UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RÔMULO MANCIOLA MELOCA

UM COMPARATIVO ENTRE FRAMEWORKS PARA MICROSSERVIÇOS

MONOGRAFIA

CAMPO MOURÃO
2017

RÔMULO MANCIOLA MELOCA

UM COMPARATIVO ENTRE FRAMEWORKS PARA MICROSSERVIÇOS

Trabalho de Conclusão de Curso de graduação apresentado à disciplina de Trabalho de Conclusão de Curso 2, do Curso de Bacharelado em Ciência da Computação do Departamento Acadêmico de Computação da Universidade Tecnológica Federal do Paraná, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. André Luis Schwerz

CAMPO MOURÃO
2017

ATA DE DEFESA DO TRABALHO DE CONCLUSÃO DE CURSO

Às **13:50** do dia **27 de novembro de 2017** foi realizada na sala **E104** da UTFPR-CM a sessão pública da defesa do Trabalho de Conclusão do Curso de Bacharelado em Ciência da Computação do(a) acadêmico(a) **Rômulo Manciola Meloca** com o título **Um Comparativo entre Frameworks para Microsserviços.**. Estavam presentes, além do(a) acadêmico(a), os membros da banca examinadora composta por: **Prof. Dr. André Luis Schwerz** (orientador), **Prof. Dr. Rafael Liberato Roberto** e **Prof. Dr. Rodrigo Campiolo**. Inicialmente, o(a) acadêmico(a) fez a apresentação do seu trabalho, sendo, em seguida, arguido(a) pela banca examinadora. Após as arguições, sem a presença do(a) acadêmico(a), a banca examinadora o(a) considerou _____ na disciplina de Trabalho de Conclusão de Curso **2** e atribuiu, em consenso, a nota _____ (_____). Este resultado foi comunicado ao(à) acadêmico(a) e aos presentes na sessão pública. A banca examinadora também comunicou ao acadêmico(a) que este resultado fica condicionado à entrega da versão final dentro dos padrões e da documentação exigida pela UTFPR ao professor Responsável do TCC no prazo de **onze dias**. Em seguida foi encerrada a sessão e, para constar, foi lavrada a presente Ata que segue assinada pelos membros da banca examinadora, após lida e considerada conforme.

Observações: _____

_____

_____

_____

_____

_____

Campo Mourão, **27 de novembro de 2017**

Prof. Dr. Rafael Liberato Roberto
Membro 1

Prof. Dr. Rodrigo Campiolo
Membro 2

Prof. Dr. André Luis Schwerz
Orientador

A ata de defesa assinada encontra-se na coordenação do curso.

Dados Internacionais de Catalogação na Publicação

M488 Meloca, Rômulo M
Um Comparativo entre Frameworks para Microsserviços/ Rômulo Manciola Meloca. Campo Mourão. UTFPR, 2017.
77. f. : il. ; 30 cm

Orientador: Prof. Dr. André Luis Schwerz.
Monografia (Graduação) - Universidade Tecnológica Federal do Paraná. Curso de Bacharelado em Ciência da Computação. Campo Mourão, 2017.
Bibliografia: f. 76 - 77.

1. Microsserviços. 2. Framework. 3. Desenvolvimento.

Biblioteca da UTFPR de Campo Mourão

Dedico este trabalho a todos os egressos e ingressos do curso que de maneira ou outra podem beneficiar-se desta obra, independente da posição que ocupem.

Agradecimentos

Agradeço a todos os que dedicaram tempo de suas vidas nesta empreitada, seja de maneira direta ou indireta, ainda que os escapasse e não o tenham percebido.

Agradeço aos professores do curso, que construíram um caminho favorável ao desenvolvimento deste trabalho lançando fundações sólidas pelas quais se pôde erigí-lo.

Agradeço ao orientador deste trabalho, que incansável e pacientemente suportou as agruras deste autor e suas mazelas, conduzindo os trabalhos por caminhos sensatos.

Agradeço aos amigos de graduação, companhia fiel, sem os quais esta jornada tornaria-se como fardo insustentável.

Agradeço a minha família, início e fim de meu trajeto por esta terra, base de todas as minhas realizações, a quem amo livre de condições.

Sobretudo, agradeço a Deus, pessoa sem a qual nada pode, existe ou detenha sentido, a quem amo livre de tempo.

O verdadeiro homem de ciência aposta sempre em todos os cavalos, e aplaude incondicionalmente o vencedor, qualquer que seja.
(CARVALHO, 2001)

Resumo

Meloca, Rômulo M. Um Comparativo entre Frameworks para Microsserviços. 2017. 77. f. Monografia (Curso de Bacharelado em Ciência da Computação), Universidade Tecnológica Federal do Paraná. Campo Mourão, 2017.

Contexto: Microsserviços estão em evidência por permitirem facilidades quanto a escalabilidade e existem diversas tecnologias que oferecem suporte a esta arquitetura devido a sua complexidade inerente.
Problema: Entretanto, o processo de escolha da tecnologia que melhor se enquadra nas necessidades de um desenvolvedor, empresa ou projeto é um desafio, especialmente quando não houver conhecimento sólido da arquitetura ou das características da tecnologia.
Objetivo: Assim, o objetivo deste trabalho é comparar o suporte ofertado por dois importantes *frameworks* para o desenvolvimento de aplicações sob essa arquitetura.
Método: Para isso, implementou-se um cenário fictício de chamadas de táxi nos *frameworks* KumuluzEE e Spring Cloud & NetFlix OSS, com o objetivo de comparar o apoio em relação à aspectos funcionais e não-funcionais da arquitetura.
Conclusões: Ao término deste trabalho, verificou-se o *framework* KumuluzEE contempla menos tópicos da arquitetura de microsserviços, entretanto, é de mais fácil aprendizado, de modo que seu uso é mais indicado aos novatos ou cujas aplicações sejam simples. Ao contrário, o Spring *framework* é de mais difícil compreensão, é mais complexo e mais completo em relação às características destacadas na arquitetura de microsserviços, sendo mais indicado para aplicações robustas.
Palavras-chaves: Microsserviços. *Frameworks*. KumuluzEE. Spring Cloud & NetFlix OSS

Abstract

Meloca, Rômulo M. A Comparison between Frameworks for Microservices. 2017. 77. f. Monograph (Undergradute Program in Computer Science), Federal University of Technology – Paraná. Campo Mourão, PR, Brazil, 2017.

Context: The microservices architecture is at its peak due to the facilities it provides for scalability and there are several technologies that support this architecture due to its inherent complexity.
Problem: However, the choice process of a an tecnology that better fits in the needs of developers, enterprise or project is a challenge, specially when developers dont have a solid knowledge of the architecture of microservices.
Objective: Therefore, the objective of this work is to compare the support of two important frameworks for microservices.
Method: For that, a fictitious taxi calls scenario was implemented in both frameworks KumuluzEE e Spring Cloud & NetFlix OSS with the objective of compare the frameworks' support for both functional and non-functional architecture aspects.
Conclusions: At the end of this work, we verified that the framework KumuluzEE implements less functionalities of the architecture of microservices, however its knowledge easyness make them more indicated for newcommers and simple applications. In other hand, the frameworks Spring is harder to understand, but it is more complete about the microservices architecture functionalities, being more indicated to robust applications.
Keywords: Microservices. Frameworks. KumuluzEE. Spring Cloud & NetFlix OSS

Lista de Tabelas

Lista de Figuras

Lista de Siglas

ACM	*Association for Computing Machinery*
AMQP	*Advanced Message Queuing Protocol*
API	*Application Programming Interface*
AWS	*Amazon Web Services*
BO	*Business Object*
ESB	*Enterprise Service Bus*
GPS	*Global Positioning System*
HTTP	*Hypertext Transfer Protocol*
IEEE	*Institute of Electrical and Electronics Engineers*
IP	*Internet Protocol*
JPA	*Java Persistence API*
JSON	*JavaScript Object Notation*
MVC	*Model-view-controller*
RPC	*Remote Procedure Call*
SMS	*Short Message Service*
URL	*Uniform Resource Locator*
XML	*eXtensible Markup Language*
YAML	*YAML Ain't Markup Language*

Lista de Acrônimos

ACID	*Atomicity, Consistency, Isolation, Durability*
CAOPLE	*Caste-centric Agent-Oriented Programming Language and Environment*
CAP	*Consistency, Availability, Partition tolerance*
CRUD	*Create, Read, Update, Delete*
DevOps	*Developer and Operations*
POJO	*Plain Old Java Object*
REST	*Representational State Transfer*
SOA	*Service Oriented Architecture*

Sumário

Capítulo 1

Introdução

Não é novidade que o termo microsserviços (*microservices*) injetou grande entusiasmo no mundo da computação. Determinada circunstância verifica-se nos diversos casos de sucesso como a NetFlix[1], Amazon[2] e SoundCloud[3]. Este advento, como qualquer novidade, carece de ponderadas análises para solidificar-se e frutificar em meio ao ecossistema (modelos arquiteturais e suas interações) já existente.

1.1. Contexto

O fenômeno dos microsserviços enquadra-se ao recente impulso que as *start-ups* trouxeram ao mundo, explorando os métodos ágeis de desenvolvimento unido à cultura DevOps (do inglês *Developer and Operations*) em resposta às demandas do mercado. Aplicações desenvolvidas sob o modelo de *start-ups* e/ou com a forma de métodos ágeis tendem a ser planejadas para pequenas quantidades de usuários, de modo que um dos grandes desafios destas empresas é fazer com que sua tecnologia não se degrade com o tempo e consiga acompanhar o crescimento da empresa.

Sumariamente, constata-se um movimento que contribui para que o desenvolvimento inicie-se em tamanho reduzido e depois sofra, incrementalmente, a necessidade de expandir-se e servir mais usuários. Há, portanto, um movimento que induz aos desenvolvedores de software cogitar a hipótese dos microsserviços.

Microsserviços é uma arquitetura de software derivada da SOA (do inglês *Service Oriented Architecture*) cuja finalidade é reduzir o acoplamento entre os diversos módulos de uma aplicação com o propósito de facilitar sua escalabilidade. Ao seu oposto dá-se

[1] https://www.netflix.com/
[2] https://www.amazon.com.br/
[3] https://soundcloud.com/

o nome de aplicação monolítica, cujos diversos módulos residem na mesma aplicação e comunicam-se diretamente por meio chamadas de métodos. Na arquitetura monolítica nenhuma funcionalidade do sistema existe e opera por si só. Como efeitos colaterais da adoção da arquitetura de microsserviços adquirem-se benefícios (por exemplo, escalabilidade e responsabilidade) e malefícios (por exemplo, dificuldade na refatoração e dificuldade na comunicação distribuída) a depender do aspecto sobre o qual se olha (de maneira que há um limiar a ser estudado entre as curvas da arquitetura monolítica e da arquitetura de microsserviços considerando o número de usuários pelo tempo de resposta).

Deste modo, uma aplicação desenvolvida sob esta arquitetura identifica-se por possuir vários sistemas de software, cada qual com uma única e bem específica responsabilidade. A decisão para a cisão de um módulo deve sempre ser a análise da relação de perdas e ganhos que há em cindir a aplicação ou mantê-la aglutinada. Em linhas gerais, vale sempre a necessidade de cada aplicação. Isto é: uma vez que uma parte da aplicação precise ser escalada ou sofra constante modificação, pensa-se em decompô-la. Do contrário, mantenha-se a aplicação aglutinada.

Embora a literatura aponte que ao assumir uma arquitetura de microsserviços, pode-se adicionar mais 70% de sobrecarga (UEDA et al., 2016) ao sistema devido ao *overhead* que há ao lidar com requisições HTTP, existem classes de aplicações que cumprem melhor seu papel sob a égide dos microsserviços (por exemplo, aquelas em que a consistência intermitente dos dados não é crucial). De modo que uma importante decisão de uma empresa é a adoção ou não da arquitetura de microsserviços. Decidir migrar ou não sua aplicação para a nuvem também deve ser uma escolha delicada que a empresa terá que fazer.

1.2. Problema

Em um sistema monolítico existem muitos arquivos de código, uma vez que sua responsabilidade é grande e agrega muitas funcionalidades. Ao contrário, na arquitetura de microsserviços existem muitos serviços e cada um com poucos arquivos de código, uma vez que sua responsabilidade é pequena e responde por apenas uma funcionalidade. Ao adotar a arquitetura de microsserviços, os desenvolvedores vêem-se face a maior facilidade ao realizar testes unitários, no sentido de que poucos serão os arquivos a serem analisados e pouco complexa será a aplicação, uma vez que parte do todo. Por outro lado, há dificuldades na realização dos testes de integração, uma vez que todo o cenário (muitas vezes complexo), precisa estar em atividade para que os testes sejam realizados. Na arquitetura de microsserviços há facilidade na manutenção, uma vez que cada serviço possui poucos arquivos de código, mas há dificuldade na implantação, uma vez que muitos sistemas precisam estar em operação para que todo o sistema funcione corretamente. Em suma, a arquitetura é complexa.

Naturalmente, tecnologias podem ser empregadas para a amenização dos *trade-off's* e

da complexidade geral da arquitetura. E as tecnologias que serão escolhidas para desenvolver-se sob esta arquitetura deverão ser muito bem ponderadas, dado que cada uma possui seus próprios *trade-offs* no contexto de uma empresa e seu produto, de certo que, também, cada tecnologia tem um propósito a atender.

Quer trate-se de uma aplicação cujo desenvolvimento parta da arquitetura de microsserviços, quer trate-se da migração de uma aplicação para a arquitetura de microsserviços, as tecnologias que serão adotadas para sua composição deverão ser analisadas em minúcias para que o desenvolvimento de um produto não venha a ser surpreendido pela falta de suporte de algum aspecto da arquitetura por parte da tecnologia escolhida.

Felizmente, existem diversos tipos de tecnologias que podem ser empregados para uma implementação que possua a arquitetura de microsserviços. Contudo, analisar seus pontos positivos ou negativos não é uma tarefa simples, especialmente para a agilidade que o mercado demanda.

1.3. Objetivo

O objetivo deste trabalho é comparar dois *frameworks* para microsserviços. Dentre todas as ferramentas disponíveis para microsserviços, observou-se sua popularidade e adoção, além de razões históricas e práticas. Por esses critérios escolheu-se os *frameworks* Kumuluz e Spring Cloud & NetFlix OSS escritos para a linguagem de programação Java.

Implementou-se um cenário fictício que muito assemelha-se às motivações que levam à opção pela arquitetura de microsserviços. O cenário, de chamadas de táxi, foi implementado duas vezes, uma para cada *framework* ao passo em que foram sendo analisados sob uma série de critérios. Os critérios, importantes para uma empresa que deseje escolher entre uma ou outra tecnologia para o desenvolvimento sob a arquitetura de microsserviços, dividem-se em aspectos funcionais e não-funcionais da arquitetura.

1.3.1. Justificativa

A razão pela qual este trabalho interessa aos arquitetos e desenvolvedores de software firma-se em sua qualidade de descrever os detalhes da arquitetura de microsserviços, discutir suas implicações práticas e motivações históricas. Por certo, dado sua complexidade, o desenvolvimento nesta arquitetura deve ser facilitado pelo suporte que determinadas tecnologias ofertam aos desenvolvedores, de modo que compará-las sob este prisma faz-se necessário e interessa aos desenvolvedores.

1.4. Contribuição

Com a implementação e análise dos *frameworks* observou-se que o *framework* KumuluzEE não é indicado para aplicações robustas enquanto demora estabelecer-se ao passo em que, face seus exemplos claros e facilidade de aprendizagem, é recomendado para aplicações simples e, em especial, novatos em microsserviços.

Com relação ao Spring Cloud & NetFlix OSS pôde-se verificar um plano de fundo vasto e sólido no qual os programadores podem orientar-se e estar mais amparados. Contudo, verificou-se que o *framework* não é muito receptivo aos novatos no que concerne a facilidade de uso e aprendizagem da ferramenta, dado o volume de abstrações existentes e para os quais o usuário iniciante não espera comportar-se. Uma vez compreendido a forma de trabalho no Spring *framework*, sua velocidade de produção é grande, mas o grau de liberdade que o usuário possui é menor.

1.5. Organização

Esta monografia está organizada em três outros capítulos. No Capítulo 2 explana a arquitetura de microsserviços, demonstrando suas motivações históricas, seus detalhes práticos, características e *trade-offs*; O Capítulo 3 apresenta as tecnologias para microsserviços, apontando suas classificações e apresentando cada um dos *frameworks* que serão comparados; O Capítulo 4 detalha o método desta pesquisa, elicitando o cenário, os critérios de avaliação e os detalhes sobre como será implementado o cenário proposto; Os resultados serão detalhados no Capítulo 5; e o Capítulo 6 sumariza os pontos centrais desta investigação.

Capítulo 2

A Arquitetura de Microsserviços

Para bem definir a arquitetura de microsserviços, cabe descrever as raízes de seu desenvolvimento, exemplificá-la e explanar seus pormenores, além de ofertar ponderações a seu respeito, conforme organiza-se as seções que seguem.

2.1. Perspectiva Histórica

A arquitetura de microsserviços é derivada do estilo arquitetural orientado a serviços (SOA) (SCHULTE; NATIS, 1996) e aquela muito compartilha desta, de modo que importa de início ater-se aos seus detalhes. Com a mudança na adoção de tecnologias, SOA foi proposta para responder aos problemas de heterogeneidade de linguagens, permitindo interface com sistemas legados, por meio de um protocolo de comunicação bem definido. Determinada estratégia permitia ainda que maior poder de processamento pudesse ser atribuído ao sistema na medida em que etapas de seu *workflow* fossem divididas em serviços.

A motivação central para a derivação da SOA para microsserviços concentra-se na escalabilidade da aplicação. Quando a aplicação precisa atender mais usuários, é possível escalá-la de duas maneiras: verticalmente, adicionando mais recursos ao hardware hospedeiro; e horizontalmente, com hospedeiros adicionais. Note-se que há um limite tecnológico e monetário para a escalabilidade vertical, de modo que em certo momento faz-se necessário a adoção de estratégias para escalar a aplicação horizontalmente.

No tocante à escala horizontal, a teoria tridimensional da escalabilidade proposta em (ABBOTT; FISHER, 2009) ilustra com um cubo posto nos três eixos sobre os quais uma aplicação pode ser escalada. A Figura 2.1 extraída de (RICHARDSON, 2014) materializa a teoria em questão. Nela pode-se compreender que uma aplicação, no intuito de atender mais usuários, pode ser escalada em três sentidos. No **eixo duplicação** a aplicação é clonada produzindo novas instâncias que são executadas simultaneamente sob um balanceador de

carga (*load balancer*) que direciona as requisições para uma ou outra instância, a depender de sua capacidade e disponibilidade. No **eixo decomposição** é possível cindir a aplicação em diversos em serviços. Por fim, no **eixo particionamento**, os dados da aplicação são segregados entre vários bancos de dados (*sharding*). Naturalmente, uma aplicação pode ser escalada em um ou mais eixos.

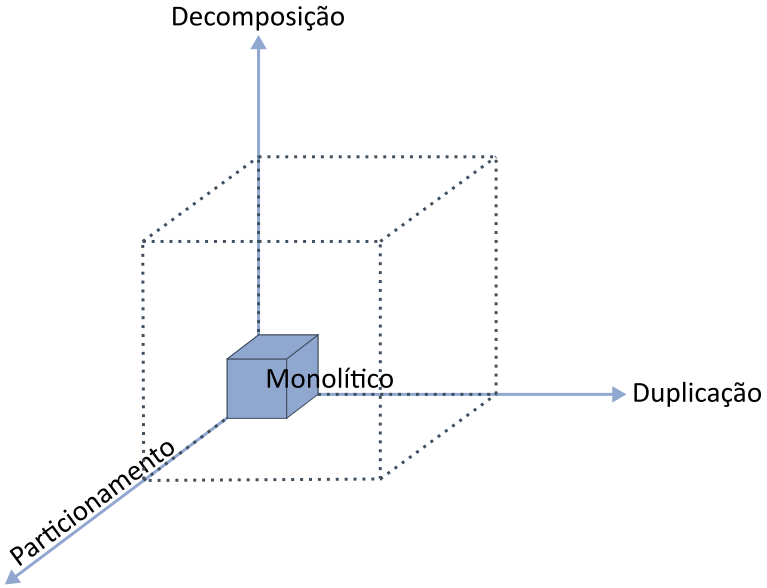

Figura 2.1. Cubo da escalabilidade (RICHARDSON; SMITH, 2016).

