Do monolítico ao microserviço

A transformação das equipes e sistemas

Gabriel Bugés Garcia[1], Carlos Pagani[2]

Instituto Federal de Educação, Ciência e Tecnologia de São Paulo (IFSP)

Campus Hortolândia – SP – Brasil

ggarcia0298@gmail.com, pagani@ifsp.edu.br

Abstract. The need for safe, functional and rapidly developing applications forced some changes in the software development model, both in management aspects and in architectures, methodologies and tools. Microservices, Cloud Computing, DevOps and Agile Methodologies are essential points in this transformation. In the course of this paper, the reader will delve deeper into the themes mentioned and find a practical guide for migration between monolithic architectures and microservices, with theoretical basis and examples of best practices.

Resumo. A necessidade de aplicações seguras, funcionais e com rápido desenvolvimento forçou algumas mudanças no modelo de desenvolvimento de software, tanto em aspectos gerenciais como em arquiteturas, metodologias e ferramentas. Podemos destacar as tecnologias como os micros serviços, Computação em Nuvem, DevOps e Metodologias Ágeis como pontos essenciais nessa mudança. Este trabalho se aprofunda no tema de micros serviços e elabora um guia prático para migração baseado em fontes teóricas e estudo de caso, explicitando exemplos de melhores práticas.

1. Introdução e Motivação

Devido à crescente demanda por armazenamento de dados, segurança, processamento e disponibilidade, a Computação em Nuvem vem se tornando cada vez mais presente como infraestrutura, serviço e plataforma. Isto fez com que diversas empresas como Amazon, Microsoft e IBM lançassem seus serviços de nuvem. [Muhammad Raza, 2020].

O crescimento da Computação em Nuvem alinhado às práticas DevOps trouxeram visibilidade a outro tópico, os micros serviços. Trata-se de uma arquitetura que vêm sendo aderida em massa por grandes empresas como Amazon e Netflix [Chris Richardson, 2017], visando a substituição de aplicações monolíticas para alcançar maior segurança, qualidade de código e escalabilidade. Essa mudança pode ser sentida também na forma organizacional das empresas.

Como mencionado no livro *Microservices for Java Developers* [Rafael Benevides e Christian Posta], os clientes demandam um serviço cada vez mais personalizado das empresas de tecnologia, e para suprir a entrega de softwares ágeis é necessário começar com uma mudança organizacional. Além de fazer citação a lei de Conway, dizendo que nenhuma pesquisa sobre microsserviços está completa sem partir do ponto que "Organizações que desenvolvem sistemas de software tendem a produzir sistemas que são cópias das estruturas de comunicação dessas organizações".

O desenvolvimento deste trabalho tem como motivação produzir um guia de migração com pontos de ação para transformar um sistema monolítico em um sistema de microsserviços, exibindo a necessidade de uma mudança na estrutura organizacional e na forma de pensar ao desenvolver uma nova aplicação.

2. **Fundamentação Teórica**

Nesse tópico serão fundamentados conceitos essenciais para a compreensão deste trabalho.

2.1. **Microsserviços**

A proposta da arquitetura de sistemas orientada a microsserviços é desenvolver sistemas que sejam mais flexíveis, escaláveis e com manutenção mais simples do que as arquiteturas de sistemas monolíticas normalmente utilizadas. A filosofia desta arquitetura é baseada em realizar apenas uma tarefa, mas com maestria, onde os serviços são focados em realizar uma única função [Marcos Machado, 2018]. Trata-se de uma abordagem onde cada componente do sistema é desenvolvido separadamente, cada um executando seu próprio processo e se comunicando para formarem em conjunto a aplicação final.

> *Em 2022, as arquiteturas de microsserviços darão suporte a 90 por cento de todos os aplicativos - melhorando a capacidade de projetar, depurar, atualizar e aproveitar o código de terceiros (de acordo com o IDC). Os microsserviços e a conteinerização de aplicativos fornece escalabilidade de operação independente, disponibilidade de sistema incomparável e lançamentos rápidos de novos serviços sem grandes reconfigurações. [Cisco, 2019].*

2.2. **Arquitetura Monolítica**

Monolítico: Que consta de uma única pedra [Dicionário Aurélio, 2015].

A proposta da arquitetura monolítica tem como fundamento compartilhar recursos entre os módulos, onde toda a modularização ocorre na mesma máquina. Os módulos compartilham recursos de processamento, memória, bancos de dados e arquivos. Toda a sua lógica para lidar com uma solicitação é executada em um único processo [Fowler, 2015].

Segundo James Lewis as aplicações monolíticas são construídas em três partes principais: a interface de usuário do lado do cliente (que consiste em páginas HTML e JavaScript executadas em um navegador na máquina do usuário), um banco de dados (que consiste em muitas tabelas inseridas em um sistema de gerenciamento de banco de dados comum, geralmente relacional), e um aplicativo do lado do servidor. O aplicativo do lado do servidor lida com as solicitações HTTP, executa a lógica do domínio, recupera e atualiza dados do banco de dados, seleciona e preenche as visualizações HTML a serem enviadas para o navegador. Esse aplicativo do lado do servidor é monolítico - um executável lógico único.

2.3. **DevOps**

O DevOps é a combinação de filosofias culturais, práticas e ferramentas que aumentam a capacidade de uma empresa de distribuir aplicativos e serviços em alta velocidade: aperfeiçoando produtos em um ritmo mais rápido do que o das empresas que usam processos tradicionais de desenvolvimento de software e gerenciamento de infraestrutura [Amazon, 2017]. Segundo Carlos Mattos (Diretor Regional da Microsoft), no lugar de times de desenvolvimento e infraestrutura isolados, surge a abordagem DevOps, na qual as duas equipes compartilham responsabilidades e trabalham em conjunto, acelerando drasticamente o processo de desenvolvimento. Isso possibilita identificar e corrigir problemas de performance mais rapidamente.

2.4. **Computação em nuvem**

Computação em nuvem trata-se de uma tecnologia que permite acesso remoto a softwares, armazenamento de arquivos e processamento de dados por meio da internet, sendo assim uma alternativa à execução em um computador pessoal ou servidor local.

> *A computação em nuvem é a entrega sob demanda de poder computacional, armazenamento de banco de dados, aplicações e outros recursos de TI por meio de uma plataforma de serviços de nuvem via Internet com uma definição de preço conforme o uso [Amazon, 2017].*

Conceitos de Computação em Nuvem foram essenciais para o desenvolvimento do trabalho, sendo que representa aspectos cruciais sobre a computação e as metodologias de trabalho, sendo esta inovação um dos grandes facilitadores do surgimento da metodologia DevOps.

