável ou a primeira.
- Transação bem-sucedida - Transações após a transação pivô, garantidas de ter sucesso e que não precisam de transações de compensação.

Embora sagas tenham sido originalmente previstas como um mecanismo para ajudar a resolver os vários problemas das transações de longa duração sobre uma única base de dados, o modelo funciona perfeitamente para coordenar mudanças ao longo de vários microsserviços [Newman (2019)].

3.7.1 *Exemplo de uma Saga*

Considerando como exemplo uma aplicação de gestão de carros e encomendas de produtos para carros, imagine-se que um utilizador pretende realizar uma encomenda que contém produtos para o seu carro. É necessário verificar que existe inventário suficiente, retirar dinheiro relativo à encomenda do utilizador, retirar do inventário estes produtos e por fim submeter a encomenda, como se pode observar pela Figura 30.

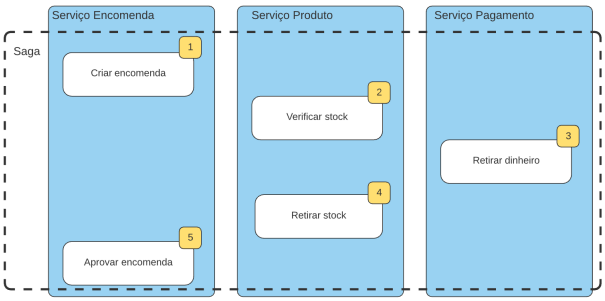


Figura 30: Exemplo de uma saga.

Cada participante da saga comunica de forma assíncrona, usando-se mecanismos de comunicação como *message broker* ou comunicação por eventos. Quando uma transação local termina, este microsserviço publica uma mensagem que desencadeia o próximo passo da saga. Desta forma garante-se que os participantes estejam pouco acoplados e que cada transação da saga é executada, mesmo que algum participante esteja temporariamente indisponível [Richardson (2018)].

3.7.2 *Lidar com Falhas*

Um dos desafios das sagas é recuperar de uma falha. Numa saga é necessário considerar que possa haver erros e que uma delas possa falhar [Richardson (2018)].

Existem duas formas de recuperação, para trás e para a frente. Uma recuperação para trás implica reverter a falha e tudo o que foi feito para trás. É preciso escrever transações de compensação para se poderem desfazer as alterações já realizadas pelas transações locais [Garcia-Molina and Salem (1987)]. Uma recuperação para a frente permite que se parta do ponto onde a falha ocorreu e continuar o processamento [Newman (2019)].

- Transações de compensação

  As transações que garantem as propriedades ACID conseguem facilmente retroceder caso algum problema aconteça, porque o retrocesso ocorre antes de se ter submetido as alterações, como se nada tivesse sido realizado [Newman (2019)].

  Com sagas é mais complicado e não é possível efetuar isto de forma automática, porque cada transação é local a um microsserviço e as modificações são submetidas para a sua base de dados. Para esta situação existem as transações de compensação [Richardson (2018)].

  Imagine-se que na Figura 30, a transação número quatro falha porque, entretanto, foi realizada outra encomenda com os mesmos produtos e deixou de existir inventário para esta encomenda. É preciso retroceder no que já se realizou, tal como ter retirado dinheiro ao utilizador pela encomenda.

  Para se retroceder nesta saga, é necessário a escrita de transações de compensação. Uma transação de compensação é uma operação que desfaz uma transação que foi já previamente submetida. Para retroceder neste exemplo de efetuar uma encomenda é preciso ter uma transação de compensação para cada uma que efetuou alterações [Newman (2019)].

  Uma saga executa as transações de compensação por ordem reversa das transações originais. Caso a transação $(n+1)$ falhar, é preciso que as n transações anteriores sejam revertidas. Para cada uma destas transações T_i, existe uma transação de compensação C_i que desfaz as mudanças da transação T_i. Para reverter os efeitos das n transações, a saga tem que executar cada C_i por ordem inversa, da C_n para a C_1 [Richardson (2018)].

  É preciso sublinhar que estas transações de compensação não tem o mesmo efeito que um *rollback* de uma transação normal de base de dados. O *rollback* é executado antes de a transação ter submetido as mudanças para a base de dados, como se nada tivesse acontecido, enquanto qu nestas transações de compensação, é criada uma nova transação para reverter as mudanças realizadas pela transação original [Newman (2019)].

  Todavia, nem sempre é possível reverter totalmente uma transação [Newman (2019)]. Como um exemplo, um dos passos de realizar uma encomenda pode envolver o envio de um correio eletrónico que indique que a sua encomenda foi efetuada com sucesso. Caso se tenha que retroceder nas ações realizadas, não é possível desfazer o envio de um correio eletrónico. Nestas situações, a transação de compensação

pode ser o envio de um segundo correio eletrónico que indica que ouve um problema com a encomenda e que esta foi cancelada.

- Reordenar os passos da saga

  Uma reordenação dos passos envolventes numa saga pode significativamente diminuir os cenários de se ter que recuperar de uma falha [Newman (2019)]. Na Figura 30, a transação quatro pode ser reordenada e ser efetuada aquando da transação dois, como se observa na Figura 31. Desta forma, previne-se o possível erro de não conseguir retirar o inventário relativo à encomenda, porque já outra encomenda o retirou enquanto era executada a transação três de retirar o dinheiro.

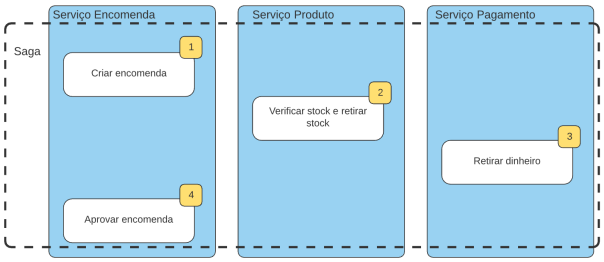


Figura 31: Ordenação dos passos de uma saga.

  Desta forma, evita-se ter que efetuar um retrocesso da transação três e simplifica-se em muito os cenários de recuperação [Newman (2019)]. Deixa de ser necessário escrever uma transação de compensação para a transação três.

- Conjugar recuperação para a frente e para trás

  Existem situações em que pode ser apropriado ter uma mistura dos tipos de recuperação de falhas, porque algumas falhas podem necessitar de ser recuperadas para trás e outras para a frente [Newman (2019)].

  Imagine-se que no processo de uma encomenda o último passo é o envio da encomenda. Se por alguma razão, ocorrer um problema no envio da encomenda, seja porque não existe espaço nas carrinhas de entrega ou por outra razão qualquer, não é correto reverter todo o processo da realização de uma encomenda. Neste caso, o melhor a fazer é voltar a tentar executar a ação de envio da encomenda.

3.7.3 *Implementação de Sagas*

Na implementação de uma saga é necessário ter em conta a coordenação dos passos desta. Quando uma saga inicia, é necessário selecionar o primeiro participante e indicar-lhe para executar a sua transação local. Quando este participante terminar a sua execução, a saga tem que selecionar o próximo participante e indicar para este executar a sua transação local e assim por adiante até a saga terminar todos os seus passos. Caso alguma transação local falhar, a saga deve executar as transações de compensação por ordem reversa [Richardson (2018)].

Para implementar esta lógica existem duas alternativas:

- Orquestração

  A lógica de coordenação é centralizada. Um orquestrador está encarregue de enviar as mensagens para cada participante de uma saga indicando que operações executar.

- Coreografia

  A decisão de coordenação e sequência está distribuída entre os participantes da saga.

3.7.4 *Orquestração*

Sagas que usam orquestração, possuem um coordenador central denominado orquestrador, que está encarregue de definir a ordem de execução, de indicar quando um microsserviço deve executar a sua transação local e de despoletar as transações de compensação caso necessário. Este controla o que acontece e quando conseguindo-se ter uma boa visibilidade do que acontece durante uma saga [Newman (2019)].

Em contraste com a saga apresentada na Figura 31, na Figura 32 é possível ver como se comporta uma saga que usa orquestração.

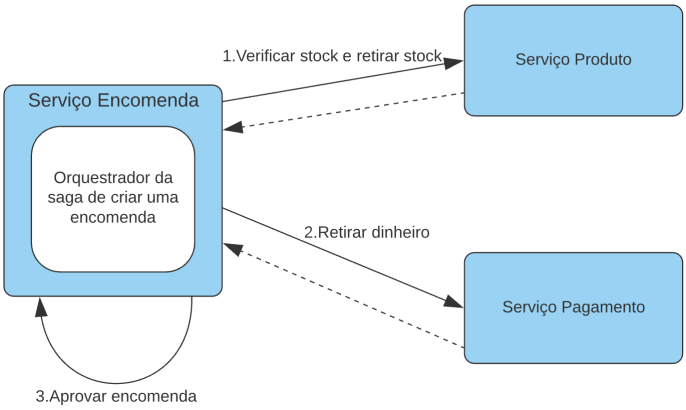


Figura 32: Saga usando orquestração.

Na Figura 32 pode-se observar o orquestrador desta saga de criar uma encomenda no microsserviço encomenda. Este sabe que microsserviços são necessários para que a saga possa ser realizada e decide que chamadas efetuar. Estes orquestradores tendem a ficar à espera da resposta do microsserviço que pediram para efetuar a sua chamada local, para saberem o que fazer a seguir, seja efetuar o próximo passo da saga ou então recuperar de uma falha.

De notar que na Figura 32, no último passo, o orquestrador envia uma mensagem ao próprio microsserviço onde se encontra. O orquestrador, visto que se encontra já no microsserviço encomenda, pode diretamente realizar este passo, porém para ser consistente, este trata o próprio microsserviço como qualquer participante [Richardson (2018)].

Uma saga orquestrada apresenta algumas vantagens. As dependências são mais simples, não existindo dependências cíclicas. O orquestrador invoca os participantes da saga e estes não invocam o orquestrador nem

sabem dos outros participantes, simplificando a lógica de negócio, dado que esta lógica de coordenação se encontra na totalidade no coordenador e não espalhada pelos participantes, como se verá na secção 3.7.5. A lógica de negócio é mais simples e não tem conhecimento das sagas em que participam [Richardson (2018)].

Esta alternativa possui, no entanto, algumas desvantagens. Com orquestração, os microsserviços vão estar mais acoplados, pois o orquestrador necessita de ter conhecimento dos que participam na saga. Outro problema é que como esta alternativa possui a lógica de coordenação centralizada, pode acontecer que exista lógica que devia estar nos microsserviços que acaba por ficar no orquestrador, levando a que estes tenham pouco comportamento e apenas recebam ordens do orquestrador [Newman (2019)].

3.7.5 *Coreografia*

Sagas que usam coreografia distribuem a responsabilidade e lógica de coordenação pelos participantes da saga [Newman (2019)]. Neste tipo de sagas, não existe nenhum coordenador central que diz o que cada participante da saga deve fazer. Cada participante subscreve eventos de outros participantes e reagem a estes [Richardson (2018)]. Na Figura 33 é possível ver um exemplo de como seria uma saga usando coreografia para a criação de uma encomenda.

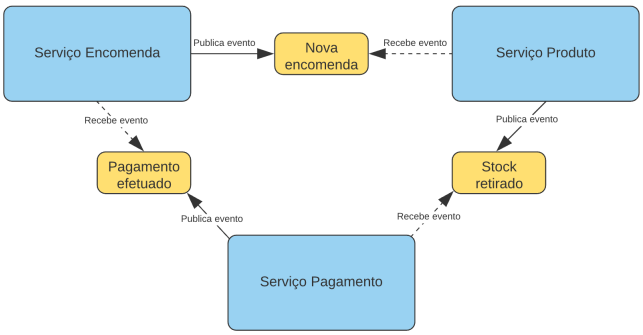


Figura 33: Saga usando coreografia.

É possível constatar na Figura 33 que os microsserviços reagem a eventos que recebem. Conceptualmente, estes eventos são publicados para todo o sistema, não apenas para um microsserviço em específico, sendo que os interessados subscrevem e os recebem [Newman (2019)].

Observando com detalhe para a Figura 33, é possível ver que o microsserviço encomenda publica um evento indicando que foi efetuada uma nova encomenda. O microsserviço produto que está subscrito a este evento, sabe que foi efetuada uma nova encomenda. Com isto, sabe que é preciso verificar se existe inventário para esta ser efetuada e subtraí-lo. Caso tudo corra com sucesso emite um novo evento no sistema indicando que este foi retirado e o processo continua. Contudo, caso não seja possível retirar, o microsserviço produto tem que emitir um evento que informe que não foi possível esta remoção e começa a recuperação desta falha.

Uma saga que usa coreografia apresenta vantagens como simplicidade e menos acoplamento entre microsserviços, porque os participantes publicam e subscrevem a eventos e não tem conhecimento direto uns dos outros [Richardson (2018)].

Também apresenta algumas desvantagens. É mais complicado de compreender o processo, porque ao contrário da orquestração não existe um único lugar onde a saga está definida. Com coreografia, a lógica de coordenação está espalhada pelos microsserviços, o que torna mais complicado a perceção do funcionamento desta. Uma segunda desvantagem é que pode criar dependências cíclicas entre microsserviços, pois cada participante subscreve aos eventos de outros participantes. Apesar de não ser necessariamente problemático, dependências cíclicas são um problema de arquitetura da aplicação [Richardson (2018)].

3.7.6 *Conjugar as duas implementações*

Apesar de sagas que usam orquestração e as que usam coreografia diferirem, é possível misturar estas duas implementações. Pode existir uma saga que funcione melhor com orquestração e outra que funciona melhor com coreografia. É possível até ter uma única saga que possui as duas implementações [Newman (2019)].

3.7.7 *Falta de isolamento*

Uns dos problemas das sagas é que estas são ACD (atomicidade, consistência, durabilidade), faltando-lhes a propriedade de isolamento. A propriedade de isolamento, como abordado na secção 3.6.1, assegura que o resultado, de executar várias transações concorrentes, é o mesmo caso estas fossem executadas numa ordem qualquer [Richardson (2018)].

A falta de isolamento nas sagas existe porque mal uma transação local de uma saga seja completada, outra saga qualquer pode visualizar estas modificações mesmo que esta não tenha terminado,o que pode causar dois problemas. Outras sagas podem acabar por modificar os dados acedidos por uma saga enquanto esta é executada e outras sagas podem realizar leituras de dados de uma saga que ainda não terminou a sua execução [Richardson (2018)].

Esta falta de isolamento pode causar uma anomalia, que pode levar a um funcionamento inesperado da aplicação. Uma anomalia é quando uma transação realiza leituras ou escritas de dados de maneira diferente da que acontecia caso fossem feitas uma de cada vez. Quando uma anomalia ocorre, o resultado de executar sagas concorrentemente difere caso estas fossem executadas por uma certa ordem [Richardson (2018)].

Existem três anomalias que podem ocorrer.

- *Lost updates* - Uma saga modifica dados sem ter visto as mudanças feitas por outra saga nestes dados.
- *Dirty reads* - Uma saga ou transação efetua leituras de dados modificados por uma saga que ainda não acabou esta atualização.
- *Fuzzy/nonrepeatable reads* - Dois passos diferentes de uma saga fazem leituras dos mesmos dados e obtém resultados diferentes porque, entretanto, outra saga realizou alterações a estes dados.

Como resultado desta falha de isolamento, a aplicação deve usar medidas, para prevenir ou reduzir o impacto de anomalias causadas por acessos concorrentes. Deve-se escrever sagas de modo a prevenir estas anomalias

ou minimizar o impacto destas. Algumas medidas implementam isolamento ao nível da aplicação, outras reduzem o risco da falta de isolamento [Richardson (2018)].

Existem algumas medidas, apresentadas a seguir, para colmatar esta falha de isolamento.

- Bloqueio semântico

  Quando se usa a medida de bloqueio semântico, as transações compensáveis colocam um identificador em qualquer registo criado ou atualizado. Este indica que o registo não acabou ainda de ser atualizado e pode vir a sê-lo. Este identificador pode ser um *lock* que previne que outras transações acedam a este registo ou um aviso que indica a outras transações que devem tratar este registo com alguma cautela. Este identificador é removido por uma transação bem-sucedida, visto que se tem garantias de que a saga terminará com sucesso ou então por uma transação de compensação, quando a saga está a recuperar de uma falha [Richardson (2018)].

  Além disso, é preciso decidir como outras sagas vão lidar com estes registos bloqueados.

  Uma opção pode passar por a saga falhar e informar o cliente para tentar de novo mais tarde. É de fácil implementação, mas acrescenta complexidade no cliente. Outra opção seria a saga ficar bloqueada até este identificador deixar de existir. Desta forma remove-se a complexidade no cliente de ter que voltar a tentar executar a operação relativa à saga. O problema é que a aplicação necessita de gerir estes identificadores e implementar um algoritmo de deteção de *deadlocks* que os resolva [Richardson (2018)].

- Atualizações comutativas

  Outra medida é desenvolver as operações de atualização para serem comutativas, significando que podem ser executadas numa ordem qualquer. Esta medida é útil porque elimina as anomalias de *lost updates* [Richardson (2018)].

- Visão pessimista

  Na medida de visão pessimista, os passos de uma saga são reordenados para minimizar o risco de acontecerem *dirty reads* [Richardson (2018)]. É semelhante ao que se constatou na secção 3.7.2 sobre a reordenação de passos de uma saga.

- Reler o valor

  Esta medida de reler o valor previne a anomalia de *lost updates*. Uma saga que use esta medida efetua uma nova leitura de um valor antes de o atualizar, verificando que não foi, entretanto, modificado. Caso tenha sido modificado, a saga é abortada e executada novamente [Richardson (2018)].

- Registar versões

  A medida de registar versões, regista as operações realizadas a um registo, para poderem ser reorganizadas. É uma forma de tornar operações que são não comutativas em comutativas. Desta forma, os pedidos realizados para alterar este registo são ordenados por ordem de chegada e são depois executadas pela ordem correta [Richardson (2018)].

3.8 PADRÕES MIGRAÇÃO - BASE DE DADOS

Os padrões mencionados na secção 3.3 abordam várias estratégias para extrair funcionalidade para microsserviços. Contudo, é preciso pensar também na base de dados.

Vão ser abordados alguns padrões arquiteturais relacionados com a base de dados.

3.8.1 *Monolítico com agregados expostos*

Os padrões mencionados anteriormente na secção 3.4 não procedem a nenhuma separação do esquema da base de dados, apenas apresentam várias formas de vários microsserviços acederem a uma mesma base de dados partilhada.

Se se considerar um microsserviço como um encapsulamento lógico associado a um ou mais agregados, como mencionado na secção 2.2.1, é preciso considerar também que este deve possuir o código relacionado com as mudanças de estado destes agregados, bem como os dados deste que devem estar na sua própria base de dados. Contudo, se o nosso microsserviço extraído necessitar de interagir com um agregado que ainda se encontra no monolítico, é preciso que este seja exposto para poder ser acedido [Newman (2019)].

Para isso existe um padrão que permite que o monolítico possua agregados que expõem uma API para o exterior do monolítico.

Como se observa na Figura 34, o microsserviço *B* foi extraído do monolítico. Todavia, necessita de dados do agregado *A* que se encontra no monolítico. Neste caso, através de uma API, ou outro mecanismo, o agregado *A* expõem uma API que permite ao microsserviço *B* aceder à informação que precisa [Newman (2019)].

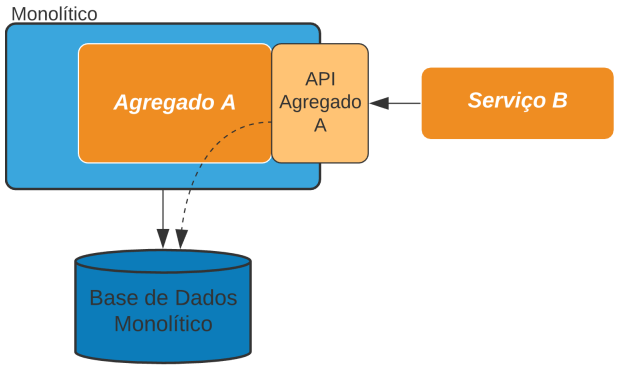


Figura 34: Monolítico com agregados expostos.

Desta forma está-se a permitir que microsserviços exteriores ao monolítico possam efetuar leituras dos dados daquele agregado e até alterar ou inserir. É possível, no entanto, restringir a informação exposta e as ações que se podem realizar [Newman (2019)].

Um ponto interessante deste padrão, é que ao se expor estas APIs do monolítico para microsserviços exteriores, está-se potencialmente a descobrir um futuro microsserviço. Na Figura 34 temos o agregado *A* a expor uma API para que o microsserviço *B* possa aceder a dados deste. Pode-se concluir que provavelmente o

melhor a fazer é extrair posteriormente este agregado *A* para um microsserviço. É evidente que se ao extrair este agregado *A* do monolítico, alguma parte do monolítico precisar de dados deste, este terá que novamente ser mudado para usar este novo microsserviço [Newman (2019)].

Este padrão é vantajoso para situações em que novos microsserviços necessitam de dados que ainda se encontram na base de dados do monolítico. Ao extrair um microsserviço do monolítico, ter este a realizar chamadas ao monolítico leva a um maior trabalho e complexidade do que se fosse diretamente à base de dados deste, mas no futuro prova-se benéfico e vantajoso, pela razão de que se realiza chamadas a um agregado que no futuro pode vir a se tornar num microsserviço que possuirá a sua própria base de dados [Newman (2019)].

3.8.2 *Mudança de domínio dos dados*

No padrão anterior, falou-se como abordar quando um novo microsserviço precisa de aceder a dados que o monolítico possui. Mesmo assim, existem casos em que temos dados que estão no monolítico que certamente deviam estar no novo microsserviço extraído, como se mostra na Figura 35 no passo 1.