Um primeiro passo de um arquiteto pode ser escalar a aplicação no eixo duplicação, uma vez que sua execução não é muito custosa em uma aplicação monolítica[1]. Sua adoção, no entanto, faz com que todas as funcionalidades do sistema sejam duplicadas ainda que não estejam sobrecarregadas. Seu tempo de testes e implantação tende a crescer em função do inchaço da aplicação, o que não configura-se como desejável, especialmente em um contexto de implantações frequentes.

Quando insuficiente a escala, outra opção é, além da duplicação, dividir a aplicação monolítica em componentes menores, conhecidos como serviços – eixo decomposição onde a SOA atua. Neste eixo a escala suprirá a demanda (volume de requisições) até o momento em que o gargalo da aplicação passar a residir em seu banco de dados.

Explorar o eixo particionamento significa espalhar os dados da aplicação por todos os serviços, o que implica na necessidade de um gerenciamento robusto da consistência.

O Teorema CAP (do inglês *Consistency, Availability, Partition tolerance*) (também conhecido como Teorema de Brewer (FOX; BREWER, 1999), ilustrado na Figura 2.2, reúne as possibilidades existentes para o gerenciamento da consistência dos dados. Um triângulo em cujos vértices se lê Consistência, Disponibilidade e Tolerância a Partições, determina que apenas um de seus lados pode ser escolhido para um sistema, ou seja, Consistência e Disponibilidade, Consistência e Tolerância a Partições ou Disponibilidade e Tolerância a

[1] Nome dado a um sistema composto de uma única aplicação de software

Partições. Ao optar-se pelo particionamento dos dados, somente é possível garantir uma das suas outras duas propriedades: disponibilidade ou consistência. Não há como particionar os dados, mantê-los sempre consistentes e ainda disponíveis em razão dos mecanismos de travas que devem haver para impedir acessos a estados inconsistentes.

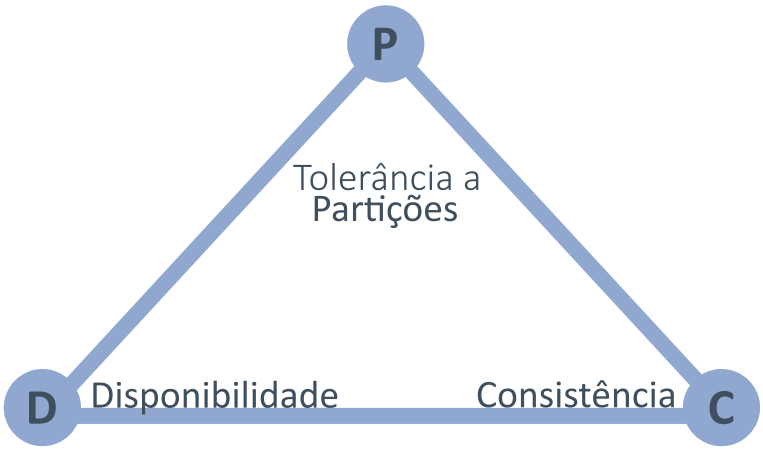

Figura 2.2. Teorema de Brewer.

Ao optar-se pelo particionamento dos dados e que sua consistência seja intermitente, em um contexto de sistemas distribuídos no qual se insere a SOA, transações distribuídas devem ser adotadas, o que retarda em muito a vazão do sistema. Entretanto, há cenários em que esta configura-se como opção única de um sistema.

Ao optar-se pelo particionamento dos dados e que estejam sempre disponíveis, sua consequência é um sistema cuja consistência não é garantida ininterruptamente, dado a ausência das travas que revogam a disponibilidade dos dados em estados inconsistentes. Isto quer dizer que algumas instâncias da base de dados serão inconsistentes. Mecanismos a nível de aplicação ou nível de dados devem ser adotados para corrigir o estado de inconsistência, tornando-o consistente. Essa é a ideia que cerca o conceito de consistência eventual. Em outras palavras, o sistema eventualmente estará consistente, mas janelas de inconsistências são permitidas e aceitas.

A modelagem dos dados na SOA não provê preparo para tolerar partições, uma vez que os serviços são muito dependentes uns dos outros, e seu protocolo de comunicação não é sucinto. Deste modo, a SOA evoluiu para suportar problemas que demandam alta escalabilidade.

Como ilustra a Figura 2.3, houve-se a necessidade de outorgar aos serviços independência, autossuficiência, tamanho reduzido e comunicação sucinta. Nesta arquitetura as teorias de escalabilidade não perdem seu significado. Isto significa que adotar microsserviços mas não segregar os dados nada mais faz que assemelhá-lo ao SOA no sentido de que não haverá escala nos três eixos, nada poderá fazer com o gargalo no banco de dados e os manterá dependentes em termos de desenvolvimento, teste e implantação. É bem verdade que seria possível o usufruto dos benefícios secundários tais como separação da responsabilidade de

cada serviço sob uma interface bem definida (encapsulamento), mecanismo de comunicação leve e facilidade na manutenção, contudo a arquitetura não cumpriria seu propósito.

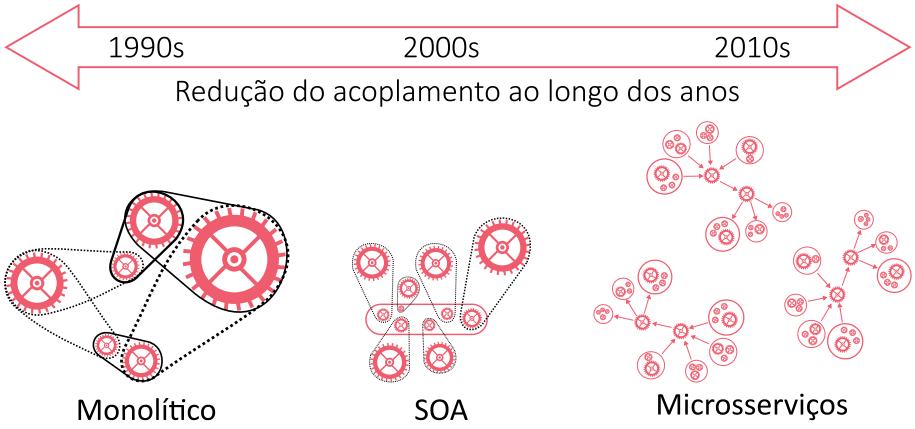

Figura 2.3. Evolução do acoplamento intra-arquitetural (MORRISON, 2015).

O termo microsserviços (*microservices*) originou-se em um *workshop* de arquitetos de software ocorrido em 2011 próximo à Veneza (FOWLER; LEWIS, 2014). Naquela oportunidade, os participantes do *workshop* trocavam experiências acerca dos estilos arquiteturais dos sistemas de software com que tinham contato. Ao decorrer dos anos, o termo tem se consolidado como uma forma particular de desenvolver aplicações de software baseada em um conjunto de serviços que são independentes e implantáveis automaticamente. Esse conjunto é organizado de acordo com a aptidão de cada serviço que são autossuficientes em termos de controle de linguagens e de dados. Neste trabalho os termos microsserviços e serviços são tratados como sinônimos.

A arquitetura orientada a microsserviços difere-se do SOA pela utilização de um mecanismo leve e individual para a comunicação dos serviços em detrimento do barramento de serviços (ESB (do inglês *Enterprise Service Bus*)) comumente encontrado na arquitetura orientada a serviços. O barramento mencionado infla-se com o suporte a vários protocolos no intento de prover abstração. Um único barramento lida com a troca de mensagens entre todos os serviços. Além disso, a arquitetura de microsserviços propõe que os dados de cada microsserviço sejam gerenciados por si próprio, de modo que seja garantida a independência de cada serviço em nome da escalabilidade do sistema e reúso dos serviços. Outrossim, a responsabilidade de cada serviço deve ser singular, conferindo-lhe tamanho reduzido, alta coesão e baixo acoplamento. Por coesão entende-se a consonância que as classes de um serviço possuem entre si, revelando a unicidade ou multiplicidade de responsabilidades do serviço. Por acoplamento entende-se a dependência que um serviço possui de outro. Graus maiores de acoplamento entre os serviços dificultam o desenvolvimento e a manutenção do sistema, podendo indicar graus menores de coesão de cada serviço, uma vez que não demonstram-se autocontidos por apresentarem elevado grau de acoplamento.

É importante ressalvar que a arquitetura de microsserviços não pretende postular

um formato rígido mas que, ao contrário, pretende fornecer aos arquitetos de software um modo de projetá-los que, naturalmente, possui especificidades e benefícios que devem ser ponderados conforme as necessidades e possibilidades do produto e de seus desenvolvedores. Assim sendo, não há, em microsserviços, uma regra rígida para sua construção, mas uma recomendação direcionada aos arquitetos de software para que evitem um formato monolítico, em atenção à teoria tridimensional da escalabilidade, conforme explicado anteriormente.

Com efeito, à medida em que há o distanciamento da origem dos eixos de escalabilidade, adicionam-se diversas necessidades ao sistema, que serão expostas no decorrer deste capítulo.

2.2. Perspectiva Arquitetural

Enquanto que alguns termos da arquitetura SOA foram mantidos na arquitetura de microsserviços, outros foram deixados. De igual maneira, outros termos foram incluídos. A este trabalho caberá explanar apenas os termos aplicáveis à arquitetura de microsserviços.

A arquitetura de microsserviços é uma arquitetura de sistema distribuído pertencente ao modelo arquitetônico cliente-servidor – uma vez que derivado do SOA – e compõe-se de tantas camadas quanto houverem serviços. Além dos serviços esperados pela demanda da regra de negócio do sistema a ser implementada, muitos outros serviços de suporte devem fazer intermédio entre eles a fim de garantir-se alta disponibilidade do sistema e a escalabilidade que primordialmente motiva a escolha da arquitetura de microsserviços.

Cada microsserviço possui seus próprios dados, sua própria regra de negócio e uma interface bem definida, de modo que cada serviço possa ter seu próprio repositório, controlador de versões, ciclo de vida, controlador de tarefas, sua própria equipe de desenvolvimento e implantação.

As trocas de mensagens entre os microsserviços deverão ser leves e sucintas, a confiar na clara definição da interface que cada microsserviço apresenta. Espera-se que um microsserviço não dependa de outros e que o número de mensagens trocadas seja reduzido e que não digam respeito a regra de negócio. Por exemplo, espera-se que um microsserviço de notificações (e-mail ou SMS (do inglês *Short Message Service*)) possa atuar independentemente, mesmo que outros serviços não estejam funcionando. Para isso, é necessário que esse serviço possua todos os dados necessários para o envio dessas notificações aos usuários do sistema. De maneira análoga, espera-se que um microsserviço qualquer possa normalmente requisitar essa função do microsserviço de notificações. Não espera-se, por exemplo, que o microsserviço de notificações precise requisitar ao microsserviço de gestão de usuários o nome vinculado a um identificador qualquer, para que ele possa completar sua tarefa.

A consequência de ter microsserviços independentes é a redundância dos dados entre eles. Note-se que esta arquitetura de redundâncias, enquanto paraleliza as funções do

sistema, incorre em possíveis incongruências que devem ser gerenciadas pelos arquitetos do software conforme postula o teorema de Brewer (abordado na Seção 2.1) e as propriedades transacionais[2] segundo a criticidade da operação para a regra de negócio implementada no sistema de software. Isto é, caberá ao arquiteto de software responsável pela gestão do sistema analisar se o conjunto de dados e a regra de negócio do sistema permite sua cisão e consistência eventual em nome da escalabilidade do sistema. Mais sobre as consequências da arquitetura pode ser lido na Seção 2.3.

Para um sistema que permita-se arquiteturar-se em microsserviços é razoável que haja consistência eventual no nome de um usuário, que estará espalhado entre os microsserviços. No entanto, é inadmissível que a consistência seja eventual, por exemplo, no pagamento do serviço prestado. Tais ponderações levam ao desenho de uma arquitetura cindida em dados cuja consistência pode ser eventual em alguns microsserviços e intermitente naqueles onde é imprescindível mantê-los consistentes a todo momento. No seio da modelagem do sistema, quando feito sob a arquitetura de microsserviços, determinada qualidade de decisões são avaliadas e tomadas em razão de um bem maior que se deseja prover – a escalabilidade.

Alguns dos serviços de suporte podem ser implementados já junto ao primeiro processamento, que ocorre na primeira camada do sistema, isto é, os computadores dos clientes. Embora espera-se que a primeira camada trate apenas da entrada de dados e apresentação de resultados, não é de todo ruim conceber a implementação dos serviços de suporte nesta camada, salvo suas debilidades – que serão expostas ao longo desta seção.

A primeira camada pode tratar-se de dispositivos móveis, computadores pessoais, televisões inteligentes ou quaisquer outros aparelhos que façam interface imediata com o usuário do sistema.

As linhas que seguem incumbem-se de pontuar cada um dos serviços de suporte, que rodeiam os serviços. Será apresentando o balanceador de cargas, seguido das implicações impressas nos dados da aplicação e a comunicação em suas formas que haverão de mantê-los (os dados) coerentes. Há ainda, em seguida, uma descrição sobre os componentes Disjuntor, API (do inglês *Application Programming Interface*) *Gateway*, Descobridor de Serviços, Monitoramento e métricas, Serviço de Autenticidade, Serviço de Configurações e detalhes relativos à implantação dos sistemas de software que compõem os microsserviços da aplicação.

2.2.1. Balanceador de Cargas

O fator determinante para a adoção da arquitetura de microsserviços é a escalabilidade. Determinado fator pressupõe que o sistema atenda muitos usuários e requisições. Este fator – a escalabilidade – implica que o sistema adote algum mecanismo para distribuir as requisições

[2] Atomicidade, Consistência, Isolamento e Durabilidade (ACID (do inglês *Atomicity, Consistency, Isolation, Durability*)), demonstrado por (HAERDER; REUTER, 1983)

dos usuários ao sistema dentre as instâncias capazes de processá-las. O número de instâncias deve satisfazer o teto do quociente entre o número de requisições esperado e a quantidade de requisições que cada instância suporta. Além disso, a aplicação deve ser elástica para suportar números maiores e menores de requisições sem desperdício de recursos.

A arquitetura de microsserviços conta com diversos serviços independentes. Isto quer dizer que em duas direções da tridimensão da escalabilidade (conforme discorrido na Seção 2.1) a própria arquitetura já fornece meios práticos para escalá-la.

Contudo, um cenário em que a arquitetura de microsserviços por si só (dotada de mais serviços, e portanto maior grau no eixo de escala, que a SOA) não é suficiente para atender à demanda, há de se clonar cada serviço sobrecarregado no quociente ideal e balancear a carga do serviço.

Sua implementação pode ser feita tanto junto ao servidor, como junto ao cliente. O balanceador de cargas pode, ainda, ser implementado de maneira rudimentar na camada de transporte por meio de um *firewall* no qual as requisições o atinge (dividindo o serviço entre instâncias por faixas de IP).

As implicações práticas em escalar a aplicação por meio do *firewall* é que, muito embora sua velocidade seja exemplar, a gama de configurações possíveis é minimizada em vista do balanceamento de carga no lado do servidor.

Implementar o balanceador de cargas no lado do cliente alivia a sobrecarga no lado do servidor, contudo, deverá estar presente em todas as interfaces com os usuários sob o risco de ter seu código exposto (quando configurar-se como um problema) e versões desatualizadas do balanceador. Além disso, haverá menos flexibilidade em um código implementado no lado do cliente.

2.2.2. Comunicação

Uma vez adotada a arquitetura de microsserviços, o mecanismo pelo qual os serviços comunicam-se, naturalmente, deve dar-se por meio de um barramento de rede sob uma interface de rede e um protocolo de comunicação. Assim se procede pois não configura-se como ideal que processos distintos compartilhem alocamento em memória física (distinguindo-se do *socket*). Também não é a intenção dos microsserviços lançar mão de uma memória compartilhada distribuída. Abstraída toda forma de comunicação em redes de computadores, os microsserviços podem comunicar-se de diversas maneiras.

Cada maneira pode ser enquadrada em uma classificação. (COULOURIS et al., 2013) ofertam seis aspectos sobre os quais é possível classificar a comunicação inter-processos de um sistema distribuído, conforme enumera-se a seguir.

1. **Quanto ao número de destinatários:** O número de destinatários de uma mensagem pode ser um ou muitos, apontando para uma comunicação individual ou difusão de

mensagens;

2. **Quanto ao sincronismo:** A comunicação pode ser síncrona ou assíncrona exigindo bloqueios enquanto a mensagem demora ser entregue ou não;
3. **Quanto à confiabilidade:** Determinadas aplicações podem exigir que a entrega das mensagens seja garantida;
4. **Quanto a ordenação dos pacotes:** Outras ainda demandam que as mensagens sejam entregues na mesma ordem em que foram enviadas;
5. **Quanto ao acoplamento espacial:** O acoplamento espacial diz respeito ao conhecimento mútuo do endereço dos interlocutores, apontando para uma comunicação direta ou indireta entre eles;
6. **Quanto ao acoplamento temporal:** O acoplamento temporal refere-se à necessidade de ambos processos estarem on-line ao mesmo tempo.

Usualmente, em razão de sua leveza, simplicidade e vasta adoção, tende-se a usufruir do próprio Protocolo de Transferência de Hipertexto (HTTP (do inglês *Hypertext Transfer Protocol*)), abaixo da Transferência de Estado Representacional (REST (do inglês *Representational State Transfer*)) no âmbito dos microsserviços. Neste formato, há um estilo de comunicação individual, síncrona, confiável, ordenada, direta e temporalmente acoplada. Em determinadas ocasiões tal classificação da comunicação não atende as necessidades da funcionalidade a ser implementada, de sorte que faz-se necessário outro protocolo de comunicação que não o REST. Há, ainda, outro modo de implementar a comunicação interprocessos denominado invocação remota de métodos (RPC (do inglês *Remote Procedure Call*)), que necessita de um protocolo mais complexo que o HTTP.

O REST é um estilo arquitetural no qual são implementados quatro métodos, a saber, PUT, GET, POST e DELETE, operando sobre os mesmos métodos do HTTP a depender de sua implementação e cumprem o papel de casá-los com as operações CRUD (do inglês *Create, Read, Update, Delete*), usualmente necessárias em aplicações que gerenciam dados. A identificação das requisições são feitas por meio de uma URL (do inglês *Uniform Resource Locator*) direcionada ao serviço e geralmente listas são acessadas pelo plural dos substantivos (por exemplo, para o contexto usuário uma URL terminada em /users retornaria a lista de todos os usuários e a URL terminada em /users/1 retornaria os dados do usuário cujo identificador seja o número um).

Com a finalidade de possibilitar a heterogeneidade do sistema, representações externas de dados podem ser utilizadas. O processo de transcrever para um formato heterogêneo é denominado *marshalling* e ao seu oposto *unmarshalling*. Algumas formas de representação externa de dados são simplesmente linguagens, como JSON (do inglês *JavaScript Object Notation*)[3] e XML (do inglês *eXtensible Markup Language*)[4], e outras são compiladas e

[3] http://www.json.org/json-pt.html
[4] https://www.w3.org/XML/

interpretadas por programas, cujas API's devem encontrar-se em cada serviço interlocutor. Este último é o caso do Protocol Buffers[5].

O meio pelo qual as mensagens podem trafegar varia de acordo com a classificação da comunicação, conforme já explicitado. Cabe, portanto, aos arquitetos do software constatar qual tipo de comunicação melhor ajusta-se em sua aplicação. Mecanismos como filas, filas de prioridade e filas de difusão facilitam a escalabilidade e heterogeneidade de sistemas. *Buffer's* de mensagens podem ser implementados a fim de aumentar a velocidade do sistema.

Certamente há preocupações concernentes a segurança no tráfego das informações serializadas, de modo que mecanismos de criptografia e autenticação devem ser adotados conforme a vulnerabilidade do sistema, sua localização e a extensão do meio de comunicação.

2.2.3. Disjuntor

Tratando-se de muitos usuários, é inadmissível permitir que o sistema fique indisponível. Supondo um cenário em que um serviço está em seu limite de capacidade de processamento, quanto mais requisições o atingirem, deste ponto em diante, tanto mais sobrecarregado ele estará, fazendo com que nem mesmo as requisições anteriores já em processamento possam ser finalizadas. Sem a informação de que o serviço requisitado está colapsando, os serviços requisitantes continuam a enviar requisições e, pior, repetidas vezes, quando seus *timeout's* de espera pela resposta são atingidos. Em decorrência de uma sobrecarga pela vazão do sistema, um serviço pode ser desligado automaticamente. Sob esse aspecto, comumente junto ao balanceador de cargas, encontra-se também um disjuntor (*circuit breaker*), que implementa uma máquina de estados encarregada de checar a responsabilidade dos serviços e impedir que requisições os atinjam, se falhos. Um diagrama da máquina de estados pode ser visualizado na Figura 2.4, conforme elaborada no trabalho de (FOWLER, 2014).