2.6. **Lei de Conway**

Melvin Edward Conway é um cientista e programador que desenvolveu o que ficou conhecido como Lei de Conway: **“**Organizações que desenvolvem sistemas de software tendem a produzir sistemas que são cópias das estruturas de comunicação dessas organizações” [Melvin E. Conway, 1968].

A lei de Conway foi necessária para explicar mudanças vividas no gerenciamento de equipes e sua função dentro das metodologias ágeis, uma vez que a reestruturação das equipes foi necessária para atender a demanda do mercado.

2.6. **Docker**

Docker é um sistema de virtualização que possibilita o empacotamento de uma aplicação ou pacote inteiro dentro de um container, que pode ser utilizado em qualquer ambiente que possua a ferramenta instalada [Cristiano Diedrich, 2015].

Um de seus objetivos é reduzir o tempo de *deploy,* complexidade da infraestrutura e problemas originados de diferenças entre ambientes, o ambiente de desenvolvimento se torna muito mais parecido ao ambiente de produção quando usado docker.

3. **Metodologia**

Para que o trabalho pudesse ser desenvolvido com maestria a primeira atividade realizada foi a definição de um cronograma a ser seguido e definição de pontos que serão abordados na pesquisa.

O desenvolvimento e estruturação da pesquisa foi executado com o auxílio de da ferramenta Trello, possibilitando a aplicação do Kanban de forma eletrônica, sendo que este é um facilitador para uma melhor visualização dos pontos executados e planejamento de ações posteriores.

Após a obtenção e fundamentação dos conhecimentos necessários foi então escrito o artigo, levantando temas abordados na introdução, modificando o Kanban a cada passo dado.

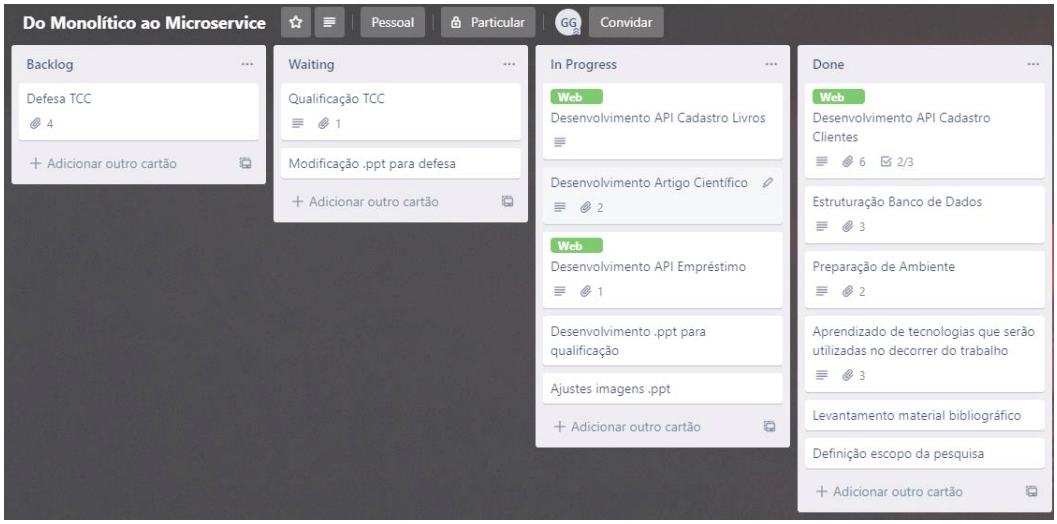

Figura 1. Kanban criado na ferramenta Trello.

No decorrer do trabalho foram utilizadas três ferramentas principais para executar o desenvolvimento, integração e testes da aplicação. Sendo elas: Spring tool Suite (IDE onde foi desenvolvido a código da aplicação), Postman (aplicação que simplifica chamadas HTTP, mais detalhes no decorrer do trabalho) e PgAdmin (*Client* SQL onde é possível gerenciar o banco de dados e suas tabelas). As ferramentas serão mais detalhadas no tópico 10.1 deste trabalho.

Tabela 1. Ferramentas utilizadas.

Ferramenta	Funcionalidade
Spring tool Suite	IDE para codificação da aplicação
Postman	Testes da aplicação através de chamadas HTTP

PgAdmin	Gerenciamento do banco de dados
Trello	Ferramenta Kanban para organizar as atividades

4. **Contextualização do cenário atual**

O avanço da tecnologia afeta diretamente a vida dos profissionais ligados a ela, dentre os desafios cotidianos podemos destacar: a necessidade de entregar mais em menos tempo, sistemas com uma disponibilidade cada vez mais alta, modelos de segurança aprimorados, uma interface simples e chamativa e outros pontos exigidos por um mercado cada vez mais concorrido. Conceitos que outrora foram considerados sólidos caem em questionamento com o surgimento de novas arquiteturas e maneiras de se desenvolver. O mercado mudou, segundo a empresa SiriusDecisions 67% de todas as compras realizadas são feitas pela internet [Carolyn Galvin, 2018], as *fintechs* (conhecidas como bancos digitais) forçaram os bancos consolidados no mercado a se adaptarem digitalmente, as plataformas de ensino a distância (EAD) cresceram 17,6% entre 2016 e 2017 e alcançaram cerca de 1,8 milhão de alunos [Talita Cavalcante, 2018].

Para suprir tamanha demanda, algumas ferramentas e metodologias foram desenvolvidas ou resgatadas, a velocidade em que são aplicadas geram dúvidas entre os que estão pouco familiarizados e muitas vezes não é visível a ligação entre os processos utilizados. No decorrer do trabalho estes processos e tecnologias serão apresentados em forma de guia para uma possível migração.

5. **Arquitetura e equipes no monolítico**

Um dos padrões de arquitetura mais consolidados no mercado e amplamente utilizado é o Monolítico[Martin Fowler, 2014]. As aplicações monolíticas podem ser separadas em três partes principais: Interface de usuário, em desenvolvimento Web podemos exemplificar como JavaScript ou HTML rodando no navegador do usuário. Banco de dados, geralmente relacionais onde ficam armazenados todos os tipos de dados referentes a aplicação. *Server-side*, onde a aplicação lida com o servidor através de requisições HTTP, acionando serviços e populando a Interface de usuário. No que se diz respeito à aplicação que roda no servidor temos um único executável, quando uma alteração é realizada torna-se necessário fazer uma nova compilação e *deploy.*

Este modelo arquitetural implica geralmente na criação de especialistas, onde os componentes da equipe se especializam em uma das três partes principais, gerando assim profissões como administradores de banco de dados, desenvolvedores focados na experiência de usuário, desenvolvedores focados em middlewares, engenheiros de software entre outros. Devido a esse modelo as equipes têm um tamanho médio de 15 pessoas [Alexander Sergeev, 2016] e se organizam geralmente seguindo o modelo cascata.

