



**CENTRO UNIVERSITÁRIO INSTITUTO DE EDUCAÇÃO
SUPERIOR DE BRASÍLIA - IESB
BACHAREL EM CIÊNCIA DA COMPUTAÇÃO**

MATEUS TYMONIUK ALMEIDA

APLICAÇÃO CONSTRUÍDA EM UMA ARQUITETURA DE MICROSERVIÇOS

Orientador: M. Sc. Cristiano Soares de Aguiar

BRASÍLIA - DF
2018

MATEUS TYMONIUK ALMEIDA

TRABALHO DE CONCLUSÃO DE CURSO II

APLICAÇÃO CONSTRUÍDA EM UMA ARQUITETURA DE MICROSERVIÇOS

Estudo apresentado como pré-requisito para obtenção do título de Bacharel em Ciência da Computação pelo Centro Universitário Instituto de Educação Superior de Brasília, como requisito parcial para à aprovação.

Orientador: M. Sc. Cristiano Soares de Aguiar

Brasília – DF

2018

MATEUS TYMONIUK ALMEIDA

APLICAÇÃO CONSTRUÍDA EM UMA ARQUITETURA DE MICROSERVIÇOS

Estudo apresentado como pré-requisito para
obtenção do título de Bacharel em Ciência da
Computação pelo Centro Universitário
Instituto de Educação Superior de Brasília,
como requisito parcial para à aprovação.

Brasília, 23 de maio de 2018.

Banca Examinadora

Orientador

M. Sc. Cristiano Soares de Aguiar

D. Sc. Cicero Roberto Ferreira de Almeida

M. Sc. Flavia Maria Alves Lopes

AGRADECIMENTOS

Dedico este trabalho à minha família e à minha companheira, pela compreensão das noites escrevendo e pelo apoio e acalento, também agradeço ao meu professor pela paciência na orientação acerca da construção do trabalho.

RESUMO

No mundo de hoje, onde os negócios mudam constantemente, as aplicações construídas precisam acompanhar a velocidade destas mudanças para que consigam agregar valor ao negócio do cliente. Aplicações comerciais, são comumente construídas na forma de um monólito, um pedaço de código que contém toda a lógica do negócio. Essa abordagem faz com que a criação e consequente implantação de um projeto seja simples, inicialmente. Porém a medida que a base de código cresce, e são necessárias mudanças nesse monólito, como modificações para corrigir erros, novas funcionalidades ou uma demanda por escalar a aplicação, ela começa a mostrar alguns problemas. Mudanças em uma base de código muito grande se tornam arriscadas de se fazer, pois pode haver acoplamento em algum ponto da aplicação, e é difícil conhecer toda a base de código para saber o que a mudança pode afetar. E normalmente para escalar este tipo de aplicação, é necessário criar um outro servidor que seja uma cópia daquele onde a aplicação está instalada, ou seja, ela escala adicionando mais máquinas, o que é chamado escalonamento horizontal.

Uma saída para estes problemas era a arquitetura orientada a serviços (SOA), que tem como foco a reutilização de componentes e a quebra da aplicação em serviços que colaboram entre si para um propósito maior. Porém esta abordagem também possuía suas complicações, visto que havia ainda uma discussão em como construir uma aplicação concisa em SOA, como quebrar a aplicação em serviços da melhor forma, os protocolos de comunicação utilizados e a granularidade dos serviços, evitando o acoplamento entre suas implementações.

Então surge o termo microsserviços, como uma outra arquitetura a serviços, que possui uma abordagem de dividir os serviços em torno das capacidades do negócio, onde os times são donos dos serviços durante todo seu ciclo de vida. Há uma procura por serviços que sejam pequenos o suficiente para serem reescritos em poucas semanas ou descartados quando necessário. Sua natureza em forma de serviços colaborativos, permite que sejam escalados apenas os serviços que demandam mais processamento, ou que demandam uma alta disponibilidade e não podem ficar fora do ar.

Neste trabalho é feita uma análise sobre uma aplicação que utiliza esta arquitetura, para determinar características chave necessárias para uma aplicação que for utilizar, ou que queira migrar para uma arquitetura de microsserviços.

Palavras-chave: Arquitetura. Serviços. Software. Microsserviços.

ABSTRACT

Today, where business changes constantly, applications need to keep up with the speed of those changes, so they can aggregate value to the client business. Commercial applications, are usually built in a monolith, a piece of code that contains all the business logic. This approach makes the creation and deployment of a Project to be simple, initially. However, when the codebase starts to grow, and changes to this monolith need to be made, like bugfixes, new functionalities or a demand to scale the application, it starts to show some problems. Changes on a large codebase are risky, because it may have a high coupling at some point of the application, and it is hard to know it entirely, to know where exactly it will break. And usually, to scale those kind of applications, it is needed to add another server, containing a copy of the application, that is, it scales by adding more machines to it, what is called horizontal scaling.

One way out of these problems was the service oriented architecture (SOA), that focus on component reusability and to split the application on collaborative services that serve to a bigger purpose. But, this approach do have it's complications, because there is still a discussion on how to build a concise application with SOA, how to break the services in a better way, the communication protocols that were utilized and the granularity of the services, avoiding coupling in the implementation.

It comes then microservices, as another service oriented architecture, that focuses on splitting the services around business capabilities, where teams own the service during their full lifetime. There is a search for services that are small so that they could be rewritten in a few weeks or thrown away when needed. Their nature as collaborative services, allows to scale only the services that need more processing, or cannot have downtime associated.

It was made an analyze about a application that uses this architecture, to determine key characteristics needed to any application that is willing to use or want to migrate to an microservice architecture.

Keywords: Architecture. Service. Software. Microservices.

LISTA DE ILUSTRAÇÕES

Figura 1 - Componentes de uma arquitetura monolítica e de microserviços	17
Figura 2 - Evolução da arquitetura de aplicações	19
Figura 3 - Lei de Conway em ação	27
Figura 4 - Divisão dos times conforme as capacidades de negócio	28
Figura 5 - Gerenciamento de dados, monolítico e microserviços	30
Figura 6 - Pipeline básico de uma build	30
Figura 7 - Como microserviços permitem a adoção de novas tecnologias	31
Figura 8 - Escalando onde é necessário	33
Figura 9 - Uberjar	37
Figura 10 - Divisão dos serviços da aplicação	39
Figura 11 - Subindo o serviço do front-end	41
Figura 12 - Registro do front-end no Consul	42
Figura 13 - Inicializando Hystrix no front-end	42
Figura 14 - Consul com front-end registrado	42
Figura 15 - Subindo o serviço do catálogo	42
Figura 16 - Registro do catálogo no Consul	42
Figura 17 - Subindo o serviço de pedido	43
Figura 18 - Registro do pedido no Consul	43
Figura 19 - Subindo serviço de usuário	43
Figura 20 - Registro de usuário no Consul	43
Figura 21 - Consul com os quatro serviços registrados	43
Figura 22 - Tela inicial da aplicação	44
Figura 23 - Tela de criação de usuário	44
Figura 24 - Tela catálogo	45
Figura 25 - Tela carrinho vazio	45
Figura 26 - Tela usuário criado com sucesso	46
Figura 27 - Tela selecionando um item	46
Figura 28 - Tela carrinho com um item	46
Figura 29 - Tela checkout	46
Figura 30 - Tela de pedidos	47
Figura 31 - Inicializando outro serviço de catálogo	47
Figura 32 - Registro do segundo catálogo no Consul	47
Figura 33 - Dois catálogos registrados no Consul	48
Figura 34 - Serviços de catálogo listados pelo Ribbon	48
Figura 35 - Catálogos finalizados	48
Figura 36 - Recuperação do catálogo pelo Hystrix	49

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
BPEL	Business Process Execution Language
CPU	Central Processing Unit
HTTP	Hypertext Transfer Protocol
LAN	Local Area Network
REST	Representational State Transfer
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SPI	Service Provider Interface
TI	Tecnologia da Informação
URL	Uniform Resource Locator
WAN	Wide Area Network
WAR	Web application ARchive
WS-Choreography	Web Service Choreography
XHTML	eXtensible Hypertext Markup Language

SUMÁRIO

1	INTRODUÇÃO.....	11
1.1	JUSTIFICATIVA.....	12
1.2	PROBLEMA.....	12
1.3	OBJETIVOS GERAIS	14
1.4	OBJETIVOS ESPECÍFICOS	14
2	FUNDAMENTAÇÃO TEÓRICA	15
2.1	SISTEMAS DISTRIBUÍDOS	15
2.2	ARQUITETURA DE SOFTWARE.....	17
2.3	ARQUITETURA SINGLE TIER.....	20
2.4	ARQUITETURA CLIENTE-SERVIDOR.....	21
2.5	ARQUITETURA ORIENTADA À SERVIÇOS (SOA).....	21
2.6	ARQUITETURA DE MICROSERVIÇOS	23
2.6.1	Arquitetura monolítica.....	25
2.6.2	Características e benefícios da arquitetura de microserviços	26
2.6.3	Benefícios da arquitetura de microserviços	31
3	DESENVOLVIMENTO.....	35
3.1	PROJETO	35
3.2	FUNCIONALIDADES DA APLICAÇÃO	35
3.2.1	Criar um novo usuário	35
3.2.2	Catálogo	35
3.2.3	Pedido.....	36
3.3	TECNOLOGIAS UTILIZADAS	36
3.3.1	Java	36
3.3.2	Wildfly Swarm	36
3.3.3	Consul.....	37
3.3.4	ShrinkWrap	38
3.3.5	Hystrix.....	38
3.3.6	Ribbon	38
3.3.7	JSF	39
3.3.8	Hibernate	39
3.4	ARQUITETURA GERAL DA APLICAÇÃO	39
3.5	IMPLEMENTAÇÃO DOS SERVIÇOS	40
3.5.1	Microserviços de catálogo, pedido e usuário.....	40
3.5.2	Microserviço do <i>front-end</i>	41

4	RESULTADOS	41
5	CONCLUSÃO.....	50
	TRABALHOS FUTUROS.....	51
	REFERÊNCIAS.....	52

1 INTRODUÇÃO

A forma de se desenvolver aplicações vem melhorado constantemente durante os anos. As pessoas sempre buscam formas de melhor modelar suas aplicações para que atendam melhor às necessidades do mundo real, aprendendo com o que passou e adotando novas tecnologias, sempre atentos ao que uma nova onda de empresas de TI vêm fazendo para atender às necessidades tanto do cliente quanto de seus desenvolvedores. (NEWMAN, 2015)

Durante anos, usou-se da arquitetura monolítica para o desenvolvimento de sistemas. Até hoje, muitas aplicações são construídas ainda desta forma, pois é uma maneira natural de se pensar uma aplicação, construída como uma peça única, que geralmente consiste de três partes centrais: uma interface com o cliente, um banco de dados e uma camada de lógica de negócios, que é a aplicação do lado do servidor. Aplicações construídas desta forma, recebem requisições HTTP do lado do cliente, processam sua lógica, fazem leituras e alterações no banco de dados, e devolvem as novas telas para o cliente. Construindo desta forma, podemos então separar a lógica em classes e métodos, fazer testes na aplicação e desenvolver um pipeline para garantir que a aplicação será devidamente testada e implantada no servidor. Pode-se também, escalar horizontalmente a aplicação, adicionando mais máquinas com instâncias dela, que ficam atrás de um sistema de balanceamento de cargas, que gerencia as requisições. (LEWIS; FOWLER 2014)

O problema desta abordagem pode surgir à medida que a aplicação cresce, quando temos que gerenciar um código-fonte muito grande, onde a manutenção torna-se mais difícil de acordo com o crescimento e pode piorar se o código não foi feito pensando no baixo acoplamento e alta coesão de seus componentes. Pode haver ainda, a necessidade de melhoria de partes da aplicação, onde novas tecnologias poderiam ser aplicadas para melhorar o desempenho ou a interface, ou até mesmo a necessidade de escalar outras partes que lidam com processos críticos da aplicação. Com a arquitetura monolítica estas melhorias se tornam difíceis de serem aplicadas.

Newman (2015) fala que com o tempo, foram sendo discutidas novas abordagens para a construção de aplicações, pensando em vários pontos, como design orientado ao domínio da aplicação, entrega contínua de componentes, virtualização on-demand, automação de infraestrutura, pequenos times autônomos

para gerenciar o código da aplicação e escalabilidade de sistemas. Então surge a arquitetura de microsserviços pensando no melhor uso das aplicações para resolver essas questões.