Nestes casos, é preciso transferir os dados deste novo microsserviço para que deixem de ser do domínio do monolítico e sejam dele, visto que é lá que estes devem estar e serem geridos. É necessário mudanças no monolítico para passar agora a tratar o novo microsserviço como quem possui estes dados [Newman (2019)]. Quando o monolítico necessitar de realizar leituras ou alterações destes, deve agora interagir com o microsserviço, como se pode observar na Figura 35 no passo 2.

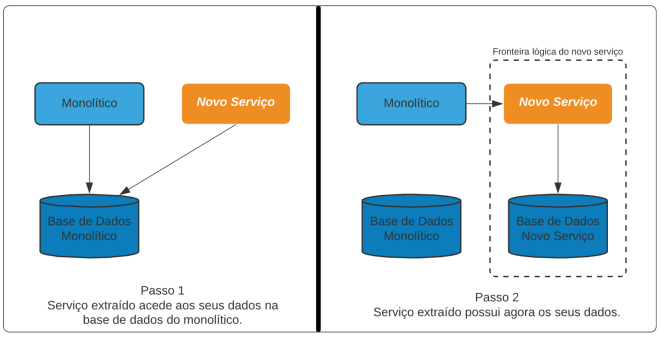


Figura 35: Mudança de domínio dos dados.

Separar estes dados do já existente esquema de base de dados pode ser algo bastante complexo. É preciso considerar a remoção de chaves estrangeiras, o uso de transações, entre outros, algo já mencionado na secção 3.5.

Este padrão deve ser utilizado quando um novo microsserviço extraído realiza alterações a dados que lhe deviam pertencer e que este não os possui. Neste caso, os dados devem ser movidos para o novo microsserviço criado, sendo este responsável pela sua manutenção e gestão [Newman (2019)].

3.8.3 *Sincronização dos dados*

Muitos dos padrões de migração abordados na secção 3.3 possuem a vantagem de ser possível retroceder para a implementação antiga após se ter mudado para a nova. Ainda assim, existe o problema de quando os dados que o novo microsserviço gere terem que estar sincronizados com os que estão no monolítico [Newman (2019)].

Na Figura 36 temos um exemplo desta situação onde se extrai o microsserviço *A* do monolítico. Todavia, tanto o novo microsserviço, como o monolítico, gerem os mesmos dados. Esta situação pode acontecer quando se pretende que seja possível um retrocesso para a implementação antiga. É preciso então que estes dados estejam sincronizados e também considerar o quão importante é ter os dados consistentes entre eles [Newman (2019)].

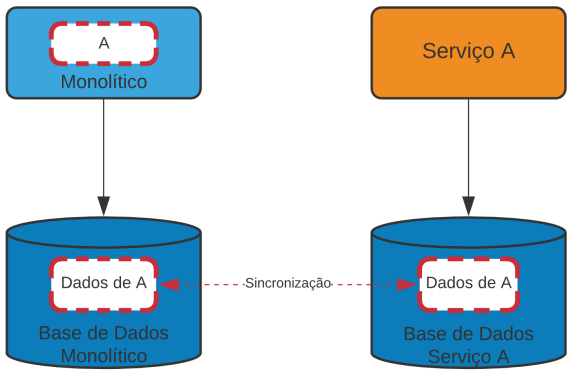


Figura 36: Sincronização dos dados.

Caso estes devam estar totalmente consistentes e sincronizados, o melhor é manter os dados num único lugar, tendo provavelmente o novo microsserviço a aceder diretamente à base de dados do monolítico, podendo recorrer ao padrão de base de dados com vistas, mencionado na secção 3.4.2. Quando se pretender proceder à mudança para o novo microsserviço, pode-se usar o padrão de mudança de domínio dos dados, mencionado anteriormente na secção 3.8.2.

Outra possibilidade seria considerar ter estas duas bases de dados em sincronia, como se tem na Figura 36. Leva a maior complexidade, pois tanto o monolítico, como o microsserviço, podem realizar escritas sobre os seus dados e terem ambos que estar sincronizados [Newman (2019)].

3.8.4 *Sincronização dos dados na aplicação*

Transferir dados de um lugar para outro é um processo bastante delicado, dependendo do quão valiosos são [Newman (2019)]. Quando se tratam de transferir dados relativos a contas bancárias ou registos médicos, por exemplo, é importante pensar com cuidado como se procederá a esta transferência.

Este processo pode realizar-se nos seguintes passos:

- **Cópia total de dados:**

  O primeiro passo consiste em realizar uma cópia total dos dados da base de dados para a nova.

Se for possível parar a aplicação, a cópia dos dados é mais fácil e direta. Caso não seja possível parar a aplicação é preciso realizar um *snapshot* do estado da base de dados e aplicá-lo na nova base de dados. Cria um desafio, pois enquanto foi realizada esta cópia, podem ter sido introduzidos novos dados na base de dados que não estavam no *snapshot*. É necessário registar as mudanças realizadas enquanto foi efetuada a cópia, para poderem ser aplicadas após esta [Newman (2019)]

- **Sincronizar escritas em ambas as bases de dados:**

  Estando agora as duas bases de dados em sincronia, é necessário assegurar que a aplicação escreva corretamente em ambas as bases de dados [Newman (2019)], como se mostra na Figura 37.

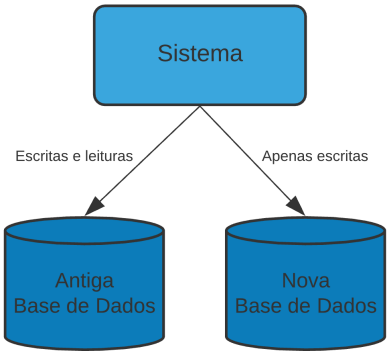


Figura 37: Sincronizar escritas em ambas as bases de dados.

- **Redirecionar escritas e leituras:**

  Encontrando-se a nova base de dados num correto funcionamento, pode-se alterar a aplicação para considerar a nova base de dados como aquela para onde devem ser realizadas as escritas e leituras. É evidente que as bases de dados devem continuar em sincronia, que continuem a realizar as escritas em ambas as bases de dados para que caso ocorra algum problema se tenha uma reserva [Newman (2019)].

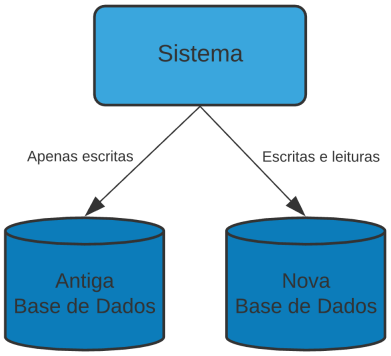


Figura 38: Redirecionar escritas e leituras.

Este padrão é vantajoso quando se considera separar primeiro o esquema da base de dados antes de separar o código [Newman (2019)].

3.8.5 *Rastrear escritas*

Este padrão de rastreio de escritas é uma variação do padrão apresentado na secção 3.8.4. Com este padrão, a base de dados principal é movida de forma incremental, permitindo que existam duas durante a migração [Newman (2019)].

Neste padrão, o novo microsserviço possui os mesmos dados que o monolítico. O monolítico continua a registar e a manter os novos dados localmente, mas quando existem mudanças assegura que estes novos dados são escritos no novo microsserviço através da API do microsserviço, como se pode observar pela Figura 39. É necessário efetuar mudanças ao código do monolítico para que aceda ao novo microsserviço para realizar esta sincronização de dados. Quando toda a funcionalidade estiver a usar o novo microsserviço como quem possui estes dados, é possível removê-los do monolítico [Newman (2019)].

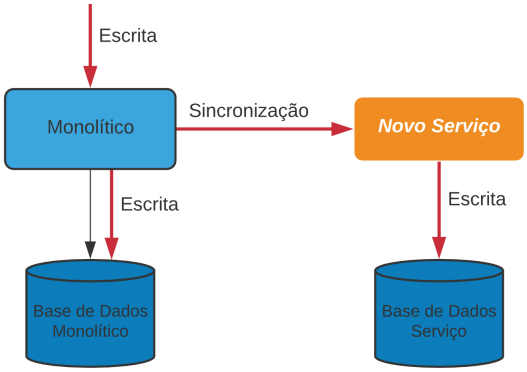


Figura 39: Rastrear escritas.

Este padrão permite que se proceda a uma migração incremental, reduzindo o impacto de cada mudança realizada. Este permite que se comece por manter apenas um pequeno conjunto de dados sincronizados e incrementar este conjunto temporalmente, enquanto se altera e se incrementa gradualmente as ligações a este novo microsserviço [Newman (2019)], como está exemplificado na Figura 40.

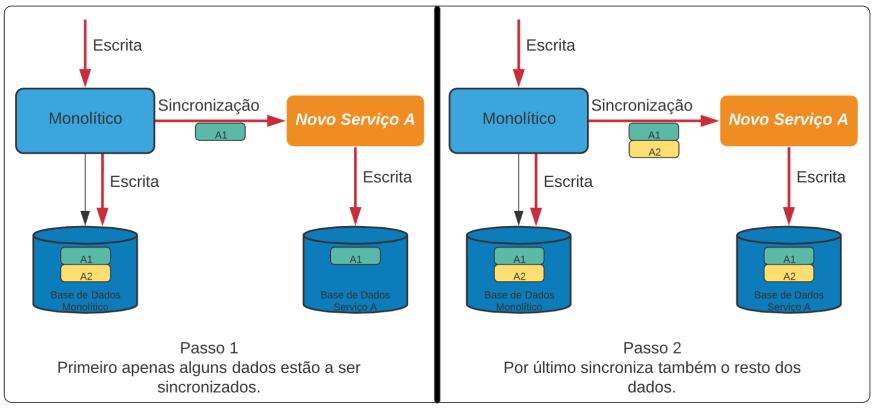


Figura 40: Sincronizar incrementalmente os dados do monolítico para o novo microsserviço.

Outros microsserviços que necessitem destes dados, podem ir buscar tanto ao monolítico como a este novo microsserviço, dependendo da informação que precisam. Se precisam de informação que está ainda apenas no monolítico, tem que ir obrigatoriamente a este, podendo apenas ir ao novo microsserviço quando este estiver a sincronizar os dados que estes pretendem [Newman (2019)].

Um problema que é preciso ter em conta neste padrão e em outras situações de dados duplicados é a inconsistência destes. Existem algumas hipóteses como:

- Realizar escritas para uma das bases de dados

  Todas as escritas são realizadas para uma das bases de dados. Os dados são sincronizados com a outra quando as escritas terminarem [Newman (2019)].

- Enviar as escritas para ambas as bases de dados

  Todos os pedidos de escrita realizados são enviados para ambas as bases de dados. Para isto, quem realiza os pedidos envia para ambas ou existe um intermediário que trata do envio [Newman (2019)].

- Enviar escritas para qualquer base de dados

  Os pedidos de escrita são enviados para qualquer base de dados, sendo que depois estas tratam de sincronizar os dados entre os sistemas [Newman (2019)].

  Esta hipótese traz mais complexidade, pois é preciso implementar sincronização nas duas bases de dados e não apenas numa.

Em todas as hipóteses apresentadas acima irá existir algum atraso na consistência dos dados, levando assim a uma eventual consistência, isto é, eventualmente os dados ficarão consistentes [Newman (2019)].

3.9 DEPENDÊNCIAS ENTRE COMPONENTES

Até agora, falou-se sobre como extrair microsserviços do monolítico e como lidar com a base de dados, porém é preciso saber por onde começar e quais dos microsserviços devem ser os primeiros a serem extraídos. Quando se tiver o código e o código de acesso à base de dados organizado e separado para cada componente, deve-se começar a pensar quais módulos extrair primeiro para microsserviços. Como mencionado anteriormente na secção 3.1, sugere-se seguir uma migração incremental de modo a diminuir o número de problemas e erros possíveis.

Usando como exemplo uma aplicação de gestão de carros e encomendas de produtos para carros, podemos ter vários componentes, representados por *packages* na Figura 41. Componentes para gerir os utilizadores, os carros, as encomendas realizadas, os produtos que existem e respetivo inventário e um que gere as notificações que tem que ser apresentadas aos utilizadores, tal como quando tem que efetuar uma mudança de óleo no carro ou levar o carro à inspeção.

Desenhando um diagrama de *packages*, apresentado na Figura 41 é possível obter uma visão da estrutura do sistema, que permite concluir que módulos são mais fáceis e mais difíceis de extrair. Se se pretender começar

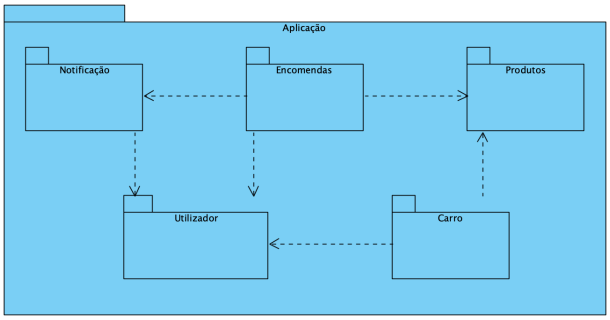


Figura 41: Dependências entre componentes.

por extrair o componente utilizador pode-se perceber que este recebe várias chamadas de outros componentes, o que implica que várias partes dos outros componentes teriam que ser modificadas. Seria preciso modificar bastante código de todos os outros componentes que realizam chamadas locais a este e que teriam que passar a ser chamadas remotas.

É mais fácil começar por extrair aqueles componentes que contém menos dependências, de modo a modificar o mínimo de outros componentes no início da separação para microsserviços [Newman (2019)].

Como se pode constatar na Figura 41, o componente do carro não possui outros componentes dependentes dele, o que o torna num componente mais fácil de extrair para um microsserviço. Desta forma reduz-se a quantidade de mudanças a ter que realizar noutros módulos [Newman (2019)].

4

DESENVOLVIMENTO DA APLICAÇÃO MONOLÍTICA

O objetivo deste tema de dissertação é o desenvolvimento de uma plataforma de *e-commerce* que seja escalável, segura e responsiva. Pretende-se que a aplicação seja desenvolvida segundo uma arquitetura monolítica, migrando posteriormente alguns componentes de modo a seguir uma arquitetura de microsserviços.

No presente capítulo será estabelecido qual o objetivo da plataforma, quais os utilizadores alvo e partes interessadas. A seguir são reunidos os requisitos da aplicação, tanto funcionais como não funcionais. Consequentemente irá ser feita a modelação da plataforma, começando a seguir o desenvolvimento de um protótipo desta segundo uma arquitetura monolítica. O desenvolvimento será bem estruturado, seguindo as orientações apresentadas no capítulo 3 para se poder migrar facilmente para uma arquitetura de microsserviços.

4.1 DEFINIÇÃO DA PLATAFORMA

Esta plataforma designada *Marketplace* disponibiliza a fornecedores e consumidores de serviços uma plataforma única e comum onde possam, no caso dos fornecedores, publicitar os seus serviços e no caso dos consumidores procurar e usufruir destes. O *Marketplace* disponibiliza uma plataforma para fornecedores verificados partilharem a sua informação e os seus serviços. Deve permitir a procura e navegação dos vários serviços, que se encontram por categorias. Também estas categorias devem ser passíveis de procura e navegação.

Os fornecedores podem:

- Registar-se no sistema.
- Autenticar-se no sistema.
- Gerir os seus serviços, tal como adicionar, remover, editar e eliminar

Os consumidores devem conseguir:

- Registar-se no sistema.
- Autenticar-se no sistema.
- Procurar por serviços
- Requisitar um serviço.

4.2 UTILIZADORES

Nesta secção descrevem-se os utilizadores do sistema, que possuem interesse ou desempenham algum papel na plataforma.

4.2.1 *Utilizadores Diretos*

- **Convidado** - Qualquer pessoa que não esteja registada no sistema. Apenas vai poder ver os serviços existentes e navegar na plataforma.
- **Consumidor** - Qualquer pessoa que se registe no sistema com o propósito de usufruir dos serviços.
- **Fornecedor** - Qualquer pessoa que se registe na plataforma com o propósito de providenciar os seus serviços.
- **Administrador** - Administradores responsáveis de gerir a aplicação. Estes estão encarregues de adicionar/remover categorias, aprovar/rejeitar avaliações, entre outros.

4.2.2 *Utilizadores de Suporte*

- **Equipas de Desenvolvimento** - Engenheiros informáticos especializados em áreas como base de dados, arquiteturas de *software* e desenvolvimento Web.
- **Equipas de manutenção** - Engenheiros informáticos especializados em desenvolvimento Web, base de dados, segurança, entre outros, mantendo o sistema em funcionamento.

4.3 REQUISITOS

De modo a melhor delinear e percecionar os requisitos que o sistema reúne, foram definidos os requisitos divididos pelos vários intervenientes no sistema, ou seja, os requisitos estarão associados ao **Convidado**, **Consumidor**, **Fornecedor** e **Administrador**.

Também foram definidos alguns requisitos não funcionais.

Estes requisitos encontram-se na secção de apêndice A.

4.4 TECNOLOGIA

A aplicação foi desenvolvida em Java , usando a *framework* Spring Boot para o desenvolvimento do *backend*. Esta tecnologia permite um desenvolvimento rápido e eficiente, facilitando tarefas como integração com a base de dados, criação de REST API e um servidor Tomcat. O IDE escolhido para o desenvolvimento da aplicação foi IntelliJ IDEA .

Para motor de base de dados usou-se MySQL , dado que o modelo relacional é adequado para a representação dos dados.

Para integração com a base de dados usou-se Spring Data JPA , uma abstração de JPA (Java Persistence API), facilitando o acesso e integração com a base de dados MySQL.

Utilizou-se também Spring Security para controlar a autenticação e autorização dos utilizadores, bem como filtrar os pedidos realizados à aplicação.

É também utilizado autenticação de dois fatores, recorrendo à biblioteca Google Auth . Este tipo de autenticação, assente no princípio de "algo que o utilizador sabe e algo que o utilizador tem", é baseado no algoritmo *Time-based One-Time Password* (TOTP).

Para compilação da aplicação utilizou-se Gradle , um sistema de código aberto que permite automatizar a compilação, teste e instalação de aplicações.

Para a instalação da aplicação usou-se Docker , um sistema de virtualização de *software* que os coloca em contentores e Kubernetes para orquestrar a instalação dos contentores.

Para o desenvolvimento do *frontend* utilizou-se JavaScript , mais precisamente a *framework* React . O desenvolvimento do *frontend* encontra-se fora do âmbito deste tema de dissertação, sendo que é abordado durante o documento algumas alterações a realizar no *frontend* devido à migração para microsserviços.

4.5 MODELO DE DOMÍNIO

Tendo os requisitos levantados e sabendo que tipos de utilizadores irão comunicar com a aplicação, desenvolveu-se um modelo de domínio apresentado na Figura 42. Neste representam-se as principais entidades e os relacionamentos entre estas, bem como alguns atributos de algumas entidades.

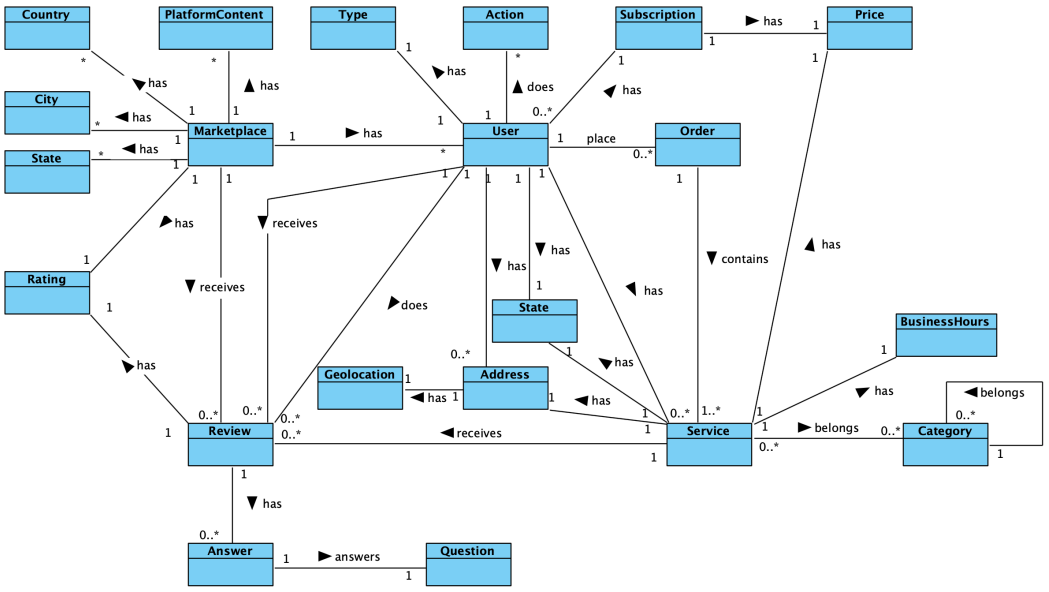


Figura 42: Modelo de domínio.

Ao observar a Figura 42, pode-se concluir que temos a aplicação denominada *Marketplace*, que possui utilizadores, avaliações, uma média de avaliações e conteúdo da plataforma, como termos de utilização, entre outros. Um utilizador pode ter serviços, efetuar uma encomenda de serviços e efetuar/receber avaliações. Além disso, um utilizador pode ter um tipo, efetuar ações, ter uma subscrição que tem um preço, um estado e um endereço que possui uma geolocalização. Um serviço pode pertence a uma categoria, possuir avaliações, um endereço, um estado, um preço e horário de funcionamento. Uma categoria pode pertencer a outra categoria, o que a torna numa subcategoria dessa. Uma avaliação tem um valor de avaliação, pode ter várias perguntas, que consequentemente tem uma resposta.