6. **Reestruturação das equipes - Lei de Conway**

Para dar início na transformação digital a estruturação das equipes que atendem a projetos necessita ser reformada. Uma das metodologias resgatas e adaptadas para esta execução tem como fundamento a Lei de Conway, criada e apresentada por Melvin E. Conway através do livro *How do Committees Invent* de 1968. Sua lei pode ser resumida da seguinte forma: "Qualquer empresa que projeta um sistema inevitavelmente produz um projeto cuja estrutura é uma cópia da estrutura de comunicação da organização".

Com base nesse pensamento (aprimorado pela metodologia ágil, que será apresentada posteriormente neste trabalho) as equipes devem ser divididas em grupos menores. A organização de especialistas (figura 2) deve ser desfeita para que sua estrutura se aproxime mais do produto final, o microserviço.

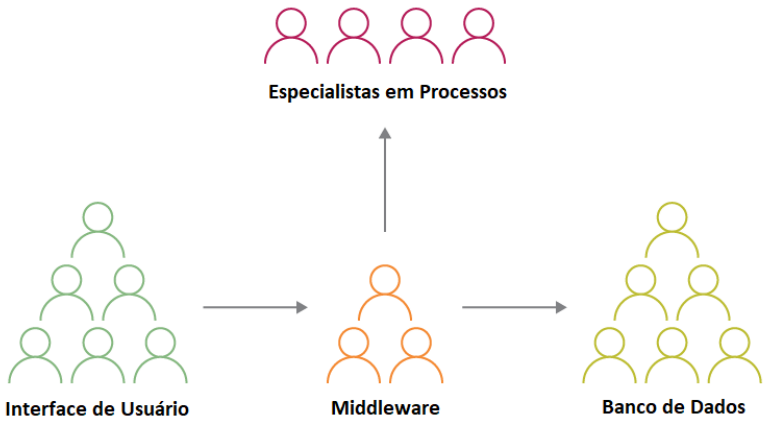

Figura 2. Equipes focadas em especialistas.

7. **Reestruturação das equipes - Ágil**

Em meados dos anos 2000 o debate sobre a necessidade de entregas mais rápidas estava em seu auge, os processos estavam lentos e o mercado exigindo velocidade e qualidade nas entregas. Para se adaptar a essa necessidade um grupo de pessoas que envolvia CEOs, criadores de metodologias e processos juntos a cientistas ligados à tecnologia se reuniram nos Estados Unidos para debaterem ideias, o que resultou em um manifesto apelidado posteriormente de Manifesto Ágil.

O manifesto possui 4 pontos principais que serviram de base para desencadear a transformação em processos e gerenciamento de equipes, estes fundamentos são: Indivíduos e interações mais que processos e ferramentas, software em funcionamento mais que documentação abrangente, colaboração com o cliente mais que negociação de contratos, responder a mudanças mais que seguir um plano [Manifesto Ágil, 2001].

Para se manter competitivo é necessária uma reestruturação nas equipes (figura 4). Times que possuem mais de doze pessoas não são aptos para aplicação da metodologia ágil de forma eficaz, sendo assim as equipes devem ser reduzidas e novas funções criadas visando uma maior autonomia e redução de processos, o tamanho ideal das equipes passa a ser de 5 a 9 pessoas.

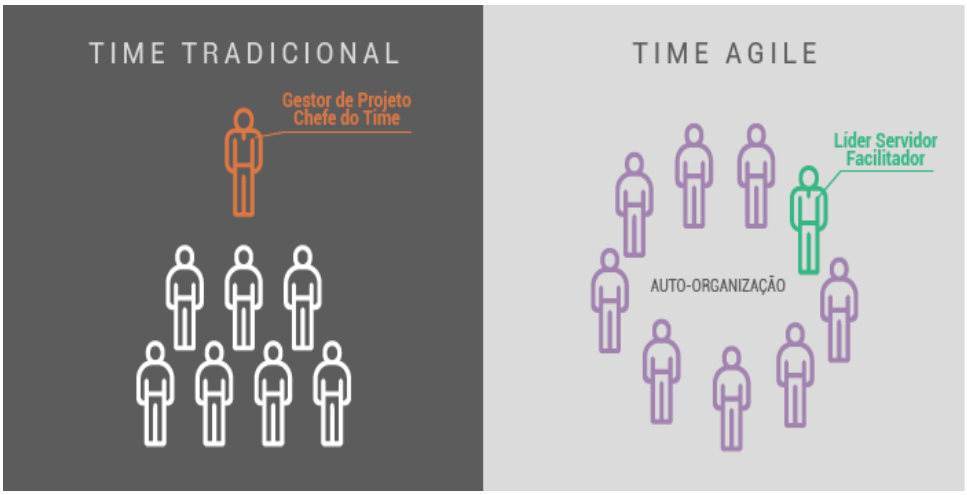

Figura 4 - Reestruturação de equipes no Ágil [Jan Kuyper, 2017].

8. **Estudo de Caso**

Após a reestruturação das equipes, o próximo passo referente ao guia de migração está ligado à reestruturação da aplicação. A divisão de grandes equipes em pequenos grupos facilitou a modularização de sistemas complexos, o que consiste em dividir sistemas monolíticos em módulos menores (microsserviços) e atribuir equipes responsáveis para cada um deles.

Para melhor ilustrar esse processo foi desenvolvido a modularização de um sistema bibliotecário monolítico que possui as seguintes funções: Gerenciamento de usuário, gerenciamento de livro e empréstimo. Foi utilizado a linguagem Java devido a sua popularidade (figura 5) com o auxílio do framework Spring Boot, ideal para o desenvolvimento de aplicações em microsserviços. Por não ser o foco da pesquisa, aspectos sobre a linguagem e suas características técnicas não serão aprofundados no decorrer deste trabalho. Como banco de dados foi utilizado Postgres por ter sido utilizado durante as aulas práticas no decorrer do curso. Além das tecnologias citadas foi também utilizado o Angular para a criação de uma interface de usuário visando a ilustração das funcionalidades dos microsserviços. Salientando que o cenário é fictício e montado exclusivamente para ilustração deste trabalho.

May 2020	May 2019	Change	Programming Language	Ratings	Change
1	2	^	C	17.07%	+2.82%
2	1	v	Java	16.28%	+0.28%
3	4	^	Python	9.12%	+1.29%
4	3	v	C++	6.13%	-1.97%
5	6	^	C#	4.29%	+0.30%
6	5	v	Visual Basic	4.18%	-1.01%

Figura 5 - Índice TIOBE 2020.

8.1. **Estrutura da aplicação monolítica**

A aplicação legado foi construída em meados de 2008 utilizando tecnologias viáveis para a época. A interface de usuário foi desenvolvida utilizando Windows Forms, C# Framework 4.5 e como banco de dados MySQL.

