UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS
UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO
ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

Rafael Hens Ribas

ENGENHARIA DE SOFTWARE NO CONTEXTO DE *MICROSERVICES*

São Leopoldo
2017

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO

ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

Rafael Hens Ribas

ENGENHARIA DE SOFTWARE NO CONTEXTO DE *MICROSERVICES*

Trabalho de Conclusão de Curso apresentado como requisito parcial para a obtenção do título de Especialista em Engenharia de Software, pelo curso de Pós-Graduação Lato Sensu em Engenharia de Software da Universidade do Vale do Rio dos Sinos – UNISINOS.
Orientador: Prof. Dr. Mateus Raeder

São Leopoldo

2017

Engenharia de Software no Contexto de *Microservices*

Rafael Hens Ribas

Unidade Acadêmica de Educação Continuada – Universidade do Vale do Rio dos Sinos (Unisinos)

Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

rafaelhribas@edu.unisinos.br

Abstract. *Nowadays, the use of software is increasingly ubiquitous in our daily lives through the use of computers, notebook computers, tablets and, above all, smartphones and the increasing use of cloud computing. Where the software development process is constantly changing in relation to the architectures of this and how they should always propose new improvements regarding distribution, modularization and reuse. The new paradigm of development that emerged is the use of microservices architecture, in which the components of a traditional monolithic software are divided into independent services where this article approaches this theme through the theoretical and practical foundation and a research with developers to the respect to the knowledge and practice of this new architecture.*

Resumo. *Atualmente a utilização de software está cada vez mais onipresente em nosso cotidiano através da utilização de computadores, notebook, tablets e, principalmente, smartphones e pela crescente utilização da computação em nuvem. Onde o processo de desenvolvimento de software sempre está em constante mudanças em relação as arquiteturas deste e de como devem sempre propor novas melhorias em relação a distribuição, modularização e reutilização. O novo paradigma de desenvolvimento que surgiu é a utilização da arquitetura de microservices, na qual divide-se os componentes de um tradicional software monolítico, em serviços independentes onde este artigo aborda este tema através do embasamento teórico, prático e por uma pesquisa com desenvolvedores a respeito do conhecimento e prática desta nova arquitetura.*

1. Introdução

Atualmente em nossa sociedade a utilização de softwares está cada vez mais presente e necessária, sejam como os tradicionais servidores, computadores pessoais, notebooks, smartphones, tablets e mais recentemente através da utilização, cada vez mais abundante, da computação em nuvem. Neste tipo de tecnologia, um dos principais objetivos de utilização é a rapidez de processamento de tarefas que economizam nossos tempos e aumentam nossa capacidade de execução de tarefas.

A história das arquiteturas de software tem como foco por mudanças progressivas em direção a distribuição, modularização e baixo acoplamento, tendo assim, com o objetivo de aumentar a reutilização e a robustez do software independente do público alvo ou da forma de distribuição do mesmo. (DRAGONI, 2017).

Uma das arquiteturas de softwares com maior utilização atualmente é a arquitetura monolítica na qual é uma única aplicação de software desenvolvida em camadas, nas

quais podem ter responsabilidades próprias que interagem entre si, onde essas são disponibilizadas na forma de um único programa executável. Sendo assim está arquitetura não é responsável somente por uma única tarefa e, sim, por todas as tarefas necessárias para a execução de uma determinada ação. Tem-se como benefícios desta arquitetura a sua simplicidade aparente de desenvolvimento pois a mesma, mesmo em camadas, é vista como única solução assim, instintivamente, tornando mais simples a sua compreensão e de como implementar as suas funcionalidades. Tem-se, também, a facilidade para a disponibilização deste software, em qualquer ambiente de execução, devido a saída do executável único juntamente com demais arquivos, ou seja, bibliotecas necessárias como um todo. Além da sua simplicidade para a realização do escalonamento sendo esse necessário apenas executar múltiplas cópias deste executável (DRAGONI, 2017).