De referir que as entidades **Country**, **City** e **State** existem no sistema para ter informação sobre os países, cidades e estados onde os serviços e bens se podem encontrar. É útil para a procura, visto que o utilizador pode filtrar por localização.

4.6 DIAGRAMA DE USE CASES

Com os requisitos levantados e especificados, desenvolveu-se um Diagrama de Use Cases apresentado na Figura 43. Neste, representam-se os utilizadores da aplicação e as funcionalidades de cada um.

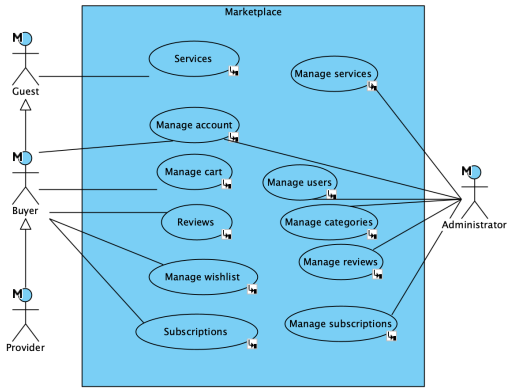


Figura 43: Diagrama de use cases.

Os sub-diagramas de use cases encontram-se na secção de apêndice B.

4.7 DIAGRAMA DE CLASSES

Este percurso até aqui permite-nos desenvolver o diagrama de classes relativo ao nosso sistema. Este modelo é independente da tecnologia usada na fase de desenvolvimento. Este diagrama de classes é apresentado na Figura 44.

O diagrama de classes é constituído por:

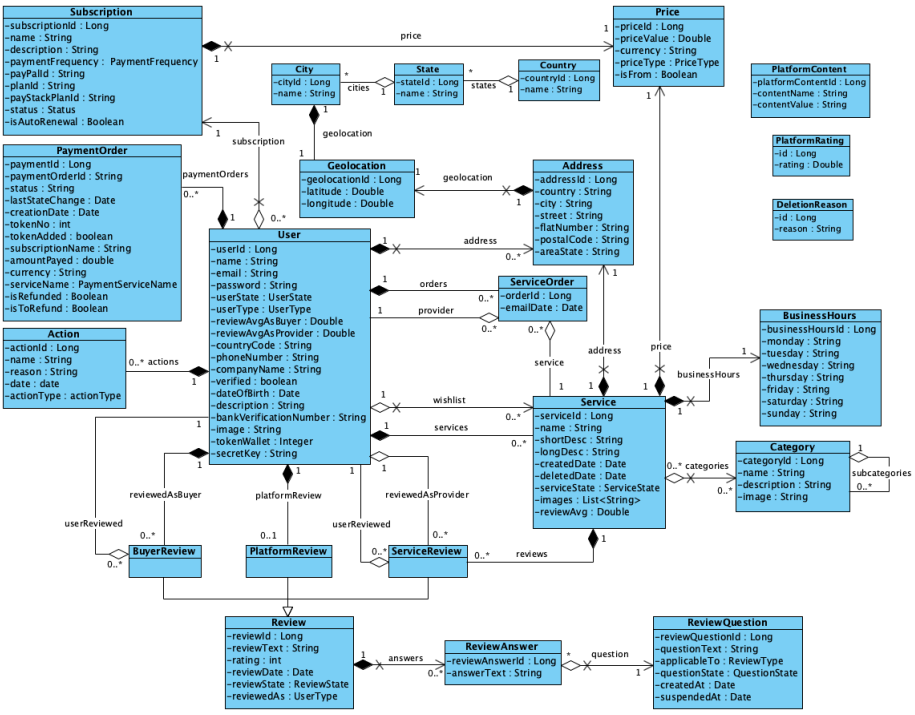


Figura 44: Diagrama de classes.

- **User**: Representa todos os utilizadores. Este possui vários parâmetros como nome, email, password, informação telefónica, possível informação sobre a empresa caso seja um fornecedor, entre outros. Para além destes destacam-se os seguintes:
  - **UserState**: Representa o estado do utilizador, podendo este não estar verificado, ativo, inativo, eliminado ou rejeitado.
  - **UserType**: Representa o tipo do utilizador, que pode ser consumidor, fornecedor ou administrador. Com esta propriedade, controlam-se os acessos e funcionalidades que cada tipo de utilizador pode ter e realizar.

  Um utilizador pode também ter outras propriedades:
  - **Services**: Representa os serviços criados por este utilizador. Estes são os serviços que, como fornecedor, disponibiliza na plataforma.
  - **Wishlist**: Representa a lista de desejos de um utilizador. São os serviços que este pretende guardar para mais tarde visualizar.
  - **Actions**: Representa as ações que este utilizador realizou na plataforma, para se poder mostrar aos administradores que ações foram realizadas por um determinado utilizador e que efeito produziu.
  - **PaymentOrders**: Representa os pagamentos realizados pelo utilizador na plataforma.
  - **Subscription**: Representa a subscrição que o utilizador possui.

  - **Address**: Representa as moradas do utilizador. Este pode ter mais que uma morada, pois pode ter a sua morada como consumidor e como fornecedor.
  - **Orders**: Representa os pedidos de contactos que este utilizador realizou.
  - **ReviewedAsBuyer**: Representa as avaliações que o utilizador recebeu como consumidor por ter usufruído de serviços.
  - **PlatformReview**: Representa a avaliação que o utilizador efetuou à plataforma.
  - **ReviewedAsProvider**: Representa as avaliações que o utilizador recebeu como fornecedor por fornecer os seus serviços.

- **Service**: Representa um serviço. Este possui um nome, descrição curta, descrição longa, a data em que foi criado, possível data em que foi eliminado, imagens e a média de avaliações recebida. Além disso, tem também o estado do serviço, que pode estar ativo, suspendo pelo próprio utilizador, suspenso por um administrador ou eliminado.

  Um serviço pode ter também as seguintes propriedades:

  - **Address**: Representa a morada do serviço.
  - **Price**: Representa o preço do serviço.
  - **BusinessHours**: Representa o horário do serviço.
  - **Categories**: Representa as categorias a que este serviço pertence.
  - **Reviews**: Representa as avaliações recebidas por este serviço.

- **Category**: Representa uma categoria, que possui um identificador, nome, descrição e uma imagem. Uma categoria pode ser subcategoria de outra categoria e pode ter várias subcategorias.

- **BusinessHours**: Representa o horário de um serviço. Este possui uma *string* para cada dia da semana, representando para cada dia de que horas a que horas o serviço está disponível.

- **Address**: Representa uma morada. Esta possui o país, cidade, rua, código postal, região, o número de porta e uma geolocalização.

- **Geolocation**: Representa uma geolocalização. Esta possui uma latitude e longitude.

- **Price**: Representa um preço. Este possui o valor, moeda, tipo e se é o valor inicial ou fixo. Em relação ao tipo de preço, este pode ser fixo, por hora ou por dia.

- **ServiceOrder**: Representa um pedido pelos contactos de um serviço. Este possui um identificador e uma data em que deve ser enviado um correio eletrónico a pedir por uma avaliação dos serviços usufruídos e prestados. Tem também a referência para o serviço e fornecedor para o qual o pedido foi realizado.

- **Review**: Representa uma avaliação, que possui um texto, uma classificação, a data em que foi feita, um estado e o tipo de utilizador que a fez. O estado da avaliação pode ser aprovada, pendente ou rejeitada. Além disso, uma avaliação pode também ter um conjunto de respostas que são referentes às questões de avaliação que existem na plataforma.

  Uma avaliação pode ser de três tipos:

  - **BuyerReview**: Representa uma avaliação que um fornecedor deu a um consumidor. Esta avaliação ao consumidor apenas pode ser feita se este tiver usufruído de um serviço do fornecedor. Para além das propriedades adquiridas da classe *Review* tem também uma referência para o utilizador que recebeu a avaliação.
  - **ServiceReview**: Representa uma avaliação que um consumidor deu a um serviço. Esta avaliação apenas pode ser realizada se o consumidor tiver usufruído do serviço. Neste caso, o consumidor avalia o serviço e por consequência está também a avaliar o fornecedor. Para além das propriedades adquiridas da classe *Review* tem também uma referência para o utilizador que recebeu a avaliação.
  - **PlatformReview**: Representa uma avaliação à plataforma. Cada utilizador pode apenas realizar uma avaliação à plataforma.

- **ReviewAnswer**: Representa as respostas às questões de avaliações. Estas possuem um identificador, um texto e a questão a qual é respondida.

- **ReviewQuestion**: Representa as questões de avaliações que existem na plataforma. Estas possuem um texto, que representa a pergunta, data de criação e suspensão, um estado e a que tipo de avaliação pode ser aplicada. Em relação ao estado da questão, este pode estar ativo ou suspenso.

- **Action**: Representa as ações realizadas na plataforma. Estas possuem um nome, uma razão pela qual foi efetuada, uma data e o tipo da ação, que pode ter sido realizada no *backoffice*, por um fornecedor, ou relativa a pagamentos.

- **Subscription**: Representa os pacotes de subscrições disponíveis na plataforma. Estes possuem um nome, descrição e uma frequência de pagamento, que pode ser diária, mensal, trimestral, semestral ou anual. Tem também informação sobre os serviços de pagamento PayPal e Paystack , podendo os pagamentos ser efetuados para uma ou para outra. Por último tem informação sobre o seu estado, que pode ser criada, ativa ou inativa, um preço e se é renovada automaticamente ou não.

- **PaymentOrder**: Representa os pagamentos efetuados na plataforma, registando todos os pagamentos. Possui o identificador da encomenda, estado, data de criação e modificação (caso tenha que ser revertido), nome da subscrição, valor a pagar, serviço em que foi pago e se foi ou vai ser uma devolução.

- **Country**: Representa os países que se encontram na plataforma. Um país pode ter vários estados. Este é necessário para que na parte de procura se possa apresentar os possíveis países onde se podem encontrar serviços e bens.

- **State**: Representa os estados que se encontram na plataforma. Cada estado pertence a um país e pode ter várias cidades. Isto é necessário para que na parte de procura se possa apresentar os possíveis estados onde se podem encontrar serviços e bens.
- **City**: Representa as cidades que se encontram na plataforma. Cada cidade pertence a um estado e possui uma geolocalização. Este é necessário para que na parte de procura se possa apresentar as possíveis cidades onde se podem encontrar serviços e bens.
- **PlatformContent**: Representa o conteúdo da plataforma, como termos de utilização, condições, *cookies*, entre outros. Possui um nome e um texto.
- **PlatformRating**: Representa a avaliação que a plataforma contém.
- **DeletionReason**: Representa as razões pelas quais os utilizadores se eliminaram da plataforma. Um utilizador pode aquando da sua eliminação da plataforma, indicar qual a razão.

4.8 MODELO DE DADOS

O modelo de dados referente à aplicação encontra-se apresentado na Figura 45

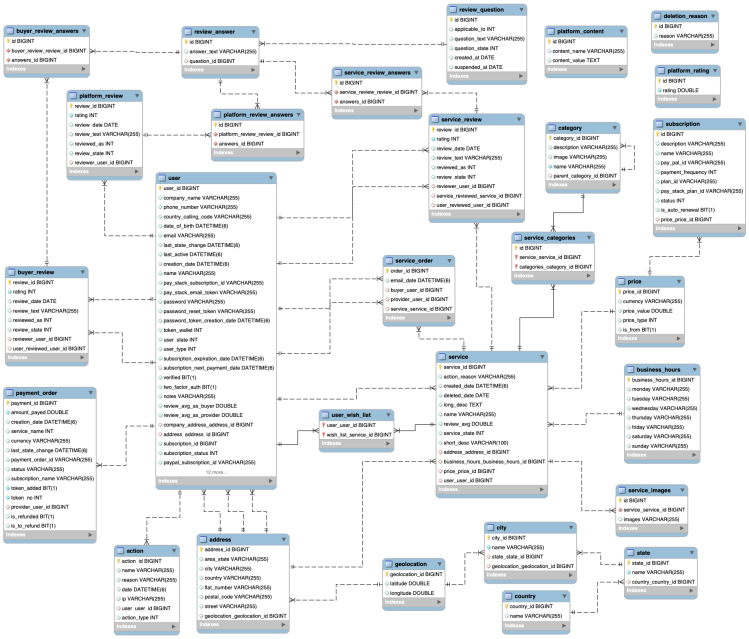


Figura 45: Modelo de dados.

4.9 DIAGRAMA DE *packages*

Foi desenvolvido o diagrama de *packages*, onde se representam os *packages* existentes no sistema e as ligações que existem entre estes. Este diagrama é apresentado na Figura 46 e pode-se observar que existem dois tipos de dependências. Uma das dependências é a de *import*, que significa que um pacote importa funcionalidade de outro pacote. A outra dependência é a de *access*, que significa que o pacote acede a funções do outro pacote.

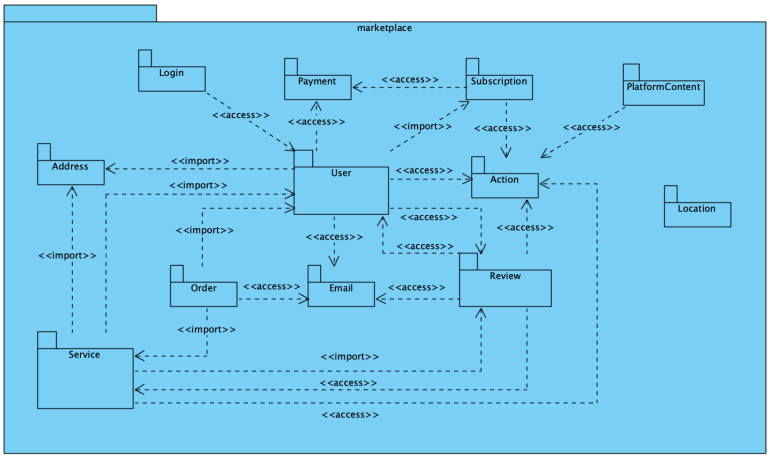


Figura 46: Diagrama de *packages*.

Este diagrama será útil quando se proceder à migração para uma arquitetura de microsserviços.

4.10 DESENVOLVIMENTO DO BACKEND

O objetivo deste tema de dissertação passa pelo desenvolvimento de uma aplicação seguindo uma arquitetura monolítica, transformando-a posteriormente numa arquitetura de microsserviços. Para isso, ir-se-á preparar a aplicação monolítica para ser mais fácil a mudança para uma arquitetura de microsserviços, seguindo várias diretrizes apresentadas no capítulo 3.

4.10.1 *Método de desenvolvimento*

O desenvolvimento do *backend* seguiu os princípios da metodologia Agile [Stellman and Greene (2014)], uma abordagem iterativa à gestão de projetos e desenvolvimento de *software*, que permite entregar valor aos clientes de forma mais rápida. Ao invés de se apostar numa instalação total e final da aplicação, entrega-se valor em pequenas quantidades e iterativamente. Os requisitos de sistema, planos e resultados são avaliados continuadamente, para se poder responder às mudanças de forma rápida.

Mais concretamente, utilizou-se Scrum [(Stellman and Greene, 2014, p. 71)], uma especificação da metodologia Agile. Esta descreve um conjunto de reuniões a realizar, ferramentas e diretrizes para auxiliar as equipas a gerir e

estruturar o seu trabalho. Scrum providencia um processo para identificar o trabalho necessário a realizar, quem irá realizar o quê, como deve ser feito e quando se espera que esteja completo.

4.10.2 *Organização de* packages

Ao invés de seguir uma organização de *packages* como se pode observar na Figura 47, em que todos os componentes do mesmo tipo ficam agrupados num mesmo *package*, vai-se preparar a aplicação para ser mais fácil a migração para uma arquitetura de microsserviços.

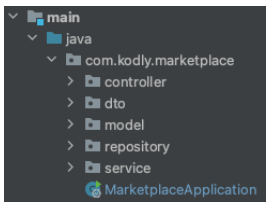


Figura 47: Organização de *packages* conforme o tipo do ficheiro.

Seguindo as diretrizes apresentadas nas secções 3.2 e 3.5, o desenvolvimento da aplicação seguirá uma organização de *packages* para que todos os componentes e funcionalidades relacionados com um determinado componente permaneçam no mesmo *package*, como se pode observar pela Figura 48. Estes componentes são bons candidatos a serem extraídos do monolítico para representarem um serviço independente. A definição de componentes baseia-se na definição de agregados e *Bounded Context*, apresentados nas secções 2.2.1 e 2.2.1 respetivamente. De referir que alguns dos *packages* constatados na Figura 48 não constituem possíveis componentes a serem extraídos, mas sim um conjunto de configurações e funcionalidades comuns a mais do que um componente. Além disso, nem todos os componentes se encontram separados em *packages* com o intuito de formarem, no futuro, um serviço independente. Podem fazer parte de um serviço, principalmente se este for apenas acedido por esse serviço.

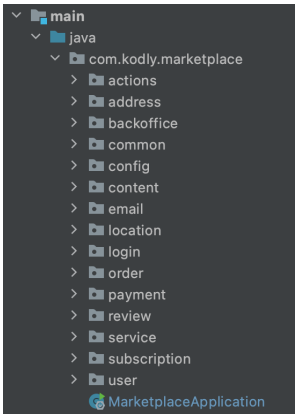


Figura 48: Organização de *packages* de acordo com componentes.

Dentro de cada *package* que representa um componente, as *packages* vão ser organizadas da forma como se pode observar na Figura 49, sendo que pode variar, tendo alguns acréscimos, para lidar com outras situações.

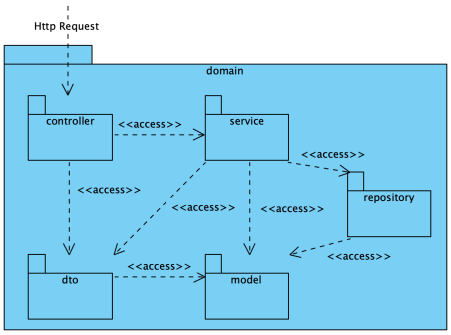


Figura 49: Organização de uma *package*.

A organização de *packages* desta forma, deve-se à arquitetura proposta do *Spring Boot*. Os controladores estão encarregues de receber os pedidos HTTP da API. Estes acedem aos métodos que necessitam dos serviços, que podem ir buscar ou efetuar mudanças aos dados, através dos repositórios e da representação dos modelos. Ainda de referir que existem os DTOs (*Data Transfer Object*), que constituem representações dos modelos, para que o exterior não veja toda a informação, mas sim apenas aquilo a que deve ter acesso.

4.10.3 *Conexão à base de dados*

Spring Boot possui o ficheiro *application.properties* que permite configurar a aplicação, sendo possível também realizar adições a este que podem ser acedidas no código. Este permite também configurar a base de dados através da definição dos atributos apresentados no código 4.1. Neste, define-se o utilizador e palavra passe, bem como o URL de acesso à base de dados. De referir que a palavra passe encontra-se exposta no exemplo, mas é possível recorrer a ferramentas que tratem de encriptar estes dados sensíveis. Para além disso, define-se que a base de dados a utilizar é MySQL , especifica-se a *driver* e indica-se que a geração das tabelas é de *update*, o que faz com que caso exista alguma mudança, esta seja replicada para o esquema da base de dados. Caso se esteja em ambiente local e se tenha a base de dados a correr localmente, o acesso a esta é através do *localhost*.

```
spring.datasource.username=theusername
spring.datasource.password=thepw
spring.datasource.url=jdbc:mysql://localhost:3306/mydb?
    createDatabaseIfNotExist=true
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.
    MySQL8Dialect
```

Código 4.1: Definição da base de dados

Com isto, tem-se a base de dados configurada e a aplicação consegue aceder a esta.

4.10.4 *Acesso à base de dados*

Como mencionado na secção 4.4, foi usado Spring Data JPA para acesso à base de dados MySQL . Esta é uma abstração que reduz significativamente a quantidade de código necessário para acesso à base de dados. Em vez de se implementar DAOs (*Data Access Object*) basta apenas ter *interfaces* que com a ajuda das anotações do Spring Boot , facilitam em muito a escrita de código, dado que fornecem já várias funcionalidades. Ainda como vantagens, temos uma diminuição das funções a ter que gerir e manter, uma maior consistência nos acessos aos dados e nas configurações para acesso à base de dados.

Na Figura 50, pode-se ver um exemplo de um repositório, que é uma *interface* que atua como DAO. É possível ver que esta estende a *interface JPARepository*, o que habilita a que o Spring Data JPA possa encontrar esta *interface* e criar automaticamente uma implementação desta.

Esta implementação automática fornece métodos CRUD. Além disso, é possível criar métodos personalizados, como se pode observar na Figura 50 nas duas funções que se encontram definidas.

```
@Repository
public interface ServiceReviewRepository extends JpaRepository<ServiceReview, Long> {

    @Query("SELECT sr FROM ServiceReview sr " +
            "WHERE (:name IS NULL OR sr.serviceReviewed.name LIKE %:name%) " +
            "AND (:reviewerName IS NULL OR sr.reviewer.name LIKE %:reviewerName%) " +
            "AND (:reviewState IS NULL OR sr.reviewState = :reviewState)")
    Page<ServiceReview> findAllByServiceReviewedAndReviewerAndReviewState(@Param("name") String serviceName, @Param("reviewerName") String reviewerName, @Param("reviewState") ReviewState reviewState, Pageable pageable);

    @Query("SELECT sr FROM ServiceReview sr " +
            "WHERE (:serviceId IS NULL OR sr.serviceReviewed.serviceId = :serviceId) " +
            "AND  (:userReviewedId IS NULL OR sr.userReviewed.userId = :userReviewedId) " +
            "AND (:rating IS NULL OR sr.rating = :rating)")
    Page<ServiceReview> findByServiceReviewedAndUserReviewedAndRating(@Param("serviceId") Long serviceId, @Param("userReviewedId") Long userReviewedId, @Param("rating") Integer rating, Pageable pageable);
}
```

Figura 50: Repositório de avaliações de serviço.