Com o decorrer do tempo a aplicação sofreu alterações vindas de desenvolvedores diferentes, o que agravou em lentidão e maior complexidade do código gerando algumas barreiras para sua modernização.

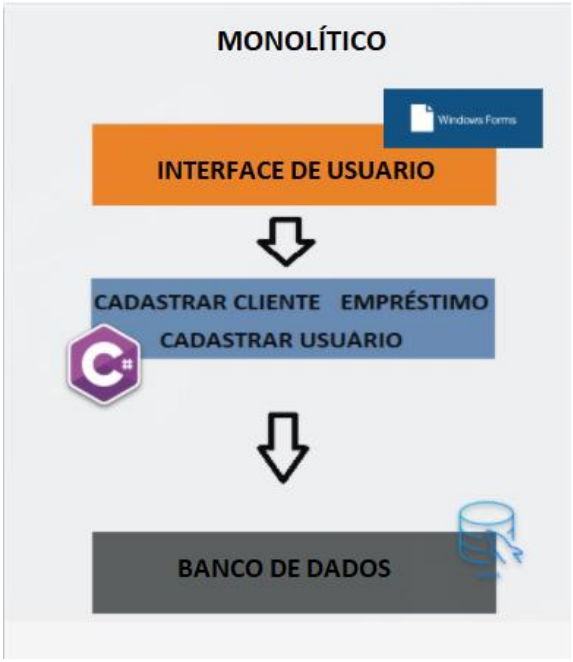

Figura 6 - Estrutura da aplicação.

Toda a aplicação está contida em um único banco de dados MySQL e organizado em duas tabelas diferentes, a primeira contendo informações de usuário e a segunda contendo informações de livros. O usuário pode cadastrar, alterar e excluir um cliente ou livro e também realiza um empréstimo ou devolução.

8.2. **A modularização para microsserviços**

Com a apresentação da aplicação foi possível identificar de maneira prática as três camadas essenciais desta arquitetura. Sendo elas: **Apresentação,** onde está a parte de interação com o usuário. **Negócio**, onde está contida toda a regra de negócio e inteligência da aplicação. **Dados**, onde é realizado todo o acesso a base de dados.

O primeiro passo para a realização da conversão de uma arquitetura monolítica para microsserviços é entender sua real necessidade, levando em conta os possíveis ganhos e dificuldades. O fato da arquitetura monolítica ser mais antiga não a torna ultrapassada e muito menos inutilizável, apenas uma abordagem com características diferentes. Além disso, é necessário estudar de forma minuciosa os gastos durante a migração, deve ser levado em conta o treinamento dos desenvolvedores, compreensão da regra de negócio como um todo, custos de infraestrutura e separação das funcionalidades de forma independente. Algumas das causas para a migração da aplicação proposta são:

- A cada alteração, toda a aplicação deve ser compilada novamente
- Dificuldade ao escalar aplicação
- Barreiras na adoção de novas tecnologias
- Dificuldades ao trabalhar com *deploy* contínuo.
- Tempo para *deploy* em nuvem.

8.3 **A estrutura das aplicações**

O monolítico foi dividido em três microsserviços e cada um deles possui responsabilidade específica. Pensando dessa forma foram criados os microsserviços gerenciar livro, gerenciar cliente e realizar empréstimo. Todos foram construídos utilizando a linguagem Java com framework Spring Boot, porém a linguagem não influencia o resultado final, tendo em vista que a comunicação é realizada através de requisições Http. Na figura 7 é possível encontrar um diagrama do ecossistema criado com a migração.

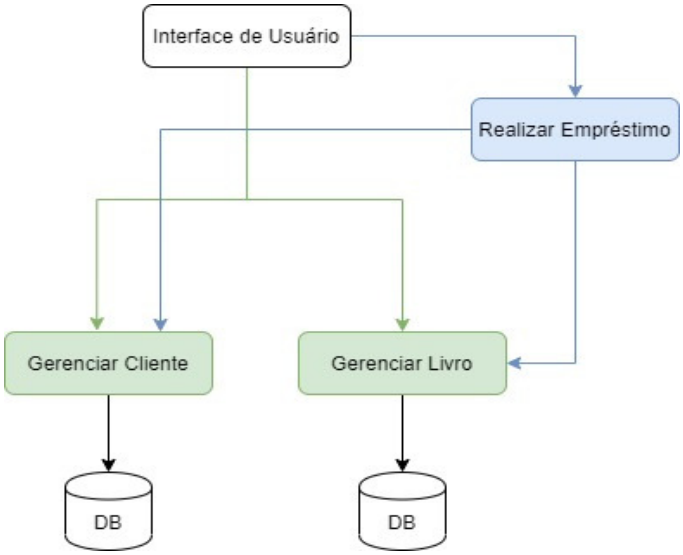

Figura 7 - Estrutura dos microsserviços.

Para a padronização da arquitetura foi utilizado o *Clean Architecture* desenvolvido por Robert Cecil Martin. A estrutura tem como objetivo organizar o código por camadas, definindo melhor a responsabilidade de cada nível e facilitando para uma possível migração no futuro.

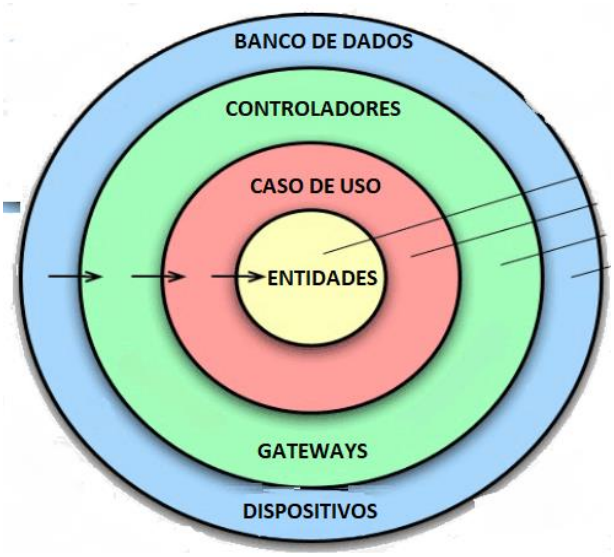

Figura 8 - Adaptado de Clean Architecture [Robert Cecil Martin, 2012].

As camadas internas não se comunicam com as externas, a camada *Entities* não se comunica com a *Use Cases* por exemplo, mas o contrário sim, a comunicação entre *Gateways* e UI é totalmente possível. Desta forma é necessário o desenvolvimento de interfaces que serão implementadas posteriormente pela regra de negócio, o que gera um maior desacoplamento e organização do código.