O primeiro estado representa o momento em que o circuito está fechado – e portanto transmitindo corrente elétrica, em sua analogia ao disjuntor – e as requisições estão sendo atendidas por um determinado serviço que se esteja sendo monitorado pelo disjuntor. À medida em que as requisições continuam sendo atendidas, a máquina de estados permanece em circuito fechado. Um determinado limiar de falhas de responsabilidade é tolerado para um serviço, de modo que enquanto esse limiar não é atingido, a máquina de estados permanece fechando o circuito.

Uma vez que o limiar de falhas é atingido, a máquina de estados faz transição para o estado de circuito aberto – no qual a corrente elétrica deixa de inundar o mecanismo, em sua analogia – no qual as requisições destinadas ao serviço monitorado deixam de ser entregues, pelo disjuntor, a fim de permitir que o serviço em questão deixe de estar sobrecarregado. O disjuntor encarrega-se de responder ao requisitante que o serviço encontra-se indisponível.

[5] https://developers.google.com/protocol-buffers/

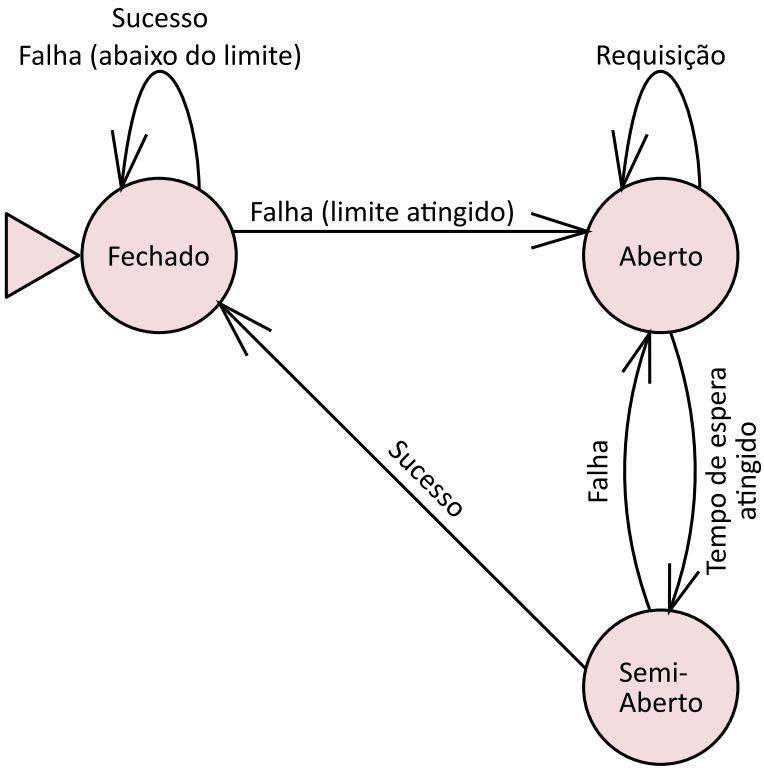

Figura 2.4. Máquina de estados do Disjuntor (FOWLER, 2014).

Um tempo determinado é configurado para que o disjuntor tente invocar o serviço cujo circuito encontra-se aberto, levando o circuito a um estado semiaberto. Se a tentativa de invocação é falha, o circuito retorna ao estado aberto. Caso contrário, a tentativa é bem-sucedida, o circuito fecha-se e o serviço volta a responder aos seus requisitantes.

Como a implementação do disjuntor costuma acompanhar o balanceador de carga, também esta ferramenta pode ser implementada tanto no lado do servidor como no lado do cliente. Suas implicações são análogas às discorridas na Seção 2.2.1, ou seja, alívio de processamento no lado do servidor, dificuldade de manutenção e exposição do código-fonte contra sobrecarga no servidor, facilidade de manutenção e proteção do código-fonte.

2.2.4. API *Gateway*

Justamente pela razão de servir muitos usuários, um sistema deve ir de encontro às preferências de cada classe de usuários a que serve, de modo que muitas interfaces de interação com o usuário são adotadas. Cada interface com o usuário captura os eventos e dispara as requisições aos serviços desejados orquestrando-as. À medida em que o número de serviços e/ou o número de dispositivos do lado do cliente aumenta, a dificuldade em gerenciá-los e manutení-los aumenta proporcionalmente, pois cada um dos dispositivos deveria fazer inúmeras requisições a inúmeros serviços.

Um sistema nessas condições padece no quesito segurança por deixar a regra de negócio muito próxima do cliente (para os casos em que expor o código-fonte for um problema),

especialmente em sistemas cuja linguagem escolhida não é compilada. Além disso, mudanças significativas na regra de negócio implicariam na refatoração de todas as interfaces com o usuário, cenário em que usuários cujo aplicativo não estivesse atualizado, usuários poderiam não conseguir usufruir do sistema ou, pior que isso, acarretar problemas para o sistema. Sob o ponto de vista dos desenvolvedores, saber que usuários podem não atualizar o aplicativo de interface traciona o desenvolvimento em muitos aspectos pelo receio em modificá-lo. Neste caso, em que a regra de negócio espalha-se por diversos dispositivos, o princípio do reúso de código é frontalmente violado. O encapsulamento da regra de negócio é nulo. Adicionar dispositivos no lado do cliente pode ser demasiado laborioso.

Para sua solução, um API *Gateway* é encontrado sempre que diversas aplicações precisam fazer interface com os diversos serviços presentes no sistema. O API *Gateway* atua unificando os serviços sob um único ponto de entrada ao sistema. A Figura 2.5 esquematiza a presença do API *Gateway* e sua interação com o conjunto. O dispositivo do usuário conhece o endereço e estabelece conexão com o API *Gateway* e este, por conseguinte, reúne as informações necessárias para a renderização das informações no dispositivo e as envia ao dispositivo. Procedimento análogo ocorre quando requisições são feitas ao sistema.

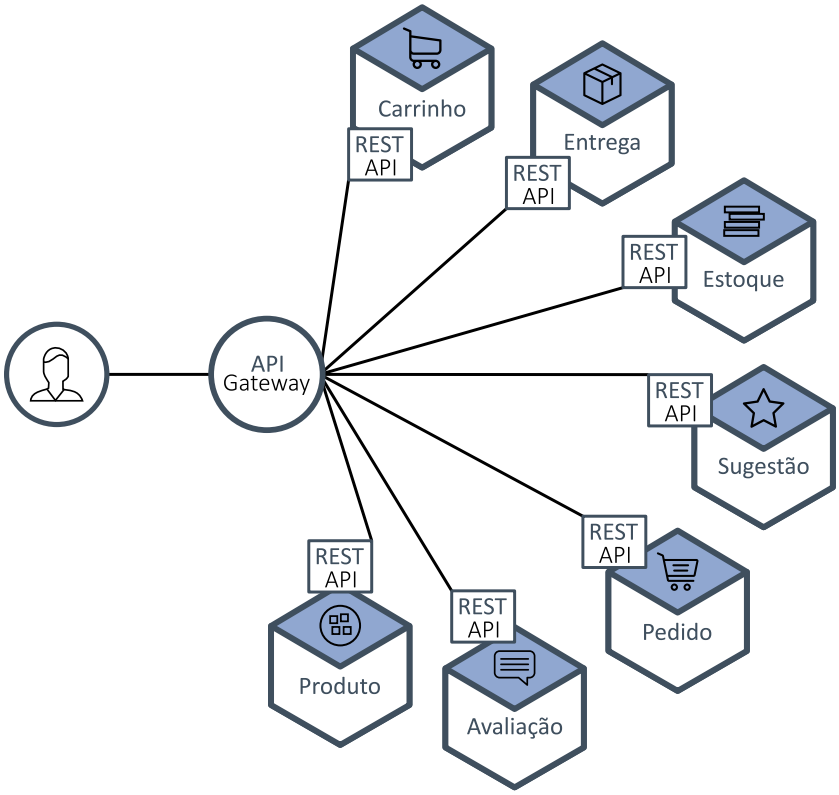

Figura 2.5. Funcionamento do API *Gateway* (RICHARDSON; SMITH, 2016).

Existe ainda uma motivação adicional para a adoção de um API *Gateway* que é o cenário onde os dispositivos que comunicar-se-ão com o sistema não possuem suporte ao protocolo REST (ou qualquer outro que se desejar padronizar). Com o API *Gateway* há a

padronização em REST no lado do servidor e a comunicação da melhor forma possível com o dispositivo. Note-se que, dado as diferenças, sutis ou não, entre os diversos dispositivos pelos quais a aplicação é acessível é interessante que haja um API *Gateway* para cada dispositivo. Assim procedendo, o balanceamento de carga e a escalabilidade são mais efetivos no que se refere ao identificar da origem da anomalia em termos de requisições.

Há ainda a possibilidade de implementar o API *Gateway* no lado do cliente uma vez que é comum vê-lo junto aos outros serviços de balanceamento de carga e disjuntor. O arquiteto de software deverá ter ciência do raciocínio detalhado nos parágrafos anteriores.

2.2.5. Descobridor de Serviços

À medida que o número de serviços cresce ou à medida em que os serviços precisam ser escalados, armazenar o endereço IP (do inglês *Internet Protocol*) e porta torna-se um problema organizacional. Se um serviço mudar de IP será necessário a refatoração de todo o sistema. O mesmo aplica-se ao adicionar-se novos serviços. Como solução, faz-se necessário um mecanismo que armazene o endereço IP e a porta de destino de cada um dos serviços, de modo que baste a cada serviço conhecer o endereço IP do Descobridor de Serviços e indagá-lo pelo IP que endereça o serviço requerido.

Há também uma particularidade na computação em nuvem em que os hospedeiros dos serviços não raras vezes mudam de endereço IP. Quanto mais frequente as mudanças ou quanto maior o número serviços, tanto mais transtorno tais situações acarretarão. Também nisto consiste um motivo determinante para a adoção de um Descobridor de Serviços.

Descobridores de Serviços podem ser implementados de maneira que os microsserviços registrem-se, que o próprio Descobridor de Serviços ou que outrem o faça. Neste último caso denomina-se *3rd Party Registration*. O registro por parte do Descobridor de Serviços implica em fazê-lo monitorar a rede. O registro por outrem pode-se ser aplicado quando os registros são feitos somente sob demanda (ou seja, conforme os serviços vão sendo requisitados e não constam entrada no registro).

Algumas implementações do Descobridor de Serviços podem ofertar mecanismos de injeção de dependência[6] no qual não é necessário lidar explicitamente com os endereços IP.

Embora o Descobridor de Serviços usualmente encontre-se em endereço estático, é possível implementá-lo no lado do cliente.

2.2.6. Serviços Periféricos

Uma vez que muitos microsserviços comporão o sistema, faz-se importante que uma interface gráfica apresente a situação de cada microsserviço, o tráfego de rede com o qual está lidando, o fluxo de requisições e tantas quantas mais informações forem necessárias à equipe de

[6] Reduz o acoplamento de sistemas injetando código em tempo de execução

gerenciamento da infraestrutura do sistema. Com este propósito, um serviço responsável por monitorar os microsserviços é indispensável.

A fim de garantir a segurança entre as trocas de mensagens por parte dos serviços, há a necessidade de um serviço de autenticidade, que impede que mensagens de origem adversa sejam tratadas como reais.

À medida em que o sistema cresce, gerenciar senhas, variáveis de ambientes e nomes de bancos passa a ser um problema. Com a finalidade de evitar o espalhamento da informação ao longo do sistema, faz-se o uso de um gerenciador de configurações.

2.2.7. Implantação

Assumindo a arquitetura de microsserviços, fatidicamente assume-se também um elevado número de serviços a serem implantados, dado o tamanho reduzido de cada um. É verdade que a implantação de um sistema em seu ecossistema pode ser simples dado todo o aparato de serviços de suporte que foram apresentados, no entanto, tratando-se de diversas linguagens, diversos bancos de dados, diversas variáveis de ambiente, diversos sistemas hospedeiros em diferentes versões, implantar (**ou testar**) um serviço passa a ser um problema. Não há razões para deixar de pensar em automatizar o processo de implantação em um cenário como este, em que existem muitos serviços e cada um carece de necessidades distintas. Para além do número de serviços, muitos são os casos em que o ciclo de vida de um serviço exige entregas frequentes.

Para enfatizar o problema em seu todo, cabe mencionar a cultura DevOps (BALALAIE et al., 2016). Nela é proposto que o time de desenvolvimento trabalhe em conjunto com o time de operações (isto é, gerentes de infraestrutura) sob um formato de métodos ágeis (BECK et al., 2001) em times pequenos. Este formato condiz muito com a arquitetura de microsserviços no que concerne o tamanho reduzido dos serviços que, naturalmente, são manutenidos por times com o tamanho máximo de duas pizzas (time menor ou igual a doze desenvolvedores (FOWLER; LEWIS, 2014)). Tanto a cultura DevOps como os métodos ágeis têm a intenção de fazer entregas (do produto) constantes, de maneira que o custo de implantar o sistema deve ser minimizado. Para tanto, testes devem ser automatizados sob um mecanismo de integração contínua e a implantação envolta em um ambiente idêntico ao do desenvolvimento sobre o qual sua construção também seja automatizada.

Fazer com que cada serviço seja implantando em um servidor dedicado pode não ser de fácil gerenciamento. Por outro lado, executar vários serviços em um mesmo hospedeiro pode eventualmente gerar incompatibilidades em versões de bibliotecas utilizadas na aplicação, além de conflitos de portas, configurações e variáveis de ambiente. Para que a infraestrutura de *hardware* não padeça nesses quesitos admite-se a aplicação das técnicas de virtualização ou conteinerização.

A virtualização dá-se por meio de um *hypervisor* que emula máquinas virtuais a executarem sistemas operacionais isolando-as. Neste modo, sistemas operacionais inteiros devem ser executados para que um serviço seja erigido.

Nesse sentido, vigora muito o uso de contêineres, que, em sua analogia, isolam o ambiente de execução de um serviço sob um formato externo idêntico aos outros, que podem ser transportados e vinculados à uma fonte de energia. Sua estrutura interna (como variáveis de ambiente, versões de bibliotecas e interpretador de código-fonte) é encapsulada em um formato descrito por uma linguagem definida e sua construção é automatizada por comandos simples. Sua carcaça é facilmente substituível e não são necessários sistemas operacionais inteiros para executarem os serviços. De maneira diferente do *hypervisor*, a conteinerização não provê isolamento, de modo que questões ligadas à segurança ainda são discutidas, sob a alegação de que a inviolabilidade do sistema deve residir na aplicação em benefício da leveza do contêiner (no que tange sua execução e seu tamanho).

Ainda mais ideal é que o aparato de implantação também faça o monitoramento da saúde do serviço, garantindo-lhe reinícios automáticos em casos de falha.

Indubitavelmente a complexidade de um sistema aumenta à medida em que este distancia-se da origem dos eixos de escalabilidade, de modo que imprescinde expor seus busílis. Assim o faz a seção seguinte.

2.3. Discussão

Para o bem ou para o mal a arquitetura de microsserviços, conforme demonstrada nas seções precedentes, a um só tempo em que alivia a complexidade de alguns elementos, outrora presente pela derivação da SOA, incrementa em muitos pontos a complexidade de outros. Suas implicações miram não somente a organização dos dados e manutenção de sua consistência e o fluxo de requisições, mas também estes mesmos aspectos no que se refere à equipe de desenvolvimento e todo o ecossistema no qual o projeto existe. Análises como a de riscos, custos e recursos devem ser feitas no mesmo ensaio em que os clientes do projeto podem ser surpreendidos por seu custo, que eleva-se em cada incremento e interações do cenário.

Os microsserviços carregam em si, primordialmente, a facilidade em se escalar a aplicação, haja visto que os serviços são pequenos, independentes e desacoplados. Este é, sem dúvidas, o ponto crucial da arquitetura e o benefício que leva os arquitetos a optarem por ela apesar de suas consequências.

Ao assumir uma arquitetura desacoplada em termos de processos, o mecanismo que conecta um software a outro impõe sobrecarga (*overhead*) ao conjunto, ainda que a interface de ambos esteja bem definida e o protocolo de rede seja sucinto. Há ainda questões ligadas a falhas na rede, seja por intervenção humana ou por aspectos relacionados à volatilidade dos nós intermediários.

No que concerne os dados armazenados no conjunto, há de se recordar que determinadas classes de aplicação não permitem que seus dados sejam particionados, como sugere a arquitetura de microsserviços quando determina a autossuficiência dos serviços. Uma vez que adotada a arquitetura de microsserviços em sua totalidade, ter seus dados particionados, replicados e distribuídos ao longo do sistema faz com que mecanismos de restabelecimento da consistência dos dados sejam adotados, quando a adoção de transações distribuídas não for uma opção em virtude da opção pela disponibilidade dos dados.

No que concerne a responsabilidade de um microsserviço, há de se notar que responsabilidades pequenas implicam em serviços menores, o que pode deixar de ser uma solução e passar a ser um problema dado o tamanho da equipe, sua interação e estilo de desenvolvimento, além de importâncias menores como o ambiente de implantação. A regra mais forte que dita a responsabilidade do microsserviço é o balanceamento da equação entre oferta e demanda. Equacionando-se as incógnitas proporcionará ao sistema uma interface bem definida para cada serviço abstraindo sua lógica, encapsulando sua regra de negócio e fornecendo um formato de entrada e saída especificados e raramente modificados. Sua implicações são prover ao time de desenvolvimento facilidade em manutenir o sistema (impedindo inclusive que programadores inexperientes cometam erros mais grosseiros), facilidade em desenvolvê-lo (no que diz respeito à divisão clara da responsabilidade), facilidade em testá-lo e depurá-lo, além do desacoplamento de código que permite seu reúso. Em contrapartida torna-se mais difícil executar os testes de integração (uma vez que todo o sistema deverá estar sendo executado) e, ainda, dificulta a refatoração geral da aplicação (uma vez que haverão vários projetos).

No que concerne a equipe de desenvolvimento do sistema, há de se frisar que sua interação e experiência devem ser arrojadas ao ponto de dominar os conceitos inscritos neste trabalho e compreender o sistema em ambas abordagens *top-down* e *bottom-up* a fim de desenvolvê-los satisfatoriamente de modo a permitir a integração do sistema. Os conceitos da cultura de desenvolvimento de software intitulada DevOps são facilmente aplicáveis. Um dos paradigmas que melhor conversa com a arquitetura é o desenvolvimento ágil combinado com técnicas de integração contínua e entregas constantes. Obviamente implantações devem ser automatizadas.

Por derradeiro, cumpre salientar que determinado grau de complexidade que verifica-se nesta arquitetura enseja a adoção de tecnologias que a reduzam ao ponto de torná-la factível no compasso que o mercado exige. O capítulo a seguir encarrega-se de explanar as classes de tecnologias existentes cuja finalidade seja oferecer suporte à arquitetura de microsserviços.

Capítulo 3

Tecnologias para Microsserviços

Verificou-se no capítulo anterior que a arquitetura de microsserviços é complexa ao ponto de justificar-se a elaboração e adoção de software intermediário, que faz interface entre os conceitos e os desenvolvedores.

Ora, evidentemente, adicionar-se camadas de código a um sistema contribui para que ele seja mais 'pesado', isto é, lento. Ponderadas as finalidades do uso, se coerentes, a adoção de *frameworks* faz-se necessária.

Neste trabalho, o termo *framework* não é tido em conotações diversas da implementação de sistemas, como o são, em sua tradução, modos de trabalho e formas de desempenhá-lo.

Os termos *framework*, ferramenta a API não confundem-se: *frameworks* são camadas de código cuja finalidade é abstrair determinadas funcionalidades que são comuns aos desenvolvedores de determinadas classes de aplicações e opera não pela provisão de interface com outros sistemas, mas opera pela abstração de complexidades do próprio sistema a ser desenvolvido; as *ferramentas* não acoplam-se ao código-fonte, mas executam de maneira independente e proveem interfaces definidas usufruindo de mecanismos de comunicação inter-processos; as *APIs*, por sua vez, são justamente uma camada de código intermediário que cria uma interface de abstração, desenvolvida pelos desenvolvedores de um serviço já em produção, para o acesso à um determinado recurso externo à aplicação.

Há ainda outra categoria de ferramentas que são serviços ofertados por sistemas em nuvem. Nos serviços em nuvem há o cenário análogo às ferramentas, no sentido de operar sem o acoplamento ao código-fonte, e que disponibilizam APIs para diversas linguagens a fazer conexão aos serviços ofertados pelo provedor de nuvem. Os serviços ofertados pelos provedores de nuvem são escritos em linguagens que independem ao programador do sistema que os utilizará, fazendo bastar que haja uma API desenvolvida para a linguagem que o usuário programador está a utilizar.