1.1 JUSTIFICATIVA

Houve uma primeira tentativa de se resolver os contratempos que surgiam quando uma aplicação monolítica se tornava grande demais, que foi a criação da arquitetura orientada à serviços, ou SOA, que definia a divisão da aplicação em componentes, que são agora serviços, com fronteiras bem-definidas, que se comunicavam através da rede, com foco na reutilização do software. Porém, apesar de SOA ser uma ideia sensível, há uma falta de consenso em como construir bem um SOA. Muitos dos problemas deste tipo de arquitetura vinham dos protocolos de comunicação (e.g., SOAP), middleware proprietário, a falta de um guia sobre granularidade de serviços, ou a falta de orientação sobre achar o melhor ponto para dividir o sistema. (NEWMAN, 2015).

Dos conceitos de SOA e sistemas distribuídos, emerge um novo, se utilizando das vantagens dos anteriores e tentando resolver as dificuldades de implementação de ambos. Arquitetura de microsserviços vem como uma evolução na forma de se construir software, com foco na reutilização de componentes pequenos e reutilizáveis, que se comunicam pela rede através de protocolos leves de comunicação, compartilhando nada além de suas interfaces, para garantir uma maior coesão no sistema como um todo, juntamente com baixo acoplamento.

1.2 PROBLEMA

Aplicações monolíticas podem ser boas, mas com o crescimento das aplicações sendo implantadas em nuvem, e o uso de metodologias ágeis no desenvolvimento, foi aumentando o ritmo e a necessidade da mudança (e consequentemente, sua implantação) nas aplicações. Com aplicações monolíticas, muitas vezes há um pacote que é gerado, contendo toda a lógica de negócios, camada de apresentação, lógica de acesso ao banco de dados, ou seja, tudo que é necessário para a aplicação funcionar. Dentro dela, pode-se ainda utilizar bibliotecas de códigos

disponibilizadas por terceiros, colocando-as dentro da aplicação para realizar alguma tarefa. Ao passo que a aplicação cresce, para que seja implantada uma nova versão é necessário realizar uma construção nova da aplicação, para que se possa então implantá-la. Seja a mudança uma correção de uma pequena falha, ou um travamento por vazamento de memória que finaliza o processo da aplicação no servidor, para aplicar esta mudança em produção, é necessária a *build* da aplicação como um todo, o que dependendo do tamanho, consome tempo para gerar e implantar, e caso seja necessário desfazer a implantação desta versão, a aplicação deve ser parada para voltar o servidor para o estado anterior (caso não haja um espelhamento, a aplicação deve ficar fora do ar por alguns minutos).

Com o crescimento da base de código, é possível que funções que realizam certas tarefas se repitam em lugares distintos, sendo necessário criar abstrações para funções mais gerais, onde haverão dependências deste código espalhadas pela aplicação. Ainda, um código grande é difícil de manter, e caso seja necessário a inserção de novas pessoas no projeto para fazer sua manutenção, pode ser necessário um tempo apenas para que a pessoa se habitue com o código e os padrões adotados pela empresa para a construção dele (algumas vezes sendo necessário que outra pessoa seja alocada para auxiliar nesta tarefa), o que vai fazer com que ela seja pouco produtiva no início.

Aplicações monolíticas crescem e evoluem na linguagem em que foram inicialmente construídas durante o desenvolvimento do projeto. Isto as torna menos propensas a adoção de novas tecnologias num estágio posterior ao início do projeto ou até mesmo quando ela já está em um estado mais maduro. Tecnologias vêm evoluindo ao longo dos anos. Caso a aplicação tenha sido escrita em Java, e por algum motivo se quer trocar um serviço por NodeJS por uma necessidade comercial ou para tirar proveito da nova tecnologia, a arquitetura monolítica dificulta um pouco este processo, não apenas por integração de código, mas também para os testes e a implantação da aplicação, sendo ainda que num projeto podem haver dependências de diferentes versões de uma biblioteca, o que pode tornar o processo mais trabalhoso. É possível que a aplicação seja completamente reescrita com outra tecnologia, mas este é um processo que leva tempo e é bastante arriscado.

1.3 OBJETIVOS GERAIS

O presente trabalho objetiva a análise da arquitetura de microsserviços, voltada para a construção de um sistema.

1.4 OBJETIVOS ESPECÍFICOS

Será feita uma análise bibliográfica acerca da arquitetura de software monolítico, orientada a serviços e microsserviços. Serão analisadas neste trabalho características como: componentização do software via serviços, resiliência, escalabilidade, facilidade de implantação, manutenção e evolução. Uma aplicação construída nesta arquitetura será analisada, e serão apontados aspectos como as fronteiras dos serviços e sua independência de implementação e implantação, como se dá a comunicação entre eles, como são guardados os dados pertinentes à cada um.

2 FUNDAMENTAÇÃO TEÓRICA

Serão tratados neste capítulo os conceitos pertinentes à compreensão do tema do trabalho.

2.1 SISTEMAS DISTRIBUÍDOS

Segundo Tanenbaum e van Steen (2007), dois avanços tecnológicos começaram a mudar a situação dos sistemas de computação. Em meados da década de 1980 o surgimento dos microprocessadores de grande capacidade, sendo que algumas dessas CPUs tinham a capacidade de computação de um *mainframe* - isto é, um grande computador central -, mas por uma fração do seu preço.

O segundo desenvolvimento foi a das redes de computadores de alta velocidade. Redes locais, ou LANs (local-area networks), permitem que centenas de máquinas localizadas dentro de um edifício sejam conectadas de modo tal que pequenas quantidades de informação possam ser transferidas em alguns microssegundos, e maiores quantidades de dados, podem ser movimentadas entre máquinas a taxa de 100 milhões a 10 bilhões de bits/s. Redes de longa distância, ou WANs (wide-area networks), permitem que milhões de máquinas no mundo inteiro se conectem a velocidades que variam de 64 Kbits/s a gigabits por segundo (TANENBAUM; VAN STEEN, 2007).

Coulouris et al. (2013) completa que as redes de computadores estão por toda parte, sendo a Internet uma delas, assim como muitas das redes das quais ela é composta. Redes de telefones móveis, redes corporativas, redes de fábrica, redes em campus, redes domésticas, redes dentro de veículos, todas elas, tanto separadamente como em conjunto, compartilham as características básicas que as tornam assuntos relevantes para o estudo de sistemas distribuídos. O resultado dessas tecnologias é que atualmente não somente é viável, mas também fácil, montar sistemas de computação compostos por grandes quantidades de computadores conectados por uma rede de alta velocidade. Esses sistemas costumam ser denominados redes de computadores ou sistemas distribuídos, em comparação com os sistemas centralizados (ou sistemas monoprocesados) anteriores, que consistem

em um único computador, seus periféricos e, talvez alguns terminais remotos. (TANENBAUM; VAN STEEN, 2007).

Para Coulouris et al. (2013), definimos um sistema distribuído como aquele no qual os componentes de *hardware* ou *software*, localizados em computadores interligados em rede, comunicam-se e coordenam suas ações apenas enviando mensagens entre si. Essa definição simples abrange toda a gama de sistemas nos quais computadores interligados em rede podem ser distribuídos de maneira útil.

Já Tanenbaum e van Steen (2007), possuem uma caracterização sem ser muito específica: “Um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente”.

Essa definição, segundo eles, tem vários aspectos importantes, sendo o primeiro deles que um sistema distribuído consiste em um conjunto de componentes autônomos, e um segundo aspecto é que os usuários, que podem ser pessoas ou programas, enxergam o sistema como uma peça única. Não há nenhuma regra quanto ao tipo de computadores que podem ser utilizados e como eles são interconectados.

Outros três pontos acerca dos sistemas distribuídos são a concorrência (execução concorrente numa rede de computadores), a inexistência de um relógio global para coordenar as ações do programa, que como citado anteriormente, neste tipo de sistema se dá por troca de mensagens, e as falhas independentes que ocorrem, pois, todo sistema é passível de falha, porém nos sistemas distribuídos as falhas são diferentes. As falhas na rede resultam no isolamento dos computadores que estão conectados a ela, mas que isso não significa que eles pararam de funcionar, e que na verdade os programas existentes talvez não consigam detectar se a rede falhou ou ficou demasiadamente lenta. Cada componente deste sistema pode falhar independentemente, deixando os outros ainda em funcionamento (COULOURIS et al., 2013).

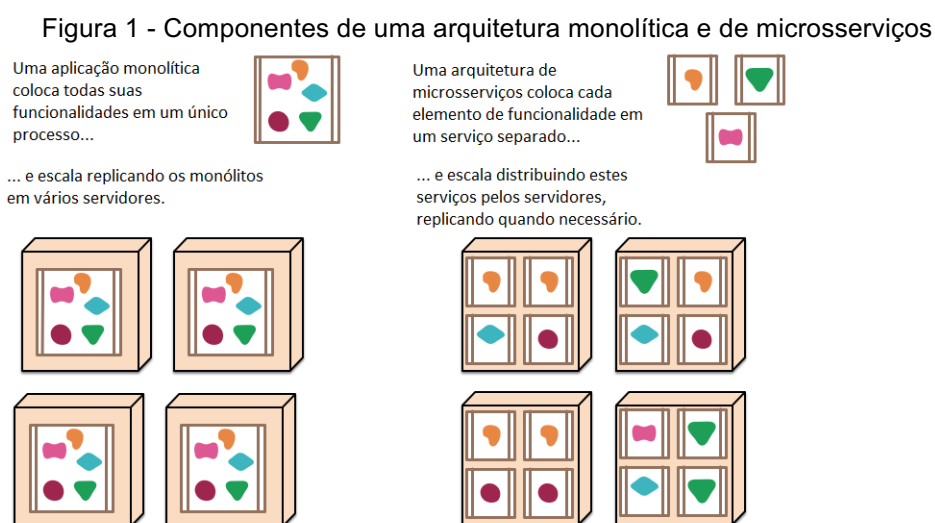
Tanenbaum e van Steen (2007) apontam que a principal meta de um sistema distribuído é facilitar aos usuários, e às aplicações, o acesso a recursos remotos e seu compartilhamento de maneira controlada e eficiente. O termo “recurso” é bastante abstrato, mas caracteriza bem o conjunto de coisas que podem ser compartilhadas de maneira útil em um sistema de computadores interligados em rede. Ele abrange desde componentes de *hardware*, como discos e impressoras, até entidades definidas pelo *software*, como arquivos, bancos de dados e objetos de dados de todos os tipos, e

isso inclui o fluxo de quadros de vídeo proveniente de uma câmera de vídeo digital ou a conexão de áudio que uma chamada de telefone móvel representa.

2.2 ARQUITETURA DE SOFTWARE

Arquitetura de software remete à estrutura geral do software e as formas em que esta estrutura provê integridade conceitual para um sistema (SHAW; GARLAN, 1995). De acordo com Pressman (2010), em sua forma mais simples, arquitetura é a estrutura ou organização dos componentes de um programa (módulos), a maneira como esses componentes interagem, e as estruturas de dados que são utilizadas pelos componentes.

Na figura 1, é mostrada a forma como estão organizados os componentes de uma aplicação monolítica e sua relação com a escalabilidade da aplicação, e ao lado, como uma aplicação em microsserviços é dividida em termos de componentes, e como cada um pode ser escalado individualmente.



Fonte: Lewis e Fowler (2014)

Shaw e Garlan (1995) descrevem um conjunto de propriedades que devem ser especificadas como parte de um *design* arquitetural:

- a) Propriedades estruturais. Este aspecto da representação do design arquitetural define os componentes de um sistema (e.g., módulos, objetos, filtros) e a maneira na qual estes componentes estão dispostos

e interagem um com o outro. Por exemplo, objetos são criados para encapsular tanto os dados quanto o processamento que manipula estes dados, e eles interagem via a invocação de métodos;

- b) Propriedades extrafuncionais. A descrição do *design* arquitetural deve endereçar como a arquitetura alcança os requerimentos para performance, capacidade, confiabilidade, segurança, adaptabilidade, e outras características do sistema;
- c) Famílias de sistemas relacionados. O *design* arquitetural deve se basear em padrões que são repetíveis, e que são comumente encontrados no design de famílias de sistemas similares. Em resumo, o design deve possuir a habilidade de reutilizar blocos da construção arquitetural.