4.10.5 *Funcionamento de um pedido*

Nesta secção demonstra-se o funcionamento de um pedido à REST API da plataforma utilizando a *framework* do Spring Boot , nomeadamente a funcionalidade de criar uma avaliação a um serviço.

Para isso, é preciso criar o *endpoint* para ser possível à *interface* de chamar este para efetuar a criação de uma nova avaliação. Pode-se observar na Figura 51 a definição da classe *ReviewController* que possui as anotações:

- **@RestController**: É uma anotação que incluí a anotação *@Controller*, indicando que é um controlador, mais precisamente um controlador REST.
- **@RequestMapping**: É uma anotação que indica que todos os *endpoints* definidos neste controlador tem como prefixo a *string* especificada, neste caso "*/review*".

```
@RestController
@RequestMapping("/review")
public class ReviewController {

    private final ServiceReviewService serviceReviewService;
```

Figura 51: Controlador para avaliações.

Tendo o controlador definido, pode-se proceder à definição do *endpoint*, que se pode observar na Figura 52. É possível ver que este *endpoint* recebe um *token* que serve para autenticar e autorizar o utilizador e ainda para saber quem foi o utilizador que efetuou a ação. Recebe também através do *path* o identificador do serviço a ser avaliado e através do corpo do pedido a informação necessária para efetuar a avaliação, como o texto, a avaliação dada e possíveis respostas que tenham sido respondidas em relação às perguntas de avaliação.

```
@PostMapping(path = "/service/{serviceId}")
@ResponseStatus(HttpStatus.CREATED)
public ReviewShowDto reviewService(@RequestHeader("Authorization") String authorizationToken, @PathVariable("serviceId") Long serviceId, @RequestBody ReviewSetDto reviewSetDto) {
    return serviceReviewService.reviewService(authorizationToken, serviceId, reviewSetDto);
}
```

Figura 52: *Endpoint* para nova avaliação a serviço.

Com isto, falta criar a função que trate efetivamente de criar a nova avaliação na plataforma. Esta pode ser observado na Figura 53. Esta função busca, através do *token*, o utilizador que efetuou a avaliação, verificando que este não é o fornecedor do serviço, visto que não pode avaliar o seu próprio serviço. Uma vez verificado, calcula a nova média de avaliações, tanto para o fornecedor do serviço, como para o próprio serviço. A seguir, cria a nova avaliação e guarda estes novos cálculos das avaliações tanto no fornecedor como no serviço. Por último, devolve uma representação desta nova avaliação.

```
public ReviewShowDto reviewService(String authorizationToken, Long serviceId, ReviewSetDto reviewSetDto) {
    String email = getEmailFromBearerToken(authorizationToken);
    User reviewer = userService.findByEmail(email);
    Service service = serviceService.findById(serviceId);
    User serviceOwner = service.getUser();

    if (reviewer.getUserId().longValue() != serviceOwner.getUserId().longValue()) {
        Double newProviderAverage = ReviewUtils.calculateAverageForNewProviderReview(serviceOwner, reviewSetDto.getRating());
        Double newServiceAverage = ReviewUtils.calculateAverageForNewServiceReview(service, reviewSetDto.getRating());

        List<ReviewAnswer> answers = reviewSetDto.getAnswers() == null ? null : reviewSetDto.getAnswers().stream().map(this::reviewAnswerSetDtoToReviewAnswer).collect(Collectors.toList());

        ServiceReview newReview = serviceReviewRepository.save(ServiceReview.builder() ServiceReview.ServiceReviewBuilder<...>
                .reviewText(reviewSetDto.getReviewText()) capture of ?
                .rating(reviewSetDto.getRating())
                .reviewer(reviewer)
                .userReviewed(serviceOwner)
                .reviewedAs(UserType.BUYER)
                .serviceReviewed(service)
                .answers(answers)
                .build());
        logger.debug("Review Service successfully added. ID: " + newReview.getReviewId());
        userService.saveNewAverageAsProvider(serviceOwner, newProviderAverage);
        logger.debug(String.format("New Average after creating review (%f) to Provider %s", newProviderAverage, serviceOwner.getUserId()));
        serviceService.saveNewAverage(service, newServiceAverage);
        logger.debug(String.format("New Average after creating review (%f) to Service %s", newServiceAverage, service.getServiceId()));
        return new ReviewShowDto(newReview);
    } else {
        logger.error("Reviewer " + reviewer.getUserId() + " can't be the owner of the service " + service.getServiceId());
        throw new ResponseStatusException(HttpStatus.FORBIDDEN, "Cannot review own service.");
    }
}
```

Figura 53: Função para criação de nova avaliação de serviço.

Este processo está também representado no diagrama de sequência que se pode observar na Figura 54.

4.10.6 *Controlo de versões de esquema de base de dados*

Estando já num estado avançado da aplicação, decidiu-se desativar a geração automática das tabelas da base de dados através das anotações do Spring Boot . Estando a desenvolver a aplicação segundo uma metodologia Scrum, como se falou na secção 4.10.1, existem sempre muitas modificações nos requisitos, propriedades novas que se pretende numa entidade, outras que devem sair, entre outros. É necessário por isso que se consiga gerir estas modificações de uma forma simples e fácil, pois estando já a aplicação em produção, não se pretende que o esquema da base de dados seja eliminado e gerado de novo, nem que se tenha que recorrer a comandos SQL

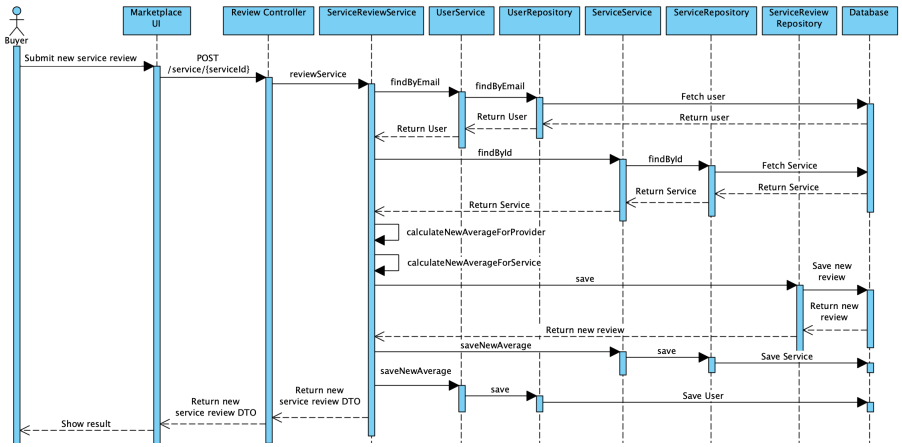


Figura 54: Diagrama de sequência de nova avaliação de um serviço.

para realizar as alterações necessárias. Para isso, passou-se a usar Liquibase , uma ferramenta de código aberto que permite monitorizar, gerir e aplicar mudanças ao esquema da base de dados. Desta forma, conseguimos com que o esquema da base de dados evolva de forma segura.

É necessário primeiro desativar a geração automática do esquema de base de dados. Para isso, basta definir a propriedade *spring.jpa.hibernate.ddl-auto* para *none*, como se observa no código 4.2.

```
spring.jpa.hibernate.ddl-auto=none
```

Código 4.2: Desativação da geração automática do esquema de base de dados

Para o funcionamento do Liquibase , existem um ou mais *changeLogs*, um ficheiro XML ou JSON, que possuem indicações do que realizar e estão atentos às mudanças realizadas neles para poderem efetuar essas mudanças no esquema de base de dados.

Na Figura 55 é possível ver um exemplo de um *changeLog* em JSON que cria uma tabela na base de dados, neste caso a tabela do preço. De notar que se encontra num *changeSet*, que uma vez submetido, não deve ser modificado. Caso se precise de realizar alterações à base de dados, deve-se criar um novo *changeSet* para contemplar essas alterações.

Com isto, falta apenas indicar de onde ler as configurações. Isto pode ser concretizado, definindo o atributo apresentado no código 4.3.

```
spring.liquibase.changeLog=classpath:/db/db.changelog.xml
```

Código 4.3: Definição do ficheiro do Liquibase

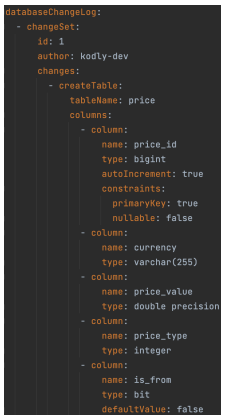


Figura 55: Liquibase - Exemplo de *changeLog* usando JSON.

4.11 INSTALAÇÃO DA APLICAÇÃO

Para se ter a aplicação disponível, é preciso efetuar a instalação desta. Para isso, é necessário instalar a base de dados, o *backend* e o *frontend*. Para a instalação da aplicação vai-se usar Kubernetes . A instalação da base de dados encontra-se no apêndice C.

4.11.1 *Backend*

Para a instalação do *backend*, deve-se primeiro ter a certeza que o *backend* consegue aceder à base de dados. Para isso, no ficheiro *application.properties* é preciso definir a localização da base de dados da forma como se apresenta no código 4.4. É possível verificar que na parte do *host* tem-se ${MYSQL_URL}, porque dado que a instalação ocorre no mesmo *cluster*, este tem acesso à resolução da localização da base de dados através do nome do serviço da base de dados, passado como variável de ambiente.

```
spring.datasource.url=jdbc:mysql://${MYSQL_URL}:3306/mydb?
    createDatabaseIfNotExist=true
```

Código 4.4: Ligação do backend à base de dados

Tendo isto, é necessário compilar a aplicação e colocá-la num ficheiro JAR. Para isso, utilizou-se Gradle para a geração do ficheiro JAR, efetuando o código apresentado em 4.5.

```
gradle build
```

Código 4.5: Geração do ficheiro JAR

A seguir cria-se uma imagem *docker* para que depois se possa colocar esta num *container* e instalar a aplicação. A imagem é criada através de um *Dockerfile* onde se define o que deve ir para dentro do *container*, podendo ser constatado no código apresentado em 4.6.

```
FROM openjdk:11-jdk-slim
EXPOSE 8080

COPY build/libs/*.jar app.jar

RUN apt-get update && \
    apt-get install -y curl

ENTRYPOINT ["java","-jar","/app.jar"]
```

Código 4.6: Dockerfile do backend

Tendo a imagem criada, pode-se passar à construção da imagem, através do comando apresentado em 4.7.

```
docker build -t mrluiigi/marketplace:monolith .
```

Código 4.7: Construção da imagem do backend

Com a imagem criada, pode-se enviar esta para o Docker Hub , um repositório onde se pode gerir imagens *docker*, através do comando apresentado em 4.8. Também é possível utilizar a imagem localmente.

```
docker push mrluiigi/marketplace:monolith
```

Código 4.8: Envio da imagem para o DockerHub

No código apresentado em 4.9, pode-se constatar a definição da instalação, que cria um *container* com o *backend*, definindo o nome do *container*, o número de réplicas, a porta em que vai estar exposta e as variáveis de ambiente.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: marketplace-back
spec:
  replicas: 1
  selector:
    matchLabels:
      app: marketplace-back
  template:
    metadata:
      labels:
        app: marketplace-back
    spec:
      containers:
        - name: app
          image: mrluiigi/marketplace:monolith
          ports:
```

```
        - name: http
          containerPort: 8080
      env:
        - name: JASYPT_ENCRYPTOR_PASSWORD
          value: SaracenSecretKey
        - name: SPRING_PROFILES_ACTIVE
          value: dev
        - name: FRONTEND_URL
          value: http://frontend-service
        - name: MYSQL_URL
          value: mysql
      imagePullPolicy: Always
```

Código 4.9: Criação da instalação do backend

Nas variáveis de ambiente pode-se ver que se define o URL do *frontend*, isto porque a funcionalidade de envio de correios eletrónicos precisa por vezes de saber o endereço do *frontend* para criar URL específicos. Também se passa o nome do serviço da base de dados MySQL , para que o *backend* possa resolver o nome deste e aceder corretamente.

Com a instalação realizada, basta apenas criar o serviço para que o *backend* tenha um *endpoint* estático para poder ser acedido. No código apresentado em 4.10, pode-se observar a definição deste serviço, que seleciona a aplicação denominada *marketplace-back* e a expõe na porta 80.

```
apiVersion: v1
kind: Service
metadata:
  name: marketplace-back
spec:
  selector:
    app: marketplace-back
  ports:
    - port: 80
      targetPort: 8080
  type: LoadBalancer
```

Código 4.10: Criação do serviço do backend

Tendo tudo configurado, basta correr o comando apresentado em 4.11 para correr esta instalação, assumindo que o nome do ficheiro onde se encontram as definições apresentadas nos códigos anteriores é denominado *monolith.yaml*.

```
kubectl apply -f monolith.yaml
```

Código 4.11: Instalação do backend

Com isto, pode-se aceder ao URL do *backend* realizando-se o comando apresentado em 4.12.

```
minikube service marketplace-back --url
```

Código 4.12: Acesso ao backend

Tendo o resultado da execução do comando apresentado no código 4.12, pode-se aceder ao Swagger como se observa na Figura 56.

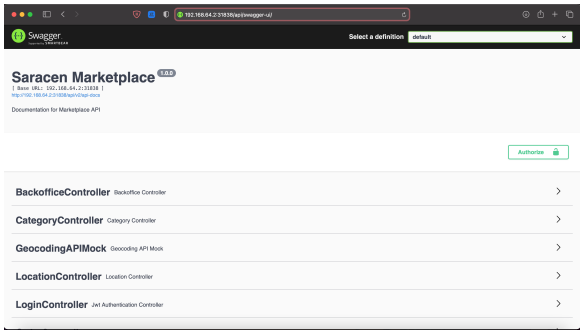


Figura 56: Acesso ao url do backend.

4.11.2 *Frontend*

A instalação do *frontend* é possível ser observada no apêndice D.

De referir que neste apêndice é possível observar no *dockerfile* apresentado no código D.1, que se copia o código relativo ao *frontend* para o *container*, procedendo à sua instalação. Além disso, tem-se também a configuração de um *reverse proxy*, neste caso NGINX , que está encarregue de redirecionar os pedidos corretamente para o *backend* e que vai ser bastante útil quando se começar a extrair microsserviços do monolítico.

Em relação ao *reverse proxy*, este está configurado para que todos os pedidos feitos para */api* sejam redirecionados para o *backend*. Na aplicação do *frontend* todos os pedidos são efetuados para si próprio com o sufixo */api*, para que no *reverse proxy* se possa intercetar estes pedidos e redirecionar corretamente para o *backend*. Isto é possível, porque o *frontend* tem acesso ao nome do serviço do *backend* e pode resolver a localização deste. Para isso, na configuração do NGINX , deve-se realizar o que está no código 4.13, assumindo que o serviço do *backend* denomina-se *marketplace-back*, como foi demonstrado no código 4.10.

```
location /api {
    proxy_pass http://marketplace-back;
}
```

Código 4.13: Configuração do Nginx

5

MIGRAÇÃO DA APLICAÇÃO PARA MICROSSERVIÇOS

Com a aplicação monolítica desenvolvida e o processo de como se procederá à migração definido, pode-se efetivamente partir para a migração da aplicação para uma arquitetura de microsserviços.

5.1 PROCESSO DE MIGRAÇÃO

Em primeiro lugar, aborda-se o processo de migração, ou seja, o que foi realizado para preparar a aplicação, como será realizada, que padrões de migrações serão usados, entre outros.

É preciso mencionar que este processo de migração começou já quando se procedeu ao desenvolvimento da aplicação monolítica. Como mencionado na secção 4.10.2, organizou-se os *packages* de modo que todos os componentes e funcionalidades de um determinado agregado ficassem agrupados. Desta forma, define-se já bons candidatos a serem extraídos do monolítico para constituírem um microsserviço.

Existem, no entanto, acoplamento entre estes *packages*, devido às interações e ligações entre eles, que vão ter que ser lidados quando forem removidos do monolítico. Começando a pensar na migração para microsserviços, é fundamental o desenho do diagrama de *packages* entre componentes no monolítico. Por isso, em primeiro lugar importa modelar este diagrama e posteriormente começar a migração. Este diagrama permite ter uma visão geral da arquitetura da aplicação, em termos de ligações que existem entre componentes, o que permite perceber quais são os componentes mais fáceis e os mais difíceis de migrar. Além disso, é possível saber que componentes vão ser necessários de ser modificados com a extração de um componente para um microsserviço.

Uma vez definido o diagrama de *packages*, começa-se a migração para uma arquitetura de microsserviços. Seguindo as indicações apresentadas na secção 3.1, vai-se proceder a uma migração incremental, para ser mais fácil o retrocesso caso algum problema ou erro aconteça.

O processo passou por começar por extrair um componente que não dependia de outros componentes, mas que possuía quem dependesse dele, sendo o componente de email escolhido para tal. Como segundo serviço, extraiu-se do monolítico um componente que não possuía quem dependesse dele, mas que dependia de componentes do monolítico, como foi o caso do componente de *login*. Por último, extraiu-se um componente que possuía dependentes e que dependia de outros, sendo o componente de avaliações o escolhido. Desta forma, abordam-se as possibilidades que os componentes podem ter em termos de dependências, mostrando como foi efetuada a sua migração.

Numas extrações ir-se-á seguir o padrão *Strangler Fig Application* apresentado na secção 3.3.1. Desta forma, temos a antiga e a nova implementação a coexistir em simultâneo, dando tempo à nova implementação de crescer e de ser bem implementada, podendo ser testada devidamente. Quando a nova implementação estiver completa, pode-se proceder à remoção da antiga e usar a nova. Caso algo corra mal, estas modificações não tiveram efeito na aplicação, visto que ainda não foi usada na aplicação e é fácil o retrocesso.

Noutros casos, o componente a ser extraído possui vários dependentes no monolítico. Nestes casos, será adotado o padrão de migração denominado *branch by abstraction*, para efetuar a migração de componentes que são bastante utilizados no monolítico, recorrendo ao uso de abstrações.

Por último, também existem situações onde os serviços extraídos necessitam de informação que se encontra no monolítico. Para isto, ir-se-á recorrer ao padrão denominado monolítico com agregados expostos, apresentado na secção 3.8.1.

Em relação à base de dados, optar-se-á por uma separação física, dado que se pretende mais robustez, não se quer um ponto único de falha e se pretende que este serviço seja totalmente independente do exterior. Esta questão foi abordada na secção 3.5.1.

De referir que no nosso caso prático, esta migração para microsserviços é efetuada após o término do MVP (minimum viable product) da aplicação e não se aplicou mais modificações a este em termos de novas funcionalidades. Contudo, existem situações em que esta migração pode ocorrer enquanto se vai adicionando valor à aplicação ou até mesmo quando já se encontra em produção, o que torna este processo mais complicado, pois é preciso contemplar estas mudanças para os novos serviços extraídos do monolítico e garantir que se mantém a aplicação disponível para os utilizadores.

5.2 DEPENDÊNCIAS ENTRE COMPONENTES

Antes de se começar a migração, é importante que se tenha uma visão geral da aplicação monolítica, isto é, que se perceba as dependências que cada componente tem e de quais depende. Na Figura 46 na página 71 é possível observar este diagrama de *packages* relativo à aplicação monolítica desenvolvida.

O ideal é começar por extrair aqueles mais exteriores, aqueles que não possuem componentes dependentes deles próprios. Na Figura 46 na página 71, pode-se ver que os candidatos a serem removidos são o **Login**, que apenas depende do **User** e que não tem nenhum que dependa dele. Outros candidatos são o **Subscription**, **Action**, **Email**, entre outros. De reparar que temos o componente **Location** que não tem nenhum que dependa dele e não depende de nenhum outro componente. Estes componentes são os mais fáceis de extrair, visto que provavelmente não se efetuará mudanças noutros componentes e neles próprios.

Todavia, nem sempre se pretende iniciar a extração por aqueles que possuem poucas dependências. Pode-se precisar extrair um componente que tenha mais dependências, porque se precisa de mais desempenho ou outra razão qualquer. Neste caso, na Figura 46, um componente mais difícil de extrair seria o **User** ou o **Service**, visto que possuem várias dependências para outros como para eles próprios.

5.3 EXTRAÇÃO DO COMPONENTE DE EMAIL

O primeiro microsserviço a extrair será um componente que não depende de ninguém, mas que tem quem dependa dele. Um bom exemplo disso é o componente **Email**, que possui vários componentes no monolítico que dele dependem.

Este componente está encarregue de tratar do envio de correios eletrónicos. Visto que se extrairá um componente que possui dependentes dele, será necessário modificar o monolítico para que este passe agora a efetuar as chamadas ao microsserviço remoto e não localmente.

5.3.1 *Desenvolvimento da funcionalidade*

De modo a desacoplar este microsserviço dos restantes, usar-se-á um *message broker*, que permite uma comunicação assíncrona. Desta forma, os microsserviços que necessitarem da funcionalidade de envio de correios eletrónicos, não necessitam de ter conhecimento do microsserviço de *email*. Estes tem apenas conhecimento do *message broker*, que trata da receção e envio das mensagens. Para o caso prático, RabbitMQ é o escolhido como *message broker*.

No anexo E é possível observar a instalação deste e a configuração em Spring Boot .

Com o RabbitMQ operacional, deve-se escutar no microsserviço de *email* os pedidos de envio de correios eletrónicos.

Um dos possíveis correios eletrónicos a ser enviado para um utilizador é o de quando este se regista na plataforma. É enviado para o utilizador um correio eletrónico que indica que este foi registado na plataforma com sucesso e que precisa de ativar a conta, seguindo-se uma ligação que o leva para o *frontend* para efetuar a ativação da sua conta.

Foi criado um componente em Spring Boot que escuta a fila por possíveis mensagens lá colocadas. Este componente é apresentado na Figura 57. Neste pode-se ver que este escuta a fila definida nas propriedades do Spring Boot e que tem uma função definida, denominada *sendRegisterEmail*, que recebe um objeto que possui as informações necessárias para um correio eletrónico de registo. Para tal, a função deve ter a anotação *@RabbitHandler*.