8.3.1. **Banco de dados**

Os serviços não possuem dependência entre si, possibilitando a utilização de tecnologias diferentes para cada um deles, o banco *clientes* está hospedado em um servidor Postgres enquanto o banco *livros* poderia estar hospedado em um servidor MySQL. Assim como nos microsserviços, a lógica do banco também está totalmente segregada por assunto e não possuem dependência entre si, sendo executadas sem um ponto de falha em comum.

Diferente da aplicação monolítica, cada serviço possui o seu próprio banco de dados, podendo estar espalhado em máquinas diferentes ou virtualizado em containers, uma prática comum é a criação de máquinas virtuais com o auxílio do Docker para a criação de instâncias de diversos bancos de dados, criando a infraestrutura necessária e eliminando a necessidade de máquinas robustas. Com o Docker é possível a criação de máquinas virtuais leves através de funcionalidades isoladas de um S.O. A figura 8 mostra uma instância de um banco Postgres rodando na porta 5432, porém poderiam haver diversas instâncias para todos os bancos de dados da aplicação.

```
Gabriel Garcia@DESKTOP-PLJRTK8 MINGW64 /f/Arquivos de Programas/Docker Toolbox
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
     NAMES
9fc91e5539f3        postgres            "docker-entrypoint.s…"   3 minutes ago       Up 3 minutes        5432/tcp
     nifty_gould
```

Figura 9 – Exemplo de instância Docker.

9. **Código da aplicação – Gerenciar Cliente**

Neste subtópico a aplicação será analisada em uma visão técnica, exibindo o código e lógica da aplicação. A estrutura e funcionamento das três APIs (ou microsserviços) são parecidos por serem construídos em *Clean Architecture*, para evitar a repetição apenas o fonte da API "gerenciar cliente" será abordado. Lembrando que a aplicação exibida é a forma final da migração do caso de uso da biblioteca e o código em sua versão completa está disponibilizado em um repositório Git e pode ser acessado pelo link https://gitlab.com/projeto-biblioteca.

As camadas da aplicação serão apresentadas seguindo a ordem da figura 8, partindo do centro e indo sentido a parte externa.

9.1. **Camada Entity**

Nesta camada as entidades da API são definidas, são objetos que serão tratados na aplicação, cada objeto possui apenas os atributos daquela entidade. A camada *Entity* é a central da arquitetura Clean (figura 8).

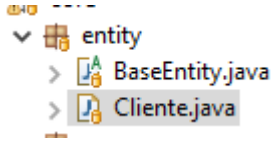

Figura 10 - Camada Entity API Gerenciar Cliente

A classe Cliente possui apenas construtores, como citado anteriormente. Abaixo estão alguns dos atributos desta classe.

```
private String nome;
private Long cpf;
private String telefone;
```

Outras classes da camada *Entity* podem ser encontradas no pacote *dataprovider*, onde as entidades de banco de dados são definidas, através destas entidades o banco de dados da aplicação é criado com o auxílio de injeções do Spring Boot, abstraindo a necessidade de maiores configurações.

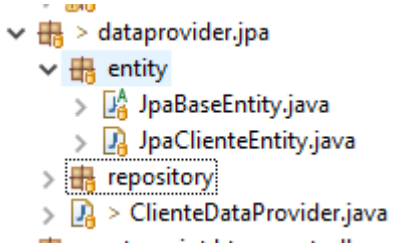

Figura 11 - Exemplo Entity de dataprovider

A classe é simples e conta com injeções Spring como é possível ver no trecho abaixo.

```
@Column(name = "nome", length = 40, nullable = false)
private String nome;
@Column(name = "cpf", length = 20, nullable = false)
private Long cpf;
```

9.2. **Camada UseCase**

Nesta camada as regras de negócio da API são definidas, dentro dela estão contidas as regras de negócio. Através dela podemos por exemplo buscar um cliente por CPF ou cadastrar um novo cliente. As classes do *UseCase* são uma implementação da camada Gateway (apresentada no próximo subtópico), que pode ser identificada dentro da arquitetura na figura 8.

Na figura 12 é possível encontrar um dos métodos da classe BuscarPorCpfUseCaseImpl, que está contida na camada *UseCase*. A classe apresentada busca clientes por CPF, realizando a chamada do fluxo do banco de dados e também fazendo tratamento de erros ou informações não encontradas.

```
@Override
public ClienteResponse executar(Long cpf) {

    ClienteResponse response = new ClienteResponse();
    response.setCpf(cpf);

    Optional<Cliente> opCliente = buscarPorCpfClienteGateway.buscarPorCpfCliente(cpf);

    if (opCliente.isPresent()) {
        response.setNome(opCliente.get().getNome());
        response.setId(opCliente.get().getId());
        response.setTelefone(opCliente.get().getTelefone());
        response.setDataNascimento(opCliente.get().getDataNascimento());
    } else {
        response.getResponse().adicionarErro(new ResponseDataErro("Cliente não encontrado com CPF " + cpf,
                ListaErroEnum.ENTIDADE_NAO_ENCONTRADA));
    }

    return response;
}
```

Figura 12 – Método que busca cliente por CPF

9.3. **Camadas Gateway e Controller**

São as camadas mais externas da API que ainda podem ser gerenciadas dentro da aplicação sem depender de contratos externos. O *Gateway* é responsável pelos contratos e interfaces a serem implementados pelo *UseCase,* ele não possui nenhuma amarração a tecnologia ou regra como podemos ver no exemplo abaixo.

```
public interface BuscarPorCpfClienteGateway {
  Optional<Cliente> buscarPorCpfCliente(Long cpf);
}
```

Já a camada *Controller* fica exposta para chamadas HTTP, através dela todo o fluxo da API é iniciado, recebe um request e retorna response a quem realizou sua chamada.

A classe ClienteController é a única dentro da camada, ela realiza o gerenciamento das requisições. Com a utilização do Spring através das notações *@GetMapping* e *@Postmapping* é possível definir o tipo de chamada HTTP que deve ser utilizado ao acessar determinado *entrypoint* (*entrypoint* é o método exposto para a chamada, o "ponto de entrada"). Segue exemplo da classe:

```
@GetMapping("/{cpf}")
Public ResponseEntity<ClienteResponse>
buscarPorCpf(@PathVariable("cpf") Long cpf) {
     ClienteResponse response =
buscaPorCpfUseCase.executar(cpf);
     return ResponseEntity.ok(response);
        }
```

9.4. **Interface de usuário**

A interface de usuário não faz parte do ecossistema da API, sendo um desenvolvimento apartado, porém pode ser representada na última camada da figura 8. Seu desenvolvimento foi realizado em Angular e realiza chamadas HTTP para a camada *Controller* (tópico 9.3). Na figura 13 é possível visualizar o resultado da interface.