Dadas as especificações destas propriedades, o *design* arquitetural pode ser representado utilizando um ou mais de um número de diferentes modelos (GARLAN; SHAW, 1993). Modelos estruturais, representam arquitetura como uma coleção organizada de componentes de um programa. Modelos de *framework* aumentam o nível de abstração do *design* tentando identificar *frameworks* repetíveis de *design* arquitetural que são encontrados em tipos similares de aplicações. Modelos dinâmicos endereçam os aspectos comportamentais da arquitetura de um programa, indicando como a estrutura, ou configuração do sistema pode mudar em função de eventos externos. Modelos de processos focam no *design* do negócio ou processos técnicos que o sistema tem que acomodar. Finalmente, modelos funcionais, podem ser usados para representar a hierarquia funcional de um sistema (PRESSMAN, 2010).

Shaw e Garlan (1996), discutem a arquitetura de software da seguinte maneira:

Desde que o primeiro programa foi dividido em módulos, sistemas de software tem arquiteturas, e programadores vêm sendo responsáveis pelas interações entre os módulos e as propriedades globais da montagem. Historicamente, arquiteturas eram implícitas - acidentes de implementação, ou sistemas legados do passado. Bons desenvolvedores de software adotaram um ou mais padrões arquiteturais como estratégias para a organização do sistema, mas eles utilizaram esses padrões informalmente e não possuem meios de torná-los explícitos no sistema resultante.

Para definir o que é arquitetura, pode-se fazer uma alusão à arquitetura de um prédio. Ao considerar este tipo de arquitetura, vários atributos diferentes vêm a mente. Em um nível mais simplista, pode-se pensar nela como a forma da estrutura física.

Mas a arquitetura é muito mais, de fato. Ela é a maneira na qual os componentes do prédio são integrados para formar um todo coeso. É a forma em que o prédio se encaixa no seu ambiente e se mistura com outros prédios na vizinhança. É o grau em que o prédio alcança seu propósito e satisfaz as necessidades do seu dono. É a estética da estrutura - o impacto visual do prédio - e a forma na qual texturas, cores e materiais são combinados para criar uma fachada externa e um ambiente de “vivência interna”. São os pequenos detalhes - o *design* das luminárias, o tipo de piso (PRESSMAN, 2010).

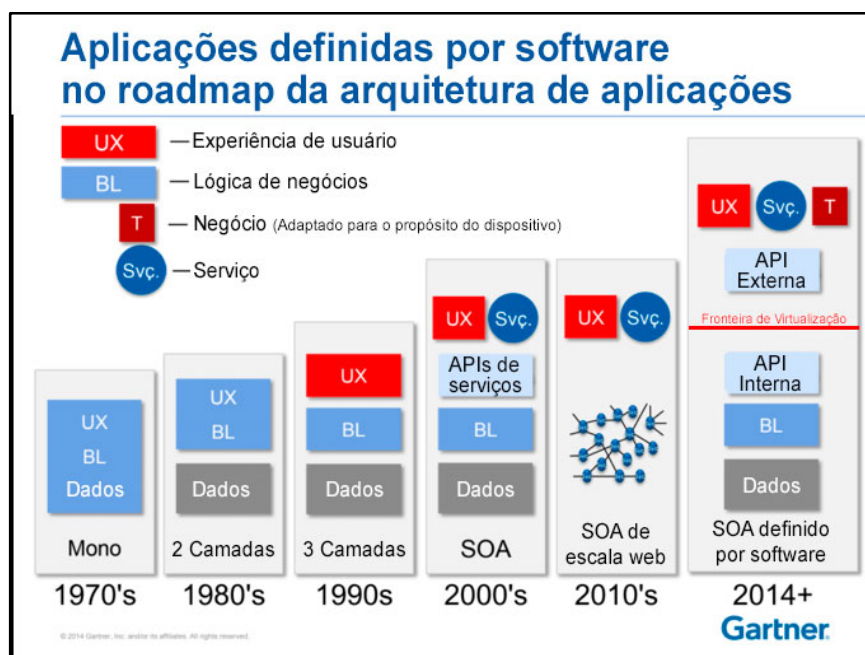
Para Tyree e Akerman (2005), arquitetura é caracterizada por milhares de decisões, tanto grandes quanto pequenas. Algumas dessas decisões são feitas logo cedo no *design* e podem ter um impacto profundo em todas as outras ações do *design*. Outras são postergadas, eliminando restrições muito severas que levariam a uma implementação pobre do estilo arquitetural (PRESSMAN, 2010).

Bass, Clements e Kazman (2003) definem a arquitetura de software de um programa é a estrutura ou estruturas de um sistema, as quais compreendem os componentes de software, as propriedades externas visíveis destes componentes, e as relações entre eles.

Como Pressman (2010) completa, a arquitetura não é o software operacional. Ao invés disso, é a representação que permite analisar a eficiência do *design* em cumprir seus requisitos, considerar alternativas arquiteturais em um estágio onde fazer mudanças no design são relativamente simples, e reduzir o risco associado à construção do software.

De acordo com o Natis (2014), há uma tendência na construção de softwares, tendo em vista a evolução das arquiteturas monolíticas construídas no modelo de camada única, ou *single tier*, passando por modelos de duas e três camadas, até o surgimento de SOA, até alcançar a arquitetura de microsserviços, como é mostrado na figura 2.

Figura 2 - Evolução da arquitetura de aplicações



Fonte: Natis (2014)

2.3 ARQUITETURA SINGLE TIER

Como mostrado na Figura 1, o modelo de arquitetura predominante na década de 1970 era o modelo de uma camada. Nele toda a lógica de negócios, camada de apresentação e persistência de dados permaneciam em um único servidor que era responsável pelo processamento.

Devido ao alto custo de aquisição de computadores na época, as empresas optavam por ter um computador central (*mainframe*) que centralizava o processamento e o armazenamento, enquanto haviam diversos terminais que agiam como interfaces de entrada e saída para acesso ao sistema.

Uma das vantagens desta abordagem é a segurança, pois todo acesso aos dados está centralizado em um ponto, e que se comunicam através de uma intranet, a partir de uma rede confiável. Outra facilidade que ela permite é a implantação da aplicação, devido o fato dos componentes estarem em um único computador.

Com o passar do tempo, o aumento da necessidade de processamento das tarefas do sistema, juntamente com o aumento de usuários, trazia a necessidade de escalar o sistema, o que tinha de ser feito comprando outra máquina, espelhando o sistema.

2.4 ARQUITETURA CLIENTE-SERVIDOR

Apesar da falta de consenso sobre muitas questões de sistemas distribuídos, algo que os pesquisadores e praticantes concordam, é pensar os sistemas em termos de clientes que solicitam serviços de servidores. (TANENBAUM; VAN STEEN, 2007). A arquitetura cliente-servidor é a mais citada quando se discute os sistemas distribuídos, de acordo com Coulouris et al. (2013).

No modelo cliente-servidor básico os processos de um sistema distribuído são divididos em dois grupos. Como Tanenbaum e van Steen (2007) descrevem, um servidor é um processo que implementa um serviço específico, e um cliente é um processo que requisita um serviço de um servidor, enviando-lhe uma requisição e na sequência esperando uma resposta. Esta interação entre eles é também conhecida como comportamento de requisição-resposta.

Considerando que muitas aplicações cliente-servidor visam dar suporte ao acesso de usuários ao banco de dados, de acordo com Tanenbaum e van Steen (2007), muitas pessoas defendem uma distinção entre três camadas: um nível de interface com o usuário, que contém a parte de apresentação do sistema ao usuário, a camada da lógica de negócios, onde normalmente estão contidas as aplicações e a camada de dados, onde são gerenciados os dados que são manipulados pela aplicação.

2.5 ARQUITETURA ORIENTADA À SERVIÇOS (SOA)

SOA é um conceito arquitetural onde todas as funções, ou serviços, são definidas utilizando uma linguagem de descrição e suas interfaces podem ser descobertas através da rede. A interface é definida de maneira neutra, em que é independente de *hardware*, sistema operacional, ou da linguagem em que o serviço foi implementado. Além disso, os múltiplos serviços que compõem um SOA colaboram para prover um conjunto de funcionalidades. Um serviço aqui quer dizer processos de um sistema operacional completamente separados. (NEWMAN, 2015).

De acordo com Erl (2009), os princípios de *design* associados a este tipo de arquitetura são:

- a) Contrato de serviço padronizado: É por onde os serviços expressam seu propósito e suas capacidades. Este seja talvez o componente mais importante da orientação à serviços, essencialmente porque ele requer que as considerações específicas do serviço sejam levadas em conta ao se projetar a interface técnica pública de um serviço e ao se avaliar a natureza e a quantidade do conteúdo que será publicado como parte do contrato oficial de um serviço;
- b) Baixo acoplamento de serviço: Este princípio defende a criação de um tipo específico de relacionamento dentro e fora dos limites do serviço, com foco constante em reduzir as dependências entre o contrato de serviço, sua implementação e os consumidores do serviço. Ele ainda permite que o *design* e a lógica de um serviço evoluam independentemente de sua implementação enquanto garante a interoperabilidade básica com os consumidores daquele serviço;
- c) Abstração de serviço: Em um nível fundamental, este princípio enfatiza a necessidade de se ocultar o maior número possível de informações e detalhes subjacentes de um serviço;
- d) Capacidade de reuso de serviço: Define os serviços como recursos corporativos com contextos funcionais agnósticos. Várias considerações de *design* são levantadas para assegurar que capacidades de serviços individuais sejam adequadamente definidas em relação a um contexto de serviço agnóstico e garantir que possam facilitar os requisitos necessários para seu reuso;
- e) Autonomia de serviço: Este princípio levanta várias questões pertinentes ao *design* da lógica, bem como ao ambiente de implementação real do serviço. As considerações sobre níveis de isolamento e de normalização do serviço são levadas em conta para alcançar uma medida conveniente de autonomia;
- f) Independência de estado do serviço: Os serviços são, idealmente, projetados para manterem informações de estado apenas quando estas forem necessárias. Aplicar este princípio requer que as medidas de independência de estado realisticamente atingíveis sejam avaliadas, com base na adequação do ambiente de arquitetura tecnológica que

fornece opções de delegação de gerenciamento de estado e de diferimento;

- g) Visibilidade do serviço: Os serviços precisam ser facilmente identificados e entendidos quando houver possibilidades de reuso. O *design* dos serviços precisa então levar em conta a “qualidade de comunicação” do serviço e suas capacidades particulares;
- h) Composição de serviços: Espera-se que os serviços sejam capazes de participar como membros de composição eficazes, mesmo que precisem ser imediatamente incluídos em uma composição.

Este tipo de arquitetura, é uma ideia sensível. SOA surgiu como uma forma de combater os desafios das grandes aplicações monolíticas. É uma aproximação a construção que promove a reutilização do *software*, onde por exemplo, duas ou mais aplicações de usuários finais poderiam usar os mesmos serviços. Ela foca em tornar mais fácil manter ou reescrever o *software*, em que teoricamente pode-se substituir um serviço por outro sem que os outros percebam, contanto que as semânticas e interfaces do serviço não sejam alteradas. (NEWMAN, 2015).

2.6 ARQUITETURA DE MICROSERVIÇOS

De acordo com Nadareishvili et al. (2016), para melhor entender o que são microserviços, precisa-se olhar de onde eles vieram. Enquanto o termo em si vem sendo usado de várias formas ao longo dos anos, a associação que ele agora tem com uma maneira particular de se construir um software veio de um encontro onde participaram alguns engenheiros de software. Nas palavras de James Lewis que estava no encontro:

Ao final do encontro de 3 dias, um de nós levantou uma questão - aquele ano estava claro que a maioria dos problemas que as pessoas enfrentavam estavam relacionados à construção de sistemas que eram muito grandes. “Como eu posso reconstruir parte disso”, “melhores maneiras de implementar um estrangulador¹”, etc.

¹ Estrangulador (Strangler) é uma metáfora usada por Fowler (2004), onde ele descreve uma forma de um sistema nascer sobre outro já existente, se aproveitando de suas funcionalidades, e aos poucos acabando com o sistema legado.

Com isso em mente, o problema se tornou “como podemos construir sistemas que são substituíveis ao invés de sistemas que temos que manter?” Nós utilizamos o termo *micro-apps*, eu me lembro.

Em suma, microsserviços são serviços autônomos e pequenos que trabalham juntos. (NEWMAN, 2015).