No âmbito de microsserviços há, ainda que a arquitetura seja recente, em toda sua

gama de classificações, ferramentas que lutam para abarcar esta fatia do mercado. Há soluções em termos de linguagens de programação para microsserviços, *frameworks*, ferramentas e soluções em nuvem. A Seção 3.1 encarrega-se de trazer à luz as classificações existentes, ordenadas por nível crescente de acoplamento ao código.

As Seções 3.2 e 3.3, por sua vez, referem-se às tecnologias que foram escolhidas para serem analisadas neste trabalho. Respectivamente KumuluzEE e Spring Cloud & NetFlix OSS. Esses *frameworks* foram selecionados pela sua grande adoção no mercado, a proximidade histórica que a linguagem de programação Java possui com os sistemas distribuídos e o apontamento da literatura de uma possível degradação de desempenho exacerbada em outras linguagens (UEDA et al., 2016).

Evidentemente, quando o desenvolvimento de uma aplicação partir de um sistema já existente, isto é, quando uma uma aplicação é migrada para a arquitetura de microsserviços, a escolha das ferramentas tomará como base as tecnologias já utilizadas pela empresa e a familiaridade da equipe de desenvolvimento para com elas. Até mesmo quando o desenvolvimento das aplicações for iniciado já sob a arquitetura de microsserviços determinados quesitos deverão ser julgados.

3.1. Classificação

As ferramentas para desenvolvimento de microsserviços mencionadas nesta seção é fruto da busca nos indexadores IEEE (do inglês *Institute of Electrical and Electronics Engineers*) e ACM (do inglês *Association for Computing Machinery*). Outros artigos relevantes em outras revistas também foram incluídos no rol de ferramentas, além de ferramentas populares que não mencionadas diretamente. É apresentado em ordem crescente de acoplamento ao código o ferramental em cada classificação existente. Por primeiro, apresenta-se as linguagens de programação para microsserviços, em seguida os *frameworks*, as ferramentas, e, por fim, as soluções em nuvem.

3.1.1. Linguagens de Programação

O resultado da busca nos indexadores retornou duas linguagens de programação preparadas para microsserviços (Jolie e CAOPLE). Seu modo de desenvolvimento é diverso da programação estruturada ou orientada a objetos, mas um modo orientado a serviços, de forma a tornar o programa naturalmente paralelizável pela sua descrição de serviços que o podem ser paralelizáveis ou concorrentes.

A saber, o paradigma de programação estruturada apresenta-se em um conjunto de funções e variáveis globais, que podem ser disjuntas em módulos e descrevem um software. Não existem resolutores de colisão de nomes de variáveis ou funções, tampouco existem

classes e objetos. As interfaces do sistema podem ser descritas e disponibilizadas em um arquivo separado para o caso de um software intermediário.

O paradigma orientado a objetos, ao contrário, preocupa-se em mapear um cenário real em um modelo computacional em que classes descrevem os objetos e suas ações. A programação orientada a objetos está calcada nos alicerces: polimorfismo, a capacidade de um objeto comportar-se de variadas maneiras em tempo de execução; herança, a capacidade de uma classe ter comportamento e/ou atributos análogos à outrem acrescido de suas especificidades; abstração, a capacidade de um método ou classe explicar seu comportamento dado seu nome e o respectivo mapeamento do cenário; e encapsulamento, a capacidade especificar as permissões de acesso aos atributos e métodos de uma classe.

Na orientação a serviços um software pode ser descrito em operações paralelas ou sequenciais, que são postas em execução pela máquina virtual da linguagem conforme a disponibilidade de hospedeiros. Neste paradigma novos hospedeiros são facilmente adicionados ou removidos e os trechos de códigos mais requisitados são facilmente atendidos. As linguagens orientadas a serviços carregam em si a ideia de concentrar esforços no desenvolvimento ao invés da implantação.

A linguagem de programação Jolie[1] está já mais consolidada, oferecendo mais suporte e documentação. No entanto, seu desempenho é baixo e exige que toda a regra de negócio esteja implementada nela, para que seja totalmente paralela. Os limites de um serviço podem não ser bem traçados, implicando em dificuldades na distinção do banco de dados ou até mesmo na concentração dos dados. A linguagem pode incluir código Java, uma vez que sua execução ocorre acima da máquina virtual Java, e também código JavaScript. Os *drivers* de conexão com o banco de dados também são *drivers* JDBC, isto é, *drivers* feitos para conexão na linguagem Java.

A linguagem CAOPLE (do inglês *Caste-centric Agent-Oriented Programming Language and Environment*) (XU et al., 2016) tem o mesmo intuito que a linguagem Jolie, entretanto ainda está em fase de desenvolvimento e, portanto, não é possível utilizá-la. Deste modo, ainda não é possível comparar o desenvolvimento nas linguagens para microsserviços.

3.1.2. *Frameworks*

Já os *frameworks* não alteram o formato da programação em sua essência, mas mesclam-se ao código produzido alterando-o sutilmente, seja pela injeção de dependências ou pelo formato decomposto em funções entranhadas pelo *framework*.

Escritos na linguagem de programação Java existem, de acordo com a busca realizada, os *frameworks* KumuluzEE e Spring Cloud & Netflix OSS, os quais são detalhados nas Seções 3.2 e 3.3, por haverem sido escolhidos para ser analisados.

[1] http://www.jolie-lang.org/

Além de *frameworks* para Java, há ainda o *framework* Seneca[2], escrito para JavaScript, que, contudo, altera razoavelmente a forma da programação. Seu modo de funcionamento é etiquetar cada função com um padrão definido pelo programador, por meio de uma função do *framework* onde são informados o padrão e a referência da função que deverá ser invocada. Os padrões etiquetados em cada função encapsulada pelo *framework* são executados conforme os padrões são coincididos, de modo a abstrair a descoberta de serviços e permitir que os trechos de código estejam localizados em diversos hospedeiros.

Outro *framework* ainda interessante é o Vert.x[3], que impõe menores acoplamento e *vendor lock-in*[4] por ser poliglota[5], no sentido de que o projeto tem mais liberdade para mudar de linguagem ou (mais recorrentemente) trabalhar com variadas linguagens de programação em cada microsserviço do sistema sem grandes dificuldades. Vert.x é também um contêiner web assíncrono e ambiente de máquina virtual de execução.

3.1.3. Soluções em Nuvem

Mais poliglotas são as soluções em nuvem, que oferecem vários serviços desde o balanceador de cargas até um ambiente de implantação conteinerizada de fácil escalabilidade. Os provedores de nuvem ganham crédito no mercado por abstraírem o gerenciamento da infraestrutura, ofertando uma interface gráfica amigável, completa e pronta para os gerentes de infraestrutura do sistema. A terceirização da computação dos dados em uma infraestrutura que não precisa ser manutenida pelos desenvolvedores em termos de hardware, rede e demais minúcias conforme o tipo de serviço escolhido (SaaS[6], PaaS[7], IaaS[8]) faz com que haja maior concentrar de forças no processo de desenvolvimento.

O rol de empresas que oferecem tais serviços é vasto o suficiente para que sejam aqui elencados apenas os mais populares e transversais até onde a pesquisa pôde alcançar: AWS (do inglês *Amazon Web Services*)[9] (gerido pela Amazon[10]), Azure[11] (gerido pela Microsoft[12]) e Google Cloud Platform[13] (gerido pela Google[14]).

---

[2] http://senecajs.org/
[3] http://vertx.io/
[4] Aprisionamento tecnológico, termo emprestado das ciências econômicas, denota o aprisionamento de uma empresa à um fornecedor seja qual for a razão.
[5] *Framework* escrito em e para muitas linguagens.
[6] Software como um Serviço, do inglês *Software as a Service*, refere-se, em geral, aos contêineres web, já preparados e abstraídos das configurações infraestruturais.
[7] Plataforma como um Serviço, do inglês *Platform as a Service*, refere-se a um ambiente mais flexível que o SaaS por permitir mais configurações.
[8] Infraestrutura como um Serviço, do inglês *Infrastructure as a Service*, refere-se ao serviço de locação de máquinas virtuais.
[9] https://aws.amazon.com/
[10] https://www.amazon.com/
[11] https://azure.microsoft.com/
[12] https://www.microsoft.com/
[13] https://cloud.google.com/
[14] https://www.google.com/

Além das API's nativas, há ainda API's *multi-cloud* que aliviam determinado *vendor lock-in* por permitir a conexão com diversos provedores de serviços em nuvem. Há de se ponderar, nesse caso, se vale aos arquitetos de software aprisionar-se em uma linguagem de programação (uma vez que não foram encontradas API's *multi-cloud* poliglotas) ou admitir o uso de mais de uma API multi-cloud, para cada linguagem adotada no sistema de microsserviços.

Seja em API's nativas, seja em API's *multi-cloud*, para além do vasto número de variáveis, um estudo com os provedores de nuvem envolve custos elevados e que devem ser considerados. Sua transversalidade e vasto rol de provedores e linguagens nos sistemas de API's nativas e *multi-cloud* torna uma comparação entre os provedores dispendiosa ou incompleta.

3.1.4. Ferramentas

Algumas soluções, no entanto não-transversais, são ferramentas independentes que, justamente por serem independentes, independe para os programadores (tal qual os serviços fornecidos pelos provedores de serviços em nuvem) a linguagem de programação utilizada e, mais ainda, não demandam API's para sua conexão, apenas uma interface da comunicação inter-processos.

A Tabela 3.1 reúne todas as ferramentas encontradas durante a pesquisa. Nela estão relacionadas as ferramentas com as características da arquitetura, demonstrando o tipo de solução que cada ferramenta provê. Cabe ressaltar que em muitos casos as ferramentas não têm a intenção de cobrir todas as característica da arquitetura de microsserviços, especializando-se em apenas uma delas.

Embora cada uma dessas ferramentas obtidas na busca detenham sua fatia de mercado e possuam sua importância no desenvolvimento de microsserviços, elas resolvem características pontuais da arquitetura de microsserviços. Sua especificidade não abarca todas as características da arquitetura de microsserviços e ferramentas que não têm o mesmo propósito não são comparáveis.

3.2. KumuluzEE

O *framework* KumuluzEE é escrito na linguagem de programação Java e tem por criador o esloveno Tilen Faganel. Seu primeiro *commit* no repositório do Git foi em quatro de Maio de 2015, o que lhe rendeu a conquista do prêmio *Duke's Choice Award* no ano. Seu repositório está hospedado no GitHub[43] sob a licença MIT[44] e seu site oficial é *https://ee.kumuluz.com*.

[43] https://github.com/kumuluz/kumuluzee
[44] https://opensource.org/licenses/MIT

		Balanceador de Cargas	*Comunicação*	*Disjuntor*	*API Gateway*	*Descobridor de Serviços*	*Manutenção da Consistência*	*Monitoramento e Méticas*	*Implantação*
	ETCD[15]	■				■			
Apache[17]	*Zookeeper*[16]	■				■			
	Airavata[18]		■						
	NGINX[19]	■			■	■		■	
HashiCorp[23]	*Consul*[20]	■				■		■	
	Serf[21]					■			
	Vagrant[22]								■
	Terraform[24]								■
	Marathon	■				■		■	
	JMQS		■						
	Kookarinrat[25]		■						
	Akka[26]			■					
Red Hat[29]	*WildFly*[27]					■			
	Hawkular[28]							■	
	OpenShift[30]								■
	Eventuate[31]						■		
	Gadea[32]						■		
	Kozmirchuk[33]						■		
Docker[34]	*Docker*							■	■
	Swarm								■
	Compose								■
	Lightbend[35]							■	
	Kubernetes[36]	■				■			■
	Dynamite[37]								■
	KVM[38]								■
	OSv[39]								■
	Guo[40]								■
	Gru[41]								■
	Ansible[42]								■

Tabela 3.1. Ferramentas para Microsserviços.

A Figura 3.1 apresenta o interesse pelo *framework* KumuluzEE ao longo do tempo. Tais dados foram obtidos e gerados em visualização gráfica pela ferramenta Google Trends[45]. Nota-se que após sua criação e conquista do prêmio *Duke's Choice Award* houveram buscas pelo *framework* e ainda hoje procura-se por ele, de modo que é importante analisá-lo. Considera-se ainda que existem empresas, segundo seu portfólio, que utilizam o *framework*, fazendo-se importante uma análise sobre ele.

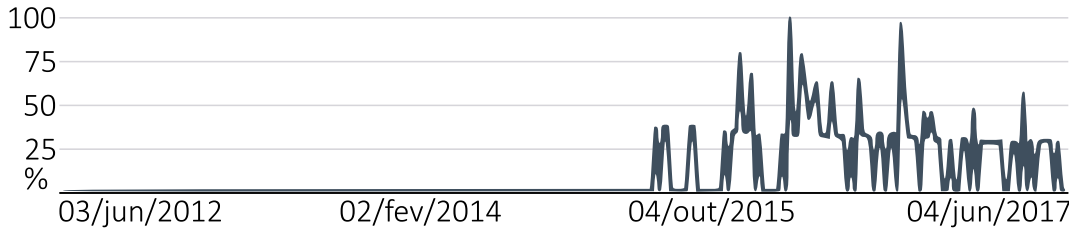

Figura 3.1. Interesse por KumuluzEE ao longo dos anos segundo o Google Trends.

[45] https://trends.google.com/trends/

A Tabela 3.2 apresenta os metadados do repositório do Git do KumuluzEE. Os dados foram observados pela página do projeto no GitHub.

Repositórios	1
Contribuidores Principais	4
Issues	12
Pull Requests	3
Commits	291
Releases	11
Stars	140
Watches	33
Forks	26

Tabela 3.2. Interação com o GitHub pelo KumuluzEE.

O *framework* pode ser utilizado adicionando-se as dependências a um projeto Java. As dependências também estão registradas no gerenciador de dependências Maven[46]. O funcionamento do *framework* se dá por injeção de dependências[47] para abstrair a descoberta de serviços. O *framework* também utiliza o servidor Web JBoss[48] integrado para que com uma aplicação *desktop* seja possível instanciar um servidor Web de maneira automática. O *framework*, ainda, engloba a aplicação e faz o gerenciamento dos recursos e ações declaradas nas classes e funções, dispensando a necessidade de um método *main*.

3.3. Spring Cloud & NetFlix OSS

O *framework* Spring Cloud é, também, escrito na linguagem de programação Java e tem um histórico maior por ser construído sobre o Spring *framework*[49], gerenciado pela Pivotal[50], sob a licença Apache 2.0[51]. Seu código-fonte está disponível no GitHub[52] e é disposto como sendo uma organização. Cada funcionalidade do Spring Cloud possui um repositório dedicado, atribuindo um nome para cada funcionalidade. Seu site oficial é *http://projects.spring.io/spring-cloud/*.

[46] https://maven.apache.org/
[47] as atribuições às variáveis anotadas são feitas em tempo de execução e pelo *framework* que resolve as anotações por meio da reflexão do método construtor padrão da classe que tipifica a variável
[48] http://www.jboss.org/
[49] http://spring.io
[50] https://pivotal.io/
[51] http://www.apache.org/licenses/LICENSE-2.0
[52] https://github.com/spring-cloud

A empresa de transmissão por *stream* de filmes e séries NetFlix[53], quando viu-se em face a um número exponenciado de requisições em seus serviços, dados que utilizava o *framework* Spring, migrou para a arquitetura de microsserviços no mesmo momento em que levou seus serviços para implantação em nuvem, na AWS. Ao migrar para uma arquitetura de microsserviços, implementou diversas das características da arquitetura de microsserviços e que lhe eram necessárias pois o *framework* Spring não oferecia suporte à arquitetura de microsserviços – à época ainda prematura. Em seguida, a NetFlix, disponibilizou o código-fonte de algumas destas implementações, que foram, naturalmente, incorporadas e encorajadas pelo *framework* Spring.

A Figura 3.2 apresenta o interesse pelo *framework* Spring Cloud ao longo do tempo. Tais dados também foram obtidos e gerados em visualização gráfica pela ferramenta Google Trends. Nota-se que seu número de interessados é expressivo e sofreu dobra no volume de sua procura com o advento dos microsserviços e colaboração do NetFlix OSS, de modo que é importante analisá-lo.

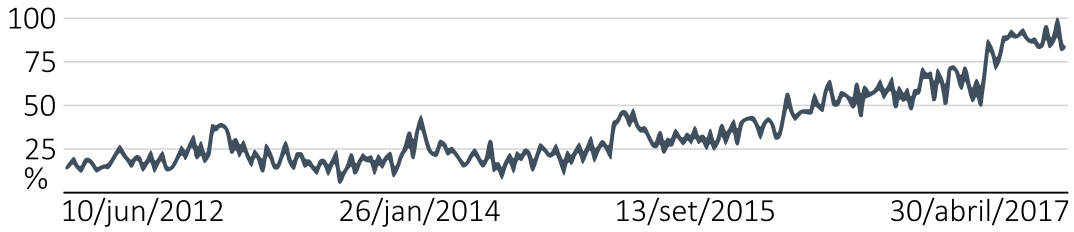

Figura 3.2. Interesse por Spring Cloud ao longo dos anos segundo o Google Trends.

A Tabela 3.3 apresenta a soma dos metadados de cada repositório do Spring Cloud, segundo disponibilizado pela API do GitHub. Estando o *framework* espalhado em setenta e cinco repositórios distintos, existem variados números de contribuidores em cada repositório, de modo que alguns são mais requisitados que outros. Os números de contribuídores variam de um à cento e um. Em todos os repositórios cerca de dez possuem números expressivos como 634 *forks*, 311 *issues* e 1051 *watches* (valores máximos e não correspondem ao mesmo repositório). O primeiro repositório é datado de sete de Maio de 2013.

O *framework* pode ser utilizado adicionando-se as dependências a um projeto que estão registradas nos gerenciadores de dependências Maven e no Gradle[54]. O *framework* Spring Cloud também encapsula a aplicação, mas, ao contrário do KumuluzEE, trabalha com um executável e um método *main*. Um servidor web é instanciado e sua implementação foi feita pelo *framework* Spring, e portanto sua dependência é indiretamente adicionada. Seu funcionamento também se dá pela inversão de controle e injeção de dependência com o registro em arquivos de configuração no formato YAML (do inglês *YAML Ain't Markup Language*).

[53] https://www.netflix.com/br/
[54] https://gradle.org/

Repositórios	75
Contribuidores Principais	6
Issues	1248
Pull Requests	-
Commits	-
Releases	-
Stars	4015
Watches	4015
Forks	3275

Tabela 3.3. Interação com o GitHub pelo Spring Cloud.

3.4. Conclusão

Em resumo, variadas são as formas de se desenvolver sob a arquitetura de microsserviços, pois muitos são os pontos de apoio que a arquitetura possui distribuídos em diversas formas. Isto justifica-se pela complexidade da arquitetura, que demanda abstrações cujo fim seja adequar o tempo de desenvolvimento de uma aplicação com a efervescência do mercado.

Pode-se desenvolver em ferramentas independentes e desacopladas do sistema que se queira implementar, linguagens de programação específicas para microsserviços ou com o recurso e advento das nuvens que fazem um misto de acoplamento e desacoplamento, conforme explicado.

Por outro lado, *frameworks* também são vastamente utilizados e abarcam (ou pretendem abarcar) a arquitetura como um todo. Sua aceitação pelo mercado é intensa pela prática do usufruto de *frameworks*, de um modo geral. Como um arcabouço, os *frameworks* sustentam uma aplicação em toda sua arquitetura pela oferta de um esqueleto com ela (a arquitetura) condizente na qual a musculatura da aplicação é dada por seus desenvolvedores, tal qual o funcionamento biológico em sua analogia.

Ora, assim como os músculos do corpo humano nada podem fazer sem a ossatura que o sustente, seria decepcionante para os desenvolvedores de um sistema que ele não pudesse mover-se em uma direção por falta de suporte do *framework* escolhido. Desta maneira, é importante, em especial aos desenvolvedores de software, conhecer o suporte que cada *framework* oferece.

No que se refere à linguagem de programação Java, os *frameworks* KumuluzEE e Spring Cloud & NetFlix OSS são notáveis em relação a adesão pelo mercado e são absolutamente comparáveis, por enquadrarem-se na mesma classificação (*frameworks*) e proporem oferecer suporte completo à arquitetura de microsserviços. O Capítulo seguinte

encarrega-se de explicitar como os *frameworks* selecionados serão avaliados.