```
@Component
@RabbitListener(queues = "${rabbitmq.queue}")
public class QueueListener {

    private final SendgridService sendgridService;

    @RabbitHandler
    public void sendRegisterEmail(RegisterEmailDto email) {
        sendgridService.sendEmail(email);
    }
```

Figura 57: Componente para escutar mensagens colocadas na fila do RabbitMQ.

De seguida, é preciso, a partir dos dados do objeto recebido, definir o correio eletrónico e enviar. Para o envio destes é utilizado SendGrid , um serviço de entrega de correios eletrónicos, recorrendo à função *sendEmail*,

apresentada na Figura 58, que recebe um objeto *Visitable*. Para esta funcionalidade de envio de correios eletrónicos, utilizou-se o padrão comportamental *Visitor*. Este padrão é utilizado quando se quer realizar uma operação a vários objetos similares entre si. Neste caso, a operação que se quer realizar é o envio de um correio eletrónico. Esta recebe um objeto, que dependendo do tipo de correio eletrónico, possui diferentes atributos.

```
@Async
public Object sendEmail(Visitable email){
    if(sendgridEnabled){
        email.accept(visitor);
        Mail mail = visitor.getMail();
        try {
            SendGrid sendGrid = instantiateSendgrid();
            Request request = instantiateRequest();
            request.setMethod(Method.POST);
            request.setEndpoint("mail/send");
            request.setBody(mail.build());
            Response res = sendGrid.api(request);
            logger.debug("Email sent");
            return res;
        } catch (IOException ex) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST);
        }
    } else {
        logger.debug("No email sent");
        throw new ResponseStatusException(HttpStatus.NO_CONTENT);
    }
}

@NotNull
public Request instantiateRequest() { return new Request(); }

@NotNull
public SendGrid instantiateSendgrid() { return new SendGrid(apikey); }
```

Figura 58: Função para envio de correios eletrónicos.

Dependendo do tipo de correio eletrónico, este é preparado, povoando os campos necessários. Na Figura 59 é possível ver-se a preparação de um correio eletrónico relativo a um novo registo. Para tal, inicializa-se este, indicando qual o endereço eletrónico e o nome do remetente. A seguir, é definido o assunto, o modelo a ser usado que foi criado na plataforma do SendGrid , a adição da variável *Validation_Key* para se poder ter acesso a esta e termina-se com uma personalização final, onde se define as partes comuns, como o nome e endereço eletrónico do utilizador.

```
public void visit(RegisterEmailDto emailDto) {
    initializeMailPersonalization(adminEmail, MTSU_ADMIN);
    String subject = "Welcome to Saracen Marketplace!";
    String registerTemplateId = "d-81f89e7361d545bb83ddb53f56f47461";

    personalization.addDynamicTemplateData("Validation_Key", emailDto.getValidationKey());
    setFinalMailPersonalization(emailDto.getUserEmail(), subject, registerTemplateId, emailDto.getUserName());
}

private void setFinalMailPersonalization(String userMail, String subject, String templateId, String userName) {
    Email to = new Email(userMail);
    mail.setTemplateId(templateId);
    if (userName != null) {
        personalization.addDynamicTemplateData(USER_NAME, userName);
    }
    personalization.addTo(to);
    mail.setSubject(subject);
    personalization.addDynamicTemplateData("Subject", subject);
    personalization.addDynamicTemplateData(SENDER_NAME, MARKETPLACE);
    mail.addPersonalization(personalization);
}
```

Figura 59: Preparação do correio eletrónico de registo.

5.3.2 *Instalação do microsserviço*

Com o microsserviço desenvolvido, pode-se proceder à instalação deste. Visto que já se tem a aplicação instalada como foi explicado na secção 4.11, basta instalar o microsserviço e redirecionar os pedidos para o microsserviço correto.

A instalação do microsserviço é semelhante à instalação do *backend*, apresentada na secção 4.11.1. A única diferença é a definição dos ficheiros de instalação em Kubernetes , nomeadamente as variáveis de ambiente. No código apresentado em 5.1, pode-se observar as variáveis de ambiente que este recebe.

```
env:
  - name: SENDGRID_API_KEY
    value: SG.8UP8x1AyQlahA7nQgiRXEQ.5NmPs21Y1DID4a6C3OpKocoZRszu-
        axJRSeed4uQCtI
  - name: RABBITMQ_URL
    value: rabbitmq
```

Código 5.1: Variáveis de ambiente da instalação do microsserviço email

5.3.3 *Modificação do monolítico*

Com a extração desta funcionalidade para um microsserviço independente e a sua instalação efetuada, é necessário realizar mudanças ao monolítico, dado que esta funcionalidade é utilizada em vários componentes do monolítico.

Deve-se configurar o monolítico conforme o que se apresentou no anexo E, para este saber o endereço do RabbitMQ de modo a conseguir publicar as mensagens de envio de correios eletrónicos.

Para se receber a variável do endereço do RabbitMQ , deve-se realizar modificações nos ficheiros de instalação do monolítico, adicionando nas variáveis de ambiente o valor desta, como se mostra no código 5.2, assumindo que o serviço é denominado *rabbitmq*.

```
env:
    - name: RABBITMQ_URL
      value: rabbitmq
```

Código 5.2: Definição do endereço do RabbitMQ na instalação do monolítico

Com o monolítico a ter este endereço, pode-se proceder à modificação das funcionalidades que necessitam do envio de correios eletrónicos. Uma destas funcionalidades é o registo de um utilizador, como se referiu anteriormente.

Para isso, criou-se o componente apresentado na Figura 60, que trata de enviar os objetos relativos aos correios eletrónicos para a fila de espera no RabbitMQ .

```java
@Component
public class SendgridService {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Value("${rabbitmq.queue}")
    private String queue;

    @Async
    public void sendEmail(Object email){
        rabbitTemplate.convertAndSend(this.queue, email);
    }
}
```

Figura 60: Componente para envio de correios eletrónicos para a fila de espera.

Com isto, na função onde se efetuava a chamada local para envio deste correio eletrónico, substituí-se por esta que realiza o envio deste objeto para a fila de espera.

Concluindo, pode-se testar a funcionalidade de registo na totalidade. O resultado esperado deve ser a receção de um correio eletrónico, como se mostra na Figura 61.

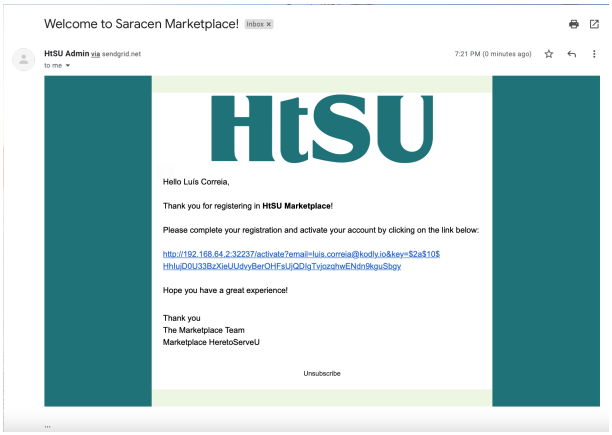


Figura 61: Receção do email de registo.

Desta forma, deixa-se de ter o monolítico a tratar do envio de correios eletrónicos e pode-se remover esta funcionalidade deste, quando se tiver a certeza que tudo se encontra correto. O fluxo da funcionalidade de registo, que também se aplica a outras funcionalidades que envolvam envio de correios eletrónicos, é apresentado na Figura 62.

5.4 EXTRAÇÃO DO COMPONENTE DE LOGIN

Chegando a este ponto, temos um microsserviço extraído do monolítico. O primeiro microsserviço não dependia de ninguém, mas possuía quem dependesse dele, o que levou a que mudanças tivessem que ser efetuadas no monolítico. Um possível candidato para a próxima extração é um componente que não tem ninguém que dependa dele, mas que depende de um ou mais componentes.

Observando o diagrama de *packages*, apresentado na Figura 46 na página 71, um bom candidato é o componente **Login**.

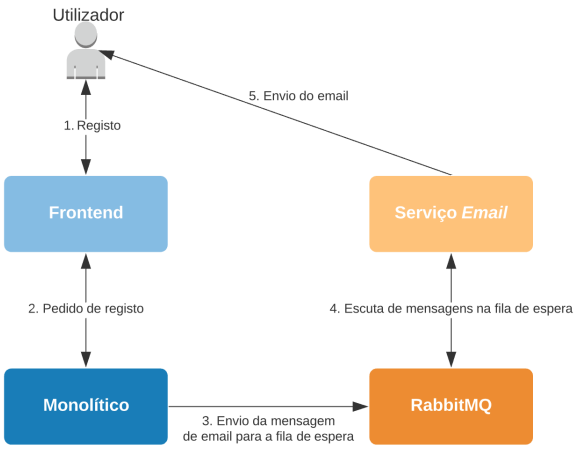


Figura 62: Fluxo da funcionalidade de registo.

É possível observar que este é dependente do componente do utilizador, o que faz com que seja necessário modificar o monolítico para este disponibilizar acesso a dados ou funcionalidade, de modo que as funcionalidades continuem a funcionar neste microsserviço, pois o que antes era feito localmente, agora tem que ser efetuado remotamente, como foi mencionado na secção 3.8.1.

Enquanto no microsserviço de email, apresentado na secção 5.3, se modificava o monolítico para este poder efetuar chamadas a funcionalidades do exterior, neste caso, ir-se-á modificar o monolítico para que o exterior possa aceder às funcionalidades deste.

O componente de *login* está encarregue da autenticação, do registo de um utilizador e administrador e de recuperar e alterar a palavra-passe. Além disso, está encarregue de gerir a autenticação de dois fatores.

5.4.1 *Modificação da autenticação*

Para que este microsserviço possa fornecer as suas funcionalidades, é necessário efetuar alterações neste e também no monolítico. Vai-se demonstrar a modificação da funcionalidade de autenticação, sendo que as outras funcionalidades possuem modificações semelhantes.

Na Figura 63 pode-se observar a definição do *endpoint /login*, definido pela função denominada *createAuthenticationToken*. Este é um método POST, devolvendo um objeto que contém o *token* de autorização, bem como outra informação. Para poder devolver este objeto, é preciso ter a certeza que o utilizador existe, está ativado e que forneceu a palavra passe correta.

Para isso, existe o serviço *userDetailsService* que possui a função *retrieveLoginUser* que está encarregue de ir buscar um utilizador a partir de um endereço eletrónico.

No monolítico, a forma de ir buscar o utilizador é ir diretamente à base de dados do utilizador, como se pode observar na Figura 64. É possível ver que através do repositório do utilizador se busca o utilizador respetivo através do endereço eletrónico. O resultado desta procura vai para um objeto opcional, que pode ou não conter outro objeto. Caso este opcional tenha algum utilizador, é porque o utilizador existe e pode então retornar este. Caso contrário, é porque o utilizador não existe e então atira uma exceção, falhando a autenticação.

```java
@RestController
@RequestMapping("/login")
@Api(tags = "LoginController")
public class JwtAuthenticationController {

    protected final Log logger = LogFactory.getLog(this.getClass());

    private final AuthenticationManager authenticationManager;
    private final JwtTokenUtil jwtTokenUtil;
    private final JwtUserDetailsService userDetailsService;

    public JwtAuthenticationController(AuthenticationManager authenticationManager, JwtTokenUtil jwtTokenUtil, JwtUserDetailsService userDetailsService) {
        this.authenticationManager = authenticationManager;
        this.jwtTokenUtil = jwtTokenUtil;
        this.userDetailsService = userDetailsService;
    }

    @PostMapping
    public ResponseEntity<JwtResponse> createAuthenticationToken(@Validated @RequestBody JwtRequest authenticationRequest) {
        User userInfo = userDetailsService.retrieveLoginUser(authenticationRequest.getEmail());
        if (userInfo.getUserState().equals(UserState.ACTIVE)) {
            authenticate(authenticationRequest.getEmail(), authenticationRequest.getPassword());
            final UserDetails userDetails = userDetailsService.loadUserByUsername(authenticationRequest.getEmail());
            final String token = jwtTokenUtil.generateToken(userDetails);
            return ResponseEntity.ok(new JwtResponse(token, userInfo.getEmail(), userInfo.getName(), userInfo.getUserType(), userInfo.getTwoFactorAuth()));
        } else {
            logger.error("User " + userInfo.getEmail() + " isn't active therefore can't login.");
            throw new UserNotActiveException("User is not active therefore can't login.");
        }
    }
```

Figura 63: Definição do *endpoint* de login.

```java
public LoginUser retrieveLoginUser(String email) {
    Optional<LoginUser> user = loginUserRepository.findByEmail(email);
    if (user.isPresent()) {
        return user.get();
    } else {
        logger.error("Email extracted from token not found");
        throw new BadCredentialsException(BAD_CREDENTIALS);
    }
}
```

Figura 64: Obter o utilizador no monolítico.

Ao extrair o componente de *login* para um microsserviço independente, este perde acesso direto à tabela do utilizador. É necessário que o componente do utilizador disponibilize alguma forma deste microsserviço aceder aos seus dados, como se observa na Figura 65. Neste caso, o componente utilizador disponibiliza um *endpoint* que devolve um utilizador através de um endereço eletrónico. Possui a mesma lógica e devolve o mesmo resultado, tendo apenas como diferença ser um pedido remoto.

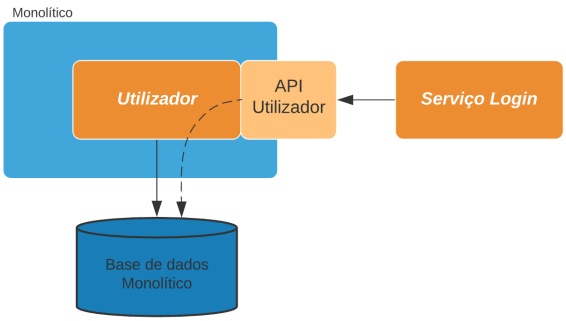


Figura 65: Componente utilizador a expor uma API no monolítico que permita acesso aos seus dados.

Com isto, no componente do utilizador no monolítico, define-se um *endpoint*, como se constata na Figura 66, que possa ser acedido e devolva um utilizador a partir de um endereço eletrónico. Este *endpoint* tem como prefixo "*/user*".

Este *endpoint* chama um serviço que tem a função apresentada na Figura 67, que realiza o mesmo código que na Figura 64, sem lidar com a presença ou não do utilizador.

```java
@GetMapping(path = "/email")
public Optional<User> getUserByEmail(@RequestParam String email){
    return userService.getUserByEmail(email);
}
```

Figura 66: Componente utilizador a expor *endpoint* para buscar um utilizador por endereço eletrónico.

```java
public Optional<User> getUserByEmail(String email) {
    return userRepository.findByEmail(email);
}
```

Figura 67: Serviço do componente utilizador para buscar um utilizador por endereço eletrónico.

Com isto, pode-se modificar a função no microsserviço de login onde se obtém o utilizador, como se apresenta na Figura 68.

```java
public User retrieveLoginUser(String email) {
    Optional<User> user = httpClient.getUserByEmail(email);

    if (user.isPresent()) {
        return user.get();
    } else {
        logger.error("Email extracted from token not found");
        throw new BadCredentialsException(BAD_CREDENTIALS);
    }
}
```

Figura 68: Obter o utilizador no microsserviço de login.

É possível observar, que ao invés de se aceder diretamente à base de dados, como se concretizava na Figura 64, é chamada uma função para ir buscar um utilizador através de um pedido HTTP. A função apresenta-se na Figura 69, sendo possível observar que é chamado o *endpoint* criado na Figura 66, passando o endereço eletrónico como parâmetro. Esta função retorna um utilizador caso venha no resultado da chamada e retorna uma exceção caso não venha nenhum utilizador ou algum erro aconteça. A variável *monolithUrl* é passada através de variáveis de ambiente, sendo esta mencionada na secção 5.4.3.

```java
public Optional<User> getUserByEmail(String email) {
    try {
        String url = monolithUrl + "/user/email";
        Map<String, String> params = new HashMap<>();
        params.put("email", email);
        User user = this.sendJsonRequest(url, HttpMethod.GET, body: null, params, headers: null, User.class);
        return Optional.ofNullable(user);
    } catch (HttpException httpException){
        throw new ResponseStatusException(HttpStatus.resolve(httpException.getResponseCode()), httpException.getMessage());
    } catch (Exception e) {
        e.printStackTrace();
        throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE, "Service Unavailable");
    }
}
```

Figura 69: Obter o utilizador no microsserviço de login através de pedido HTTP.

5.4.2 *Interação com o microsserviço de email*

O microsserviço de login possui funcionalidades que necessitam do microsserviço de email para envio de correios eletrónicos. Uma dessas funcionalidades é o registo de um utilizador ou a recuperação da palavra-passe.

Para isto, no microsserviço de login é preciso efetuar as mudanças realizadas no monolítico aquando da remoção do microsserviço de email, apresentadas na secção 5.3.3. O código relativo a correios eletrónicos que

sejam das funcionalidades de login, pode ser removido do monolítico, visto que este deixa de ser responsável por estas funcionalidades.

Na Figura 70, é apresentado o fluxo da funcionalidade de registo, em contraste com o que se apresentou na Figura 62 na página 87. Pode-se ver que o monolítico deixou de ser responsável por receber os pedidos que são agora redirecionados para o microsserviço de login, sendo apenas utilizado para a obtenção e criação de utilizadores. Também se observa que é o microsserviço de login que está encarregue de efetuar o pedido de envio do correio eletrónico para o RabbitMQ . No caso da criação de um utilizador falhar, sendo porque já existe um utilizador com o mesmo endereço eletrónico ou, porque o pedido não chegou ao monolítico e não foi possível a resposta, o envio de correio eletrónico não é efetuado.

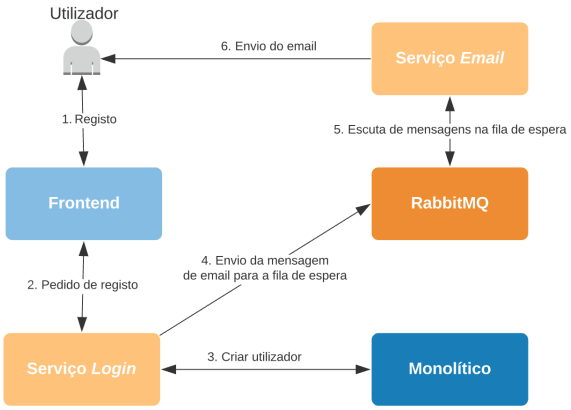


Figura 70: Fluxo da funcionalidade de registo com o microsserviço de Login.

Nesta Figura apresenta-se o fluxo da funcionalidade de registo de utilizadores, mas este aplica-se a outras funcionalidades presentes no microsserviço de login.

5.4.3 *Instalação do microsserviço*

Para a instalação do microsserviço de login, os passos são semelhantes ao que já se apresentou nas outras secções anteriormente, tal como a secção 5.3.2.

A maior diferença são as variáveis de ambiente que o microsserviço necessita. No código apresentado em 5.3, pode-se observar a definição destas variáveis de ambiente. A resolução destes endereços no microsserviço é conseguida através de Kubernetes .

```
env:
  - name: FRONTEND_URL
    value: http://frontend-service
  - name: RABBITMQ_URL
      value: rabbitmq
  - name: MONOLITH_URL
    value: http://marketplace-back
```

Código 5.3: Definição das varíaveis de ambiente do microsserviço login

5.4.4 *Redirecionamento no frontend*

O *frontend* realiza os pedidos ao monolítico. Dado que se extraiu este microsserviço do monolítico, deve-se agora redirecionar os pedidos que envolvam funcionalidades relativas com este para ele próprio. Este redirecionamento pode ser feito recorrendo ao *reverse proxy* NGINX que se encontra no *frontend*, como mencionado na secção 4.11.2, para que os pedidos vão para o microsserviço correto.

Todos os *endpoints* do microsserviço do *Login* tem os prefixos */login*, */register* e */code*. Na secção *location* da configuração do NGINX , intersetam-se todos os pedidos que tenham */api* mais os prefixos apresentados anteriormente e assim estes são redirecionados para o microsserviço de login.

Com esta mudança, este redirecionamento comporta-se da forma que se apresenta na Figura 71.

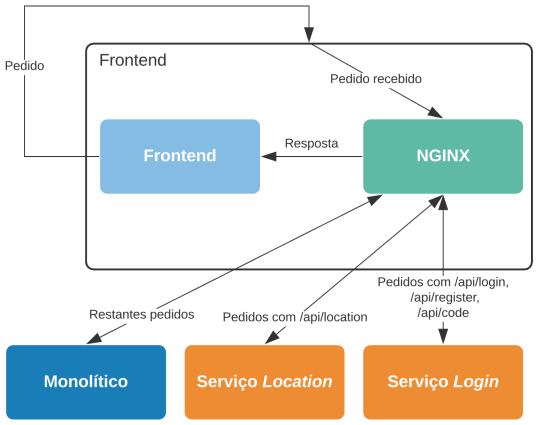


Figura 71: Redirecionamento dos pedidos para o novo microsserviço de login.

Para tal funcionar, é preciso que se tenha no ficheiro de configuração do NGINX o bloco de código que se apresenta no código 5.4.