Figura 13 – Interface de Usuário

Dentre as opções podemos realizar diversas opções, que se tornam uma espécie de gatilho para o *back-end.* Das chamadas HTTP apresentadas podemos destacar dois métodos utilizados pelo sistema:

- **GET:** O método GET solicita a representação de um recurso específico, temos como exemplo a funcionalidade Buscar Usuário;

- **POST:** O método POST é utilizado para submeter uma entidade a um recurso específico, temos como exemplo a funcionalidade Cadastrar Usuário.

10. **Testes e reprodutibilidade**

Neste subtópico abordaremos os testes funcionais realizados para validação das aplicações e sua reprodutibilidade. Será exemplificado apenas um entrypoint para que o trabalho não fique repetitivo, a estrutura e ferramentas utilizadas para todos os testes são as mesmas.

10.1 **Ferramentas**

Para facilitar a realização dos testes foi utilizado a ferramenta Postman (https://www.postman.com/) para realizar chamadas HTTP e a IDE Spring Tools 4 (https://spring.io/tools) para executar a aplicação em uma porta *localhost*. Para validação das alterações no banco de dados foi utilizado o pgAdmin III(https://www.pgadmin.org). Os códigos fontes das aplicações estão hospedados na plataforma GitLab, podendo ser acessados através do link https://gitlab.com/projeto-biblioteca.

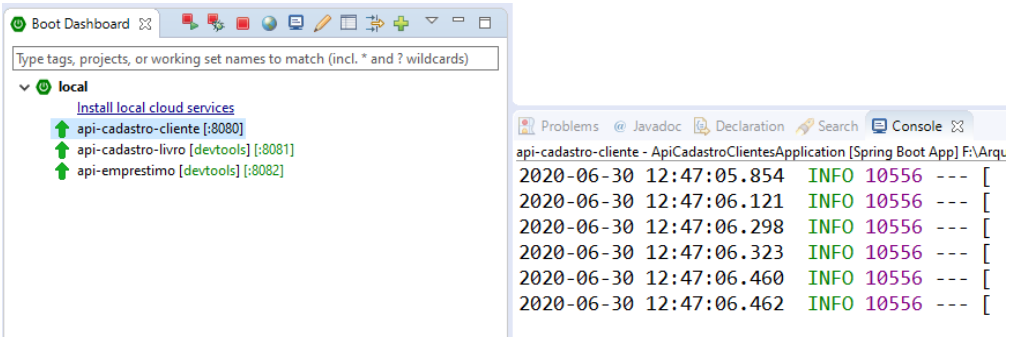

Figura 14 - APIs sendo executadas no Spring Tools 4 nas portas 8080, 8081 e 8082

Com as aplicações sendo executadas (figura 14) é possível dar início às chamadas via Postman, para isso é preciso especificar o endereço do *entrypoint* (tópico 9.3) e a porta.

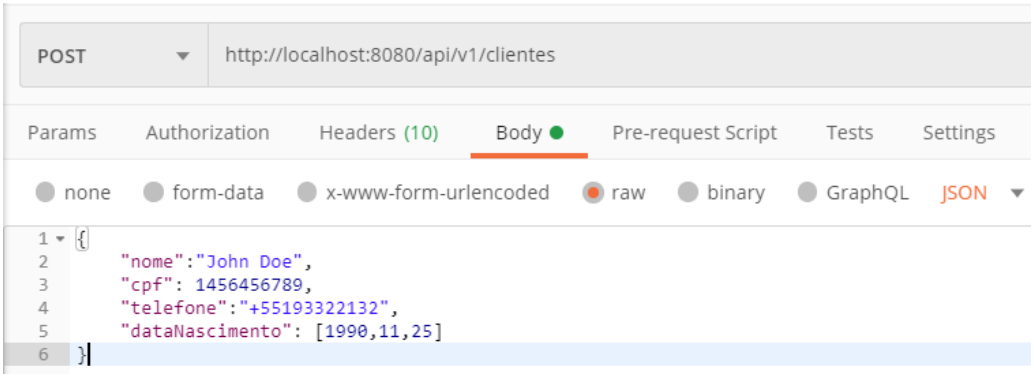

Figura 15 - Exemplo de chamada HTTP para realizar um novo cadastro através da API de Gerenciar Cliente

Como exibido anteriormente, a API Gerenciar Cliente está sendo executada na porta 8080, no corpo da requisição está sendo enviado as informações sobre o novo cliente, após isto a requisição pode ser submetida. É possível validar o status da requisição através do retorno da API.

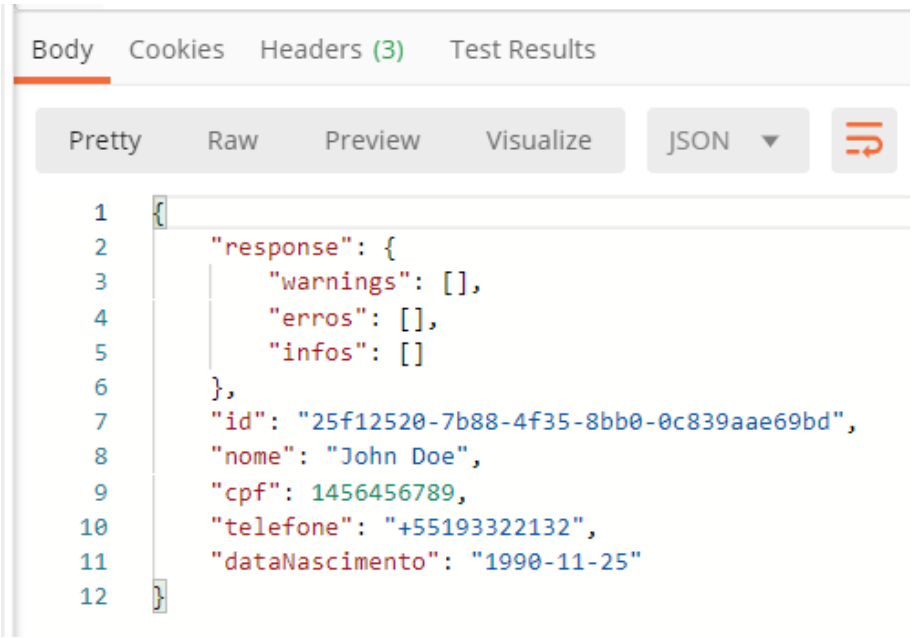

Figura 16 - Retorno da chamada

Retorno sem erros ou *warnings* com código 200, a aplicação não apresentou exceções. É possível realizar uma segunda validação através de uma query na tabela Cliente através do pgAdmin III, que retornará a mesma linha da figura 17 mostrando o registro inserido via requisição *post*.