Capítulo 4

Método de Pesquisa

Conforme demonstrado, a arquitetura de microsserviços é complexa ao ponto de justificar a adoção de tecnologias que facilitem a implementação de um sistema sob essa arquitetura. Os *frameworks* são um dos tipos de tecnologias que materializam os conceitos da arquitetura e ofertam aos desenvolvedores facilidades para a programação. Contudo, aos arquitetos de software caberá escolher um dentre os *frameworks* existentes para adotar, e esta escolha envolve grande número de variáveis que precisam ser consideradas no intuito de evitar barreiras tecnológicas que impeçam a evolução do software.

A fim de avaliar os *frameworks* selecionados, seguiu-se o método que consiste na implementação de um cenário fictício no intuito de responder às perguntas previamente fixadas (e que serão expostas neste capítulo) e que naturalmente seriam levadas em consideração por um arquiteto de software quando no deparar-se com a necessidade de escolher um *framework* para implementar um cenário de microsserviços.

O cenário elaborado foi implementado em cada *framework* – unicamente por este autor – e ao passo em que as questões de pesquisa foram observadas e respondidas, gerando um parecer técnico sobre as ferramentas. A Seção 4.1 apresenta o cenário implementado e a Seção 4.2 apresenta os critérios sob os quais os *frameworks* foram avaliados durante a implementação do cenário. A Seção 4.3 apresenta as tecnologias adotadas, além dos *frameworks*.

Para os fins deste trabalho utilizou-se as versões '2.3.0' do *framework* KumuluzEE e 'Brixton' para o *framework* Spring Cloud. As implementações encontram-se disponíveis nas organizações TaxiCalls-kumuluz[1], TaxiCalls-react[2] e TaxiCalls-spring[3] do GitHub.

[1] https://github.com/TaxiCalls-kumuluz
[2] https://github.com/TaxiCalls-react
[3] https://github.com/TaxiCalls-spring

4.1. Cenário

Implementou-se o cenário de um sistema para chamadas de táxis. Embora o cenário seja fictício e ingênuo, sua razão de ser está na necessidade intrínseca de uma arquitetura de microsserviços, pensando-se em uma *start-up* que pretenda crescer e atender muitos usuários segundo as demandas do mercado, mercado por métodos ágeis e a cultura DevOps.

O fundamento econômico do negócio é a demanda que os passageiros têm de locomover-se em segurança, de maneira rápida com um serviço de qualidade, motorista confiável e preço baixo. Em troca deste serviço os passageiros ofertam dinheiro.

Os motoristas, por sua vez, demandam dinheiro. Para isso, ofertam serviço de locomoção e demandam passageiros confiáveis e facilidade em encontrar passageiros.

Além destas, cada pessoa tem suas próprias demandas que anseiam ser satisfeitas. Os passageiros concorrem pelos motoristas e os pressionam da mesma maneira que os motoristas concorrem pelos passageiros e os pressionam para que se façam cumprir suas demandas pessoais.

A oferta equaciona-se à demanda quando ambas as demandas são atendidas. Assim sendo, um serviço que facilite o rastreio das ofertas e demandas interessa ao mercado e encontra lastro.

Determinada aplicação é um problema, do ponto de vista dos arquitetos de software, particionável e escalável em determinados aspectos e rígido em outros. A saber, gerenciamento de pagamentos demanda rígido controle de modo que a consistência não pode ser sacrificada em virtude da escalabilidade, fazendo-se memória do teorema CAP discorrido na Seção 2.1. Mais que isso, o problema pode ser abordado de maneira incremental aos moldes das *start-ups*, conforme discorrido no capítulo introdutório deste trabalho. *Stakeholders* como *Call Center* e *Admin Panel* podem ser atendidos em *releases* seguramente futuras na medida em que a aplicação lograr êxito.

As seções que seguem descrevem o sistema sob o olhar da arquitetura de software em sequência de nível de abstração do mundo real, vivo e contínuo e seu respectivo mapear para variáveis discretas.

4.1.1. Processo de Negócio

Em palavras telegráficas, o fluxo de trabalho das ações de cada *stakeholder* é descrito nos itens que seguem, separados por cada ação que pode ser iniciada.

1. Uma única vez, o motorista faz o cadastro no sistema, inserindo seus dados bancários pessoais e referentes ao veículo. Tão logo tudo esteja correto e regular no cadastro, o motorista estará apto a entrar em horário de serviço.

2. Uma única vez, o passageiro faz o cadastro no sistema inserindo seus dados pessoais e cartão de crédito e já está apto a requisitar serviços.
3. O motorista ajusta no sistema o custo por quilômetro rodado e custo cobrado por minuto de viagem.
4. O motorista informa ao sistema que está em horário de serviço a seu critério.
5. O passageiro busca por um motorista, filtrando-os por carro ou motorista de sua preferência, ou ainda por maior reputação de um sistema de avaliações. O motorista é notificado e opta por aceitar ou rejeitar o serviço. Aceitando o serviço, o usuário é notificado e seu cartão de crédito é cobrado o motorista desloca-se para a localização do passageiro e transporta-o para seu destino. Bem-sucedida a viagem, o crédito é cobrado do passageiro e há o depósito na conta do motorista. O motorista pode avaliar o passageiro. O passageiro pode avaliar o motorista.
6. O motorista deixa o horário de serviço a seu critério, informando-o ao sistema.

A Figura 4.1 apresenta um esquema do fluxo de requisições que sofre o sistema desenvolvido quando na estória[4] de uma viagem. Na Figura, cada hexágono representa um serviço e a repetição dos hexágonos representa a possibilidade que todos os serviços têm de escalar-se em várias instâncias. As setas tracejadas representam uma comunicação indireta, como é o envio de notificações para os celulares e as setas contínuas representam a comunicação direta. Os números que rotulam cada seta definem a ordem de cada troca de mensagens e as repetições do mesmo número significam que as etapas ocorrem em paralelo, pela regra de negócio.

Para efetuar uma requisição de viagem, um passageiro requisita a viagem ao API *Gateway* (1), que descobre o serviço de gerenciamento de viagens e encaminha a requisição (2), que é processada de acordo com a máquina de estados inerente à viagem e retorna os motoristas disponíveis ordenados pela proximidade (3 e 4, considerando o API *Gateway*). O passageiro escolhe o motorista e solicita seus serviços por meio da requisição ao serviço de gerenciamento de viagens (5 e 6) que é notificado pelo serviço de notificações (7).

Ao aceitar uma viagem o motorista envia uma requisição ao serviço de gerenciamento de motoristas (1 e 2), que inunda o sistema com a informação do aceite (3): ao serviço de notificação, requere a notificação ao passageiro; ao serviço de gerenciamento de passageiros e de viagens, informa o estado da viagem como aceito; e ao serviço de fatura, requere a cobrança. Todos os serviços retornam resposta positiva (4) e que é encaminhada ao motorista (5 e 6).

[4] (*stories*) Alusão aos métodos ágeis.

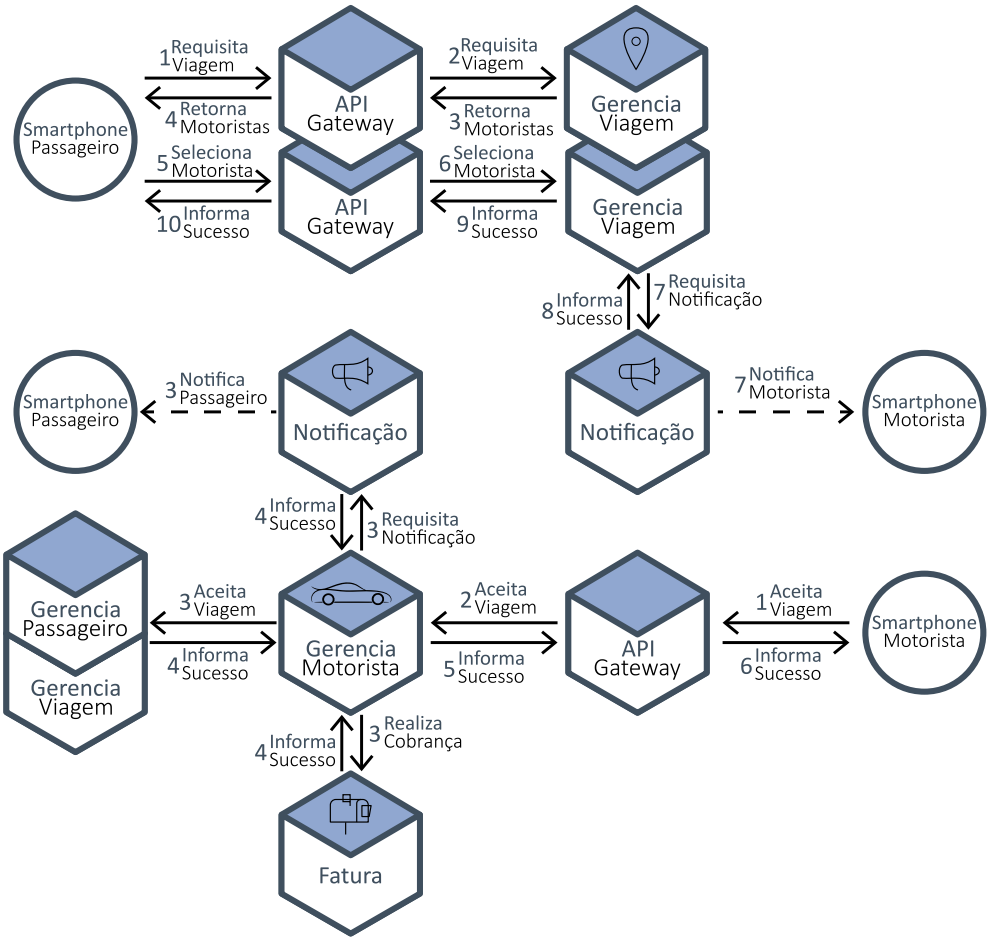

Figura 4.1. Fluxo de Requisições de uma Viagem.

4.1.2. Arquitetura

Para implementar o processo de negócio apresentado, utilizou-se um sistema distribuído na arquitetura cliente-servidor, ou melhor, na arquitetura de microsserviços.

A arquitetura conta com interfaces para dispositivos móveis para que os motoristas e passageiros tenham acesso ao sistema facilmente, qualquer que seja sua localização.

No lado do servidor, um API *Gateway* unifica todas as requisições de dispositivos REST.

O endereço IP do Descobridor de Serviços é obtido pelo nome de hospedeiro do aparato do Docker Compose. Seu funcionamento é o de armazenar a localização de cada serviço presente no sistema. Cabe a ele o repasse dessas informações para o *Gateway*.

Os serviços, contudo, não acessam diretamente os serviços devido ao disjuntor de segurança que anexado à entrada de cada serviço.

Cada serviço armazena seus próprios dados e oferta uma interface textual, ou seja uma resposta inteligível em um formato de representação externa de dados acessível pelo método GET do protocolo REST.

Cabe retomar a ideia de que as interfaces gráficas serão grosseiras e rudimentares

e limitam-se ao necessário para invocação da ação devido à ênfase que deseja-se dar aos *frameworks*. Muitas outras práticas poderiam ser adotadas para demonstrar um cenário de muitas requisições e um sistema elástico e resiliente que as suportasse, mas que, no entanto, não cruzam o campo de visão dos *frameworks* de modo que este trabalho restringi-se aos tópicos que os tocam.

4.1.3. Linha de Produção

Para implementar a arquitetura desenhada foram feitos ciclos de desenvolvimento individual para cada serviço em repositórios distintos inscritos em uma mesma organização (funcionalidade do GitHub[5]), de maneira a aproximar-se de um cenário real. Uma organização para cada *framework* avaliado e outra para as aplicações comuns a ambos.

Foi executado cada ciclo de implementação (*sprint*) duas vezes, uma para cada *framework* testado, alternando-se o *framework* que foi implementado por primeiro. Buscou-se a fixação de sistemas para o desenvolvimento de todos os ciclos que forem necessários para a sua implementação total, em vista da busca pela independência e autossuficiência de cada serviço. No entanto, um cenário real contaria com diversas equipes de desenvolvimento e o desacoplamento total entre os serviços é impossível – se assim fosse, não comporiam um sistema, que por definição opera em conjunto – de modo que foi admitido o desenvolvimento de ciclos de outros projetos para satisfazer a interação entre os serviços. Assim foi procedido para que o sistema continuamente fosse funcional e sua avaliação feita de maneira isolada e impessoal. Não houve fixação de tempo para o cumprimento dos ciclos.

4.2. Critérios de Avaliação

Ao passo que o sistema foi desenvolvido, foi-se avaliado sob critérios bem definidos e objetivos. Dois aspectos foram observados: funcionais e não-funcionais. Os aspectos funcionais descrevem, em síntese, as características da arquitetura de microsserviços, que foram descritas na Seção 2.2. Foram observados seu suporte à cada característica e uma descrição ponderada de seus detalhes e implicações para o desenvolvedor. Para os aspectos não-funcionais foram observados questões importantes para a empresa que decide por uma ou outra tecnologia, como grau de adoção, documentação, facilidade de uso e flexibilidade do *framework*.

4.2.1. Aspectos Funcionais

Quem quer que vá escolher uma tecnologia naturalmente a porá à prova sob uma série de questionamentos e critérios de desempate segundo suas próprias necessidades. No âmbito dos aspectos funcionais, foram ponderadas as indagações que seguem:

[5] https://github.com/

QP1 – O *framework* oferece suporte a um balanceador de cargas?
QP2 – O *framework* oferece alguma forma de manutenção da consistência dos dados?
QP3 – O *framework* trata a comunicação entre os serviços de forma abstraída?
QP4 – O *framework* trata a comunicação entre os serviços de variadas formas?
QP5 – O *framework* oferece a implementação de um disjuntor?
QP6 – O *framework* oferece a implementação de um API *Gateway*?
QP7 – O *framework* oferece suporte ao Descobridor de Serviços?
QP8 – O *framework* oferece alguma forma de monitoramento dos serviços?
QP9 – O *framework* trata de algum modo a segurança e autenticidade na comunicação?
QP10 – O *framework* facilita a criação de um serviço de configurações?
QP11 – O *framework* facilita a implantação dos serviços?

4.2.2. Aspectos Não-Funcionais

Ainda que determinado *framework* não ofereça suporte a uma ou outra característica da arquitetura de microsserviços, valerá ao arquiteto de software ponderar os aspectos não-funcionais das tecnologias em disputa no páreo.

Adoção

Para mensurar a adoção das ferramentas, além dos dados do repositório do GitHub apresentados nas tabelas 3.2 e 3.3, fez-se uma busca para extrair o número de repositórios no GitHub que os têm como dependência e a soma de usos apontados no gerenciador de dependência Maven.

QP1 – Quantos projetos no GitHub utilizam o *framework*?

Sendo o número de repositórios demasiado grande, não há como fazer uma busca completa para avaliar a adoção de cada ferramenta em tempo hábil. Em números concretos, até a data atual existem mais de noventa e três milhões de repositórios hospedados no GitHub criados por vinte e nove milhões de usuários. Utilizando sua API deveriam decorrer-se mais de quatro anos ininterruptos de computação, devido a sua limitação imposta de cinco mil requisições por hora para cada usuário que queira utilizá-la. Uma pesquisa mais inteligente, iterando-se sobre os usuários ao invés dos repositórios poderia completar a tarefa em três anos menos.

No entanto é verdade que a população de repositórios contempla grande número de áreas da computação, de modo que o subconjunto de repositórios que pertence à área de microsserviços é discrepante e constitui a porcentagem da população que interessa a esta pesquisa. Assim sendo, a busca pode ser otimizada utilizando-se esta amostra ao invés da população.

A imprecisão, porém, utilizando-se deste método, estará na obtenção da amostra, que será feita pela recente funcionalidade de rotulação em repositórios e que, no entanto, não garante a captura de todos os repositórios que utilizam a arquitetura de microsserviços, uma vez que cabe ao dono do repositório rotulá-lo.

Sabendo da imprecisão do método, mas com amparo no limite temporal com que este trabalho conta, coletou-se a amostra dos repositórios pelo rótulo de 'microsserviço' e operou-se sobre esse conjunto calculando (estimando) o nível de adoção que cada *framework* possui. Em um tamanho amostral de não mais que dois mil repositórios, sabe-se que repositórios não-rotulados foram desconsiderados.

Documentação

Para mensurar a qualidade da documentação da linguagem, investigou-se se:

QP2 – A documentação do *framework* contempla todas as características da arquitetura a que oferece suporte?
QP3 – A documentação do *framework* é traduzida para português?
QP4 – A documentação do *framework* é traduzida em quantos idiomas?
QP5 – O *framework* possui tutoriais de implementação básica?
QP6 – O *framework* possui tutoriais de implementação básica em todas suas características?
QP7 – A documentação contempla tópicos avançados?
QP8 – Quantas páginas de documentação o *framework* possui?
QP9 – Existe um manual para os métodos e classes do *framework*?
QP10 – Existem exemplos disponibilizados abordando cada uma das características da arquitetura?
QP11 – Os exemplos disponibilizados são didáticos e/ou ilustrados?

Facilidade de Uso

Em relação a facilidade no uso do *framework* foram observados os seguintes quesitos:

QP12 – O *framework* não interfere no modo como se desenvolve nesta linguagem?
QP13 – O *framework* opta por configuração, convenção ou parametrização?
QP14 – O *framework* utiliza injeção de dependências e inversão de controle?
QP15 – O *framework* faz o uso da API de reflexão do Java?
QP16 – Quanto conhecimento prévio do *framework* é desobrigado para sua utilização?
QP17 – O *framework* restringe o acesso a componentes internas?
QP18 – Quantas interfaces de abstração são providas pelo *framework*?
QP19 – É natural a implementação utilizando este *framework*?

Flexibilidade

É, ainda, necessário verificar se o *framework* traciona o desenvolvimento por dificultar a integração de sua solução com outras tecnologias. Nesse sentido foram investigados os tópicos a seguir:

QP20 – É possível integrar outras tecnologias ao *framework*?
QP21 – É fácil integrar outras tecnologias ao *framework*?

Para responder às questões, por primeiro foi avaliado a integração dos *frameworks* com as tecnologias utilizadas para implementar o cenário descritas na Seção 4.3 e em um segundo momento com algumas das tecnologias descritas na Seção 3.1 conforme o resultado das indagações referentes aos aspectos funcionais.

4.3. Tecnologias Auxiliares

Naturalmente não é satisfatório aplicar o método de entregas frequentes sem um aparato que o torne minimamente funcional. Destarte, serão utilizadas as tecnologias enumeradas a seguir, escolhidas pelo conhecimento prévio de seu funcionamento ou facilidade em alcançar-se o objetivo final de maneira ágil em circunstâncias não-primordiais a este trabalho.

1. **React Native**[6], um *framework* de desenvolvimento para JavaScript *back-end* (NodeJS[7]) que traduz código para Android e IOS, dispensando a necessidade de implementar uma aplicação para cada plataforma.
2. **Docker**[8], que cria contêineres (ao invés de um *hypervisor*) e encapsula a execução de um software. Sua utilização se dará na execução de cada serviço implementado.
3. **Docker Compose**, que prepara as variáveis de ambiente, cria redes virtuais, volumes de armazenamento e mapeia as portas abertas no sistema. Sua utilização se dará em todo o ambiente de teste, que automatiza a implantação.
4. **MariaDB**[9], bifurcação do banco de dados relacional MySQL cujo código-fonte é aberto. Sua utilização se dará nos serviços de gerenciamento e fatura, que exigem consistência ininterrupta.
5. **Git**, controlador de versões, que será utilizado para cada serviço implementado.
6. **GitKraken**[10], gerenciador de repositórios Git.
7. **Github**[11], hospedeiro de repositórios do Git, que será utilizado ao longo do desenvolvimento, separando-se em duas organizações, uma para cada *framework*.

[6] https://facebook.github.io/react-native/
[7] https://nodejs.org/en/
[8] https://www.docker.com/
[9] https://mariadb.org/
[10] https://www.gitkraken.com/
[11] https://github.com/

Capítulo 5

Resultados

Para a implementação do cenário proposto, adotou-se o modo de desenvolvimento exploratório, que consulta a documentação na medida em que dúvidas pontuais da implementação surgem. A tomada deste modo de desenvolvimento é importante para capturar a sensação dos desenvolvedores que assim procedem ao utilizar o *framework* e enquadra-se mais para os que estiverem a experimentar uma ferramenta para decidir adotá-la ou não. Este modo de desenvolvimento captura ainda a naturalidade do *framework* em prover o desejado, qualquer que seja, e ofertar envergadura sobre a arquitetura.

O método foi executado com a alternância de *framework* para o início de cada passo seguinte dentre o rol de tarefas menores que compuseram a ordem de implementação. Esse modo de desenvolvimento foi importante para eliminar vícios deste autor que pudesse influenciar a percepção de facilidade inicial de implementação de cada aspecto julgado em cada um dos *frameworks*. Embora massante, esse estilo de desenvolvimento pôde capturar a sensação do desenvolvedor em cada *framework* e sua respectiva alternância.