```
location /api/login {
    proxy_pass http://login;
}
location /api/register {
    proxy_pass http://login;
}
location /api/code {
    proxy_pass http://login;
}
```

Código 5.4: Redirecionamento dos pedidos para o novo microsserviço

Com estas modificações, é necessário reiniciar o *frontend*, para este ter as mudanças efetuadas na configuração do *reverse proxy* e se poder testar e ter a certeza que é o microsserviço de login que recebe os pedidos relativos a ele.

Para se ter a certeza, pode-se proceder à remoção de todo o código relativo ao microsserviço de login que se encontra no monolítico, proceder à sua instalação e verificar no *frontend* os pedidos continuam a ser efetuados com sucesso.

Outra opção seria verificar o histórico de mensagens do microsserviço de login, que se encontra em modo depurador, para se ver se os pedidos relativos a este chegam. Para isso, pode-se proceder ao comando apresentado em 5.5, onde *login_ID* é o identificador do serviço de login.

```
kubectl logs <login_ID> --follow
```

Código 5.5: Histórico de mensagens do microsserviço login

5.5 EXTRAÇÃO DO COMPONENTE DE AVALIAÇÕES

A próxima extração do monolítico, será um componente que possui dependentes dele próprio e que depende de outros. Observando o diagrama de dependências, apresentado na Figura 46, um bom candidato é o componente de avaliações.

O componente de avaliações está encarregue de gerir as avaliações, incluindo as que são efetuadas aos consumidores, aos serviços e à plataforma. Além disso, também está encarregue de gerir as respostas de avaliação.

É possível observar que o componente de avaliações é dependente dos componentes de utilizador, ações, email e serviço. É dependente do utilizador e do serviço porque necessita de informação destes, quando se quer saber a informação sobre uma avaliação, para se saber qual foi o serviço avaliado e quem foi o utilizador que efetuou à avaliação ou qual a recebeu. Depende do email para poder enviar correios eletrónicos acerca de avaliações, por exemplo, quando um utilizador avalia a plataforma, é enviado um correio eletrónico a agradecer a avaliação à plataforma. Por último, depende do componente de ações para poder registar as ações realizadas na plataforma, tal como novas avaliações, alterações destas ou eliminações.

Estas dependências faz com que seja necessário modificar o monolítico para se disponibilizar *endpoints*, de modo que o microsserviço de avaliações possa aceder às funcionalidades que precisa, como foi mencionado na secção 3.8.1. Além disso, é possível ver que o componente do utilizador e do serviço dependem deste, porque estes componentes precisam de saber quais são as avaliações referentes a um utilizador e a um serviço. No monolítico quando é realizado um pedido para saber a informação de um utilizador, é necessário também saber quais são as avaliações que este recebeu. O mesmo acontece para os serviços.

Por último, estas dependências encontram-se também a nível de dados. As avaliações podem ser de três tipos, uma avaliação a um consumidor (Buyer Review), a um serviço (Service Review) e à plataforma (Platform Review). É possível observar na Figura 45, que existem relações entre as tabelas que representam estas avaliações e as tabelas do utilizador e do serviço, o que implica que a extração deste componente do monolítico, leve também a uma separação do modelo de dados.

Como mencionado na secção 5.1, seguir-se-á um processo de migração incremental para diminuir os problemas que apareçam.

A seguir, demonstra-se a extração da funcionalidade de avaliar um serviço, sendo que para o resto das funcionalidades, o processo é semelhante.

5.5.1 *Separação do modelo de dados*

É boa prática a extração dos dados relativos ao microsserviço para este também. Era possível continuar a ter os dados relativos às avaliações no monolítico, como referido na secção 3.4, sendo que não se perdia as vantagens associadas com o modelo relacional, mas acaba-se por não ter uma separação total, por não se ter um microsserviço coeso, dado que não possui os seus próprios dados.

A separação destes dados implica a quebra das relações referidas anteriormente, algo mencionado na secção 3.5.3. As colunas nas tabelas das avaliações que atuam como chaves estrangeiras para os utilizadores e para os serviços, deixam de ser chaves estrangeiras, visto que agora não vão poder ter referência para estas, dado que se encontram em esquemas de base de dados diferentes. Todavia, continuam a existir estas colunas, porque a identificação destes utilizadores e serviços necessita de continuar a ser efetuada.

Na Figura 72, pode-se observar o modelo de dados relativo às avaliações. Este modelo de dados representa o esquema de base de dados do microsserviço de avaliações. Pode-se constatar que as tabelas são equivalentes às que se apresentam na Figura 45, exceto que em vez de chaves estrangeiras nas tabelas *buyer_review*, *platform_review*, *service_review*, temos apenas uma coluna que representa a identificação do utilizador ou do serviço. De notar, que entre as tabelas do próprio esquema, estas continuam a se relacionar, recorrendo às chaves estrangeiras.

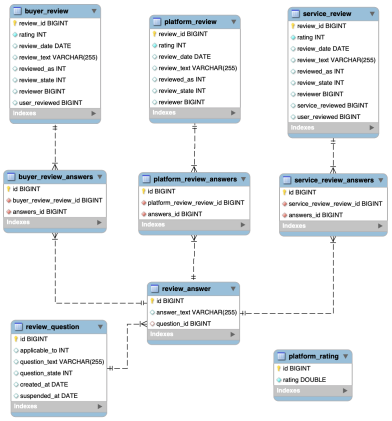


Figura 72: Modelo de dados do microsserviço de avaliações.

Acrescenta-se complexidade, visto que se tem que gerir estas colunas que não são chaves estrangeiras, mas que atuam como tal e manter a consistências no sistema. Este tema é abordado na secção 3.5.3, onde se apresentam algumas opções para cada situação.

5.5.2 *Diagrama de sequência de nova avaliação*

A plataforma disponibiliza a possibilidade de um consumidor avaliar um serviço. Para isso, no *backend*, existe um *endpoint* que permite a realização desta avaliação. Na Figura 54 é possível observar o diagrama de sequência de uma nova avaliação de um serviço. No *frontend*, basta chamar este *endpoint* e passar-lhe os argumentos necessários. Como este componente de avaliações está a ser extraído, não se poderá aceder diretamente aos componentes do utilizador e do serviço para ir buscar a informação necessária.

Na Figura 73, temos um novo diagrama de sequência, que em contraste com o apresentado na Figura 54, contempla o microsserviço de avaliações extraído do monolítico.

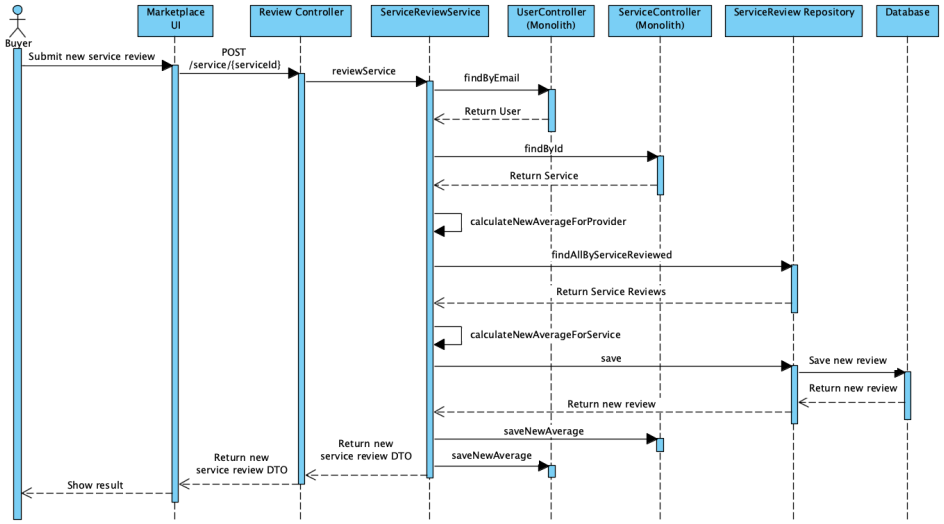


Figura 73: Diagrama de sequência de avaliação de um serviço com a separação do microsserviço de avaliações.

É possível observar que se continua a ter que ir buscar o utilizador que efetuou a avaliação e o serviço avaliado, para se poder proceder às alterações necessárias. No entanto, estas buscas não são locais, mas sim remotas, o que implica realizar alterações no monolítico para este providenciar os *endpoints* que permitam realizar estas buscas por via remota. A funcionalidade destes *endpoints* continua a ser a mesma que se encontra apresentada na Figura 54.

Além disso, pode-se observar que existe uma busca extra pelas avaliações que o serviço recebeu, porque quando um serviço é avaliado, a sua média de avaliações tem que ser atualizada em função da que possui e da nova que recebeu. No monolítico, a busca pelo serviço a ser avaliado devolvia também a lista de avaliações que este recebeu, porém, com a separação do esquema de base de dados, os serviços deixaram de estar ligados às suas avaliações, pelo que a busca por um serviço não possui nenhuma informação sobre as suas avaliações.

Na função apresentada na Figura 74, temos a implementação da funcionalidade apresentada na Figura 73 .

É preciso referir que esta função é bastante semelhante à função que se encontra no monolítico, exceto as linhas 86 e 87, devido às chamadas extra.

De resto, a função é igual, porque o resto da lógica está encapsulada nos serviços do utilizador e do serviço. Optou-se por se ter os serviços de cada entidade representados no microsserviço, para se poderem ter encapsulados os métodos relativos às chamadas remotas.

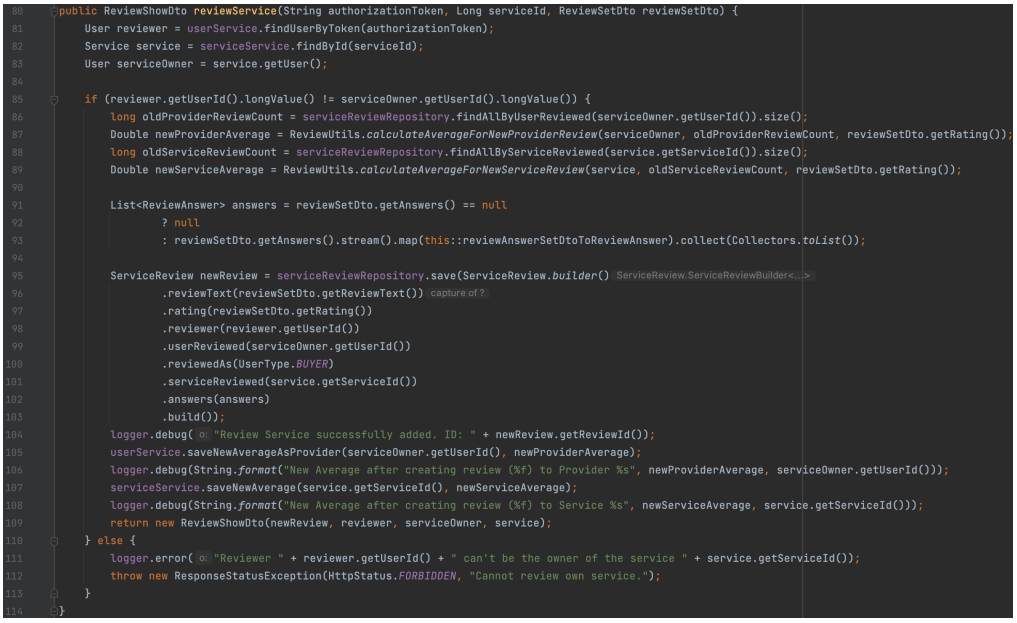


Figura 74: Função para realizar nova avaliação de serviço.

5.5.3 *Realização das chamadas remotas ao monolítico*

Para a função definida anteriormente, é necessário desenvolver as funções que vão realizar as chamadas remotas ao monolítico, para se obter o utilizador e serviço. De notar que na Figura 74, as linhas 105 e 107 representam funções que também efetuam chamadas remotas ao monolítico, para atualizar a média de avaliações.

Para a busca por um utilizador ao monolítico, é necessário criar um *endpoint* neste que permita que o exterior possa realizar esta operação. Para isso, aplica-se o padrão de monolítico com agregados expostos, mencionado na secção 3.8.1.

Visto que se recebe o *token* de autorização, este é o parâmetro que se passa ao *endpoint* para se saber qual o utilizador que efetua a avaliação. Na Figura 75, pode-se observar a definição deste *endpoint* no monolítico, que devolve um opcional, dependendo se o utilizador com aquele *token* existe ou não.

```
@GetMapping(path = "/user/auth/token")
public Optional<User> getUserByAuthorizationToken(@RequestParam String token){
    return userService.getUserByAuthorizationToken(token);
}
```

Figura 75: *Endpoint* no monolítico que devolve um utilizador.

Com este *endpoint* definido no monolítico, no novo microsserviço pode-se desenvolver a função que trata de realizar o pedido remoto para obter o utilizador. Na Figura 76, pode-se observar as funções *findUserByToken* e

getUserByToken, sendo que a primeira chama a segunda, tratando da possibilidade deste utilizador não existir. Na segunda função é possível observar que é realizado um pedido ao *endpoint* que foi definido no monolítico. A resolução do endereço do monolítico é feita através de Kubernetes , recebendo-o como variável de ambiente.

```
public User findUserByToken(String authorizationToken) {
    return getUserByToken(authorizationToken).orElseThrow(() -> {
        logger.error( o: "User not found.");
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "User not found.");
    });
}

private Optional<User> getUserByToken(String authorizationToken) {
    try {
        String url = monolithUrl + "/api/user/auth/token";
        Map<String, String> params = new HashMap<>();
        params.put("token", authorizationToken);
        User user = httpClient.sendJsonRequest(url, HttpMethod.GET, body: null, params, headers: null, User.class);
        return Optional.ofNullable(user);
    } catch (HttpException httpException){
        HttpStatus httpStatus = HttpStatus.resolve(httpException.getResponseCode());
        throw new ResponseStatusException(httpStatus != null ? httpStatus : HttpStatus.SERVICE_UNAVAILABLE, httpException.getMessage());
    } catch (Exception e) {
        e.printStackTrace();
        throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE, "Service Unavailable");
    }
}
```

Figura 76: Chamada remota para ir buscar um utilizador ao monolítico.

Com isto, temos o microsserviço a conseguir fazer ligação com o monolítico para poder ir buscar o utilizador pretendido. Para as outras funções mencionadas, o processo é semelhante.

O ideal é que com a continuação da migração para uma arquitetura de microsserviços, esta ligação seja realizada para o microsserviço que esteja encarregue do componente dos utilizadores.

5.5.4 *Modificação do monolítico*

Com a extração do microsserviço de avaliações, é necessário referir que foi necessário efetuar alterações no monolítico. Algumas destas modificações foram já apresentadas nas secções anteriores. Estas alterações foram efetuadas para que o microsserviço extraído pudesse aceder a informação que necessita que ainda se encontra no monolítico.

Contudo, como se pode constatar pelo diagrama de *packages* apresentado na Figura 5.2, existem componentes no monolítico que necessitam do componente de avaliações. Por isso, foi necessário efetuar alterações ao monolítico, para que as funções que dependiam deste, fossem mudadas para realizar a chamada ao microsserviço.

Para isso, o monolítico deve saber o endereço do microsserviço de avaliações. É preciso criar uma propriedade no ficheiro *application.properties* que receba o valor desta variável, como se observa no código 5.6.

```
review.service.host=${REVIEW_URL}
```

Código 5.6: Propriedade para endereço do microsserviço de avaliações

Para se receber esta variável, deve-se realizar modificações na instalação do monolítico, adicionando nas variáveis de ambiente o valor desta, como se mostra no código 5.7, sabendo que o microsserviço de avaliações é denominado *review*.

```
env:
    - name: REVIEW_URL
      value: http://review
```

Código 5.7: Passagem do endereço do microsserviço de avaliações na instalação do monolítico

Com estas alterações, pode-se começar a modificar as funções que necessitam do microsserviço. Para isso, utiliza-se o padrão de migração *Branch by Abstraction* apresentado na secção 3.3.2.

No monolítico, existe a funcionalidade de se ir buscar toda a informação de um utilizador. Nesta informação encontram-se as avaliações que este recebeu por ser consumidor de serviços, que tem agora de ser adquiridas do microsserviço de avaliações.

Na Figura 77, apresenta-se a funcionalidade que se pretende alterar, recorrendo ao padrão *Branch by Abstraction*.

Figura 77: Funcionalidade no monolítico a se alterar para realizar mudanças ao novo microsserviço.

Os passos efetuados para se migrar esta implementação seguindo este padrão são os seguintes.

1. **Criar uma abstração da funcionalidade a ser substituída e implementá-la**

   O primeiro passo é criar uma abstração que tenha as funcionalidades que se pretendem extrair, como se observa na Figura 78. Neste caso, pretende-se que a abstração possua a funcionalidade *getReceivedBuyerReviews*.

Figura 78: Criar abstração da funcionalidade a migrar.

   A seguir, implementou-se esta abstração, colocando-se na função o que já era feito, como se pode constatar na Figura 79.

Figura 79: Implementação da abstração da funcionalidade a migrar.

2. **Mudar as chamadas à funcionalidade para se usar a nova abstração**

   Com a abstração e a implementação criada, é necessário agora redirecionar as chamadas para esta, que passa por iniciar o objeto nos componentes onde esta função é chamada, atribuindo-lhe como resultado a nova implementação, como se apresenta no código 5.8.

```
this.reviewServiceAbstraction = new
    ReviewServiceLocalImplementation();
```

Código 5.8: Iniciar abstração no componente onde se usa a funcionalidade

De seguida, basta substituir na função *getReceivedBuyerReviews* a atual chamada pela nova implementação da abstração, como se observa na Figura 80.

```
public List<ReviewShowDto> getReceivedBuyerReviews(String authorizationToken) {
    String email = jwtTokenUtil.getEmailFromAuthorizationString(authorizationToken);
    User user = findByEmail(email);
    return reviewServiceAbstraction.getBuyerReviews(user);
}
```

Figura 80: Mudar chamadas à funcionalidade para a nova abstração.

3. **Criar uma nova implementação da abstração**

   Quando o microsserviço seja retirado do monolítico, têm-se que alterar a forma como estas avaliações são adquiridas. Para isso, cria-se uma nova implementação da abstração, apresentada na Figura 81. A implementação da função *getReceivedBuyerReviews* realiza uma chamada remota ao microsserviço de avaliações.

```
@RequiredArgsConstructor
public class ReviewServiceRemoteImplementation extends ReviewServiceAbstraction {

    private final BuyerReviewService buyerReviewService;

    @Override
    public List<ReviewShowDto> getBuyerReviews(User user) {
        return buyerReviewService.getBuyerReviews( reviewerId: null, user.getUserId());
    }
}
```

Figura 81: Nova implementação da abstração.

4. **Mudar a abstração para usar a nova implementação**

   Com o novo microsserviço implementado, pode-se alterar o monolítico para se utilizar a nova implementação da abstração, modificando a maneira como a abstração é instanciada. Em vez do que se apresentou no código 5.8, deve-se efetuar da forma como se apresenta no código 5.9.

   Caso se pretenda, pode-se adicionar um mecanismo que permita facilmente mudar de uma implementação para outra.

5. **Remover a implementação antiga**

   Após se verificar que tudo funciona como esperado, pode-se proceder à remoção da implementação antiga da abstração.

   Caso se pretenda, é possível também efetuar a remoção da abstração.

```
this.reviewServiceAbstraction = new
    ReviewServiceRemoteImplementation(buyerReviewService);
```

Código 5.9: Iniciar abstração com a nova implementação

5.5.5 *Instalação do microsserviço*

Para a instalação do microsserviço de avaliações, os passos são semelhantes ao que já se observou nas outras secções apresentadas anteriormente, tal como a secção C para a instalação da base de dados e a secção 5.3.2 para a do microsserviço.

A maior diferença prende-se com as variáveis de ambiente que o microsserviço precisa. No código apresentado em 5.10, pode-se observar a definição destas variáveis de ambiente.