Output pane

Data Output | Explain | Messages | History

	id character varying(40)	cpf bigint	data_nascimento timestamp without time zone	nome character varying(40)	telefone character varying(13)
1	25f12520-7b88-4f35-8bb0-0c839aae69bd	1456456789	1990-11-25 00:00:00	John Doe	+55193322132

Figura 17 - Novo cliente inserido no banco de dados

11. **Vantagens e desvantagens.**

Neste tópico serão abordados as vantagens e desvantagens de cada arquitetura. É importante ser mencionado que o objetivo da pesquisa não é a definição da melhor arquitetura e sim explorar e as características de cada uma, fornecendo ao leitor um embasamento sobre os ganhos e dificuldades em sua adoção, deixando a seu critério a decisão de realizar ou não a migração.

A figura 18 resgata as duas arquiteturas já exploradas anteriormente e exibe de forma gráfica a diferença entre as duas arquiteturas, onde pode ser ressaltado a diferença em que a lógica de negócio a o acesso ao banco de dados é realizado em cada uma das arquiteturas.

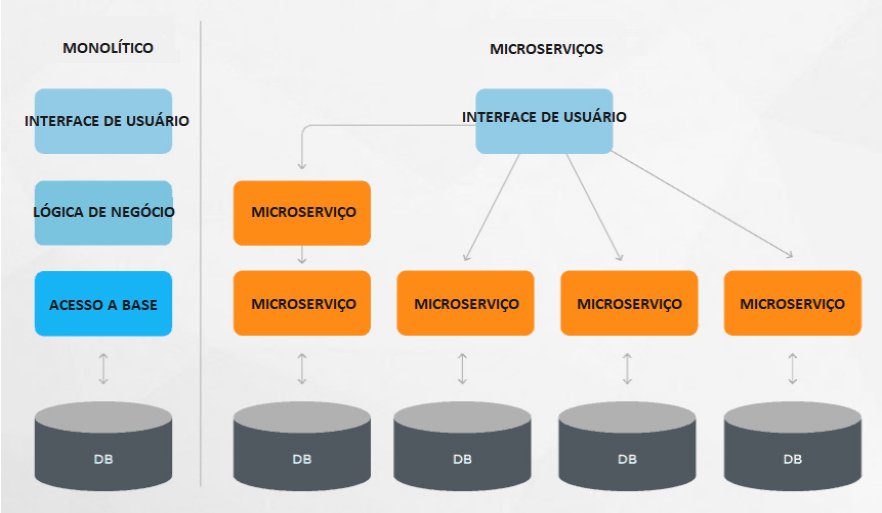

Figura 18 - Traduzido de Microservices vs. Monolithic Architectures

[Dan KUSNETZKY- 2017]

11.1. **Monolítico**

"Se não está quebrado, por que consertar? " - Robert Atkins.

A evolução do modo de desenvolver não significa que a arquitetura monolítica se tornou ultrapassada, mesmo sendo mais antiga ainda é possível utilizá-la em diversos cenários sendo desnecessário a migração desenfreada de todas as aplicações.

Baseado no artigo "microsserviços: dos grandes monólitos às pequenas rotas" de Lucas Santos e "*Microservices*" de Martin Fowler, o monolítico quando comparado a arquitetura em microsserviços possui as seguintes vantagens:

Simplicidade da arquitetura: A aplicação não possui muitas camadas, o que torna seu desenvolvimento mais simples (notável na figura 18 ao observar a coluna do monolítico).

Fluxo de publicação simples: Possui uma esteira simples, tendo em vista que a publicação será alterada, compilada e publicada por inteiro (notável na figura 18 ao observar a coluna do monolítico).

Tecnologia: Toda aplicação possui a mesma divisão de camadas e a linguagem é uma só, o que facilita a coesão da equipe.

Mesmo com benefícios significativos, o monolítico também possui defeitos que podem pesar na hora de decidir a arquitetura a ser utilizada na aplicação, dentro as suas desvantagens podemos destacar:

Build: No caso de alteração em um módulo toda a aplicação deve ser compilada novamente, o que pode levar tempo para aplicações maiores.

Ponto de falha: Como mencionado anteriormente os módulos possuem dependência, ou seja, se uma parte da aplicação for "quebrada" o restante também deixa de funcionar.

Aculturamento: Um novo desenvolvedor ao ter o primeiro contato com a aplicação encontrará uma maior dificuldade, já que a aplicação é densa e seu entendimento como um todo demanda tempo.

11.2. **Microsserviços**

É um erro supor que toda nova aplicação deve ser construída em microsserviços, mesmo possuindo diversas vantagens este tipo de arquitetura pode trazer inúmeros problemas quando não utilizada em um cenário ideal. Segundo Clare Boothe Luce - A simplicidade é a maior sofisticação - a ideia dos microsserviços é facilitar e agilizar o desenvolvimento, uma vez que esse resultado não é obtido toda a arquitetura da aplicação deve ser revisada.

Baseado no artigo de Martin Fowler, já citado anteriormente, e *Microservices in a Nutshell* de Philipp Hauer podemos destacar alguns pontos que devem ser levados em consideração na adoção da arquitetura:

Deploys independentes: Apenas o módulo afetado precisar passar pelo processo de *deploy*, tornando o build mais rápido e focado nas alterações, o que facilita não só o *deploy*, mas também os testes e uma entrega contínua.

Escalabilidade: Os serviços são tratados de formas independentes na escalabilidade, dessa forma quando um módulo estiver sofrendo de sobrecarga apenas ele pode ser escalonado para suprir a demanda, sendo esta ação "invisível" para os outros módulos.

Desenvolvimento paralelo: Equipes podem atuar em módulos diferentes simultaneamente de maneira independente, eliminando a necessidade de realizar *merge* no mesmo projeto (notável na figura 1).

Independência em caso de falhas: A falha de um módulo não quebra a aplicação como um todo, caso o módulo de pagamento de um e-commerce quebre os

usuários poderão continuar fazendo *login* normalmente sem saber de tal falha, a menos que tentem utilizá-lo.

Como citado anteriormente, apesar de diversas vantagens, a arquitetura orientada a microsserviços também possui aspectos desvantajosos que devem ser considerados no início do projeto para redução de problemas posteriormente. Dentre eles podemos destacar:

Estrutura e Cultura: Para o desenvolvimento de uma aplicação em microsserviços uma série de fatores estruturais devem ser configurados, como mecanismos de entrega contínua (que são afetados pela cultura da empresa) e servidores *Cloud* (necessários para agilizar o desenvolvimento e entrega).

Segurança: A segurança na comunicação entre os serviços deve receber atenção, uma vez que a comunicação acontece com mais *endpoints* do que quando todos os serviços estão condensados em uma única aplicação.