A ordem de implementação das funcionalidades foi traçada a começar pelo microsserviço que lida com os passageiros e a abertura de cada rota pelas quais o serviço interage ou deve interagir. A ordem de apresentação dos resultados segue a estruturação dos aspectos da arquitetura, conforme foram apresentados, além ainda de contar com uma seção dedicada às reflexões a respeito da arquitetura de microsserviços em si, outra para alguns detalhamentos sobre decisões de projeto importantes para reprodutibilidade dos resultados e uma conclusão na qual se pode chegar a partir das reflexões expostas.

5.1. Aspectos Funcionais

A Tabela 5.1 sintetiza as respostas obtidas pela execução do método, as quais são detalhadas as características da arquitetura. '✓' representa afirmativa à questão de pesquisa arrolada,

do contrário '—'. Casos onde há mais de um ícone '✓' significam que variadas formas são providas a funcionalidade, tantas quantos ícones houverem.

Questão de Pesquisa	*KumuluzEE*	*Spring Cloud*
QP1 *Oferece suporte a um balanceador de cargas?*	—	✓✓
QP2 *Oferece manutenção da consistência dos dados?*	—	—
QP3 *Trata a comunicação entre os serviços de forma abstraída?*	✓	✓✓
QP4 *Trata a comunicação entre os serviços de variadas formas?*	—	—
QP5 *Oferece a implementação de um disjuntor?*	—	✓
QP6 *Oferece a implementação de um API Gateway?*	—	✓✓
QP7 *Oferece suporte ao Descobridor de Serviços?*	—	✓
QP8 *Oferece alguma forma de monitoramento dos serviços?*	—	✓
QP9 *Trata a segurança e autenticidade na comunicação?*	—	✓
QP10 *Facilita a criação de um serviço de configurações?*	—	✓
QP11 *Facilita a implantação dos serviços?*	—	✓

Tabela 5.1. Respostas às Questões de Pesquisa Funcionais

5.1.1. Balanceador de Cargas

QP1 – O *framework* oferece suporte a um balanceador de cargas?

O *framework* KumuluzEE não possui uma implementação própria para o balanceador de cargas e sugere o uso da ferramenta Apache ZooKeeper[1] pela provisão de uma interface de abstração para ela disponibilizada nos seus exemplos.

O Apache ZooKeeper, atua como um descobridor de serviços que responde o endereço da instância do serviço requisitado que estiver desocupado, cumprindo exatamente o papel de um balanceador de cargas.

[1] https://zookeeper.apache.org/

Importa salientar que está sendo implementada uma versão própria do descobridor de serviços por parte da Kumuluz e que não pôde-se avaliar se determinado serviço também se encarregará do balanceamento de cargas.

O Spring Cloud, ao contrário, oferta uma implementação do balanceador de cargas por meio da anotação *@LoadBalanced* em um gerenciador REST (*RestTemplate*) ou automaticamente por meio dos clientes *@FeignClient*.

Faça-se menção ao mérito do Spring Cloud no momento em que também disponibiliza uma interface de abstração para o Apache ZooKeeper. A interface não foi avaliada.

5.1.2. Dados da Aplicação

QP2 – O *framework* oferece alguma forma de manutenção da consistência dos dados?

Com relação aos dados da aplicação, ambos os *frameworks* não provisionam nenhum meio para gerir a duplicação de informação e manutenção da consistência dos dados, ficando tão somente a encargo dos programadores.

Cabe observar que o *framework* Spring provê algumas interfaces de abstrações para o gerenciamento de classes POJO (do inglês *Plain Old Java Object*) (*@Repository*), mas isto somente para o ambiente interno de cada serviço. Note-se que determinadas abstrações existem em razão do plano de fundo (*background*) ofertado pelo Spring *framework* (e não Spring Cloud).

5.1.3. Comunicação

QP3 – O *framework* trata a comunicação entre os serviços de forma abstraída?

Em relação ao formato de comunicação entre os serviços, em ambos os *frameworks* não é preciso lidar explicitamente com uma representação externa de dados. O *framework* KumuluzEE por padrão retorna o formato JSON, mas por padrão recebe como entrada texto plano (*Mime Type plain/text*), de modo que é possível omitir a anotação *@Produces* mas é preciso informar a anotação *@Consumes* em cada método que lida com uma rota. O *framework* Spring Cloud por padrão produz e consome o formato JSON, fazendo não ser preciso informar o formato, quando optar-se pela escolha do formato JSON.

Com relação à comunicação em si, existem duas formas para criação de uma requisição no Spring Cloud *framework*: chamada (*RestTemplate*) para um endereço, informando classe para reflexão (no retorno); ou chamada para um cliente *@FeignClient* (herdado pela NetFlix), cuja dependência é injetada e internamente construída (tratando-se de uma interface Java)

bastando fazer-se coincidirem os caminhos virtuais (URLs), assinaturas dos métodos e nomes do serviço entre os interlocutores. É importante mencionar que o nome de um serviço é definido via arquivo YAML de configuração.

Para o KumuluzEE *framework* há somente a possibilidade da criação de requisições mediante a informação da classe de retorno para reflexão.

Com relação às especificidades, o KumuluzEE requer que seja definido o *MIME type* no cabeçalho do protocolo HTTP para uma requisição POST via cURL[2], por exemplo *Content-Type: application/json* no parâmetro *–head*. Não testou-se se a chave/valor precisa ser adicionada em uma requisição AJAX. Para as requisições intra-*framework* nenhuma especificação é necessária. O Spring Cloud não requer nenhuma especificação quando se utiliza o padrão por ele adotado (JSON).

Por outro lado, o Spring Cloud, embora não necessite que seja especificado o cabeçalho HTTP anunciando o *Content-type* como JSON, requer que os parâmetros dos métodos para POST sejam anotados com *@RequestBody*.

Em ambos os *frameworks* não é preciso informar a classe para a reflexão (*Reflection*) quando na passagem de parâmetros (antes falava-se do retorno), nem é preciso fazer recurso a outro tipo de retorno que não o simples retorno do objeto a ser serializado.

Há ainda outra peculiaridade entre os *frameworks* que faz balancear a relação de perdas e ganhos por conferir, o KumuluzEE, a possibilidade de fazer colisões entre rotas criadas, diferentemente do Spring. No KumuluzEE, por exemplo, é possível criar uma rota para *drivers/id* e outra para *drivers/available*. Também não há maiores dificuldades quando se insere mais de um caractere "/" no caminho, ou seja *drivers//available*.

QP4 – O *framework* trata a comunicação entre os serviços de variadas formas?

Com relação às diversas formas de comunicação, não encontrou-se comunicação para múltiplos destinatários, somente encontrou-se o envio de mensagens síncronas, confiáveis, ordenadas, acopladas temporal e espacialmente em ambos os *frameworks* nativamente.

Nos exemplos tomados como base encontrou-se a utilização da ferramenta RabbitMQ[3] junto às aplicações em Spring. A ferramenta RabbitMQ é desenvolvida pela Pivotal, assim como o *framework* Spring. Além de não tê-la utilizado na aplicação implementada, não fora encontrada a dependência para RabbitMQ junto aos exemplos tampouco encontrou-se o local onde se poderia estar configurado detalhes para o uso da ferramenta. Acredita-se que a simples adesão da dependência Maven AMQP (do inglês *Advanced Message Queuing Protocol*) faz com que todas as mensagens internas à aplicação sejam trocadas segundo o protocolo AMQP, o que de fato seria uma abstração muito eficiente pela transparência e parametrização absoluta. Isto teoriza-se porque deixar de executar o contêiner RabbitMQ na aplicação de

[2] https://curl.haxx.se/
[3] https://www.rabbitmq.com/

exemplo mencionada faz com que o sistema não erija-se. A ferramenta RabbitMQ suporta mensagens assíncronas, publicação e subscrição, filas de mensagens para múltiplos clientes e RPC, além de filas seletivas por meio dos protocolos HTTP, AMQP, STOMP[4] e MQTT[5]. Um vasto rol de linguagens de programação é suportado.

5.1.4. Disjuntor

QP5 – O *framework* oferece a implementação de um disjuntor?

Ambos os *frameworks* oferecem a implementação do disjuntor e basta fazer uso das anotações e escrever a rotina de tratamento da exceção no próprio serviço requisitante, ao invés de haver a necessidade de lançar várias instâncias de um serviço que faça o monitoramento e controle da passagem de requisições para o serviço requisitado. Embora haja tutorial e implementações já prontas, no KumuluzEE ainda não é possível utilizá-lo dado que sua dependência ainda não foi publicada no Maven.

No Spring Cloud há a anotação *@HystrixCommand* (herdada pela NetFlix) onde se informa o nome do método que tratará uma falha na requisição. O método anotado deve tratar de requisições via *RestTemplate*. O tratamento via interfaces *@FeignClient* também é possível pela provisão de um parâmetro de *fallback*[6] na anotação, onde se informa a classe a tratar a falha.

No KumuluzEE também é possível provisionar método ou classe para tratamento de falhas, contudo somente é possível fazê-los pela anotação dos métodos que utilizam requisições com o aparato do *ClientBuilder* (javax.ws.rs e interferências do KumuluzEE), uma vez que não há mecanismos de abstração para as chamadas.

Não se conhece a natureza da implementação de ambos os disjuntores, mas seu funcionamento é interno ao serviço requisitante. Para a garantia da resiliência da arquitetura é importante que haja um intervalo de tempo para fechar o circuito. Qualquer tempo dentro do intervalo deve ser respondido pelo disjuntor. Se porventura as implementações internas continuarem a redirecionar as rotas e somente tratar os erros colaborará para a inundação e queda do sistema.

[4] https://stomp.github.io/
[5] http://mqtt.org/
[6] *fallback* no sentido de 'bater em retirada', muito presente no contexto militar e que representa a rota de fuga quando a rota principal deixa de operar.

5.1.5. API *Gateway*

QP6 – O *framework* oferece a implementação de um API *Gateway*?

O *framework* Spring Cloud oferece a implementação de um API *Gateway* que é semelhante ao modo da criação de requisições.

Sobretudo, há o *@ZuulProxy*, herdado da NetFlix, que tão somente encaminha as requisições para os serviços competentes a partir da transparente invocação da URL do serviço alvo com o prefixo escolhido. Por exemplo, uma aplicação Web ou Móvel que queira autenticar-se (*login*) requisitará o serviço de autenticação (<authentication-service>/authenticate), mas sem conhecer seu endereço, o fará ao API *Gateway*, que tão somente disponibilizará a mesma URL acrescida de um prefixo qualquer (<gateway-service>/authentication/authenticate) tomando os mesmos parâmetros e devolvendo mesmo retorno que a assinatura do serviço alvo.

O *framework* KumuluzEE não possui nenhuma implementação para o API *Gateway* e somente é possível fazê-lo (assim como no Spring também é evidentemente possível fazê-lo) por meio da construção de um serviço que comporte-se abrindo rotas para todos os serviços que deverão ser acessados externamente.

5.1.6. Descobridor de Serviços

QP7 – O *framework* oferece suporte ao Descobridor de Serviços?

O Kumuluz não possui suporte ao Descobridor de Serviços, ao invés encoraja o uso do Apache ZooKeeper quando provê exemplos contando com uma interface de abstração para utilização da ferramenta para o registro e descoberta dos serviços. Nele os serviços registram-se a si próprios.

Conforme já mencionado, o *framework* KumuluzEE está desenvolvendo uma implementação própria do descobridor de serviços.

Em contra-partida, o *framework* Spring Cloud oferece o *Eureka* (herança da NetFlix) para o registro e descoberta dos serviços.

5.1.7. Serviços Periféricos

QP8 – O *framework* oferece alguma forma de monitoramento dos serviços?

No Spring Cloud todos os serviços que incluírem o motor HTML Thymeleaf terão rotas para */beans*, */env*, */health*, */metrics* e */trace*, que dão uma série informações sobre o status dos

serviços. O caminho */health*, por exemplo, retorna campos úteis como estado dos serviços de configuração, descoberta de serviços, e disjuntor além dos serviços disponíveis. */metrics* retorna campos como consumo de memória, tempo de execução, número de classes, dados do hospedeiro, como número de CPUs e, especialmente, métricas referentes ao número de requisições.

Há ainda, no Spring Cloud uma interface visual Web para monitoração do status dos serviços que estão registrados no descobridor de serviços *Eureka*.

Outra interface gráfica provida pelo aparato do Spring Cloud *framework* é o visualizador Web do status dos disjuntores.

O *framework* KumuluzEE não oferta nenhum método de monitoramento dos serviços, nem visual e nem em algum formato textual.

QP9 – O *framework* trata de algum modo a segurança e autenticidade na comunicação?

O *framework* Spring Cloud oferta anotações como *@EnableAuthorizationServer* e *@EnableWebSecurity* para registrar serviços autenticados e suas permissões de acesso. É preciso que haja um serviço dedicado a isso para atualização de *tokens* de segurança.

O *framework* KumuluzEE não provê nenhum método para garantia da segurança entre as requisições. Uma implementação rudimentar dele poderia ser feita pela substituição de rotas *GET* para *POST* (e não via cabeçalho HTTP) e que seja enviado um *token* e comparado via variáveis de ambiente. Determinada implementação alivia a complexidade de um serviço dedicado à segurança.

QP10 – O *framework* facilita a criação de um serviço de configurações?

O *framework* Spring Cloud possui um serviço de configurações, que inclusive possui (pode possuir) chave de segurança para seu acesso, naturalmente informada via variáveis de ambiente para fins de parametrização e segurança para publicação do código-fonte.

O *framework* KumuluzEE dificulta a criação de um serviço de configurações, uma vez que suas configurações são informadas via parâmetro de execução e não lidas pelo método principal (que não existe no KumuluzEE).

5.1.8. Implantação

QP11 – O *framework* facilita a implantação dos serviços?

O KumuluzEE inicializa seu próprio servidor HTTP e implanta os serviços com base nas anotações no código, de modo que não é preciso criar uma classe com um método público

estático, sem retorno e com nome ‘main’ e que leva como parâmetro um vetor do tipo String, mas executa por meio do caminho de classes (*classpath*) das dependências, e não a partir da construção de um executável JAR.

Determinada especificidade é, a bem da verdade, interessante para facilitar a refatoração de código pois há um único modo de executar qualquer que seja a aplicação feita sob a égide do *framework* KumuluzEE. Contudo, em um cenário onde não se planeje a compilação e execução via conteinerização, a ausência de um único arquivo pode ser causa de dificuldades ao programador, especialmente tratando-se de vários serviços (que podem ter diferentes versões e portanto acarretar erros se a opção por compartilhamento das dependências em uma única pasta for adotada), dado que todas as dependências deveriam ser carregadas junto dos arquivos compilados (.class).

Embora o *framework* Spring opere de maneira análoga, seu modo de funcionamento requer um método público estático de nome ‘main’ sem retorno e que tome um vetor do tipo String como parâmetro, já que o produto de sua compilação (arquivo JAR) é executável. Uma chamada para um método de assinatura análoga ao método ‘main’ presente na classe *SpringApplication* inicializa os serviços carregando os argumentos passados ao executável (java -jar <service-name>.jar arg_1 arg_2 ... arg_n).

No KumuluzEE não há nenhuma importação para dependências dentro do código-fonte, mas sua execução é prejudicada se os arquivos de dependência não estiverem na pasta correta, já no Spring algumas dependências não utilizadas precisam ser removidas, já que o Spring tenta automatizar alguns processos, como a conexão ao banco de dados, a partir da simples presença da dependência do JPA (do inglês *Java Persistence API*).

Objetivamente falando, não há nenhum mecanismo, em nenhum dos *frameworks*, que facilite a implantação dos serviços: ambos precisam ser compilados pelo Maven (no caso do Spring é possível que seja feito pelo Gradle) e executados a seu modo. Conflitos de porta e de variáveis de ambiente precisam ser resolvidos pelos programadores com um mecanismo de virtualização, conteinerização, adoção de computadores físicos independentes ou organização das portas (determinada organização faria necessitar que fossem refletidas na aplicação). No Spring as configurações podem ser feitas no método principal, mas comumente são feitas via arquivo de configuração. No KumuluzEE, na ausência de método principal, as configurações são parametrizadas na execução.

Contudo, algumas dessas sutilezas, como a utilização de um arquivo JAR e existência de método principal, tornam mais fácil a implantação. Por exemplo optando-se em utilizar o Elastic Beanstalk da AWS, cuja implantação se dá pelo upload do arquivo JAR ou WAR, seria mais fácil para uma aplicação feita sob o *framework* Spring. Faça-se menção que, em razão de dificuldades como essa, os serviços têm passado a oferecer a opção de provisionamento de

serviços também via imagem Docker (Azure[7], AWS[8], Google Cloud Platform[9]).

Destaque-se aqui a parcela de importância de um mecanismo de gerenciamento dos serviços, como o é o Docker Compose, que facilita muito o gerenciamento de várias instâncias de um mesmo serviço.

5.2. Aspectos Não-Funcionais

A Tabela 5.2 apresenta os resultados às questões de pesquisas não funcionais. '✓' representa afirmativa à questão de pesquisa arrolada, do contrário '—'. Múltiplos '✓' quantificam o nível de adequação à questão de pesquisa. Por sua vez, ✗é utilizado para especificar que determinado critério de determinada questão de pesquisa não foi satisfeito, ✪ do contrário.

5.2.1. Adoção

QP1 – Quantos projetos no GitHub utilizam o *framework*?

É importante explicar o motivo da fatoração com a qual apresentou-se os resultados da **QP1**: o operador à direita apresenta o número de repositórios que foram encontrados utilizando o *framework* em questão, conquanto que o operador à esquerda apresenta o número de repositórios encontrados mas que são de uso interno ao *framework*.

O repositório encontrado que utiliza o KumuluzEE é: alexandreafon/algaworks-curso-jsf-primefaces-essencial-kumuluzee.

Para o Spring Cloud encontrou-se 21 repositórios de uso internos dos quais um está vinculado à companhia Pivotal Cloud Foundry (pivotal-cf), outro em Spring Cloud Incubator (spring-cloud-incubator), outro em Spring Petclinic community (spring-petclinic), além dos 18 restantes na própria organização Spring Cloud.

A coleta dos dados foi feita no dia 13/10/2017.

5.2.2. Documentação

QP2 – A documentação do *framework* contempla todas as características da arquitetura a que oferece suporte?

O Spring *framework* é completo em sua documentação e nenhum ponto deixa a ser coberto na documentação. Há, inclusive, um arquivo de documentação diferente para cada versão do *framework*.

[7] https://azure.microsoft.com/pt-br/services/service-fabric/
[8] https://aws.amazon.com/pt/elasticbeanstalk/
[9] https://cloud.google.com/container-builder/

Crit.	*Questão de Pesquisa*	*KumuluzEE*	*Spring Cloud*
Adoção	**QP1** *Projetos no GitHub*	1 + 14	71 + 21
Documentação	**QP2** *Contempla todas as características da arquitetura a que oferece suporte?*	✓	✓
	QP3 *É traduzida para português?*	—	—
	QP4 *É traduzida em quantos idiomas?*	0	0
	QP5 *Tem tutoriais de implementação básica?*	✓	✓
	QP6 *Possui tutoriais de implementação básica em todas suas características?*	✓	✓
	QP7 *Contempla tópicos avançados?*	—	✓
	QP8 *Número de páginas*	—	298
	QP9 *Existe um manual para as componentes?*	—	—
	QP10 *Existem exemplos abordando cada uma das características da arquitetura?*	✓	✓
	QP11 *Os exemplos disponibilizados são didáticos e/ou ilustrados?*	—	—
Facilidade de Uso	**QP12** *Não interfere no modo de desenvolvimento da linguagem?*	✗✓✓✓	✓✗✗✗
	QP13 *Opta por configuração, convenção ou parametrização?*	Parametrização	Configuração
	QP14 *Utiliza injeção de dependências e IoC?*	✓	✓✓
	QP15 *Faz o uso da API de reflexão do Java?*	✓	✓✓
	QP16 *Quanto conhecimento prévio é desobrigado para utilizá-lo?*	✗✗✗✓✓✓	✗✗✓✓✓✓
	QP17 *Restringe o acesso à componentes internas?*	✓	✓
	QP18 *Interfaces de abstração providas?*	✓✓✓✓✗✗✗	✗✓✓✓✓✓✓
	QP19 *É natural a implementação?*	✗✗✗✓✓✓	✓✓✗✗✗✗
Flexibil.	**QP20** *É possível integrar outras tecnologias?*	✓	✓
	QP21 *É fácil integrar outras tecnologias?*	✓	—

Tabela 5.2. Respostas às Questões de Pesquisa Não-Funcionais.