```
env:
  - name: JASYPT_ENCRYPTOR_PASSWORD
    value: SaracenSecretKey
  - name: MYSQL_URL
    value: review-mysql
  - name: RABBITMQ_URL
      value: rabbitmq
  - name: MONOLITH_URL
    value: http://marketplace-back
```

Código 5.10: Criação da instalação do microsserviço de avaliações

5.5.6 *Redirecionamento no frontend*

O *frontend* realiza os pedidos relativos à funcionalidade das avaliações ao monolítico. Dado que se extraiu o microsserviço de avaliações do monolítico, deve-se agora redirecionar os pedidos que envolvam funcionalidades relativas com este para ele próprio. Este redirecionamento pode ser concretizado semelhantemente ao que se efetuou na secção 5.4.4.

Todos os *endpoints* do microsserviço de avaliações tem o prefixo */review*. Como o *frontend* realiza todos os pedidos para si próprio com o sufixo */api*, na secção *location* apanha-se todos os pedidos que tenham */api/review* e assim estes são redirecionados para o microsserviço de avaliações, sendo semelhante ao que se apresentou no código 5.4.

Com esta mudança, este redirecionamento fica da forma que se apresenta na Figura 82.

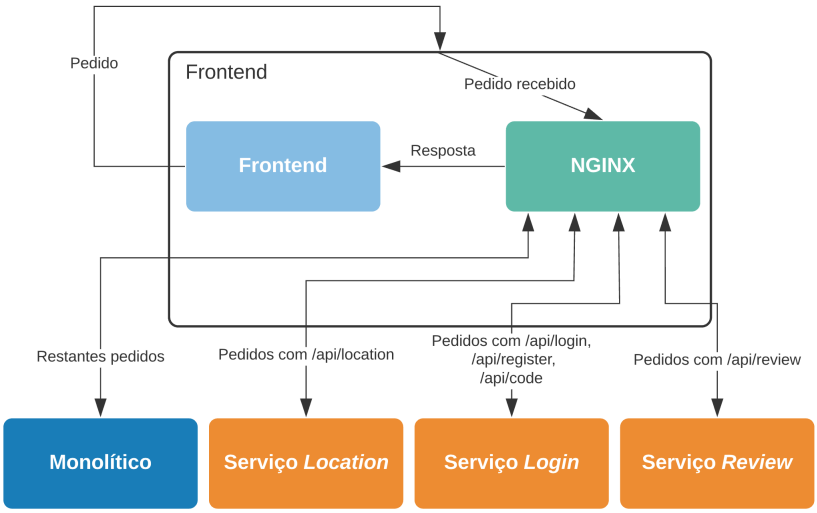


Figura 82: Redirecionamento dos pedidos para o novo microsserviço de avaliações.

5.6 DISCUSSÃO DA MIGRAÇÃO EFETUADA

Concluída a extração de alguns componentes para microsserviços é possível efetuar algumas avaliações desta migração e o impacto que teve na aplicação.

O processo começou por extrair um componente que não dependia de ninguém, mas que possuía quem dependesse dele, sendo o componente de email escolhido para tal. Como segundo serviço, extraiu-se do monolítico um componente que não possuía quem dependesse dele, mas que dependia de componentes do monolítico, como foi o caso do componente de *login*. Por último, extraiu-se um componente que possuía dependentes e que dependia de outros, sendo o componente de avaliações o escolhido.

Para as duas primeiras extrações utilizou-se o padrão de migração *Strangler Fig Application*, dado que eram componentes que se encontravam na fronteira do diagrama de *packages*. Na última extração, utilizou-se o padrão de migração *Branch by Abstraction*, pois o componente de avaliações encontrava-se mais no interior do diagrama de *packages*, o que fazia deste um componente mais complicado de extrair. Nestas extrações foram também aplicados padrões de migrações de base de dados, tal como o monolítico de agregados expostos. Em cada um, foi explicado como se procedeu à instalação de cada serviço e também as modificações que foram necessárias de realizar no *frontend*, apresentando diagramas de como as comunicações passaram a ser efetuadas entre microsserviços. Em particular, para o serviço de avaliações, foi demonstrado como se procedeu à separação do modelo de dados, envolvendo remoção de chaves estrangeiras. Foi também apresentado um diagrama de sequência da realização de uma avaliação de um serviço, contemplando a separação deste serviço de avaliações, contrastando-se com o diagrama de sequência modulado para o monolítico.

Apesar de não se ter efetuado a migração total da aplicação para uma arquitetura de microsserviços é possível já determinar o impacto que esta migração tem. Com a extração de microsserviços, é esperado que se aumentem as chamadas às bases de dados e entre estes. O que antes era feito localmente, pode agora envolver mais que um serviço ou base de dados. Deste modo, é normal não haver melhorias no desempenho da aplicação com

esta migração, pois apenas se removeu funcionalidades do monolítico, colocando-as num serviço independente. É expectável que se tenha um aumento dos tempos de resposta das funcionalidades extraídas, pois necessitam provavelmente de efetuar chamadas extra que não eram necessárias anteriormente.

Contudo, não tendo à primeira vista uma melhoria no desempenho da aplicação é possível concluir que foram alcançadas algumas vantagens da aquisição de uma arquitetura de microsserviços, que pode compensar o aumento do tempo de resposta de algumas funcionalidades.

Em primeiro lugar, aumentou-se a resiliência da aplicação. Tal deve-se ao facto de termos microsserviços independentes, que falhando, não levam à falha de toda a aplicação. Esta continua operacional, sendo que o microsserviço que falhou estará temporariamente indisponível. No nosso caso prático, se o microsserviço de *email* falhar, não irá levar à falha de toda a aplicação. As funcionalidades que envolvam o envio de correios eletrónicos, continuarão a funcionar, exceto que não será possível o envio do correio eletrónico. Quando o microsserviço de *email* voltar a estar operacional, este poderá processar as mensagens na fila de espera. É preciso que os microsserviços comuniquem de forma assíncrona, como é o caso da comunicação com o microsserviço de *email* em que se utiliza RabbitMQ .

Em segundo lugar, consegue-se escalar independentemente os microsserviços. Se se pretende que o componente de *login* tenha mais desempenho, é possível escalar este independentemente dos restantes. Enquanto na aplicação monolítica apenas se podia escalar a aplicação como um todo, neste caso pode-se escalar os microsserviços que se pretende. Além disto, aumenta-se a disponibilidade da aplicação, pois se uma instância de um microsserviço escalado falhar, têm-se outras instâncias para tomar o seu lugar. De referir que o escalonamento pode ser horizontal como vertical, em outras palavras, pode-se ter várias instâncias do mesmo, como aumentar os recursos, como memória, disco, capacidade de processamento, entre outros.

Outra vantagem com esta migração é de se poder instalar independentemente os microsserviços. Enquanto no monolítico, uma mudança implicava a instalação total da aplicação, com microsserviços, uma mudança implica apenas a instalação do serviço modificado. Desta forma temos instalações e entregas mais rápidas. Na eventualidade de falha de uma instalação, esta aconteceu apenas a um serviço isolado que pode ser facilmente revertida.

Com estas vantagens adquiridas já se consegue compensar o aumento do tempo de resposta devido a chamadas extra que se façam. É possível aumentar os recursos de um serviço e ter várias instâncias deste, de modo que se aumente o desempenho do serviço.

É preciso referir que nem sempre a migração para uma arquitetura de microsserviços é o mais acertado. Pode haver situações em que se estejam a realizar demasiados pedidos, que não compensa estar a despender mais instâncias e recursos para melhorar o desempenho do serviço e consequentemente da aplicação. É preciso avaliar cada situação e perceber qual a melhor opção.

6

CONCLUSÃO E TRABALHO FUTURO

6.1 CONCLUSÃO

Chegando ao fim desta dissertação, pode-se afirmar que foram cumpridos os objetivos previstos no plano inicial. Começou-se por analisar-se as características de uma arquitetura monolítica, apresentando os vários tipos de monolítico. A seguir apresentou-se e analisou-se a arquitetura de microsserviços, fazendo a comparação com a arquitetura monolítica, acabando por mostrar quais as vantagens e desvantagens de cada uma.

Explicou-se o que é a arquitetura de microsserviços, um conjunto de microsserviços autónomos que trabalham conjuntamente entre si para o funcionamento de um produto de *software*. Estes focam-se em ser independentemente instaláveis, significando que é possível realizar modificações, proceder à sua instalação e não ter impacto noutros microsserviços. Para conseguir este conceito, é necessário ter cuidado para não se partir interações com outros microsserviços. Estes possuem a característica de serem modulados à volta de um domínio, para que quando se tenham que efetuar alterações, estas sejam apenas efetuadas num e não em vários, o que implica que cada microsserviço tenha a posse dos seus próprios dados, não existindo bases de dados partilhadas, pois, se um microsserviço necessitar de dados de outro, não deve aceder diretamente à base de dados, mas sim efetuar uma chamada ao microsserviço que possui os dados que este pretende. Para isto, os microsserviços devem aplicar um conceito denominado ocultação de informação, que significa que devem apenas expor o mínimo para os seus consumidores, permitindo uma maior facilidade na gestão de cada um. Por último, os microsserviços são agnósticos em relação à tecnologia, podendo cada um ser implementado na tecnologia pretendida.

Todavia, para se ter um microsserviço, é necessário saber como o definir. Apresentaram-se alguns conceitos sobre *domain-driven design* (DDD), tal como agregados e *Bounded Context*. Um agregado é um conjunto de objetos de domínio que podem ser tratados como uma unidade só e que possuem um ciclo de vida, estado e identidade. O código que trata das transações de estado de um agregado deve estar agrupado. *Bounded Context* tipicamente representam uma maior fronteira organizacional. Do ponto de vista da implementação, *Bounded Context* contém um ou mais agregados. Apresentaram-se algumas formas dos microsserviços comunicarem entre si, tal como REST, RPC, *message brokers* e comunicação baseada em eventos, apresentando-se as vantagens e desvantagens de cada uma.

Contudo, microsserviços não são só vantagens, por isso apresentam-se algumas desvantagens e desafios da adoção desta arquitetura, tal como ter uma maior complexidade, pois é um sistema distribuído, levando a

maior dificuldade no desenvolvimento e configuração da aplicação. Um dos maiores desafios é a definição de um microsserviço, que se não for bem definido, pode levar a microsserviços pouco coesos e também à necessidade de modificar outros quando só se efetua mudanças num único microsserviço. Outro problema prende-se com a gestão de configurações, mais complexa, visto que existem vários microsserviços que necessitam de ser geridos e que comunicam entre si.

Abordou-se a escolha de cada uma destas arquiteturas. A decisão de se desenvolver segundo uma arquitetura monolítica, permite uma maior facilidade no início do desenvolvimento, dado que não é necessário pensar na divisão dos microsserviços e as comunicações entre componentes são todas locais. Esta arquitetura é uma boa escolha quando a aplicação a desenvolver é simples, não sendo necessário optar por uma arquitetura mais complexa, quando se pretende instalar rapidamente a aplicação, pois esta constitui um único artefacto, podendo ser compilada e colocada num servidor a correr. Outras das razões prende-se pela falta de conhecimento na arquitetura de microsserviços, pois ter-se ia que investir tempo na aprendizagem desta arquitetura, um processo que podia ser dispendioso e frustrante, pois é necessário gerir os vários microsserviços, as ligações entre estes e os eventuais erros que surjam.

Contudo, podem haver situações em que iniciar o desenvolvimento seguindo uma arquitetura monolítica não é a escolha mais acertada, pois o início do desenvolvimento de um produto de *software* é o momento exato para se começar a pensar nos diferentes microsserviços necessários e na separação destes. O desenvolvimento do monolítico até pode ser pensado para se limitar e definir bem as fronteiras destes componentes, porém, estes estão normalmente muito acoplados e cheios de ligações entre eles. A escolha de uma arquitetura de microsserviços é acertada quando se pretende uma maior rapidez no desenvolvimento da aplicação, pois constituindo um conjunto de microsserviços independentes, a equipa de desenvolvimento pode dividir-se em subgrupos e focar-se apenas no seu microsserviço respetivo, de forma autónoma e independente. Outra razão é a necessidade de se querer que uma parte da aplicação seja mais eficiente, pois pode haver um componente que seja mais consumido pelos utilizadores. Neste caso, a escolha de microsserviços é uma mais-valia, pois pode-se escalar independentemente cada um, aumentando o desempenho e eficiência.

Posto isto, abordou-se o tema da migração de uma arquitetura monolítica para uma de microsserviços. A maioria das aplicações criadas acabam por nunca ser um produto imutável, visto que existem sempre funcionalidades a serem modificadas e/ou adicionadas. Quando existe a necessidade de mudar uma linha de código no monolítico, é necessário a instalação de toda a aplicação. A adoção de uma arquitetura de microsserviços pode facilitar em muito a mudança e adição de novas funcionalidades, bem como uma entrega mais rápida para o cliente. A maior dificuldade numa migração de um monolítico para microsserviços, relaciona-se com a definição do que deve ser cada microsserviço e na mudança de comunicação entre estes. A migração de arquiteturas parte da identificação de partes do código que podem ser isoladas sem ter muito impacto no resto do código. Antes disto, é necessário compreender o porquê da necessidade de migrar de arquitetura. Esta decisão não deve ser tomada apenas porque a arquitetura de microsserviços está em voga, mas sim por razões fundamentadas.

Um dos pontos mais importantes numa migração para uma arquitetura de microsserviços é a de não a efetuar abruptamente, mas sim com mudanças pequenas e incrementais. É preciso perceber que vão acontecer erros

e que algo pode ser feito incorretamente aquando desta migração. A adoção de uma migração incremental permite que sejam reduzidos os erros e a dimensão destes. Uma boa forma de se iniciar a pensar na migração, começa pelo desenho da arquitetura de como se pretende a aplicação. Deste modo, consegue-se visualizar os microsserviços que se pretende que existam e as ligações entre estes. Consegue-se evitar problemas futuros, pois se o gráfico de dependências destas ligações contiverem ciclos é porque algo não está correto e deve-se repensar este desenho arquitetural.

A migração de um monolítico deve começar por efetuar mudanças no código. Este deve estar organizado à volta de domínios, como se abordou nas secções dos conceitos de *domain-driven design*. Abordaram-se alguns padrões de migração de código. Entre eles falou-se do *strangler fig application*, um padrão que segue a ideologia de uma migração incremental, pois permite que o novo código e o antigo coexistam em simultâneo, dando tempo ao novo para crescer e substituir por completo o antigo. A ideia deste padrão é identificar as partes do sistema que se desejam migrar, implementando-as a seguir no novo microsserviço. Com este implementado, é necessário redirecionar as chamadas que eram feitas ao monolítico, para serem agora feitas ao novo microsserviço. Abordou-se outro padrão, denominado *branch by abstraction* que permite extrair partes bastantes utilizadas e acopladas numa aplicação. A ideia passa por criar uma abstração para a funcionalidade a ser extraída, mudando a seguir as chamadas à funcionalidade para esta nova abstração. Assim sendo, pode-se criar uma nova implementação desta abstração, podendo posteriormente remover a implementação antiga.

Após se ter abordado a separação do código, introduziu-se o tema de separação da base de dados. Assim como o código, também o código relativo à base de dados deve estar devidamente separado, para se saber que partes são acedidas por quem. Microsserviços funcionam melhor quando cada um possui a sua informação, encapsulando-a e só permitindo certos acessos a esta. Muitas vezes implica que se tenha que remover chaves estrangeiras, pois as tabelas deixam de estar no mesmo esquema de base de dados, perdendo a ligação relacional que existia, o que pode levar a eventuais problemas de consistência de dados, dado que será necessário gerir estas colunas que não são chaves estrangeiras, mas que vão atuar como tal.

Com a base de dados particionada, temos que ter preocupações com operações que envolvam transações. Normalmente depende-se muito da base de dados para assegurar a consistência dos dados e para se poder executar várias operações em simultâneo, contando que são realizadas de forma atómica. Com a separação da base de dados, isto deixa de ser possível. Para colmatar este problema, apresentaram-se algumas opções, tal sagas, um algoritmo que consegue coordenar várias mudanças de estado sem necessidade de conter os recursos por um longo período de tempo.

Ainda no tema das bases de dados, falou-se de padrões de migração. Nos mencionados, destaque para o monolítico com agregados expostos, utilizado quando um microsserviço é extraído do monolítico, mas necessita de dados deste. Neste caso, o monolítico providencia através de uma API ou outro mecanismo, uma forma deste poder ter uma representação dos dados que necessita. Outros padrões incluem mudança da posse dos dados, que acontece quando um microsserviço é extraído do monolítico, mas continua a aceder à base de dados deste e também a sincronização dos dados, que acontece quando um microsserviço é extraído da base de dados e precisa dos dados que se encontram no monolítico, algo que deve ser concretizado de forma muito cuidadosa quando se encontra em produção.

Até aqui falou-se de como extrair microsserviços do monolítico, mas é importante saber por onde começar. Com o desenho de um diagrama de *packages*, consegue-se ter uma visão geral da aplicação, inclusive as dependências que existam entre componentes no monolítico, conseguindo-se obter uma visão sobre que módulos são mais fáceis e os mais difíceis de extrair. Pode-se começar por extrair aqueles módulos que contém menos dependências, adquirindo as vantagens de uma migração incremental, porém, pode haver situações em que se queira extrair um componente com mais dependências.

Ainda no tema de migração para microsserviços, é necessário perceber se se deve começar por separar primeiro a base de dados ou o código quando se quer extrair um microsserviço. Começar por extrair primeiro a base de dados, aumenta-se potencialmente o número de chamadas realizadas às bases de dados, pois onde antes se poderia realizar um pedido com um único acesso à base de dados, com esta separação, pode-se necessitar de dois ou mais acessos. É ótimo caso se tenha preocupações com a consistência dos dados ou com o desempenho da aplicação, podendo decidir se se deve continuar com a migração ou retroceder nesta. Começar por extrair primeiro o código, permite perceber quais são os dados que este novo microsserviço necessita. Existe por último a hipótese de se separar o código e o esquema da base de dados em simultâneo. Contudo, pode causar vários problemas e vai contra a abordagem de uma migração incremental.

Terminando o estado da arte, passou-se para o caso de estudo. Passou-se por apresentar o desenvolvimento da aplicação monolítica de uma plataforma de *e-commerce* que disponibiliza a fornecedores e consumidores de serviços uma única plataforma para poderem publicitar e usufruir destes. Apresentaram-se os utilizadores, os requisitos funcionais e os diagramas respetivos à modulação da plataforma. A seguir explicou-se como se procedeu ao desenvolvimento do *backend*, em função da tecnologia utilizada.

Com isto apresentado, começou-se por apresentar a migração para uma arquitetura de microsserviços. Primeiro, fez-se o desenho do diagrama de *packages*, para se saber quais são os componentes mais fáceis e difíceis de extrair. Com esta visão geral, procedeu-se à extração de componentes do monolítico para microsserviços. Por último, efetuou-se uma discussão e avaliação geral deste processo de migração.

Concluindo, pode-se afirmar que se atingiu os objetivos previstos. Apresentaram-se as arquiteturas monolíticas e de microsserviços, de forma a desenvolver a aplicação segundo estas. Estudou-se a migração de uma arquitetura monolítica para microsserviços, abordando como preparar a aplicação, o código e a base de dados. Com a aplicação monolítica desenvolvida, procedeu-se ao desenho do diagrama de *packages* para perceber as dependências entre componentes, de modo a identificar quais os mais difíceis e complicados de extrair. Após estudar o processo de migração, procedeu-se à migração de três componentes para microsserviços, utilizando os padrões de migração apresentados. Estas extrações deram para perceber bem a complexidade da arquitetura de microsserviços e a dificuldade na extração, principalmente quando um componente se encontra mais acoplado. Também foi possível perceber a complexidade na gestão de todos os microsserviços e na comunicação entre estes. Um último aspeto prende-se com a depuração que acaba por ser mais exaustiva, visto que se têm que considerar vários processos e analisá-los independentemente.

6.2 TRABALHO FUTURO

Apesar de todo o trabalho e desenvolvimento realizado nesta dissertação existem sempre aspetos que podem ser mudados e outros acrescentados. Seja porque existem novas funcionalidades que podem ser adicionadas, novas formas de realizar estas funcionalidades, novos requisitos, entre outros. O objetivo deste tema de dissertação era o estudo do estado da arte na área de *e-commerce*, tendo ênfase na arquitetura monolítica e de microsserviços e na migração de uma para outra. Para tal, ia-se aplicar alguns dos conceitos investigados, para demonstrar o processo de migração de uma plataforma de *e-commerce*, desenvolvida segundo uma arquitetura monolítica, para uma arquitetura de microsserviços.

Durante o desenvolvimento desta dissertação, foram naturalmente surgindo ideias e novas funcionalidades que se apresentam de seguida.

- Aprofundar o conhecimento na área de microsserviços.

  Para os trabalhos desta dissertação, foram lidos vários livros, tentando ao máximo captar a essência da arquitetura de microsserviços e do processo de migração para este. Porém, existem de certo muitos outros temas relacionados com a arquitetura de microsserviços que não foram abordados nesta dissertação.

  Para além disso, a área da tecnologia não estagna e está em constante desenvolvimento, pelo que a arquitetura de microsserviços vai continuar a evoluir e é necessário estar atualizado com estas evoluções.

- Continuar com a migração da aplicação para uma arquitetura de microsserviços.

  Como se pode ver, a plataforma de *e-commerce* não foi totalmente migrada para uma arquitetura de microsserviços. Apresentou-se apenas o processo de migração de alguns componentes, possuindo estes diferentes graus de dependências e aplicando alguns dos conceitos apresentados nesta dissertação.

- Aplicar outros padrões de migração.

  No estado da arte foram apresentados alguns padrões de migração para uma arquitetura de microsserviços. Alguns deles foram aplicados e outros não. Era interessante, aplicar os restantes padrões e procurar e estudar acerca de outros novos padrões de migração.

- Utilizar outros protocolos de comunicação entre microsserviços.

  Para estabelecer a comunicação entre microsserviços, foi utilizado REST e um *message broker*. Era interessante utilizar outros mecanismos de comunicação, como comunicação por eventos.

- Adicionar mais segurança aos microsserviços.

  Um aspeto importante a considerar é a adição de segurança aos microsserviços desenvolvidos. Os *endpoints* criados para uso único dos microsserviços, estão disponíveis para qualquer microsserviço, sem qualquer restrição. Era interessante limitar os acessos a certos *endpoints*.

- Simular processo de migração em produção.

Este processo de migração foi realizado localmente, o que levou a que não fosse muito preocupante a ocorrência de erros ou falta de dados. Era interessante simular o processo de migração num ambiente de produção. Para isso, seria necessário ter em conta processos de retrocesso, caso algo corresse mal na instalação de um novo microsserviço e também processos de migração de dados.

- Utilizar um API Gateway

  A arquitetura de microsserviços utilizada é uma arquitetura distribuída, não existindo uma entidade centralizada que controla todos os componentes. Cada microsserviço conhece os que precisa, não necessitando de um intermediário para comunicar com outros microsserviços.

  Contudo, era interessante experimentar a utilização de uma API Gateway, tornando a arquitetura de microsserviços numa arquitetura centralizada. Com este, tem-se uma forma fácil de mapear os pedidos, de providenciar maior segurança, sendo o único ponto de acesso à aplicação e de monitorizar e calcular métricas do sistema como um todo.

- Implementar um *log* centralizado

  Com a arquitetura de microsserviços, temos vários microsserviços que se encontram separados e em processos diferentes. A depuração torna-se assim mais complicada pois têm-se que considerar vários *logs* referentes aos vários microsserviços.

  Um aspeto importante era a criação de um *log* centralizado, de forma a existir apenas um lugar onde se tenha que efetuar a depuração, de modo que seja mais fácil e rápida a identificação e resolução de problemas.

BIBLIOGRAFIA

Bruce, M. and Pereira, P. (2018). *Microservices in Action*. Manning Publications.

Conway, M. E. (1968). How do committees invent? *Datamation*.

Doyle, K., Ferguson, K., and McKenzie, C. (2021). What is rest (representational state transfer)? https://searchapparchitecture.techtarget.com/definition/REST-REpresentational-State-Transfer. Accessed: 2020-11-25.

Erl, T. (2007). *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, USA.

Evans, E. (2003). *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., USA.

Feathers, M. (2004). *Working Effectively with Legacy Code*. Prentice Hall PTR, USA.

Fielding, R. T. and Taylor, R. N. (2000). *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California, Irvine. AAI9980887.

Fowler, M. (2004). Strangler fig application. https://martinfowler.com/bliki/StranglerFigApplication.html. Accessed: 2021-03-15.

Fowler, M. (2013). Ddd_aggregate. https://martinfowler.com/bliki/DDD_Aggregate.html. Accessed: 2021-03-10.

Fowler, M. (2014). Reporting database. https://martinfowler.com/bliki/ReportingDatabase.html. Accessed: 2021-03-21.

Fowler, M. (2015). Monolith first. https://martinfowler.com/bliki/MonolithFirst.html. Accessed: 2020-10-24.

Fowler, M. (2020). Dark launching. https://martinfowler.com/bliki/DarkLaunching.html. Accessed: 2021-03-19.

Fowler, M. and Lewis, J. (2014). Microservices. https://martinfowler.com/articles/microservices.html. Accessed: 2020-10-24.

Garcia-Molina, H. and Salem, K. (1987). Sagas. In Dayal, U. and Traiger, I. L., editors, *SIGMOD Conference*, pages 249–259. ACM Press. SIGMOD Record 16(3), December 1987.

Hammant, P. (2007). https://paulhammant.com/blog/branch_by_abstraction.html. Accessed: 2021-03-17.

IBM (2019a). Esb (enterprise service bus). https://www.ibm.com/cloud/learn/esb. Accessed: 2020-12-11.

IBM (2019b). Soa (service-oriented architecture). https://www.ibm.com/cloud/learn/soa. Accessed: 2020-12-01.

IBM (2020a). Event-driven architecture. https://www.ibm.com/cloud/learn/event-driven-architecture. Accessed: 2020-11-27.

IBM (2020b). Message brokers. https://www.ibm.com/cloud/learn/message-brokers. Accessed: 2020-11-26.

IBM (2020c). Soa vs. microservices: What's the difference? https://www.ibm.com/cloud/blog/soa-vs-microservices. Accessed: 2020-11-30.

Indrasiri, K. and Siriwardena, P. (2018). *Microservices for the Enterprise: Designing, Developing, and Deploying*. Apress.

Jansen, G. and Saladas, J. (2020). Event-driven architectures: What are they and why use them? https://developer.ibm.com/technologies/messaging/articles/advantages-of-an-event-driven-architecture/. Accessed: 2020-11-27.

Javed, A. (2019). Monolithic vs. soa vs. microservices. https://www.linkedin.com/pulse/monolithic-vs-soa-microservices-aqib-javed-/. Accessed: 2020-11-07.

Josuttis, N. (2007). *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media.

Kleppmann, M. (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media.

Matturro, B. (2020). What is remote procedure call (rpc)? https://searchapparchitecture.techtarget.com/definition/Remote-Procedure-Call-RPC. Accessed: 2020-11-25.

Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M. (2016). *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Inc., 1st edition.

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 1st edition.

Newman, S. (2019). *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Incorporated.

Newman, S. (2021). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2nd edition.

Richards, M. (2015). *Microservices Vs. Service-oriented Architecture*. O'Reilly Media.

Richardson, C. (2018). *Microservices Patterns: With examples in Java*. Manning Publications.

Sato, D. (2014). Canary release. https://martinfowler.com/bliki/CanaryRelease.html. Accessed: 2021-03-19.

Schmidt, M. ., Hutchison, B., Lambros, P., and Phippen, R. (2005). The enterprise service bus: Making service-oriented architecture real. *IBM Systems Journal*, 44(4):781–797.

Smith, S. (2013). Application pattern: Verify branch by abstraction. https://www.stevesmith.tech/blog/application-pattern-verify-branch-by-abstraction/. Accessed: 2021-03-17.

Stellman, A. and Greene, J. (2014). *Learning Agile: Understanding Scrum, XP, Lean, and Kanban*. O'Reilly.

Tilkov, S. (2015). Don't start with a monolith. https://martinfowler.com/articles/dont-start-monolith.html. Accessed: 2020-10-25.

ÍNDICE

A

REQUISITOS

A.1 REQUISITOS FUNCIONAIS

A.1.1 *Requisitos de Convidado*

1. O convidado pode procurar por serviços.
2. O convidado pode visualizar um serviço.
3. O convidado pode navegar pela plataforma.
4. O convidado pode visualizar as avaliações de um serviço.

A.1.2 *Requisitos de Consumidor*

5. O consumidor pode registar-se no sistema.
6. O consumidor pode autenticar-se no sistema.
7. O consumidor pode editar os seus dados.
8. O consumidor pode remover-se do sistema.
9. O consumidor pode procurar por serviços.
10. O consumidor pode filtrar os serviços.
11. O consumidor pode visualizar um serviço.
12. O consumidor pode adicionar um serviço ao carrinho.
13. O consumidor pode visualizar o seu carrinho.
14. O consumidor pode remover um serviço do carrinho.
15. O consumidor pode fazer uma encomenda do seu carrinho.

16. O consumidor pode adicionar um serviço à lista de desejos.
17. O consumidor pode visualizar a sua lista de desejos.
18. O consumidor pode remover um serviço da lista de desejos.
19. O consumidor pode subscrever-se a uma subscrição.
20. O consumidor pode cancelar uma subscrição.
21. O consumidor pode efetuar pagamentos relativos à sua subscrição.
22. O consumidor pode avaliar um serviço usufruído.
23. O consumidor pode avaliar a plataforma.
24. O consumidor pode responder a perguntas relativas à avaliação.

A.1.3 *Requisitos de Fornecedor*

O fornecedor pode realizar todos os requisitos do consumidor, para além dos seguintes.

25. O fornecedor pode adicionar um serviço.
26. O fornecedor pode editar um serviço.
27. O fornecedor pode remover um serviço.
28. O fornecedor pode avaliar um consumidor que usufruiu dos seus serviços.

A.1.4 *Requisitos de Administrador*

29. O administrador pode autenticar-se no sistema.
30. O administrador pode ver estatísticas da plataforma.
31. O administrador pode ver todos os utilizadores.
32. O administrador pode registar outros utilizadores.
33. O administrador pode alterar o estado de um utilizador.
34. O administrador pode ver os dados de um utilizador.
35. O administrador pode aprovar um utilizador.
36. O administrador pode desaprovar um utilizador.

37. O administrador pode ver todas as categorias.
38. O administrador pode adicionar uma categoria.
39. O administrador pode ver os dados de uma categoria.
40. O administrador pode editar uma categoria.
41. O administrador pode mudar o estado de uma categoria.
42. O administrador pode remover uma categoria.
43. O administrador pode ver todos os serviços.
44. O administrador pode ver os dados de um serviço.
45. O administrador pode alterar o estado de um serviço.
46. O administrador pode receber notificações de novos serviços.
47. O administrador pode remover um serviço.
48. O administrador pode ver todas as avaliações.
49. O administrador pode ver os dados de uma avaliação.
50. O administrador pode aprovar uma avaliação.
51. O administrador pode remover uma avaliação.
52. O administrador pode ver todas as subscrições.
53. O administrador pode ver os dados de uma subscrição.
54. O administrador pode adicionar uma subscrição.
55. O administrador pode editar uma subscrição.
56. O administrador pode remover uma subscrição.
57. O administrador pode ver todas as questões de avaliações.
58. O administrador pode adicionar uma questão de avaliação.
59. O administrador pode ativar uma questão de avaliação.
60. O administrador pode suspender uma questão de avaliação.

A.2 REQUISITOS NÃO FUNCIONAIS

A.2.1 *Requisitos de Aparência*

1. A plataforma deve apresentar a informação de forma clara.

A.2.2 *Requisitos de Segurança*

2. A plataforma deve garantir que ninguém acede às informações pessoais dos utilizadores, à exceção do próprio utilizador.
3. A plataforma deve prevenir a introdução de dados errados.

A.2.3 *Requisitos Legais*

4. A informação pessoal dos utilizadores deve ser tratada ao abrigo do Regulamento Geral da Proteção de Dados.
5. A plataforma deve respeitar as leis éticas do país onde é inserido.

B

DIAGRAMA DE USE CASES

Sub-diagramas de use cases:

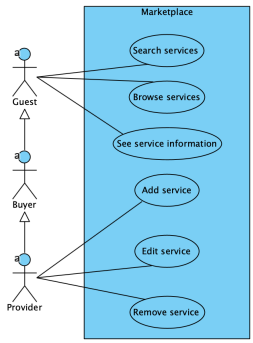


Figura 83: Sub-diagrama de use cases de serviços.

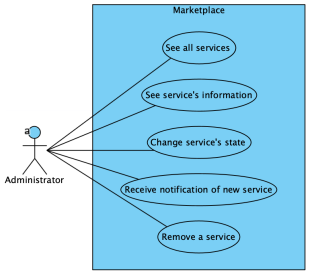


Figura 84: Sub-diagrama de use cases de gestão de serviços.

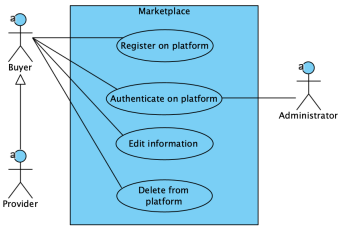


Figura 85: Sub-diagrama de use cases de gestão de contas.

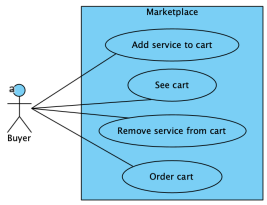


Figura 86: Sub-diagrama de use cases de gestão do carrinho.

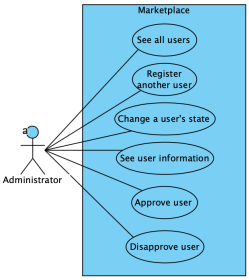


Figura 87: Sub-diagrama de use cases de gestão dos utilizadores.

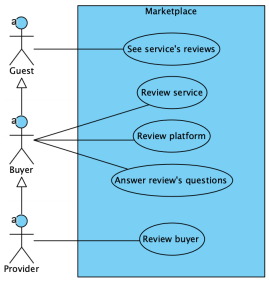


Figura 88: Sub-diagrama de use cases de avaliações.

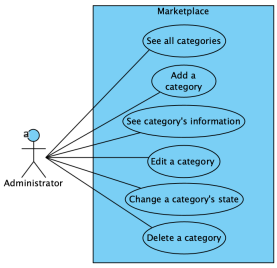


Figura 89: Sub-diagrama de use cases de gestão das categorias.

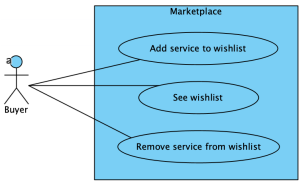


Figura 90: Sub-diagrama de use cases de gestão da lista de desejos.

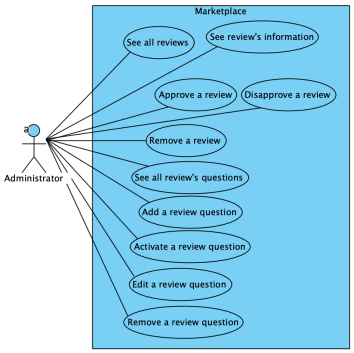


Figura 91: Sub-diagrama de use cases de gestão das avaliações.

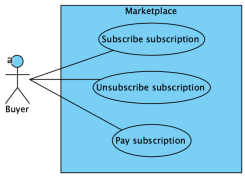


Figura 92: Sub-diagrama de use cases de subscrições.

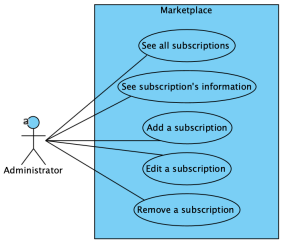


Figura 93: Sub-diagrama de use cases de gestão das subscrições.

C

INSTALAÇÃO DA BASE DE DADOS

Para a instalação da base de dados usa-se Kubernetes , necessitando de um ficheiro que contém o volume persistente, a instalação e o serviço.

O volume persistente, denominado de *Persistent Volume* (PV), é um pedaço do armazenamento do *cluster*. Este é um recurso, cujo ciclo de vida é independente do ciclo de vida do *pod* a que pertence. Este preserva os dados mesmo depois de um reinício ou eliminação do *pod*. Estes volumes persistentes são consumidos por uma reivindicação do volume persistente, denominado de *PersistentVolumeClaim* (PVC), que são basicamente pedidos para consumir esse armazenamento. Estes PVCs podem pedir uma determinada quantidade de espaço e modos de acesso a este armazenamento.

No código apresentado em C.1, pode-se observar a definição do volume persistente e a reivindicação deste volume, separados por "- - -". Neste código, destaca-se a criação do volume persistente com 10 gigabytes e a reivindicação destes 10 gigabytes.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
spec:
  storageClassName: manual
  accessModes:
```

```
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Código C.1: Criação do armazenamento da base de dados em Kubernetes

Tendo isto, pode-se proceder à instalação da base de dados. No código apresentado em C.2, pode-se ver a definição desta instalação, que cria um *container* com a base de dados, definindo o nome do *container*, o nome do utilizador, a password, o nome da base de dados, a porta em que vai correr e que volume usar para o armazenamento dos dados.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - image: mysql:latest
        name: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: root
        - name: MYSQL_USER
          value: theusername
        - name: MYSQL_PASSWORD
          value: thepw
        - name: MYSQL_DATABASE
          value: mydb
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
```

```
            persistentVolumeClaim:
              claimName: mysql-pv-claim
```