Nadareishvili et al. (2016) mostra que a ideia de James identifica três conceitos principais ao estilo de *design* de microsserviços:

- a) Microsserviços são ideais para sistemas de grande porte. Os maiores problemas que as pessoas encontravam eram em relação ao tamanho. Isto é significativo, pois o tamanho reflete uma característica particular dos microsserviços - eles são construídos para resolver problemas para sistemas que são grandes demais. Porém o tamanho é uma medida relativa, e é difícil quantificar a diferença entre pequeno, normal e grande. O que os arquitetos do encontro estavam preocupados, não era com a questão de tamanho do sistema. Ao invés disso, eles focaram na situação em que o sistema era muito grande. O que eles identificaram foi que sistemas que crescem de tamanho para além das fronteiras que lhes são inicialmente definidas, apresentam problemas quando chega a hora de fazer alguma modificação. Em outras palavras, novos problemas surgem devido à sua escalabilidade;
- b) Arquitetura de microsserviços é orientada ao objetivo. Este tipo de arquitetura não é sobre identificar uma coleção específica de práticas, ao invés disto, é um reconhecimento de que desenvolvedores de *software* estão tentando alcançar um objetivo similar utilizando uma aproximação em particular. Pode haver um conjunto de características em comum que emergem deste estilo de desenvolvimento de software, mas o foco se destinava a resolver o problema inicial de sistemas que eram muito grandes;
- c) Microsserviços são focados na substituição. A ideia que circunda a substituição de componentes ao invés de manter os que já existem chegam ao coração do que torna os microsserviços especiais.

Lewis e Fowler (2014) apontam que, apesar de não haver uma definição precisa deste estilo arquitetural, existem algumas características em comum no que diz respeito à organização em torno da capacidade do negócio, implantação automatizada, inteligência nas pontas, e controle descentralizado de linguagens e dados. Em resumo, o estilo arquitetural de microsserviços é uma abordagem para o desenvolvimento de uma única aplicação como um composto de pequenos serviços, cada um rodando em seu próprio processo, e se comunicando via mecanismos leves, geralmente uma API de recursos HTTP. Estes serviços são construídos em torno das capacidades do negócio e independentemente implantados por um maquinário totalmente automatizado. Existe um mínimo de gerenciamento centralizado por entre os serviços, e que pode ser escrito em diferentes linguagens de programação e utilizar diferentes tecnologias de armazenamento de dados.

2.6.1 Arquitetura monolítica

Para Lewis e Fowler (2014), para entender o estilo dos microsserviços, primeiro é necessário compará-lo ao estilo monolítico. Aplicações monolíticas são construídas como uma única unidade, centralizando o artefato da implantação, mas também com um único ponto de falha. Aplicações comerciais construídas desta forma normalmente são constituídas das três camadas descritas na arquitetura cliente-servidor: uma camada de interface com o usuário, uma camada de lógica de negócios e uma camada de dados. Esta camada de lógica que contém a aplicação é um monólito: um único arquivo executável.

A medida que aplicação cresce, sua base de códigos aumenta drasticamente, e começa a se tornar difícil fazer uma mudança nesta aplicação sem afetar outras partes, mesmo que ela tenha sido muito bem construída com base nos princípios do baixo acoplamento e alta coesão entre seus componentes.

A partir dos problemas encontrados nestas aplicações, surgiram os microsserviços, que consistiam em construir aplicações com base em grupos de serviços que se comunicam e interagem para poder alcançar determinado objetivo em comum.

Os serviços são independentes, possuem uma fronteira firme que os permite ser independentemente escalados, modificados e até descartados se for necessário,

podendo ser escritos em diferentes linguagens de programação e gerenciados por times independentes.

O conceito de microsserviços não é algo inovador, visto que ele traz conceitos que remetem aos princípios de design do Unix, onde a cultura do software já pregava a necessidade de se construírem programas que sejam simples de ler e escrever, evitando grandes e complicados monólitos. (RAYMOND, 2004).

2.6.2 Características e benefícios da arquitetura de microsserviços

Lewis e Fowler (2014) definem que apesar de não haver uma definição formal do estilo arquitetural de microsserviços, existem características em comum em arquiteturas que se encaixam neste título.

2.6.2.1 Componentização via serviços

Desde o começo na indústria de software, havia um desejo em construir sistemas como conjuntos de componentes interligados. Por definição, um componente é uma unidade de software que pode ser trocado ou melhorado de forma independente. (LEWIS; FOWLER, 2014).

Arquiteturas de microsserviços podem usar bibliotecas para tirar proveito do reaproveitamento, mas a forma prioritária de se alcançar tal objetivo é quebrando o *software* em serviços. Como Lewis e Fowler (2014) definem, bibliotecas são componentes ligados a um programa e chamados em memória. Já serviços são componentes fora do processo que se comunicam via chamada de serviço, ou uma chamada a um procedimento remoto.

Usar serviços em detrimento à bibliotecas tem uma razão, que é o fato de que serviços podem ser implantados de forma independente. Uma aplicação que consiste em várias bibliotecas dentro de um processo, ao haver uma mudança em um componente significa implantar novamente a aplicação. Porém em uma aplicação composta por serviços, uma mudança neste serviço vai requerer que apenas ele seja implantado. Este não é um fato absoluto, pois algumas mudanças podem afetar as interfaces dos serviços, o que vai resultar em alguma coordenação, mas o foco de uma boa aplicação de microsserviços é minimizar estes impactos.

2.6.2.2 Organizados em torno das capacidades de negócio

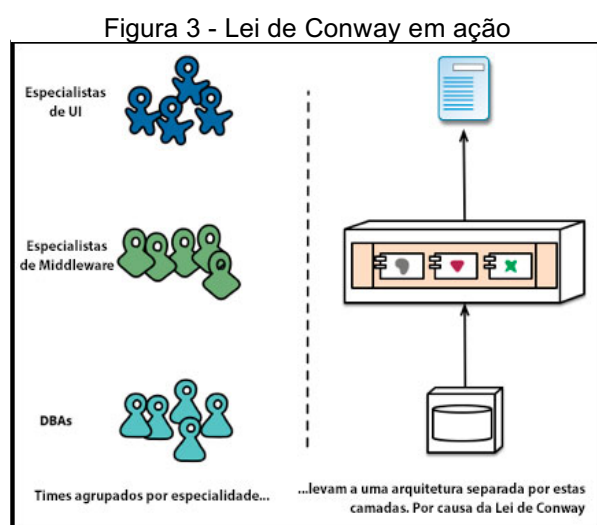
Ao buscar a divisão de uma aplicação em partes menores, há uma tendência de que esta divisão seja feita a partir das fronteiras das tecnologias, o que gera um time de especialistas em interfaces com o usuário, um time de banco de dados, e um time de lógica de negócios. Separados assim, uma mudança qualquer deve passar muitas vezes por entre os grupos, o que pode envolver certa burocracia e tempo. (LEWIS; FOWLER, 2014).

Nadareishvili et al. (2016) ao falar de como a cultura afeta a produtividade do time, cita uma frase de um artigo escrita por Melvin Conway:

Organizações em que produzem sistemas tem a tendência a construir sistemas que são a cópia da sua estrutura de comunicação.

Esta frase foi identificada em 1975 por Fred Brooks como a lei de Conway, e ela mostra o quanto é afetada a qualidade final do produto da companhia pela estrutura organizacional. Colocada de forma simples, ela diz que a comunicação dita o resultado.

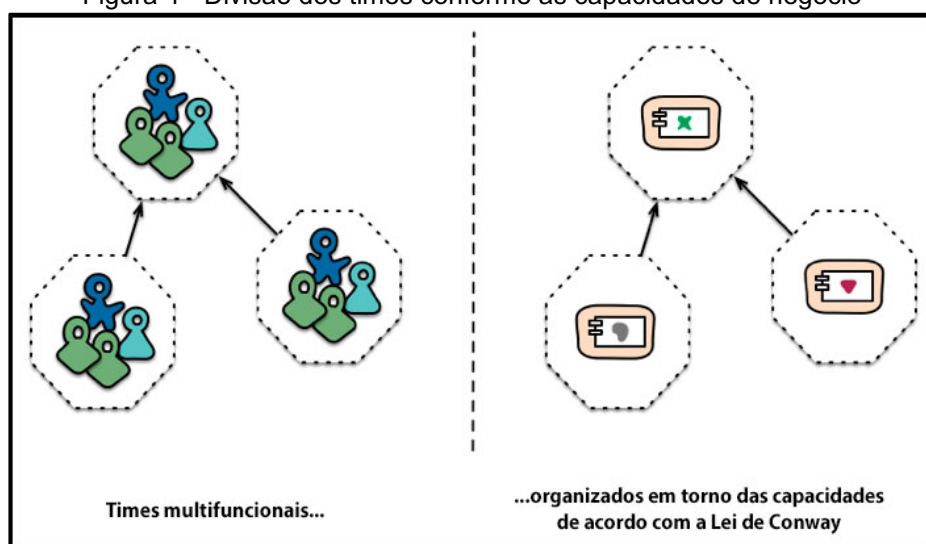
A figura 3 mostra os times de desenvolvimento separados de acordo com suas especialidades, e como essa divisão se reflete na aplicação que é construída, ou seja, como reflete a Lei de Conway para aplicações monolíticas.



Fonte: Lewis e Fowler (2014)

Microserviços possuem uma divisão diferente destas capacidades, organizando os serviços em tornos das capacidades negociais. Tais serviços são responsáveis por toda sua área de negócios, incluindo a interface com o usuário, persistência de dados e outras colaborações externas, o que leva os times que constroem estes serviços sejam compostos por profissionais das mais variadas especialidades, times multifuncionais de pessoas com habilidades em experiência de usuário, banco de dados e gerenciamento de projetos. A figura 4 mostra como times multifuncionais, organizados agora de acordo com as capacidades de negócio refletem na construção da aplicação.

Figura 4 - Divisão dos times conforme as capacidades de negócio



Fonte: Lewis e Fowler (2014)

2.6.2.3 Foco em produtos, não projetos

Ao invés de seguir o modelo de projeto em que o objetivo é entregar um pedaço de *software* onde então ele é considerado como completo e o time que o construiu é desfeito, e o *software* é deixado aos cuidados da manutenção, os times que propõem microserviços preferem a noção de que o time deve ser o dono daquele produto durante todo seu tempo de vida.

Esta ideia vai de acordo com a de dividir a aplicação em partes, que encapsulam as capacidades de negócio. Agora, ao invés de olhar para o *software* como um conjunto de funcionalidades que devem ser completadas, há uma relação que perdura, onde a questão principal é como o *software* pode ajudar seus usuários para aumentar suas capacidades de negócio. (LEWIS; FOWLER, 2014).

2.6.2.4 Pontas inteligentes, canais burros

Ao construir serviços, o ideal é que os consumidores deste serviço não conheçam a implementação interna deste serviço, pois isto levaria a um alto acoplamento no sistema. Isto quer dizer que, ao mudar algo dentro dos microserviços, os consumidores daquele serviço terão problemas ao precisar mudar também. (NADAREISHVILI ET AL., 2016).

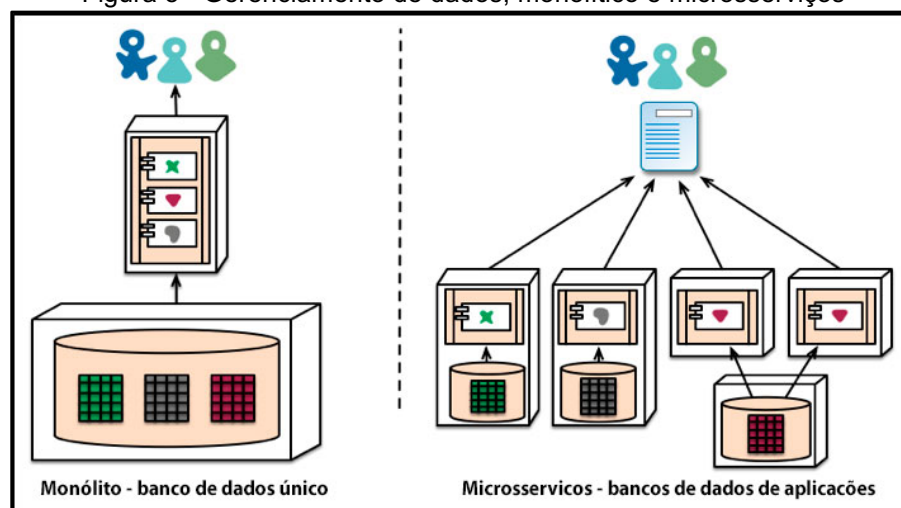
A aproximação dada pela arquitetura de microserviços para isso é a utilização de pontas inteligentes e canais burros. Aplicações são donas da sua lógica de negócios e agem como filtros no sentido Unix clássico - recebendo uma requisição, aplicando a lógica necessária, e produzindo uma resposta. Isto é coreografado utilizando protocolos REST simples, que definem que os recursos dos quais aquele serviço trata, vão se comportar sempre da mesma forma quando da chamada de um determinado método exposto para chamada, por aquele serviço, ao invés de protocolos mais complexos como o *WS-Choreography*, como definido por Burdett e Kavantzaz (2004), que consiste num modelo de informação que descreve os dados e as relações entre eles necessárias para definir uma coreografia, que descreve a uma sequência e as condições nas quais os dados trocados entre dois ou mais participantes para que atinjam um propósito útil, o BPEL que é um padrão para a montagem de um conjunto de serviços num processo a fim de reduzir os custos e a complexidade de sua integração, ou, a orquestração dos serviços por uma ferramenta central. Esta abordagem objetiva que os serviços sejam o mais desacoplados e coesos possível. (LEWIS, FOWLER, 2014).