Apesar destes benefícios citados ao adotar essa arquitetura tem-se, também, os seus pontos negativos (RICHARDSON, 2016), tais como:

- conforme o software e a equipe de desenvolvimento aumentam a sua complexidade de desenvolvimento também aumenta proporcionalmente pois são vários desenvolvedores interagindo sobre a mesma base de código fonte assim proporcionando uma inserção maior de erros pois profissionais diferentes com metas distintas podem inteirar sobre algumas partes em comum assim como a devida compreensão de como implementar a qualidade desta;
- sobrecarga de tempo para a inicialização do software pois quanto maior o seu tamanho maior a sua inicialização independente do ambiente em que se encontre juntamente com o aumento proporcional de memória para armazenamento e execução;
- dificuldade para realizar manutenções de melhoria e/ou corretiva pois é necessário gerar o executável de todo o software e não somente da camada que sofreu a alteração;
- escalabilidade do software acaba sendo um obstáculo a mais pois é necessário escalar, duplicar, todo o software e não apenas a camada que está sendo maior consumida naquele período, sendo assim, gerando um aumento no custo benefício para a empresa detentora do software;
- dificuldade de atualização tecnológica pois para atualizar como uma nova versão de tecnologia é necessário alterar todo o software e não apenas a camada que poderia se beneficiar de um novo *framework.*

Visando este aumento de capacidade de tarefas e economia de tempo o processo de desenvolvimento de software demanda por essa maior agilidade e rapidez no seu desenvolvimento. Assim, diminuindo o tempo do mesmo em relação a entregas de projetos, correções e manutenções, tem-se a arquitetura distribuída de software nomeada de *microservices* que propõe que empresas com suas equipes consigam atingir essas metas e a escalabilidade necessária nos dias atuais. (RICHARDSON, 2016).

O objetivo principal deste artigo é propor a utilização da arquitetura de software *microservices* através da comparação com o padrão monolítico descrevendo as vantagens e desvantagens específicas de cada uma e ressaltando o motivo da opção da arquitetura distribuída. Para atingir o objetivo principal, definiu-se os seguintes objetivos específicos: a) Mapear o processo de desenvolvimento de um software utilizando a arquitetura de

microservices; b) Identificar os pontos de vantagens e desvantagens da arquitetura proposta em relação a monolítica; c) Desenvolvimento de um software utilizando a arquitetura de *microservices*; d) Desenvolvimento de um software utilizando a arquitetura monolítica; e) Apresentar o resultado de uma pesquisa para recolher a opinião de profissionais da área em relação a este tema. Desta forma, a pesquisa apresenta-se através de um estudo de caso, realizando a implementação de um simples *ecommerce* usando os paradigmas de arquitetura monolítica e de *microservices* para a resolução do mesmo problema, comparando ambas arquiteturas.

Além desta introdução, o restante do artigo está organizado seguindo a seguinte forma: A Seção 2 apresenta o referencial teórico para o desenvolvimento deste artigo; a Seção 3 mostra as principais estratégias de migração entre a arquitetura monolítica para a de *microservices*; a Seção 4 mostra o desenvolvimento de um mesmo software utilizando as arquiteturas de monolítica e de *microservices*, mostrando as alterações realizadas nesta migração assim como impressões a respeito deste processo; a Seção 5 mostra o resultado de uma pesquisa aplicada com profissionais da área de tecnologia em relação ao assunto deste artigo, e; finalmente, a Seção 6 descreve as considerações finais do estudo.

2. Referencial Teórico

O referencial teórico utilizado na elaboração e execução da pesquisa realizada neste trabalho é apresentar os principais conceitos, vantagens e desvantagens na utilização do paradigma de desenvolvimento de software em *microservices*.

2.1. *Microservices*

O paradigma da arquitetura de software em *microservices* consiste, em ao invés de construir um único software monolítico, dividi-lo em várias partes pequenas, no caso serviços, e independentes que se conectam entre si de acordo com as regras de negócios e o fluxo que o software necessita satisfazer (RICHARDSON, 2016). Este conceito é uma derivação do *Bounded Context* (BC) definido pelo *Domaind-Driven Design* (DDD) *pattern* na qual define que para uma aplicação considerada larga deve ter sua implementação dividida em vários BCs e seus modelos e base de dados separadas uma das outras (TORRE, 2017).

Segundo, Richardson (2016), um serviço deve implementar apenas uma parte distinta de funcionalidades, como por exemplo, em um *ecommerce* temos os serviços de ordem de compra, produtos, clientes entre outros que possam necessitar. Cada um destes serviços é como um pequeno software que possui sua própria arquitetura e tecnologia que consiste em representar regras especificas de uma camada de regras de negócios. No qual, em alguns casos, podem expor simplesmente uma ou várias API que é consumida por outro serviço ou aplicação qualquer, como exemplo um SPA[1], ou pode ser ambas partes neste caso implementando tanto o *Backend* quanto o *Frontend*.