O KumuluzEE oferece suporte a menos funcionalidades e mesmo as mais recém-implementadas não deixam de estar documentadas. Diferente do Spring *framework*,

há somente uma documentação, independente de versão do *framework* e somente há a documentação no arquivo README.md de cada repositório no GitHub.

QP3 – A documentação do *framework* é traduzida para português?

Toda a documentação está escrita na língua inglesa, mesmo o KumuluzEE tendo sido criado por um esloveno.

QP4 – A documentação do *framework* é traduzida em quantos idiomas?

Nenhum *framework* tem sua documentação traduzida.

QP5 – O *framework* possui tutoriais de implementação básica?

O Spring *framework* possui tutoriais básicos na documentação apresentada no site para cada funcionalidade que disponibiliza, contudo não torna claro todas as possibilidades de seu usufruto. Na documentação presente no GitHub e no arquivo de documentação completo os trechos de código que exemplificam os parágrafos explicativos não chegam a compor um exemplo.

No KumuluzEE há um artigo publicado que explica o passo a passo da reprodução de seu exemplo mais simples, bem como há em seu site. Trata-se de um exemplo funcional. Há também no artigo a menção dos passos necessários para ampliar o exemplo exposto e em seu repositório oficial este outro exemplo mais complexo completamente implementado. Em seu site há dois exemplos completos e cada funcionalidade é exemplificada no GitHub com um exemplo funcional.

QP6 – O *framework* possui tutoriais de implementação básica em todas suas características?

Nenhuma característica deixa de ser documentada em nenhum dos *frameworks*

QP7 – A documentação contempla tópicos avançados?

No Spring Cloud a documentação é sólida nos tópicos avançados. Cada característica do *framework* é detalhada em minucias. Tais detalhes somente são encontrados na documentação completa de cada versão (o arquivo de documentação completo) que não é a mais visível das documentações e contempla seu propósito, ou seja, para uso avançado.

No KumuluzEE tópicos avançados não são documentados. Embora os tutoriais tentem elasticizar-se o quanto podem, não se pode dizer que os tópicos avançados são contemplados – como por exemplo, os diversos formatos de comunicação.

QP8 – Quantas páginas de documentação o *framework* possui?

Como há documentação espalhada em vários lugares no Spring (site, GitHub e completa) considerou-se somente a documentação completa. Sendo a documentação completa um arquivo HTML, simulou-se a impressão em papel A4 e contabilizou-se o número de páginas.

No Spring *framework*, considerando-se somente a documentação referente à última versão do *framework*, contabilizou-se 298 páginas A4 de documentação.

No KumuluzEE não há uma documentação completa.

QP9 – Existe um manual para os métodos e classes do *framework*?

Não foi encontrado um manual como a documentação da Oracle. No Spring *framework* há uma documentação para cada versão do *framework* e seu nível de detalhes em cada tópico é volumoso e a profundidade em cada tópico é muito fiel ao seu propósito. Há também, no Spring uma documentação em cada site

QP10 – Existem exemplos disponibilizados abordando cada uma das características da arquitetura?

Os exemplos (avançados) disponibilizados em ambos os *frameworks* abordam todas as características da arquitetura.

QP11 – Os exemplos disponibilizados são didáticos e/ou ilustrados?

Com relação aos sistemas de exemplo do KumuluzEE, até mesmo os exemplos mais complexos não traziam boa separação de classes em pacotes (de fato, todas as classes do mesmo serviço estavam no mesmo pacote), tampouco trazia separado o banco de dados um para cada microsserviço. Ainda no rol de debilidades, todas as portas dos contêineres eram expostas, comprometendo a segurança, e fazendo as requisições via rede do hospedeiro e não rede interna ao Docker. Aponta-se ainda a não-presença do docker-compose que automatiza a implantação no exemplo.

Não se sabe se o intuito do sistema de exemplo do *framework* KumuluzEE é torná-lo simples com essas decisões de projeto, mas ocorre que quem quer que faça menos reflexões que as aqui expostas sobre a arquitetura corre o risco de tomar o exemplo como um cenário ideal de microsserviços, o que não é verdade.

O *framework* Spring apresentou um exemplo menor também sucinto, mas em um exemplo mais complexo (que frise-se não foi desenvolvido pela Pivotal, mas que qualquer usuário que busque por exemplos facilmente o encontra, assim como este autor) trouxe o docker-compose inclusive com um arquivo para o desenvolvimento, onde todas as portas dos contêineres são liberadas, e outro para o ambiente de produção que toma esse cuidado em relação à segurança da aplicação e de suas informações nela contida.

5.2.3. Facilidade de Uso

QP12 – O *framework* não interfere no modo como se desenvolve nesta linguagem?

Anotações em métodos, atributos e classes e injeção de dependências não são considerados uma interferência no modo de desenvolvimento (por serem funcionalidades e conceitos mais antigos).

No mais, entretanto, o Spring *framework* pode-se dizer interferir no modo de desenvolvimento por: 1) tentar automatizar a criação da entidade de persistência unicamente a partir da existência de uma dependência para o JPA; 2) ter sua própria forma de configuração da base de dados; 3) injetar dependências a partir de interfaces Java.

À exceção do modo de implantação diversificada via *classpath* no *framework* KumuluzEE, não há interferências no modo de desenvolvimento, justamente pela opção por ferramentas ao invés de ferramentas próprias e pela ausência de algumas funcionalidades. A configuração do *EntityManager* pelo *EntityManagerFactory* para o parametrização via variáveis de ambiente não é uma interferência pois sua utilização continua sendo via configuração de uma unidade de persistência no formato XML.

QP13 – O *framework* opta por configuração, convenção ou parametrização?

O funcionamento do *framework* KumuluzEE se dá por meio da parametrização das opções desejadas na execução via *classpath*. Variáveis de ambiente relacionadas ao banco de dados podem ser parametrizadas via variáveis de ambiente e configuração do *EntityManager* em tempo de execução por meio do *EntityManagerFactory* tomando um *Map<String, String>* como parâmetro para descrever as personalizações.

O *framework* Spring Cloud é personalizado por meio de arquivos de configuração no formato YAML e ainda configuração via chamadas de método no método principal. Uma classe de configuração anotada também é responsável por automatizações da configuração do mapeamento objeto-relacional. Determinada classe de configuração é automaticamente internamente construída e executada. A classe de configuração é anotada com o caminho de um arquivo de configuração para senhas e demais dados para conexão ao banco de dados (e que admite informação de variáveis de ambiente), e ainda é anotada com o caminho do pacote no qual estão as entidades além de outra anotação para o caminho do pacote no qual encontram-se as classes de abstração para acesso às entidades.

QP14 – O *framework* utiliza injeção de dependências e inversão de controle?

Como já mencionado o *framework* Spring Cloud utiliza injeção de dependências não somente nas requisições, mas também no gerenciamento dos dados, fruto do plano de fundo oriundo

do Spring *framework*. Para o funcionamento da injeção de dependências em classes anotadas como *@Repository* ou *@Service* não é preciso explicitar método construtor padrão e a classe pode, além disso, ser uma interface e todos os métodos abstratos.

O KumuluzEE satisfaz o critério sem qualquer diferenciais.

QP15 – O *framework* faz o uso da API de reflexão do Java?

O Spring Cloud *framework* faz reflexão em interfaces, e então obtém o tipo de retorno do método e o constrói. Em decorrência da reflexão em interfaces, mais código repetido pode ser enxugado. No KumuluzEE para a reflexão ser feita é preciso informar a classe a ser construída quando no retorno de requisições a outros serviços.

QP16 – Quanto conhecimento prévio do *framework* é desobrigado para sua utilização?

Em ambos os *frameworks* é necessário conhecimento prévio de injeção de dependências e anotações. Abstraindo os detalhes de implantação, dificuldades iniciais de configuração e ainda questões ligadas aos dados e às interfaces gráficas, atendo-se isoladamente aos *frameworks*, algumas características da arquitetura funcionam de maneira transparente, como é o balanceador de cargas em ambos os *frameworks* e o disjuntor no Spring Cloud *framework*. Para a comunicação, em ambos os *frameworks*, os padrões definidos satisfazem minimamente o problema, e somente uma implementação avançada para sistemas mais escaláveis requer mais conhecimento nas formas de comunicação.

No Spring Cloud, quando há algum erro, é gerada uma mensagem de erro mais resumida ao requisitante, de modo a não deixá-lo sem resposta, compondo um desenho primitivo de disjuntor, que também é transparente ao programador novato.

Para o descobridor de serviços de ambos os *frameworks* (considerando a abstração do KumuluzEE para o ZooKeeper) não importa saber se os serviços registram-se ou se deixam-se registrar (faça-se claro que uma leitura atenta às anotações deixa isto claro, mas que o programador que tão somente e desatentamente siga os exemplos disponibilizados visualiza a descoberta de serviços de uma maneira também transparente).

Para não deixar passar desapercebida uma visão mais categórica, por alguma razão a implementação do descobridor de serviços do Spring Cloud requer que o serviço de descoberta de serviços explicitamente diga que não deve ser registrado em si próprio.

QP17 – O *framework* restringe o acesso a componentes internas?

Enquanto que no KumuluzEE toda interação é feita a partir das anotações da própria JAX-RS (não há importações para nenhuma outra biblioteca), o Spring Cloud possui uma classe

para inicialização e o restante anotações que são herança do NetFlix e do Spring *framework* (nenhuma dependência no Spring é direta para repositórios da NetFlix).

Faça-se claro que as novas implementações, como o disjuntor, no KumuluzEE, também passaram a importar normalmente determinadas interfaces.

Em resumo, em ambos os *frameworks* não há possibilidades de que os usuários utilizem indevidamente métodos, classes ou atributos de propósito internos ao *framework*.

QP18 – Quantas interfaces de abstração são providas pelo *framework*?

O Spring *framework*, além das abstrações dos aspectos funcionais da arquitetura, abstrai a tradução de (*marshalling*) e para (*unmarshalling*) um formato de representação externa dos dados. Abstrai a comunicação HTTP. Abstrai a reflexão para requisições. Abstrai o escaneamento das entidades. Abstrai a construção das classes de acesso aos dados. Abstrai a criação de rotas mas não o ordenamento de prioridade.

Além dos aspectos funcionais, o KumuluzEE *framework* abstrai a representação externa dos dados, a criação de rotas, o ordenamento de prioridade das rotas e a comunicação HTTP. Não há abstrações relacionada aos dados e nem a abstração de reflexão.

QP19 – É natural a implementação utilizando este *framework*?

Com relação à **experiência inicial** com os *frameworks*, no KumuluzEE uma anti-naturalidade é o seu modo diverso de execução via caminho de classes (*classpath*).

É importante dizer que o fato de o KumuluzEE não ser executado via método principal faz com que não seja possível executá-lo por meio de IDE's tampouco depurar o código por meio deles. Evidentemente, tratando-se de microsserviços, como são muitos os serviços participantes de uma requisição, cada um com variáveis de ambiente específicas e passíveis de conflito, faz com que de fato executar todos os serviços por meio de uma IDE não seja exatamente um caminho viável. Entretanto, existem cenários em que deseja-se avaliar somente um dos serviços executando todas as instâncias com o gerenciador contêineres ou máquinas virtuais à exceção da que se deseja depurar, o que somente seria possível no Spring Cloud. Determinado detalhe, embora sutil, deixa o usuário "cego" na execução da aplicação, e pode ser um ponto bastante negativo para o contexto de determinada empresa.

Automações com docker, docker-compose e shell script são essenciais para tornar o ambiente de construção e testes da aplicação mais amigáveis, contudo em ambos os *frameworks* perde-se muito tempo com a implantação dos serviços, ainda que os mecanismos de automação sejam capazes de remover e implantar apenas o microsserviço que se deseja reparar. Em razão do extenso plano de fundo do *framework* Spring, nele o tempo de compilação e implantação é maior que o KumuluzEE. Em números, o tempo de compilação pelo Maven no KumuluzEE é 11.261 segundos, a construção dos contêineres 4.846 segundos, e 31.804 segundos para

a implantação do sistema. No Spring Cloud o tempo de compilação é 22.340 segundos, 7.791 segundos a construção e 156.655 segundos a implantação. Constata-se lentidão de 11.079 segundos, 2.94 segundos, 124.851 segundos na compilação, construção e implantação, respectivamente no Spring Cloud em vista do KumuluzEE. É importante frisar que no processo de implementação, muitas vezes esse tempo é gasto e ele pode ser amenizado com a recompilação tão somente do microsserviço que está a se desenvolver.

Com relação às **dependências**, existem algumas diferenças entre o modo de construção do arquivo POM para satisfazer a compilação de cada *framework*, de modo que nisto também perde-se algum tempo. Para o programador iniciante no Spring Cloud verifica-se que podem haver dúvidas com relação às dependências e respectivas versões a serem inseridas no arquivo POM, isto porque as versões são nominadas (ao invés de números) e várias versões de teste são disponibilizadas nos repositórios do Maven e a última versão de testes sempre está em evidência para os iniciantes.

No KumuluzEE as dependências dificilmente são encontradas pelo *plugin* do Maven no Netbeans, além do que, embora haja um gerador POM, não é explícito que são necessárias três dependências para sua mínima execução (Jetty, CDI e JAX-RS). Tais dependências não são importadas no código-fonte. O Spring Cloud, ao contrário, é mais fácil de serem encontradas suas dependências, no entanto nem sempre as versões ideais são encontradas, pois existem classes de nomes iguais mas funções diferentes.

Portanto, com relação à facilidade no uso no que concerne a criação da aplicação, o usuário que tenta criar por si próprio o arquivo POM, incluindo as dependências necessárias ou mesmo com o gerador POM, dificilmente obtém êxito se não conhecer bem as funcionalidades que deseja, de modo que o caminho mais seguro é a opção pela modelagem de um projeto de exemplo. Do contrário, ambos os *frameworks* são menos fáceis. No quesito criação do arquivo POM, o Spring *framework* requer o uso de parâmetros específicos na árvore envolvida pela *tag* *<parent></parent>*, enquanto que essa exigência não feita no Kumuluz.

Sobre as **configurações**, no Spring Cloud, todas as configurações concentram-se (ou podem concentrar-se) em arquivos de configuração. Embora determinada prática não seja estranha ao programador que utilize configurações no formato XML para gerenciamento de conexões à Banco de Dados e gerenciamento de Servlets, o Spring adiciona mais um elemento que de configuração YAML, que poderia ser unificado com as configurações que já são definidas nas classes via anotações. Há até mesmo classes de configuração, sem método ou atributo qualquer e em que, além de algumas anotações já para configurações, anota-se o caminho para o arquivo com as configurações de fato.

No que concerne o **processo de desenvolvimento**, o Kumuluz oferta mensagens de erro mais claras. Por exemplo, há a necessidade de uma pasta que seja criada no projeto: além de constar nos tutoriais, a precisa mensagem de erro *"No 'webapp' directory found in the projects resources folder"* é exibida. De outro lado, o Spring *framework* quando não consegue

conectar-se ao banco de dados ou injetar uma dependência, lança uma exceção e não inicializa o serviço. A observação dos erros é difícil em cenários complexos com vários registros (*logs*). Quando ocorre o mesmo problema, no KumuluzEE, a aplicação ainda assim é inicializada, retorna uma mensagem de 'serviço indisponível' quando requisitado e a mensagem de erro exibida no registro (*log*) é "*... is not proxyable because it has no no-args constructor ...*".

Quando ocorre uma exceção o Spring *framework* retorna uma mensagem no formato JSON apresentando detalhes sucintos sobre o problema. Isto é fundamental para o caso em que uma equipe está desenvolvendo um microsserviço que está a interagir com um ambiente fiel ao de produção e que não possui acesso ao *log* do sistema. Alguns programadores desatentos podem tentar rastrear e consertar o erro pela mensagem sucinta e não atentarem-se à mensagem completa, perdendo tempo.

Sobre o **gerenciamento dos dados**, um modo diverso de funcionamento ocorre no Spring quando a simples presença da dependência do JPA faz o Spring tentar conexão ao banco de dados e, pior, falhar. Conclua-se desde logo que, por mais paradoxal que possa parecer, determinada situação é mais penosa para o iniciante, fazendo destacar-se disparidade entre as curvas de aprendizagem dos *frameworks*. Frise-se que não há ganho de tempo neste proceder mencionado, mas o há nas demais abstrações de POJOs mencionadas.

Ainda com relação aos dados, no Spring, as entidades anotadas com *@Entity* são dispostas em um mesmo pacote sobre o qual seu caminho é escrito na anotação de configuração da classe de configuração e a inferência das entidades presentes na aplicação é feita. Uma outra anotação aponta para o pacote que contém os gerenciadores das entidades e faz sua instanciação na inicialização, sem que o programador precise fazê-la.

À contra-mão, para o bem ou para o mal, de caso pensado ou não, KumuluzEE em nada interfere no gerenciamento dos dados.

Registre-se ainda que algumas outras perdas de tempo podem ser enfrentadas por aqueles que desconhecem o modo de funcionamento do gerenciamento dos dados no Spring, como foi o caso durante a execução do método: quando não se explicita auto reconexão à base de dados, após determinado tempo a conexão é interrompida e não mais torna a executar as consultas no banco de dados que se estiver utilizando.

Felizmente há fartura de respostas à questões recorrentes como essa, para o caso de um *framework* popular e histórico. Faça-se menção ainda à facilidade com que se cria uma base de dados emulada para testes no Spring *framework*.

Sobre a **injeção de dependências**, no *framework* Spring Cloud o modo de gerenciamento dos dados e clientes *@FeignClient* é feito pela anotação ou implementação em interfaces Java que são tranquilamente instanciadas pela injeção de dependências *@Autowired* e quaisquer métodos que nela estejam declarados são implementados pelo *framework*. Mais ainda, os métodos *findByName* e congêneres que forem declarados na interface de gerenciamento dos dados são implementados pelo *framework* e fazem coincidir o nome do

sufixo após *findBy* com o nome do atributo da classe gerenciada e justamente resgata no banco de dados a entidade ou lista de entidades que coincidirem com o parâmetro passado, a depender do tipo de retorno escrito na assinatura do método.

Há de se perpassar pelo ponto de que avaliou-se a dificuldade em criar a mesma abstração que os clientes *@FeignClient* do Spring Cloud, a fim de avaliar o mérito do *framework* Spring Cloud e também verificar se os próprios programadores poderiam facilmente implementar determinada abstração para seu próprio uso. Constatou-se que, instanciar programaticamente as interfaces Java é uma tarefa complexa ao ponto de permitir-se a duplicação de código ou a criação de abstrações menos robustas, de modo a fazer pontuar o Spring Cloud *framework* nesse critério. É importante recordar que o modo de gerenciamento do *framework* Spring Cloud para os dados também permite a injeção de dependências a partir de uma interface Java.

Embora o código Spring seja mais limpo, enfatize-se que a curva de aprendizagem para compreender o segundo modo da criação de requisições (clientes *@FeignClient*) é fortemente alongado no eixo das abscissas, face, especialmente, ao alto volume de anotações que levam as classes, métodos e campos das classes em Spring. Uma vez cumprida a curva, refatora-se código em Spring de maneira mais veloz.

Em ambos os *frameworks*, encontra-se muito presente anotações em métodos, classes e atributos. Há ainda muito frequente a atribuição de instâncias às variáveis por meio de injeção de dependências. Por tratarem-se de, respectivamente, uma funcionalidade e um conceito já mais antigos, não se pode julgar seus usos anti-naturais.

Com relação à **facilidade na comunicação**, o *framework* KumuluzEE ganha muitos pontos por fazer-se simples no quesito de facilidade em se estabelecer conexões e criar rotas. O Spring Cloud, ao contrário, possui duas formas para fazer as requisições e requer menos linhas de código para tanto, contudo possui uma curva de aprendizado maior.

Foram produzidas 7969 linhas de código no Spring Cloud e 7085 linhas de código no KumuluzEE. Apesar da maior abstração ofertada pelo Spring Cloud, existem arquivos de configuração que corroboram para o aumento do número de linhas de código, se comparado ao KumuluzEE, e ainda uma separação em arquivos diferente do KumuluzEE (e que não foi possível replicá-la) que faz elevar o número de importações; e que é parte do número observado.

5.2.4. Flexibilidade

QP20 – É possível integrar outras tecnologias ao *framework*?

Há de se destacar que, embora o exemplo mais complexo do Spring Cloud apresentava com a interface Web em HTML, CSS e JavaScript comuns, não é fácil servir HTML comum. Ao

invés, encoraja-se o uso do motor ThymeLeaf. O KumuluzEE encoraja uma versão própria do PrimeFaces, contudo o serviço de HTML comum é trivial.