Tecnologia: Os microsserviços não possuem dependência tecnologia, ou seja, podem ser desenvolvidos utilizando linguagens e *frameworks* diferentes, o que pode se tornar prejudicial na ausência de documentação e necessidade de uma equipe multidisciplinar.

Complexidade: Possui diversas camadas e configurações específicas, o que pode tornar os testes, monitoração e suporte mais complexos, exigindo profissionais mais capacitados.

7. Considerações finais

Referente ao presente trabalho, é de suma necessidade ressaltar a importância das matérias abordadas durante o curso superior de Análise e Desenvolvimento de Sistemas, onde foram aprimoradas as habilidades de analisar, desenvolver e criticar arquiteturas de sistemas e padrões de desenvolvimento, além do estudo de metodologias a respeito do gerenciamento de equipes e projetos, essenciais no desenvolvimento deste trabalho.

Como citado no desenrolar da pesquisa, o objetivo do trabalho não foi definir a melhor arquitetura, mas sim explorar seus aspectos positivos e negativos, auxiliando o leitor em uma possível tomada de decisão e esclarecendo, de forma simplificada, dúvidas sobre o tema. No início percebemos como a evolução tecnológica depende primeiramente de aspectos organizacionais, destacando a importância de um bom gerenciamento da equipe e introduzindo a lei de Conway para o leitor, posteriormente observamos como tais mudanças influenciam diretamente o desenvolvimento de uma aplicação e possíveis maneiras de aplicá-las em forma de duas arquiteturas diferentes, microsserviços ou monolíticos.

Como forma de preparação para o leitor que opte em realizar uma migração, um dos tópicos mais trabalhosos e que deve ser analisado com maior calma é o 8.3, onde é discutido a arquitetura do novo ecossistema, erros durante esse processo podem custar todo o projeto. A migração é trabalhosa e requer tempo de estudo, um dos maiores problemas encontrados por empresas que se aventuram na migração é não estimar os custos corretamente, uma vez que será necessário a capacitação dos desenvolvedores, tempo e principalmente uma boa infraestrutura Cloud, que pode ter um custo elevado

caso os microsserviços não sejam desenvolvidos e arquitetados da melhor forma possível.

A tecnologia está em constante transformação e os sistemas precisam se adequar para continuar operando de forma competitiva, a migração, mesmo que complexa em alguns pontos, pode se tornar um ganho expressivo para a empresa e seus usuários.

8. **Referências bibliográficas**

AGILE. (2004) ‘’Manifesto for Agile Software Development’’, http://agilemanifesto.org, Dezembro.

AMAZON. (2020) ‘’Desenvolvimento e operações e AWS’’, https://aws.amazon.com/pt/devops/, Dezembro.

BENEVIDES, Rafael. e POSTA, Christian. (2019) ‘’Microservices for Java Developers’’, https://developers.redhat.com/books/microservices-java-developers-hands-introduction-frameworks-and-containers, Dezembro.

CISCO. (2019) ’’Understanding Istio Service Mesh on Kubernetes’’, https://www.ciscolive.com/c/dam/r/ciscolive/emea/docs/2020/pdf/DEVNET-2022.pdf, Dezembro.

CAVALCANTE, Talita. (2017) ‘’Educação a distância cresce 17,6% em 2017’’, https://agenciabrasil.ebc.com.br/educacao/noticia/2018-09/educacao-distancia-cresce-176-em-2017-maior-salto-desde-2008, Abril.

CONWAY, Melvin. (1968) ‘’HOW DO COMMITTEES INVENT?’’, https://www.melconway.com/Home/pdf/committees.pdf, Fevereiro.

DIEDRICH, Cristiano. (2015) ‘’Docker’’, https://www.mundodocker.com.br/o-que-e-docker, Janeiro.

FOWLER, Martin. (2014) ‘’Microservices’’, https://martinfowler.com/articles/microservices.html, Fevereiro.

GALVIN, Carolyn. (2018) ‘’SiriusDecisions’’, https://go.forrester.com/research/siriusdecisions, Fevereiro.

HAUER, Philipp. (2015) ‘’Microservices in a Nutshell. Pros and Cons’’, https://phauer.com/2015/microservices-nutshell-pros-cons/, Junho.

KUSNETZKY, Dan. (2019) ‘’Microservices vs. Monolithic Architectures’’, https://rancher.com/blog/2019/microservices-vs-monolithic-architectures, Junho.

LEWIS, James. (2014) ‘’Microservices’’, https://martinfowler.com/articles/microservices.html. Acesso em: 15 fev. 2020.

MACHADO, Marcos. (2018) ‘’Qual a diferença para a Arquitetura Monolítica?’’, https://www.opus-software.com.br/micro-servicos-arquietura-monolitica, Fevereiro.

MATTOS, Carlos. (2017) ‘’O desenvolvimento de software na era moderna’’, https://blog.gft.com/br/2017/07/14/cloud-e-devops-o-desenvolvimento-de-software-na-era-moderna, Fevereiro.

MARTIN, Robert. (2012) ‘’The Clean Architecture’’, https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html, Fevereiro

NEWMAN, Sam. (2014) ‘’Demystifying Conway's Law’’, https://www.thoughtworks.com/insights/articles/demystifying-conways-law, Fevereiro.

RAZA, Muhammad. (2020) ‘’Cloud Growth in 2020’’, https://www.bmc.com/blogs/cloud-growth-trends, Dezembro.

RICHARDSON, Chris. (2019) ‘’What are microservices?’’, https://microservices.io, Abril.

SERGEEV, Alexander. (2018) ‘’An Guide to Creating and Managing Waterfall Teams’’, https://hygger.io/blog/an-essential-guide-to-creating-and-managing-waterfall-teams, Fevereiro.

TIOBE. (2020) ‘’TIOBE Index for 2020’’, https://www.tiobe.com/tiobe-index, Dezembro.

Cópia de documento digital impresso por Tavane Rodrigues (2049539) em 04/05/2021 09:54.

Documento Digitalizado Público

Artigo de Trabalho Conclusão de Curso aluno Gabriel Buges Garcia

Assunto: Artigo de Trabalho Conclusão de Curso aluno Gabriel Buges Garcia
Assinado por: Carlos Pagani
Tipo do Documento: Estudo
Situação: Finalizado
Nível de Acesso: Público
Tipo do Conferência: Cópia Simples

Documento assinado eletronicamente por:

- **Carlos Eduardo Pagani**, **PROFESSOR ENS BASICO TECN TECNOLOGICO**, em 09/03/2021 19:04:23.

Este documento foi armazenado no SUAP em 09/03/2021. Para comprovar sua integridade, faça a leitura do QRCode ao lado ou acesse https://suap.ifsp.edu.br/verificar-documento-externo/ e forneça os dados abaixo:

Código Verificador: 626641
Código de Autenticação: 06beb4c5cf