2.6.2.5 Gerenciamento de dados descentralizado

A descentralização do gerenciamento de dados quer dizer em um nível mais abstrato, que o modelo conceitual do mundo vai ser diferente por entre os sistemas. Microserviços preferem deixar que cada serviço gerencia seu próprio banco, diferentes instâncias de uma tecnologia de banco de dados ou até sistemas completamente diferentes de dados - uma abordagem chamada de *Polyglot Persistence*. (LEWIS, FOWLER, 2014). Na figura 5, há uma representação de como

os dados eram organizados na arquitetura monolítica, em que faziam parte de um conjunto apenas, onde a aplicação buscava os dados que precisavam, e na arquitetura de microsserviços, cada serviço é responsável pelos dados da camada de negócio que ele representa.

Figura 5 - Gerenciamento de dados, monolítico e microsserviços

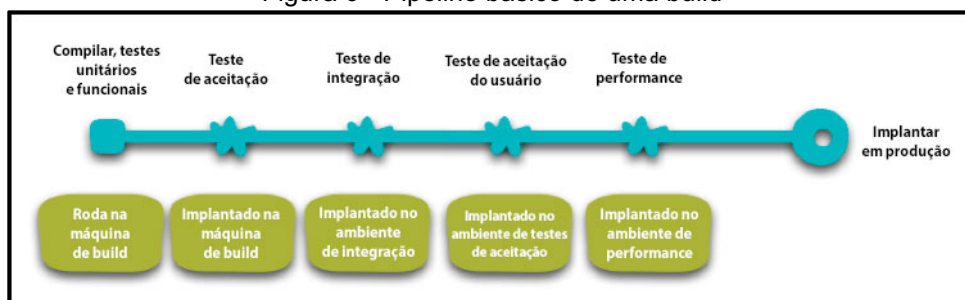


Fonte: Lewis e Fowler (2014)

2.6.2.6 Automação de infraestrutura

De acordo com Nadareishvili et al. (2016), Integração Contínua é uma técnica onde o objetivo é manter todas as partes sincronizadas, assegurando que o código novo que chega é integrado ao código já existente. Para fazer isso, o servidor de integração contínua verifica que existe um código novo, então ele baixa e garante que o código compila e passa nos testes. Na figura 6, é mostrado um *pipeline* de uma *build* de um *software*. Os times que constroem aplicações desta maneira, fazem uso extensivo de técnicas de automação de infraestrutura.

Figura 6 - Pipeline básico de uma build



Fonte: Lewis e Fowler (2014)

2.6.2.7 Projetar para a falha

Como consequência de usar serviços como componentes, as aplicações tem que ser construídas de uma forma que elas tolerem a falha de serviços. Qualquer serviço pode falhar devido a uma indisponibilidade do provedor, e então o cliente tem que responder a isto da forma mais graciosa possível. Tais situações induzem uma complexidade maior para que sejam tomados estes cuidados. O Simian Army do Netflix induz falhas nos serviços, ou até mesmo em *datacenters* durante o dia de trabalho para testar tanto a resiliência quanto o monitoramento da aplicação. (LEWIS; FOWLER, 2014). Em contrapartida, existe o Hystrix, também da Netflix, que é uma biblioteca de tolerância à falhas e latência construída para isolar pontos de acesso a sistemas remotos, serviços e bibliotecas de terceiros, para terminar a falha em cascata e permitir a resiliência em sistemas distribuídos complexos onde a falha é inevitável.

É uma preocupação constante dos desenvolvedores de aplicações distribuídas, como as falhas afetarão o usuário final. Por isso da criação destas ferramentas por parte do Netflix.

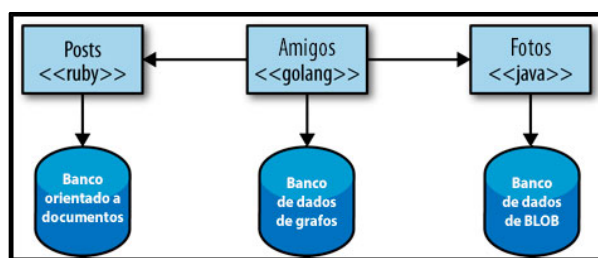
2.6.3 Benefícios da arquitetura de microsserviços

Os benefícios deste tipo de arquitetura são muitos e variados. Muitos destes benefícios podem ser vistos como adquiridos dos sistemas distribuídos. Porém, devido ao quão longe microsserviços tendem a levar os conceitos por trás de sistemas distribuídos e da arquitetura orientada a serviços, eles tendem a levar esses benefícios a um grau maior. (NEWMAN, 2015).

2.6.3.1 Heterogeneidade de tecnologias

Em um sistema composto de múltiplos serviços colaborativos, pode-se optar por utilizar diferentes tecnologias dentro de cada um. Isso permite que sejam escolhidas as melhores ferramentas para cada necessidade. (NEWMAN, 2015). Na figura 7, um exemplo de como pode ser abordada esta característica numa aplicação.

Figura 7 - Como microsserviços permitem a adoção de novas tecnologias



Fonte: Newman (2015)

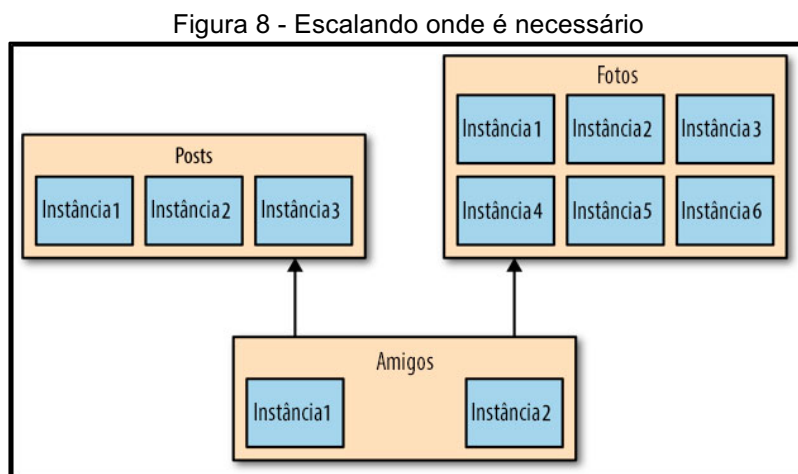
Com microsserviços é possível adotar novas tecnologias mais facilmente e entender como novos avanços podem se encaixar nas aplicações. Em aplicações monolíticas, ao ser adotado um novo framework, uma nova linguagem de programação, ou até mesmo uma nova forma de armazenamento de dados, passando do modelo relacional, para um modelo de grafos, o impacto da mudança tende a ser grande, há um alto risco associado. Com um sistema dividido em serviços, existem vários lugares onde se pode tentar a aplicação de novas abordagens, podendo-se escolher o ponto onde haja um risco menor, limitando um potencial impacto negativo. (NEWMAN, 2015).

2.6.3.2 Resiliência

Um conceito chave na engenharia de resiliência, é o chamado anteparo. Quando um componente do sistema falha, e isso não é passado adiante aos outros componentes, é possível isolar o ponto de falha e o restante do sistema deve continuar operante. Numa arquitetura de microsserviços, as fronteiras entre estes serviços se tornam os anteparos óbvios. Em sistemas monolíticos, a aplicação pode rodar em múltiplas máquinas para reduzir o risco de falha, mas com microsserviços, é possível construir sistemas que lidam com a falha de serviços e se degradam funcionalmente de acordo. No entanto, que para que o sistema em microsserviços consiga adotar essa resiliência, é necessário levar em conta as novas fontes de falhas que sistemas distribuídos têm que lidar. Uma rede pode (e irá) falhar, assim como máquinas. É preciso estar atento a isso para saber como cuidar quando acontecer, e que impactos haverão (se houverem) sobre o usuário final. (NEWMAN, 2015)

2.6.3.3 Escalabilidade

Com uma aplicação dividida em serviços, é possível escalar apenas aqueles que necessitam de maior processamento, rodando mais instâncias deste serviço sem ter que aumentar o processamento para todas as máquinas. A figura 8 mostra serviços rodando com 2, 3 e 6 instâncias.



Fonte: Newman (2015)

2.6.3.4 Constituição

Uma das promessas-chave dos sistemas distribuídos e das arquiteturas orientadas a serviço é a possibilidade da reutilização de uma funcionalidade. Com microserviços, estas funcionalidades podem ser consumidas de formas diferentes e para propósitos diferentes. Isto é especialmente importante quando se é pensado em como os consumidores irão utilizar o *software*. Foi-se o tempo onde era preciso pensar estritamente em uma aplicação desktop, ou em uma aplicação *web*. Nas aplicações de hoje em dia, é necessário saber como aproveitar melhor as capacidades para a *web*, uma aplicação nativa, uma aplicação mobile, para um *tablet* ou dispositivo vestível. (NEWMAN, 2015)

2.6.3.5 Otimizando para substituição

Com serviços individuais sendo pequenos, o custo de substituí-los por uma implementação melhor, ou até deletá-los, é mais fácil de ser gerenciado. Times que utilizam microserviços estão mais confortáveis com reescrever completamente um

serviço quando for necessário, ou acabar com ele quando não for preciso. (NEWMAN, 2015).

3 DESENVOLVIMENTO

O projeto em questão busca definir as vantagens em se utilizar a arquitetura de microsserviços para aplicações corporativas.

3.1 PROJETO

A aplicação estudada, foi feita por Heiko Braun e está disponível no Github, e também no site do Wildfly Swarm para mostrar como é estruturada uma aplicação construída com base na arquitetura de microsserviços. A aplicação é construída em Java, utilizando o Wildfly Swarm para empacotar os serviços que a compõem.

Esta aplicação consiste em uma loja virtual, onde o usuário pode se cadastrar, verificar o catálogo de produtos, adicioná-los ao seu carrinho de compras e realizar o pedido. Para isso ela conta com quatro serviços que a constituem: o serviço do catálogo, o serviço do usuário, o serviço de pedidos e o serviço de interface com o usuário. Todos os serviços são construídos de forma desacoplada, ou seja, eles desconhecem a implementação interna uns dos outros, apenas conhecem a interface de comunicação. Apesar de estarem salvos todos dentro de um mesmo projeto no repositório de controle de versão, eles podem ser independentemente construídos e modificados, sem que o funcionamento dos demais sejam afetados.

3.2 FUNCIONALIDADES DA APLICAÇÃO

As funcionalidades foram construídas para que sejam consumidas pelo serviço do *front-end*, que é o cliente dos demais serviços.

3.2.1 Criar um novo usuário

É possível inserir um novo usuário na aplicação fornecendo seus dados.

3.2.2 Catálogo

A aplicação fornece um catálogo de produtos para que o usuário possa visualizar e adicioná-los ao seu carrinho de compras.

3.2.3 Pedido

Após selecionar os produtos do catálogo ao carrinho, o usuário pode proceder com o pedido, que retornará sucesso e contabilizará os pedidos feitos desde então, ou apresentará uma falha no registro do pedido.

3.3 TECNOLOGIAS UTILIZADAS

A seguir serão descritas as principais tecnologias utilizadas nos serviços e suas funcionalidades.

3.3.1 Java

O Java é uma linguagem de programação orientada à objetos, inicialmente projetada para *softwares* de dispositivos eletrônicos, mas que acabou tendo grande destaque na construção de aplicações para *web*, sendo muito presente no mercado atualmente.