Código C.2: Criação da instalação da base de dados

Com a instalação realizada, basta apenas criar um serviço para que a base de dados tenha um *endpoint* estático para que possa ser acedida. No código apresentado em C.3, pode-se ver a definição deste serviço, que seleciona a aplicação denominada de *mysql* e a expõe na porta 3306.

```
    apiVersion: v1
    kind: Service
    metadata:
      name: mysql
    spec:
      ports:
      - port: 3306
      selector:
        app: mysql
      clusterIP: None
```

Código C.3: Criação do serviço da base de dados

Concluindo, basta correr o comando apresentado em C.4 para correr esta instalação, assumindo que o nome do ficheiro onde se encontram as definições apresentadas nos códigos anteriores é denominado de *monolith-mysql.yaml*.

```
            kubectl apply -f monolith-mysql.yaml
```

Código C.4: Instalação da base de dados

D

INSTALAÇÃO DO FRONTEND

Em relação ao *frontend*, a instalação deste passa pela criação de uma imagem *docker* para que depois se possa colocar esta num *container* e instalar a aplicação. A imagem é criada através de um *Dockerfile* onde se define o que deve ir para dentro do *container*. Tal pode ser observado no código apresentado em D.1.

```
FROM node:14 as build
ARG environment
WORKDIR /client
COPY package.json ./
RUN npm install
RUN npm install webpack@5.24.1 -g
COPY . ./
RUN npm run build:${environment}


FROM nginx:stable-alpine
ARG environment
COPY --from=build /client/build /usr/share/nginx/html
RUN rm /etc/nginx/conf.d/default.conf
COPY ./nginx/.htpasswd /etc/nginx/.htpasswd
COPY ./nginx/nginx.conf-${environment} /etc/nginx/conf.d/nginx.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Código D.1: Dockerfile do frontend

A seguir, pode-se proceder à instalação do *frontend* utilizando Kubernetes .

No código apresentado em D.2, pode-se observar a definição desta instalação, que cria um *container* com o *frontend*, definindo o nome do *container*, o número de réplicas e a porta em que estará exposta.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 1
```

```
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: mrluiigi/marketplace:front
        imagePullPolicy: Always
        ports:
          - name: http
            containerPort: 80
```

Código D.2: Criação da instalação do frontend

Com a instalação realizada, basta apenas criar o serviço para que o *frontend* tenha um *endpoint* estático para poder ser acedido. No código apresentado em D.3, pode-se observar a definição deste serviço, que seleciona a aplicação denominada *frontend* e a expõe na porta 80.

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  type: LoadBalancer
  selector:
    app: frontend
  ports:
  - port: 80
    targetPort: 80
```

Código D.3: Criação do serviço do frontend

Seguidamente, corre-se o comando apresentado em D.4 para correr esta instalação, assumindo que o nome do ficheiro onde se encontram as definições apresentadas nos códigos anteriores é denominado *frontend.yaml*.

```
kubectl apply -f frontend.yaml
```

Código D.4: Instalação do frontend

Com isto, pode-se aceder ao URL do *frontend* realizando-se o comando apresentado em D.5.

Tendo o resultado da execução do comando apresentado no código D.5, pode-se aceder à aplicação como se constata na Figura 94.

```
minikube service frontend --url
```

Código D.5: Acesso ao frontend

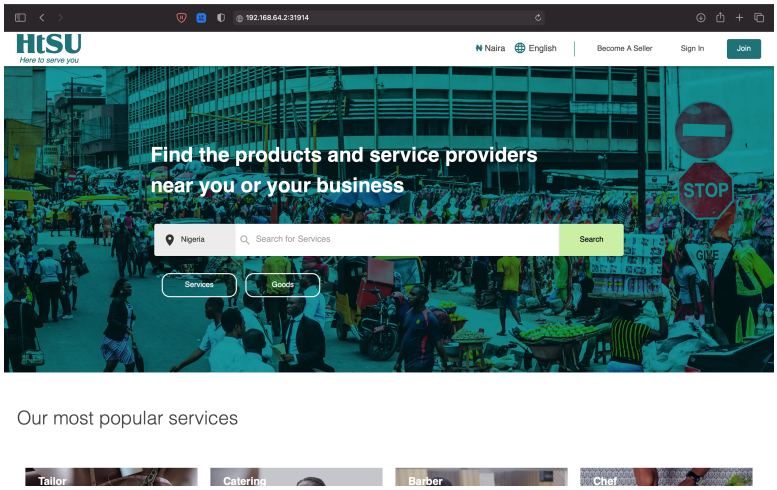


Figura 94: Acesso à *interface* da aplicação.

E

CONFIGURAÇÃO DO RABBITMQ

E.1 INSTALAÇÃO

A instalação do RabbitMQ vai ser realizada com recurso a Kubernetes .

No código apresentado em E.1, pode-se observar a definição desta instalação, que cria um *container* com o RabbitMQ , definindo o nome do *container*, o número de réplicas e as portas em que estará exposta. A porta 5672 é para os serviços poderem-lhe aceder e a 15672 é para a *interface* de acesso.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rabbitmq
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rabbitmq
  template:
    metadata:
      labels:
        app: rabbitmq
    spec:
      containers:
        - image: rabbitmq:3-management
          name: rabbitmq
          ports:
            - containerPort: 5672
            - containerPort: 15672
      restartPolicy: Always
```

Código E.1: Criação da instalação do RabbitMQ

Com a instalação definida, é preciso criar o serviço para que o RabbitMQ tenha um *endpoint* estático para poder ser acedido. No código apresentado em E.2, pode-se observar a definição deste serviço, que seleciona a aplicação denominada *rabbitmq* e a expõe nas portas apresentadas.

```
apiVersion: v1
kind: Service
metadata:
  name: rabbitmq
spec:
  ports:
    - name: "5672"
      port: 5672
      targetPort: 5672
    - name: "15672"
      port: 15672
      targetPort: 15672
  selector:
    app: rabbitmq
  type: LoadBalancer
```

Código E.2: Criação do serviço do RabbitMQ

E.2 CONFIGURAÇÃO NO SPRING BOOT

Com o RabbitMQ instalado, é preciso configurar a aplicação, para poder comunicar com este, de modo que seja possível publicar e receber mensagens da fila de espera.

Primeiro é preciso adicionar a dependência relativa à comunicação com o RabbitMQ . Em Gradle pode ser efetuado da forma que se apresenta no código E.3.

```
implementation 'org.springframework.boot:spring-boot-starter-amqp'
```

Código E.3: Adição da dependência do RabbitMQ

É preciso definir a conexão ao RabbitMQ . Para tal, pode-se efetuar o que se apresenta no código E.4. Neste, recebe-se o endereço do RabbitMQ como variável de ambiente e define-se a porta, o nome de utilizador e a palavra-passe. Também se pode definir uma propriedade onde se define o nome da fila de espera. Caso se pretenda, esta propriedade pode ser definida através das variáveis de ambiente.

```
spring.rabbitmq.host=${RABBITMQ_URL}
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
rabbitmq.queue=email
```

Código E.4: Conexão ao RabbitMQ

A seguir, deve-se definir um ficheiro de configuração onde se define a fila a ser criada, apresentado na Figura 95.

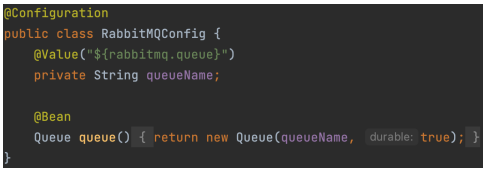


Figura 95: Definição da fila de espera no RabbitMQ

Concluindo, os componentes do Spring Boot tem acesso ao componente *RabbitTemplate*, que permite o envio e receção de mensagens. Entre as várias funções que fornece, existe a *convertAndSend* que recebe o nome da fila de espera e a mensagem, colocando esta mensagem na fila de espera.