QP21 – É fácil integrar outras tecnologias ao *framework*?

Também recorda-se a facilidade com o qual o *framework* Kumuluz tem empenhado em prover abstração e integração com outras ferramentas enquanto demora implementar suas próprias funcionalidades. Por não interferir muito na forma como se programa na linguagem, o KumuluzEE consegue integrar melhor a outras ferramentas que não as indicadas. Exemplo disso é o operar com muita naturalidade que o Kumuluz faz no que se refere ao balanceamento de cargas, mérito da ferramenta ZooKeeper.

No Spring Cloud, como há um aparato para abstração no gerenciamento dos dados, uma camada de código (*plugin*) precisa haver disponibilizada pelo *framework* para seu funcionamento.

5.3. Discussões sobre a Arquitetura

Durante o processo de desenvolvimento, ou seja, da execução do método, várias questões ligadas à arquitetura emergiram, precisaram ser refletidas e cabem ser descritas. Pode-se classificar determinadas questões quanto aos **dados**, aos **serviços** e às **equipes** e são expostas segundo esse ordenamento de maneira cumulativa face à propagação dos erros que se cometem nas camadas mais internas (dados) para as camadas mais externas (equipes).

Com relação aos **dados** da aplicação, e sendo sua particionabilidade a premissa mais crucial para a adoção da arquitetura, há de se concentrar muito esforço em seu fluxo pelo sistema: experimentou-se que a aplicação realmente precisa enrobustecer-se com a separação entre POJOs e BOs (do Inglês, *Business Object*), uma vez que uma série de processos, inevitavelmente, envolvem a lida com objetos que devem ser refletidos, e portanto requerem um método construtor – ao menos protegido (*protected*) – padrão, ou seja, sem parâmetros. Ademais, o mapeamento objeto-relacional opera pelo atributos de uma classe, mas sobretudo pelos métodos *get* em todos eles. Além disso, o mapeamento objeto-JSON–objeto-Java requer a presença dos métodos *set* para todos os campos presentes no objeto JSON.

Uma das grandes discussões é com relação ao modo de busca (*fetch*) dos dados de maneira tardia (*lazy*) ou imediata (*eager*) junto à base de dados. É também para a arquitetura de microsserviços uma grande discussão a redundância dos dados, que analogamente imporá ao sistema uma busca tardia para os dados que estiverem espalhados ao longo do sistema. Naturalmente o impacto da manutenção da consistência da redundância também custa ao sistema. Ocorre que somente determinados setores da aplicação, que constata-se não sofrerem constantes alterações, devem ser espalhados ao sistema, em benefício da velocidade pela independência do serviço.

Com relação aos **serviços**, e sendo sua independência o ponto central pelo qual se confere velocidade à aplicação cujos dados estejam espalhados ao longo de si, deve-se planejá-los em visões *bottom-up*, *top-down* e em sequência no tempo: ainda que todos os serviços devam possuir POJOs para os dados e que muitos serviços tenham interseção no conjunto de POJOs, não é verdade que eles sejam iguais, ou seja, cada serviço registrará valores e estruturas diferentes em suas bases de dados, de modo que os POJOs serão diferentes entre os serviços e não poderão estar em um mesmo módulo e que os serviços usem-no como dependência. Ocorre que dados serão trocados entre as aplicações e, por mais que determinados dados não sejam persistidos e nem mesmo preenchidos pelo requisitante da mensagem, o serializador enviará determinado campo com valor nulo e o deserializador requererá um método *set* para lidar com determinado campo, mesmo que seu corpo seja vazio.

Durante o processo de desenvolvimento, parte considerável do tempo deve ser dedicada aos dados. Assim, um planejamento profundo sobre o funcionamento dos serviços e suas interações devem ser rastreados no que se refere a seus dados, para que seja identificado (i) quais campos deverão ser persistidos em cada serviço, (ii) quais campos deverão ser obtidos de outrem e (iii) quais contarão com métodos de corpo vazio, além de (iv) mecanismos para proteção dos dados pela interface com BOs para os POJOs.

A decisão por dados que serão buscados (*fetch*) de maneira tardia (*lazy*) devem ser feitas e implementadas no API *Gateway*, e não pelo serviço, com o fim de reduzir o acoplamento entre os serviços (uma vez que não haverá comunicação entre eles) e permitir o reúso para um contexto diferente (pois caso houvesse comunicação direta, seu uso precisaria ser refatorado ou removido, e mais ainda, dificultando escrita de um contrato, ou regra de negócio, diferente do anterior ao reúso).

Mais ainda, sobre o planejamento, uma das grandes problemáticas é que um protocolo próprio deve ser utilizado para a comunicação interna dos serviços. Nesse protocolo deve-se analisar e estabelecer um limiar bem ajustado no qual seja possível reutilizar os serviços em outras ocasiões e que não seja o protocolo um empecilho a tal reúso. Sobretudo, devem ser planejadas quais interfaces de entrada (objetos POST em rotas) para cada serviço serão providas e quais tipos de dados devem ser persistidos em cada serviço (para que não crie-se dependência entre a regra de negócio e o serviço que pretende-se fazer genérico e, logo, reusável).

Sendo as responsabilidades pequenas e, consequentemente, os serviços em volume notório, não há como executá-los, quer seja para testá-los, quer seja para implantá-los ou manutení-los, sem um aparato de automações que, embora não fáceis de obtê-los, somente são obtidas em decorrência de padronizações. Algumas ferramentas são essenciais para a construção do sistema, tais como o Docker, Docker Compose e Bash.

Além disso, planejamento é essencial para que sejam distinguidos os serviços críticos

dos secundários, de modo a construir o grafo de dependência dos serviços. Neste grafo de dependências é possível visualizar as dependências necessárias para a manutenção do sistema em funcionamento e sem quais não se pode mantê-lo em execução. Para as dependências que identificar-se serem menos críticas para o bom funcionamento da aplicação é possível ofertar respostas padrão para os usuários quando em seu mal funcionamento.

Extraindo o melhor de cada exemplo disponibilizado na documentação dos *frameworks*, entendeu-se por bem organizar o código em pacotes a obedecer que um pacote seja dedicado aos recursos (URL) composto de classes para cada prefixo de caminho do recurso e que eles imediatamente requisitem aos serviços competentes, que por sua vez comunicam-se naturalmente com a base de dados. Os serviços competentes, por sua vez, são organizados em dois pacotes distintos: um para o serviço próprio e outro para fazer interface com os outros serviços do sistema. POJOs e BOs também devem ser separados em pacotes.

Com relação às **equipes** de desenvolvimento, e sendo sua dependência o mais grave empecilho para a construção de um sistema escalável, deve-se fazer conta de uma equipe, não somente experiente mas, sobretudo, perfeitamente alinhada na divisão e conquista dos objetivos do sistema: contando com repositórios segregados, experimentou-se grande labor quando na necessidade de correções e suas respectivas propagações para o sistema, o que, expandindo para um cenário real e grave, excede à imaginação e requer realismo no planejamento, boa interação na execução e adestramento da equipe.

Sobretudo, é importante que haja um comando central, em metáfora, muito bem articulado e determinado deve poder gerir o processo de desenvolvimento e sem o qual a estratégia de divisão e conquista não integra-se na recursão, como recorda (BALALAIE et al., 2016).

Tratando-se de um sistema invariavelmente grande, é necessário que a equipe de desenvolvimento menos experiente tenha acesso a métodos mais seguros e, isto é, faz-se necessário que o sistema seja protegido de violações das restrições da regra de negócio instanciando objetos somente mediante informação de seus identificadores unívocos e impossibilidade de alteração desses valores. Contudo, há ainda grande necessidade de adestramento da equipe de desenvolvimento, que naturalmente pode remover determinadas restrições e comprometer o sistema.

Tomando por exemplo a boa segregação de pacotes, gerar-se-á determinada organização que vem a oferecer uma estruturação lógica capaz de fazer-se compreender pelos programadores, face às necessidades de um ambiente corporativo em termos de recursos humanos. Determinado conhecimento é inerente à estrutura externa mas sobretudo interna dos serviços que deve ser fielmente seguida pelos programadores como no respeito entre a distinção entre POJOs e BOs, como no respeito ao modo de busca dos dados tardia, na garantia de serviços reusáveis, no seguimento aos padrões, espera por falhas de outrem e busca pelo sempre retorno de resposta ao cliente final ou intermediário, e é, determinado

conhecimento quando fielmente seguido, capaz de gerar independência entre as equipes.

Ainda que a equipe de desenvolvimento seja de um só, e a equipe de implementação deste trabalho o foi, o sistema é formado de um conjunto de módulos muito maior que um só serviço, de modo que pôde-se inclusive experimentar que decisões transversais e correções importantes são propagadas para todo o sistema e sobretudo precisam ser feitas por uma equipe bem adestrada para seguir fielmente à arquitetura de microsserviços, não violar as regras de negócio, aplicar os padrões pré-estabelecidos que possibilitam as automações e possuir determinado grau de independência pelo acolhimento de tarefas específicas e bem desenhadas que possibilitem um produto final escalável.

5.4. Decisões de Projeto

Importa ainda sumarizar que algumas decisões tiveram de ser tomadas no curso da implementação. O é uma a opção pela interface gráfica Web nativa em benefício do balanceamento de carga e em detrimento da heterogeneidade da aplicação. Determinada decisão corrobora à tese sustentada no Capítulo 3 e que trata do aprisionamento tecnológico que um *framework* impõe ao entrelaçar-se à aplicação.

Outra decisão tomada foi a manutenção do módulo de abstrações providas pelo KumuluzEE para o Apache ZooKeeper e que foi mantido como dependências de cada microsserviço, ao invés da inclusão de determinada camada em cada microsserviço. Determinada importância reside na independência das equipes e a decisão foi baseada no entendimento que, quando na criação de uma implementação própria das funções desempenhadas pelo ZooKeeper, a dependência seria simplesmente substituída. Recorde-se que os exemplos do KumuluzEE contavam com um módulo compartilhado para todos os serviços e tenha-se em mente as implicações já discutidas.

No entanto, as classes referentes ao modelo (regra de negócio) foram mantidas separadas e espalhadas ao longo dos serviços conforme a população de métodos requerida para o serviço na qual se instala. Determinada decisão foi tomada à reflexo e coerência da decisão de manter repositórios independentes para cada serviço.

Cada *framework* em seus exemplos demonstrava uma forma diferente de organizar e atribuir nome aos componentes. Ao longo do desenvolvimento refatorou-se os exemplos para coincidirem em termos de organização, conforme o formato que melhor aprouve dentre os disponíveis e, sobretudo, para satisfazerem o melhor possível à arquitetura de microsserviços, conforme sumarizada neste trabalho e desenhada ao longo de seus anos.

Decidiu-se também segregar o API *Gateway* dos microsserviços de gerenciamento dos passageiros e de gerenciamento dos motoristas para evitar o acumulo de funções. Seu propósito também centrou-se ideia de inserir funcionalidade neles por meio da provisão de abstrações na interface REST.

Não utilizou-se a implementação do API *Gateway* para no Spring, a fim de permitir a criação de serviços menos vinculados à regra de negócio.

Por exemplo, para a checagem de notificações criou-se uma rota que toma um passageiro como parâmetro POST e então há a tradução no *Gateway* para o formato que o serviço de notificações toma como argumento, que não compreende a classe *Passenger* mas a classe *CheckNotificationsRequest*. Assim procedendo, não faz-se necessário outorgar ao cliente móvel, impactando em aumento de complexidade no lado do cliente, e nem ao microsserviço de notificações, impactando em torná-lo menos reusável.

5.5. Resumo

Em resumo, este capítulo apresentou como procedeu-se para a aquisição dos resultados obtidos, seguido pela apresentação dos resultados relativos aos aspectos funcionais da arquitetura de microsserviços e então os aspectos não-funcionais. Apresentou-se resultados relativos à arquitetura de microsserviços per se e que surgiram durante o desenvolvimento do sistema seguido das decisões de projeto importantes para a reprodutibilidade do método.

Capítulo 6

Conclusão

Em precisando-se de um sistema escalável e resiliente, por atender muitos usuários, considera-se a arquitetura de microsserviços e verifica-se vasto número de ferramentas que propõe atendê-lo. Dentre as ferramentas há os *frameworks* para microsserviços que buscam satisfazer todas as características da arquitetura. Mesmo na categoria de *frameworks* há vastidão de possibilidades que demandam ser analisadas.

Neste trabalho analisou-se, por razões históricas e de adoção e comparabilidade, os *frameworks* KumuluzEE e Spring Cloud & NetFlix OSS. Ambos os *frameworks* foram comparados por meio do desenvolvimento de um cenário fictício de chamadas de Táxi, sobre os quais verificou-se o suporte de cada *framework* em cada característica da arquitetura de microsserviços (aspectos funcionais) e também questões ligadas à adoção, documentação, facilidade de uso e flexibilidade de cada *framework* (aspectos não-funcionais).

Sobre os aspectos funcionais, verificou-se que o *framework* Spring Cloud oferece suporte satisfatório às características da arquitetura de microsserviços (deixando apenas duas características menos satisfeitas). Grande parte da cobertura do Spring Cloud aos aspectos funcionais deve-se à herança do Spring *framework*, apresentando variadas formas de implementar uma mesma característica da arquitetura de microsserviços. Realça-se que o Spring *framework* (e o Spring Cloud por indução) oferta mecanismos (de complexas implementações) de abstrações avançadas e que resumem a intenção dos programadores em poucas linhas de código. Por outro lado, verificou-se que o *framework* KumuluzEE possui implementações para menos funcionalidades (deixando quatro características, considerando as recente implementações) e interfere menos no código dos programadores, mas tem evoluído no intuito de atendê-las e sem deixar de prover abstrações para ferramentas existentes enquanto demoram implementar suas próprias soluções.

Em relação aos aspectos não-funcionais verificou-se que o *framework* Spring Cloud é mais utilizado e mais documentado em função de sua popularidade e maturidade, convertendo

para si 24 pontos positivos com relação às questões de pesquisa investigadas. Verificou-se que o Spring Cloud apresenta um maior número de abstrações que proveem facilidades a usuários experientes e, em muitas vezes, são obstáculos a serem superados para a introdução de usuários novatos. Para os novatos ambos os *frameworks* não são brilhantes nos exemplos. Por outro lado, o *framework* KumuluzEE é menos documentado, mais claro e conciso nos exemplos básicos, mas sobretudo é fácil tanto para novatos quanto para programadores experientes, que deverão fazer suas próprias abstrações (que talvez não se iguale ao aparato do Spring *framework*) e usar suas próprias, ou de outrem, ferramentas que somente é possível pela baixa interferência no código-fonte por parte do *framework*. O *framework* KumuluzEE contabilizou 22 pontos positivos com relação às questões de pesquisa.

Seguramente pode-se dizer que o *framework* KumuluzEE é mais indicado para os iniciantes em microsserviços do que o *framework* Spring. O KumuluzEE é mais indicado para aplicações pequenas.

O *framework* KumuluzEE sendo menos adotado, menos documentado e menos adequado às características da arquitetura de microsserviços, de maneira alguma indica-se para projetos graves enquanto demora estabelecer-se.

Tendo sido realizado o trabalho perante o tempo de desenvolvimento da ferramenta KumuluzEE, um trabalho futuro é checar novamente as características que o *framework* suporta a fim de reconsiderar se o *framework* continua não sendo indicado para implementações mais robustas.

Existe uma possibilidade de que o KumuluzEE, como sendo um *framework* novo, possa ofertar desempenho superior ao Spring, por aproveitar-se de novas características da linguagem de programação. Seria este um trabalho futuro à este: a comparação de desempenho.

Pontue-se ainda que, tendo-se evoluído ao longo da implementação deste trabalho, não foi possível avaliar a dificuldade enfrentada para a refatoração e questões ligadas à retrocompatibilidade. Seria este outro trabalho futuro.

Outro trabalho ainda é a implementação do método com equipes de desenvolvimento, ponto crucial da arquitetura de microsserviços, a fim de testar os *frameworks* sob esse aspecto.

Neste trabalho também foram elencadas outras categorias de ferramentas para microsserviços, que também esperam ser avaliadas.

Referências

ABBOTT, Martin L; FISHER, Michael T. *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. [S.l.]: Pearson Education, 2009.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, v. 33, n. 3, p. 42–52, maio 2016. ISSN 0740-7459.

BECK, Kent; BEEDLE, Mike; BENNEKUM, Arie van; COCKBURN, Alistair; CUNNINGHAM, Ward; FOWLER, Martin; GRENNING, James; HIGHSMITH, Jim; HUNT, Andrew; JEFFRIES, Ron; KERN, Jon; MARICK, Brian; MARTIN, Robert C.; MELLOR, Steve; SCHWABER, Ken; SUTHERLAND, Jeff; THOMAS, Dave. *Manifesto for Agile Software Development*. 2001. Disponível em: <http://agilemanifesto.org/>.

BRUNNER, S.; BLöCHLINGER, M.; TOFFETTI, G.; SPILLNER, J.; BOHNERT, T. M. Experimental evaluation of the cloud-native application design. In: *Proc. IEEE/ACM 8th Int. Conf. Utility and Cloud Computing (UCC)*. [S.l.: s.n.], 2015. p. 488–493.

CARVALHO, Olavo. *O poder de conhecer*. 2001. Disponível em: <http://www.olavodecarvalho.org/semana/pconhecer.htm>.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim; BLAIR, Gordon. *Sistemas Distribuídos-: Conceitos e Projeto*. [S.l.]: Bookman Editora, 2013.

FLORIO, L.; NITTO, E. D. Gru: An approach to introduce decentralized autonomic behavior in microservices architectures. In: *Proc. IEEE Int. Conf. Autonomic Computing (ICAC)*. [S.l.: s.n.], 2016. p. 357–362.

FOWLER, Martin. *CircuitBreaker*. 2014. Disponível em: <https://martinfowler.com/bliki/CircuitBreaker.html>.

FOWLER, Martin; LEWIS, James. *Microservices*. 2014. Disponível em: <https://martinfowler.com/articles/microservices.html>.

FOX, A.; BREWER, E. A. Harvest, yield, and scalable tolerant systems. In: *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*. [S.l.: s.n.], 1999. p. 174–178.

GADEA, Cristian; TRIFAN, Mircea; IONESCU, Dan; IONESCU, Bogdan. A reference architecture for real-time microservice api consumption. In: *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*. New York, NY, USA: ACM, 2016. (CrossCloud '16), p. 2:1–2:6. ISBN 978-1-4503-4294-0. Disponível em: <http://doi.acm.org/10.1145/2904111.2904115>.

GUO, D.; WANG, W.; ZENG, G.; WEI, Z. Microservices architecture based cloudware deployment platform for service computing. In: *Proc. IEEE Symp. Service-Oriented System Engineering (SOSE)*. [S.l.: s.n.], 2016. p. 358–363.

HAERDER, Theo; REUTER, Andreas. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 15, n. 4, p. 287–317, dez. 1983. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/289.291>.

KOOKARINRAT, P.; TEMTANAPAT, Y. Design and implementation of a decentralized message bus for microservices. In: *Proc. 13th Int. Joint Conf. Computer Science and Software Engineering (JCSSE)*. [S.l.: s.n.], 2016. p. 1–6.

KOZMIRCHUK, A.; KOKOREV, A.; NESTEROV, V.; MIKHAILOVA, E. Postgresql service with backup and recovery for cloud foundry. In: *Proc. Social Media and Web (ISMW FRUCT) 2016 Int. FRUCT Conf. Intelligence*. [S.l.: s.n.], 2016. p. 1–6.

MORRISON, Alan. *Agile coding in enterprise IT: Code small and local*. 2015. Disponível em: <http://usblogs.pwc.com/emerging-technology/agile-coding-in-enterprise-it-code-small-and-local/>.

RICHARDSON, Chris. *Microservices: Decomposição de Aplicações para Implantação e Escalabilidade*. 2014. Disponível em: <https://www.infoq.com/br/articles/microservices-intro>.

RICHARDSON, Chris; SMITH, Floyd. *Microservices: From Design to Deployment*. [S.l.]: NGINX, 2016.

SCHULTE, Roy; NATIS, Yefim. Service oriented architectures. *URL http://www.gartner.com/id=302868*, 1996.

UEDA, T.; NAKAIKE, T.; OHARA, M. Workload characterization for microservices. In: *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*. [S.l.: s.n.], 2016. p. 1–10.

XU, C.; ZHU, H.; BAYLEY, I.; LIGHTFOOT, D.; GREEN, M.; MARSHALL, P. Caople: A programming language for microservices saas. In: *Proc. IEEE Symp. Service-Oriented System Engineering (SOSE)*. [S.l.: s.n.], 2016. p. 34–43.