Devido ao fato do Swarm ser implementado em Java e os serviços da aplicação todos rodarem nele, o Java foi a linguagem utilizada para a construção da aplicação.

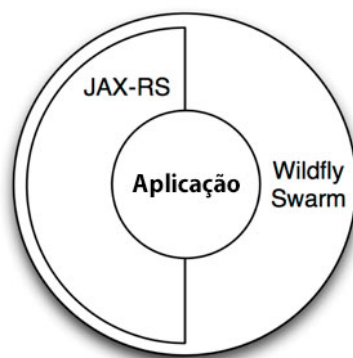
3.3.2 Wildfly Swarm

O Wildfly Swarm é um projeto que foi retirado do WildFly Java Application Server 10.1.0.Final, que foi desconstruído em partes menores, nas quais o swarm permite a reconstituição destas partes na aplicação para que então sejam criados executáveis autônomos, chamados “uberjars”.

O objetivo de tal aproximação é criar um suporte à microsserviços no qual a aplicação receba apenas as partes da JavaEE que ela necessita para funcionar.

Um “uberjar” consiste em colocar apenas o necessário de um servidor de aplicação sobre a aplicação que o utilizará. As partes desnecessárias são excluídas, o que permite uma melhor utilização de recursos. A figura 9 mostra como o Wildfly Swarm encapsula a aplicação e entrega este uberjar, como um arquivo que tem tudo que a aplicação necessita para funcionar.

Figura 9 - Uberjar



Fonte: Braun (2016a)

3.3.3 Consul

O Consul é uma ferramenta que possui vários componentes, dentre eles a descoberta e configuração de serviços na sua infraestrutura. A aplicação estudada neste trabalho faz uso de duas características chave:

- a) O registro de serviços;
- b) A verificação da saúde de um serviço: O Consul provê uma variedade de verificações de serviços, podendo ser associados a um dado serviço (checa se ele está retornando “200 OK”), ou a um nó local (verificando se a utilização de memória está abaixo de 90%), por exemplo.

3.3.3.1 Registro de serviços

Funciona como um banco de serviços, onde contém a localização de suas instâncias. Estas instâncias são inscritas no registro de serviços ao iniciarem e devem ser removidas no seu desligamento. Os clientes de um serviço buscam no registro de serviços para encontrar instâncias disponíveis de um serviço. Cabe ao registro de serviços fazer uma verificação da saúde do serviço, através de uma API, para verificar se ele pode receber requisições.

A aplicação se utiliza do padrão de auto registro, onde a instância de um serviço é responsável por se registrar ao registrador, ou seja, ao subir um serviço, ele se registra no Consul, indicando seu *host* e endereço IP, e se fazer disponível para a descoberta, e ao ser desligado, ele se remove do registro.

3.3.4 ShrinkWrap

É um projeto de código aberto patrocinado pela Red Hat. Ele provê uma API e uma SPI para criar, importar, exportar e manipular arquivos.

3.3.5 Hystrix

Em um ambiente distribuído, inevitavelmente algumas das muitas dependências de um serviço podem falhar. Hystrix é uma biblioteca que ajuda a controlar as interações entre os serviços distribuídos, adicionando tolerância à latência e lógica de tolerância a falhas. A maneira como ele faz isso é isolando os pontos de acesso entre os serviços, parando falhas em cascata que ocorrem ao longo deles e provendo opções de recuperação, o que aumenta a resiliência do sistema no geral. (CHRISTENSEN, 2013).

Criada pela Netflix, uma empresa, conforme Watson, Emmons e Gregg (2015), de serviço de transmissão de vídeos online, e pioneira em novas arquiteturas de nuvem e tecnologias que operam em uma escala massiva, e que possui a arquitetura de seu sistema construída em microsserviços, a biblioteca começou a ser desenvolvida em 2011, e hoje em dia é largamente utilizada na empresa.

3.3.6 Ribbon

Outra biblioteca criada pela Netflix, o Ribbon tem um papel importante em dar suporte a comunicação entre processos na nuvem. Ela inclui os balanceadores de carga nos serviços clientes. Abaixo algumas de suas características:

- a) Múltiplas regras de balanceamento de carga, com capacidade de agregar outras
- b) Integração com a descoberta de serviços
- c) Resiliência a falhas embutida
- d) Feito para a nuvem
- e) Clientes integrados com os balanceadores de carga

O projeto desta biblioteca está em manutenção no momento, e a empresa indicou que não está utilizando todos os componentes dela, pois substituiu alguns por soluções que não fazem parte da *NetflixOSS* (Netflix Open Source Software - onde a empresa distribui seus softwares livres). (WANG, 2013)

3.3.7 JSF

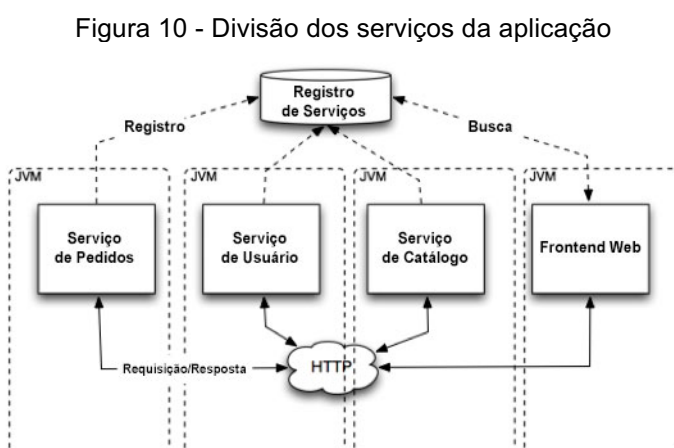
Framework para criar interfaces para aplicações Java voltadas para a *web*.

3.3.8 Hibernate

O banco utilizado pelos serviços é uma instância criada em memória pelo Hibernate, não sendo vinculada a um sistema gerenciador de banco de dados específico, utilizada apenas para pequenos testes de criação, mapeamento e manipulação de entidades.

3.4 ARQUITETURA GERAL DA APLICAÇÃO

Os quatro serviços que compõem a aplicação estão estruturados de acordo com o mostrado na figura 10:



Fonte: Braun (2016b)

Apesar do fato de que todos serviços possuem uma interface aberta para que haja comunicação entre eles, apenas o serviço do *front-end* interage com os demais. Essa comunicação é feita de forma síncrona, e é utilizado o Consul para o registro e descoberta de serviços, então logo ao subir, o serviço é registrado e o serviço do *front-end* o consulta sempre que vai fazer uma requisição aos demais serviços.

3.5 IMPLEMENTAÇÃO DOS SERVIÇOS

Os serviços descritos anteriormente descrevem as funcionalidades de cada um. Para desempenhá-las, cada serviço foi construído na forma de um microserviço que realizará determinada tarefa.

3.5.1 Microserviços de catálogo, pedido e usuário

Estes serviços implementam as funcionalidades delegadas ao serviço de catálogo, de pedidos e de usuários. Eles possuem uma implementação em comum, pois todos são constituídos por um REST *endpoint*, que contém a lógica de como acessar os seus recursos, assim como as classes que definem as entidades que vão ser guardadas no banco de dados. Todas as informações são guardadas numa instância criada em memória pelo Hibernate. Apenas o serviço de catálogo ao iniciar, realiza a inserção de dados referentes aos produtos, os serviços de pedidos e de usuários apenas guardam informações quando são feitas novas inserções por meio da sua API.

A classe principal destes serviços, é a responsável por iniciar o container do Swarm, e subir o serviço, para isso, ela busca a configuração desejada no arquivo *project-stages.yml*, que permite que sejam criadas configurações de acordo com ambientes, para que possam ser diferenciados nesse arquivo, como o nome do serviço, a porta que ele utilizará em cada ambiente e a URL (*Uniform Resource Locator*) do registrador de serviços. Após carregar as configurações, ela inicia o Swarm, e empacota o serviço em um arquivo e adiciona os recursos do banco de dados, e então registra o serviço no Consul. Por último, é feita a implantação do serviço no container do Swarm.

3.5.2 Microserviço do *front-end*

Este serviço que possui a camada de interface com o usuário. Ele também é o serviço que age como cliente dos outros: faz uma chamada ao catálogo para obter a lista de produtos, envia um usuário a ser criado para o serviço de usuário, e registra um pedido no serviço de pedidos. Essa camada de interface foi construída utilizando o JSF.

Na classe principal, este serviço também cria um container do Swarm para subir, e faz o empacotamento do projeto em um .WAR (Web application ARchive), mapeando as páginas .xhtml (eXtensible Hypertext Markup Language) que vão compor a aplicação.

Este serviço, possui um cliente para cada um dos outros. Estes clientes são responsáveis por fazerem a descoberta dos serviços no Consul, e implementarem a lógica de comunicação com o serviço na qual ele é cliente.

No *front-end*, é também onde está a criação das *threads* do Hystrix, que vão cuidar das chamadas ao serviço. Estas *threads* também são criadas nos clientes dos serviços, pois para cada serviço que deve ser chamado, é criada a sua respectiva *thread*, que fica responsável por finalizar a chamada ao serviço caso ela demore para responder e que mede o sucesso, as falhas (exceções lançadas pelo cliente) e *timeouts*, e executando uma recuperação caso alguma exceção ocorra.

Nos clientes dos serviços, por último, também está o Ribbon, que vai atuar verificando as instâncias dos serviços registradas no Consul, criando um grupo de recursos, para distribuir as chamadas entre as instâncias caso alguma chegue a falhar, e chamando o Hystrix para executar a lógica de recuperação.

4 RESULTADOS

Ao inicializar o serviço do *front-end*, pode-se passar parâmetros para determinar em que endereço ele vai estar. A porta utilizada está configurada no arquivo *project-stages.yml*, e vai ser buscada de acordo com o estágio que é determinado ao subir o serviço, conforme é mostrado na figura 11.

Figura 11 - Subindo o serviço do front-end

```
MacBook-Pro-de-Mateus:microservice mateustymoniuk$ java -Dswarm.project.stage=production -Dswarm.bind.address=127.0.0.1 -jar
./everest/target/everest-web-swarm.jar -Dconsul.host=127.0.0.1:8500
[INFO] Using project stage: production
```

Fonte: Elaborado pelo autor, 2018

A figura 12 mostra o momento que o serviço faz o seu registro no Consul:

Figura 12 - Registro do front-end no Consul

```
2018-05-08 01:14:31,024 INFO [org.wildfly.swarm.topology.consul.runtime.Advertiser] (MSC service thread 1-1) Registered service web-frontend:127.0.0.1:8085
```

Fonte: Elaborado pelo autor, 2018

O Hystrix também é inicializado antes que o serviço seja completamente implantado no Swarm, como é mostrado na figura 13:

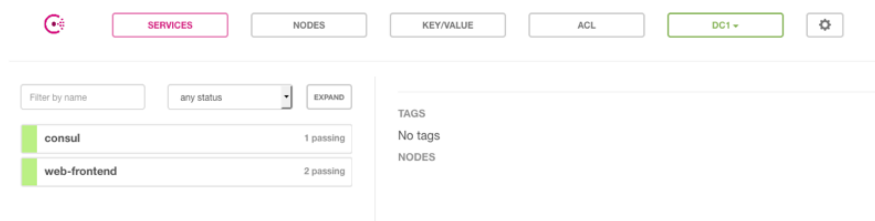
Figura 13 - Inicializando Hystrix no front-end

```
2018-05-08 01:14:32,966 INFO [stdout] (ServerService Thread Pool -- 15) Initialising hystrix ... 1.1 SEÇÃO SECUNDARIA
```

Fonte: Elaborado pelo autor, 2018

A figura 14 mostra a tela do Consul, onde é possível checar se o serviço subiu sem erros:

Figura 14 - Consul com front-end registrado



Fonte: Elaborado pelo autor, 2018

As etapas vão se repetir para os demais serviços. Na figura 15, a inicialização do serviço de catálogo:

Figura 15 - Subindo o serviço do catálogo

```
MacBook-Pro-de-Mateus:microservice mateustymoniuk$ java -Dswarm.project.stage=production -Dswarm.bind.address=127.0.0.1 -jar ./catalog/target/catalog-swarm.jar -Dconsul.host=127.0.0.1:8500
[INFO] Using project stage: production
```

Fonte: Elaborado pelo autor, 2018

Como no serviço de *front-end*, o serviço de catálogo também se registra no Consul, como mostra a figura 16:

Figura 16 - Registro do catálogo no Consul

```
2018-05-08 01:29:07,489 INFO [org.wildfly.swarm.topology.consul.runtime.Advertiser] (MSC service thread 1-4) Registered service catalog:127.0.0.1:9080
```

Fonte: Elaborado pelo autor, 2018

Na figura 17, a inicialização do serviço de pedido:

Figura 17 - Subindo o serviço de pedido

```
MacBook-Pro-de-Mateus:microservice mateustymoniuk$ java -Dswarm.project.stage=production -Dswarm.bind.address=127.0.0.1 -jar
./order/target/order-swarm.jar -Dconsul.host=127.0.0.1:8500
[INFO] Using project stage: production
```

Fonte: Elaborado pelo autor, 2018

O serviço de pedido se registra no Consul também, conforme a figura 18 mostra:

Figura 18 - Registro do pedido no Consul

```
2018-05-07 03:39:13,873 INFO [org.wildfly.swarm.topology.consul.runtime.Advertiser] (MSC service thread 1-2) Registered serv
ice order:127.0.0.1:8230
```

Fonte: Elaborado pelo autor, 2018

E o último serviço a ser inicializado, o de usuário, mostrado pela figura 19:

Figura 19 - Subindo serviço de usuário

```
MacBook-Pro-de-Mateus:microservice mateustymoniuk$ java -Dswarm.project.stage=production -Dswarm.bind.address=127.0.0.1 -jar
./user/target/user-swarm.jar -Dconsul.host=127.0.0.1:8500
[INFO] Using project stage: production
```

Fonte: Elaborado pelo autor, 2018

E como os demais, se registra no Consul, como é mostrado na figura 20:

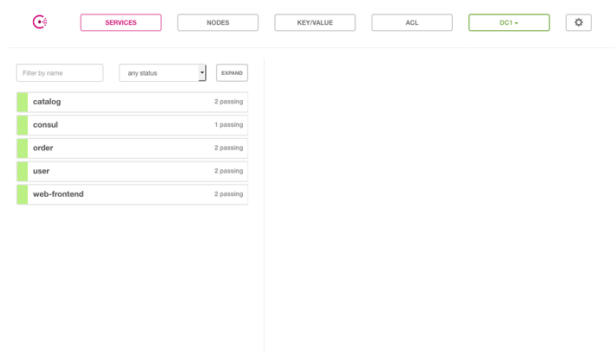
Figura 20 - Registro de usuário no Consul

```
2018-05-08 01:40:12,733 INFO [org.wildfly.swarm.topology.consul.runtime.Advertiser] (MSC service thread 1-1) Registered serv
ice user:127.0.0.1:8130
```

Fonte: Elaborado pelo autor, 2018

A figura 21 mostra a tela do Consul após o a inicialização de todos os serviços, com todos eles devidamente registrados no Consul.

Figura 21 - Consul com os quatro serviços registrados



Fonte: Elaborado pelo autor, 2018

Com os serviços implantados, é possível navegar pela aplicação. Ao visitar o endereço do *front-end*, tem-se a tela inicial da aplicação, conforme a figura 22, onde é possível verificar as funcionalidades de criar usuário, de catálogo e o carrinho, que irá chamar o pedido:

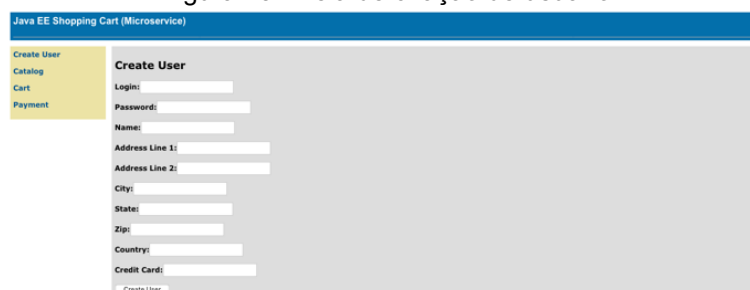
Figura 22 - Tela inicial da aplicação



Fonte: Elaborado pelo autor, 2018

Navegando entre as funcionalidades, a primeira do menu lateral é a de criar usuário, mostrada na figura 23:

Figura 23 - Tela de criação de usuário

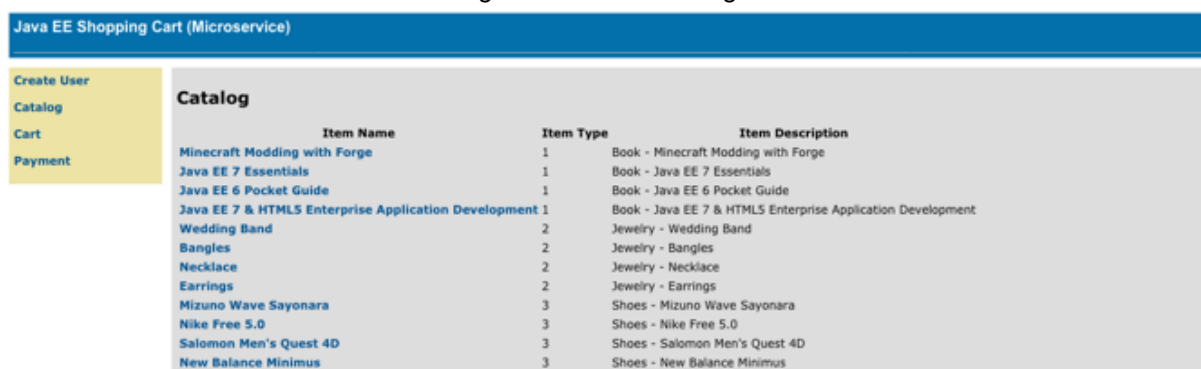


Fonte: Elaborado pelo autor, 2018

Ao inserir os dados da tela de cadastro de usuário, e clicar no botão, o serviço do *front-end* envia essas informações por meio de um método POST do HTTP para o serviço de usuário, que se encarrega de criar este usuário e salvar em seu banco.

A próxima funcionalidade do menu é a do catálogo.

Figura 24 - Tela catálogo



Item Name	Item Type	Item Description
Minecraft Modding with Forge	1	Book - Minecraft Modding with Forge
Java EE 7 Essentials	1	Book - Java EE 7 Essentials
Java EE 6 Pocket Guide	1	Book - Java EE 6 Pocket Guide
Java EE 7 & HTML5 Enterprise Application Development	1	Book - Java EE 7 & HTML5 Enterprise Application Development
Wedding Band	2	Jewelry - Wedding Band
Bangles	2	Jewelry - Bangles
Necklace	2	Jewelry - Necklace
Earrings	2	Jewelry - Earrings
Mizuno Wave Sayonara	3	Shoes - Mizuno Wave Sayonara
Nike Free 5.0	3	Shoes - Nike Free 5.0
Salomon Men's Quest 4D	3	Shoes - Salomon Men's Quest 4D
New Balance Minimus	3	Shoes - New Balance Minimus

Fonte: Elaborado pelo autor, 2018

Os dados da tela do catálogo, mostrados na figura 24, são provenientes do serviço de catálogo. Ao entrar nesta tela, o *front-end* requisita ao serviço os itens cadastrados para poder apresentá-los.

O carrinho é uma tela do *front-end* que agrupa os itens selecionados do catálogo para um posterior envio ao serviço de pedido. A figura 25 mostra o carrinho vazio, ou seja, quando não é selecionado nenhum item do catálogo.

Figura 25 - Tela carrinho vazio



Item Name	Item Count

Continue Shopping Checkout

Fonte: Elaborado pelo autor, 2018

Até o momento, apenas o serviço de catálogo foi chamado, então, deve-se criar um usuário para então fazer uma requisição ao serviço de usuário, e então selecionar um item do catálogo, adicionando-o ao carrinho e finalizando o pedido, para que seja feita a inserção no serviço de pedido.

Deve-se criar um usuário e checar se ele foi adicionado com sucesso no serviço de usuário, conforme mostra a tela na figura 26:

Figura 26 - Tela usuário criado com sucesso



Fonte: Elaborado pelo autor, 2018

O próximo passo então é selecionar um item do catálogo para poder efetuar um pedido. Ao clicar em um item do catálogo, é apresentada a tela, mostrada pela figura 27, para adicionar o item ao carrinho, selecionando a quantidade desejada:

Figura 27 - Tela selecionando um item



Fonte: Elaborado pelo autor, 2018

Selecionada a quantidade, e adicionando o item ao carrinho, ao navegar até a tela do carrinho é possível encontrar o item selecionado como na figura 28:

Figura 28 - Tela carrinho com um item



Fonte: Elaborado pelo autor, 2018

Antes de fazer o pedido, existe a tela de *checkout*, mostrada pela figura 29, para que o usuário verifique os dados da entrega e do cartão de crédito:

Figura 29 - Tela checkout

Fonte: Elaborado pelo autor, 2018

Confirmando o pedido no *checkout*, é enviado o registro ao serviço de pedido, então ele apresenta a tela com o status do pedido, e uma contagem de todos os pedidos feitos, como mostrado na figura 30:

Figura 30 - Tela de pedidos

Fonte: Elaborado pelo autor, 2018

Para verificar o funcionamento do Hystrix, no que diz respeito à recuperação caso haja alguma falha, e do Ribbon como balanceador de carga, será criado como mostrado na figura 31, outro serviço de catálogo em um outro endereço:

Figura 31 - Inicializando outro serviço de catálogo

```
MacBook-Pro-de-Mateus:microservice mateustymoniuk$ java -Dswarm.project.stage=production -Dswarm.bind.address=127.0.0.1 -Dswarm.port.offset=200 -jar ./catalog/target/catalog-swarm.jar -Dconsul.host=127.0.0.1:8500
[INFO] Using project stage: production
```

Fonte: Elaborado pelo autor, 2018

Novamente, o serviço se registra no Consul, como mostrado na figura 32:

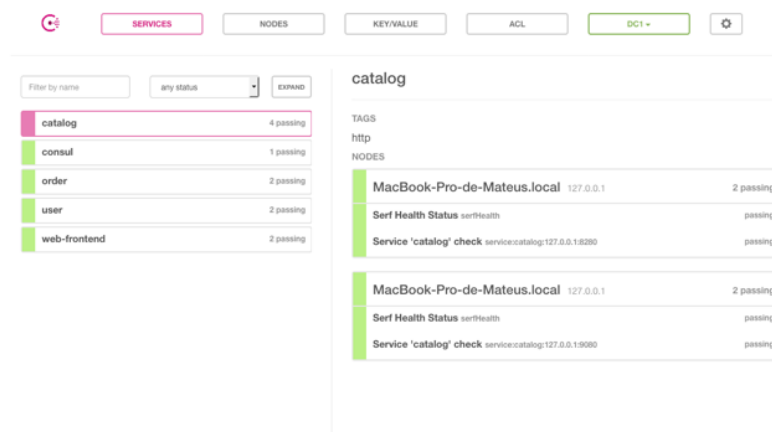
Figura 32 - Registro do segundo catálogo no Consul

```
2018-05-08 02:05:35,926 INFO [org.wildfly.swarm.topology.consul.runtime.Advertiser] (MSC service thread 1-6) Registered service catalog:127.0.0.1:8280
```

Fonte: Elaborado pelo autor, 2018

Após a inicialização do segundo serviço de catálogo, é possível checar se os dois estão registrados no Consul, conforme a figura 33 mostra:

Figura 33 - Dois catálogos registrados no Consul



Fonte: Elaborado pelo autor, 2018

Ao clicar no menu de catálogos, o Ribbon verifica e lista agora, as duas instâncias do serviço como mostrado na figura 34:

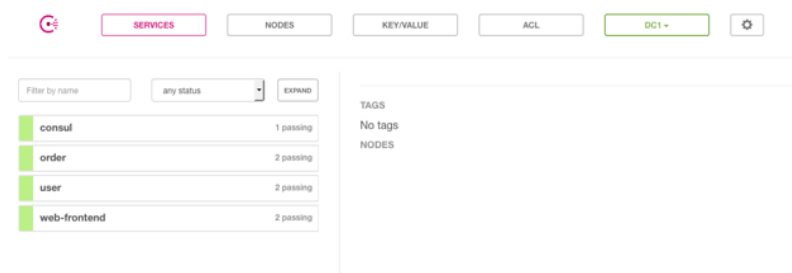
Figura 34 - Serviços de catálogo listados pelo Ribbon

```
2018-05-08 02:13:08,131 INFO [com.netflix.loadbalancer.BaseLoadBalancer] (default task-1) Client:catalog instantiated a Load
Balancer:DynamicServerListLoadBalancer:{NFLoadBalancer:name=catalog,current list of Servers=[],Load balancer stats=Zone stats
: {},Server stats: []}ServerList:null
2018-05-08 02:13:08,137 INFO [com.netflix.loadbalancer.DynamicServerListLoadBalancer] (default task-1) DynamicServerListLoad
Balancer for client catalog initialized: DynamicServerListLoadBalancer:{NFLoadBalancer:name=catalog,current list of Servers=[
127.0.0.1:9080, 127.0.0.1:8280],Load balancer stats=Zone stats: {unknown=[Zone:unknown; Instance count:2; Active connec
tions count: 0; Circuit breaker tripped count: 0; Active connections per server: 0.0;]
},Server stats: [[Server:127.0.0.1:8280; Zone:UNKNOWN; Total Requests:0; Successive connection failure:0; T
otal blackout seconds:0; Last connection made:Wed Dec 31 21:00:00 BRT 1969; First connection made: Wed Dec 31 21:
00:00 BRT 1969; Active Connections:0; total failure count in last (1000) msec:0; average resp time:0.0; 90 percentile
resp time:0.0; 95 percentile resp time:0.0; min resp time:0.0; max resp time:0.0; stddev resp time:0.0]
, [Server:127.0.0.1:9080; Zone:UNKNOWN; Total Requests:0; Successive connection failure:0; Total blackou
t seconds:0; Last connection made:Wed Dec 31 21:00:00 BRT 1969; First connection made: Wed Dec 31 21:00:00 BRT 1969;A
ctive Connections:0; total failure count in last (1000) msec:0; average resp time:0.0; 90 percentile resp time:0.0;9
5 percentile resp time:0.0; min resp time:0.0; max resp time:0.0; stddev resp time:0.0]
]}ServerList:org.wildfly.swarm.netflix.ribbon.runtime.TopologyServerList@1b7a7c1e
```

Fonte: Elaborado pelo autor, 2018

Agora, para mostrar o funcionamento do Hystrix, serão finalizadas as duas instâncias do serviço de catálogo. A figura 35 mostra o Consul sem nenhum serviço de catálogo registrado:

Figura 35 - Catálogos finalizados



Fonte: Elaborado pelo autor, 2018

E ao fazer uma requisição ao catálogo, o Ribbon identifica que não há mais instâncias do serviço, e o Hystrix devolve a lista correspondente à recuperação do serviço que ele guardou, o que pode ser visto na figura 36:

Figura 36 - Recuperação do catálogo pelo Hystrix

```
2018-05-08 02:18:17,118 INFO [com.netflix.loadbalancer.BaseLoadBalancer] (default task-5) Client:catalog instantiated a Load
Balancer:DynamicServerListLoadBalancer:{NFLoadBalancer:name=catalog,current list of Servers=[],Load balancer stats=Zone stats
: {},Server stats: []}ServerList:null
2018-05-08 02:18:17,128 INFO [com.netflix.loadbalancer.DynamicServerListLoadBalancer] (default task-5) DynamicServerListLoad
Balancer for client catalog initialized: DynamicServerListLoadBalancer:{NFLoadBalancer:name=catalog,current list of Servers=
[],Load balancer stats=Zone stats: {},Server stats: []}ServerList:org.wildfly.swarm.netflix.ribbon.runtime.TopologyServerList@
170e9015
2018-05-08 02:18:17,131 WARN [com.netflix.loadbalancer.RoundRobinRule] (default task-5) No up servers available from load ba
lancer: DynamicServerListLoadBalancer:{NFLoadBalancer:name=catalog,current list of Servers=[],Load balancer stats=Zone stats:
 {},Server stats: []}ServerList:org.wildfly.swarm.netflix.ribbon.runtime.TopologyServerList@170e9015
2018-05-08 02:18:17,132 WARN [com.netflix.loadbalancer.RoundRobinRule] (default task-5) No up servers available from load ba
lancer: DynamicServerListLoadBalancer:{NFLoadBalancer:name=catalog,current list of Servers=[],Load balancer stats=Zone stats:
 {},Server stats: []}ServerList:org.wildfly.swarm.netflix.ribbon.runtime.TopologyServerList@170e9015
2018-05-08 02:18:17,133 WARN [com.netflix.loadbalancer.RoundRobinRule] (default task-5) No up servers available from load ba
lancer: DynamicServerListLoadBalancer:{NFLoadBalancer:name=catalog,current list of Servers=[],Load balancer stats=Zone stats:
 {},Server stats: []}ServerList:org.wildfly.swarm.netflix.ribbon.runtime.TopologyServerList@170e9015
2018-05-08 02:18:17,134 WARN [com.netflix.loadbalancer.RoundRobinRule] (default task-5) No up servers available from load ba
lancer: DynamicServerListLoadBalancer:{NFLoadBalancer:name=catalog,current list of Servers=[],Load balancer stats=Zone stats:
 {},Server stats: []}ServerList:org.wildfly.swarm.netflix.ribbon.runtime.TopologyServerList@170e9015
2018-05-08 02:18:17,135 INFO [stdout] (default task-5) << Serving fallback result list >>
```

Fonte: Elaborado pelo autor, 2018

5 CONCLUSÃO

O objetivo deste trabalho foi estudar a arquitetura de microsserviços, começando pela arquitetura monolítica, e passando pela evolução para o SOA e então microsserviços.

Foram mostradas características chave que uma aplicação construída em microsserviços necessita para que não haja problemas futuros quanto a sua implantação e manutenção, já que apesar de não criar um único ponto de falha, e não ser uma base de código única, a distribuição da aplicação em diversos pontos faz com que a preocupação com a comunicação seja crucial, agora que ela depende totalmente da rede, e que sejam revisadas sempre as fronteiras da aplicação, para determinar até onde uma funcionalidade deve ser gerenciada por um serviço, ou dividida em mais partes.

A forma de pensar e gerenciar a aplicação por parte do time de desenvolvedores deve mudar, já que é preferível que times multifuncionais cuidem dos serviços, para evitar que a aplicação seja separada em camadas que englobam funcionalidades comuns aos serviços. Cada time deve determinar como vai funcionar seu serviço, e eles cuidarão da sua manutenção e até do possível descarte, visto que, o código deve ser pequeno o bastante para que um serviço seja facilmente substituído.

Ao apresentar a aplicação, foi possível observar como os serviços são implantados e a forma como se comunicam, e a importância que é de se projetar pensando na falha de um componente.

Conclui-se que a arquitetura de microsserviços é uma maneira viável de se construir aplicações, tendo em mente que a criação das fronteiras dos serviços são tarefas complexas e que podem e devem mudar com o tempo à medida que a aplicação evolui, e que os serviços determinados por essas fronteiras não devem expor detalhes internos de sua execução, onde outros serviços apenas conhecem apenas as formas de invocar seus métodos.

TRABALHOS FUTUROS

A partir do projeto desenvolvido, pode-se destacar os possíveis projetos futuros:

- Criação de um serviço de autenticação para proteger os serviços e seus recursos
- Criar um API gateway para lidar com as requisições e cuidar do balanceamento de carga
- Colocar os serviços para rodar em containers
- Fazer a implantação dos serviços em máquinas individuais, e para isso, utilizar-se da automação do processo de implantação.

REFERÊNCIAS

BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. **Software Architecture in Practice**. 2. ed. Boston: Pearson Education, Inc, 2003.

BRAUN, Heiko. **Container API, Fractions and Deployments**. 2016a. il. Disponível em: <<http://wildfly-swarm.io/tutorial/step-3/>>. Acesso em: 20 abr. 2018.

BRAUN, Heiko. **Right-size your runtime**. 2016b. il. Disponível em: <<http://wildfly-swarm.io/tutorial/step-2/>>. Acesso em: 20 abr. 2018.

BURDETT, David; KAVANTZAS, Nickolas. WS Choreography Model Overview. 2004. Disponível em: <<https://www.w3.org/TR/ws-chor-model/>>. Acesso em: 31 maio 2018.

CHRISTENSEN, Ben. **Hystrix: Defend your app**. 2013. Disponível em: <<https://github.com/Netflix/Hystrix/wiki>>. Acesso em: 06 abr. 2018.

COULOURIS, George et al. **Sistemas Distribuídos: Conceitos e Projeto**. 5. ed. Porto Alegre: Bookman, 2013.

ERL, Thomas. **SOA: Princípios de design de serviços**. São Paulo: Pearson Prentice Hall, 2009.

FOWLER, Martin. **StranglerApplication**. 2004. Disponível em: <<https://www.martinfowler.com/bliki/StranglerApplication.html>>. Acesso em: 05 set. 2017.

GARLAN, David; SHAW, Mary. **An Introduction to Software Architecture: advances in Software Engineering and Knowledge Engineering**. New Jersey: World Scientific Publishing Company,, 1993.

LEWIS, James; FOWLER, Martin. **Microservices: a definition of this new architectural term**. 2014. Disponível em: <<https://martinfowler.com/articles/microservices.html>>. Acesso em: 5 out. 2017.

NADAREISHVILI, Irakli et al. **Microservice Architecture: Aligning Principles, Practices and Culture**. Sebastopol: O'reilly Media, Inc., 2016.

NATIS, Yefim V.. **Software-Defined Architecture: Application Design for Digital Business**. 2014. Disponível em: < <https://www.gartner.com/webinar/2698619/>>. Acesso em: 30 out. 2017.

NEWMAN, Sam. **Building Microservices: Designing fine-grained systems**. Sebastopol: O'reilly Media, Inc., 2015.

PAHL, Claus; JAMSHIDI, Pooyan. Microservices: A Systematic Mapping Study. **Proceedings Of The 6th International Conference On Cloud Computing And Services Science**, [s.l.], p.137-146, 16 maio 2016. SCITEPRESS - Science and Technology Publications. Acesso em: 5 out. 2017.

PEYROTT, Sebastián. **An Introduction to Microservices, Part 3: The Service Registry: How the service registry works in a microservice-based architecture**. 2015. Disponível em: <<https://auth0.com/blog/an-introduction-to-microservices-part-3-the-service-registry/>>. Acesso em: 20 fev. 2018.

PRESSMAN, Roger S.. **Software Engineering: A practitioner's approach**. 7. ed. New York: Mcgraw-hill, 2010.

RAYMOND, Eric Steven. **The Art of Unix Programming**. Boston: Pearson Education, Inc, 2004.

RICHARDS, Mark. **Software Architecture Patterns**: Understanding Common Architecture Patterns and When to Use Them. Sebastopol: O'reilly Media, Inc., 2015.

RICHARDSON, Chris. **Pattern: Service registry**. 2017. Disponível em: <<http://microservices.io/patterns/service-registry.html>>. Acesso em: 16 abr. 2018.

SHARMA, Sourabh. **Mastering Microservices with Java**: Master the art of implementing microservices in your production environment with ease. Birmingham: Packet Publishing, 2016.

SHAW, Mary; GARLAN, David. **Software Architecture**: perspectives on an emerging discipline. New Jersey: Prentice Hall, Inc, 1996.

SHAW, Mary; GARLAN, David. Formulations and formalisms in software architecture. **Computer Science Today**, [s.l.], p.307-323, 1995. Springer Berlin Heidelberg.

TANEMBAUM, Andrew S.; VAN STEEN, Maarten. **Sistemas Distribuídos**: princípios e paradigmas. 2. ed. São Paulo: Pearson Prentice Hall, 2007.

THONES, Johannes. Microservices. **Ieee Software**, [s.l.], v. 32, n. 1, p.116-116, jan. 2015. Institute of Electrical and Electronics Engineers (IEEE). Disponível em: <<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7030212>>. Acesso em: 20 set. 2017.

TYREE, J.; AKERMAN, A.. Architecture Decisions: Demystifying Architecture. **Ieee Software**, [s.l.], v. 22, n. 2, p.19-27, mar. 2005. Institute of Electrical and Electronics Engineers (IEEE).

WANG, Allen. **Ribbon wiki**. 2013. Disponível em: <<https://github.com/Netflix/ribbon/wiki>>. Acesso em: 09 abr. 2018.

WATSON, Coburn; EMMONS, Scott; GREGG, Brendan. **A Microscope on Microservices**. 2015. Disponível em: <<https://medium.com/netflix-techblog/a-microscope-on-microservices-923b906103f4>>. Acesso em: 06 abr. 2018.