---

[1] *Single Page Application* é uma arquitetura de UI/UX para aplicações web que interagem com o usuário dinamicamente alterando o site sem a necessidade de recarregar cada página a cada interação se assemelhando as aplicações *desktops*.

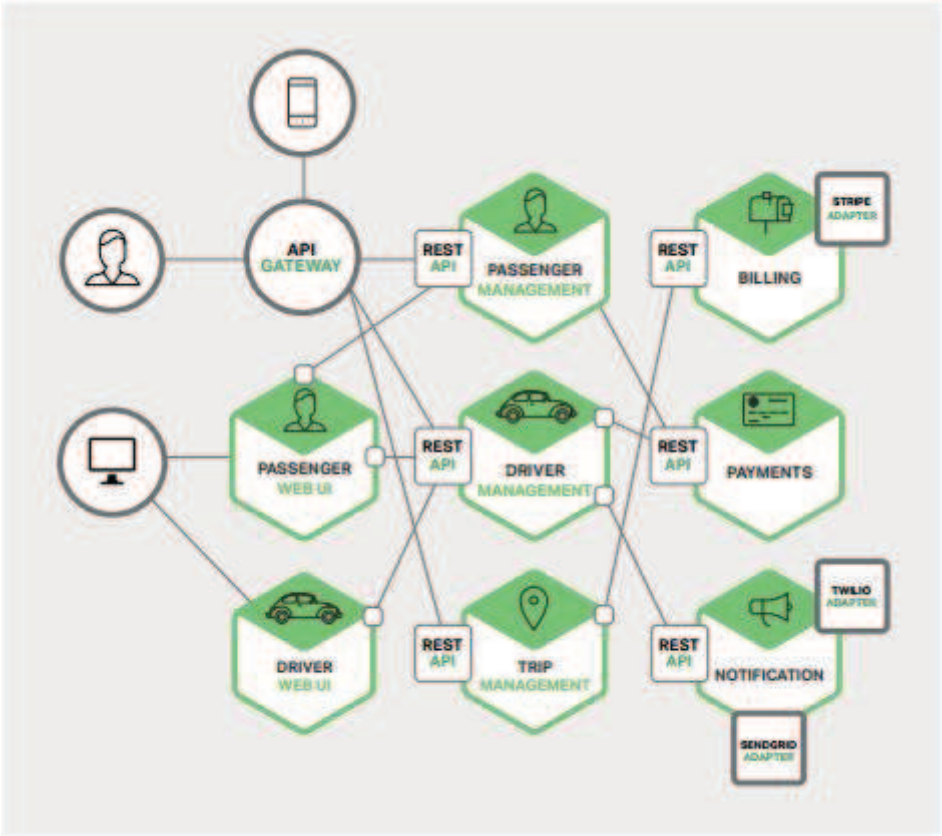

Figura 1. Representação de um software na arquitetura de *microservices*

Fonte: Richardson (2016)

Na Figura 1 o software está implementado por vários serviços próprios na qual existe um SPA para cada ator do software devidamente identificados como passageiro (*passenger)* e motorista (*driver*) assim separando, também, a implementação da interface gráfica de acordo com o contexto do ator. Cada serviço está disponibilizando APIs REST para ser consumida pela SPA ou por outra parte do software, ou seja, outro serviço. Neste exemplo tem-se também a exposição de uma API especifica para acesso via dispositivo móvel através do aplicativo próprio para cada ator além do acesso pelo SPA em si. Outro aspecto da arquitetura de *microservices* é a independência da persistência dos dados, ou seja, cada serviço que necessite armazenar dados para trabalho deve ter seu próprio banco de dados implementado e acessível somente por ele. Esta implementação da persistência de dados, em termos de tecnologia, pode-se utilizar a mais adequada para cada cenário que o serviço necessite, por exemplo, pode ser um banco de dados relacional ou NoSQL. Neste sentido este *pattern* é outro contraponto ao modelo monolítico na qual, na sua maioria, tem-se apenas um único banco de dados para persistir os dados necessários pelas regras de negócios do software. Exemplo conforme Figura 2. (RICHARDSON, 2016).

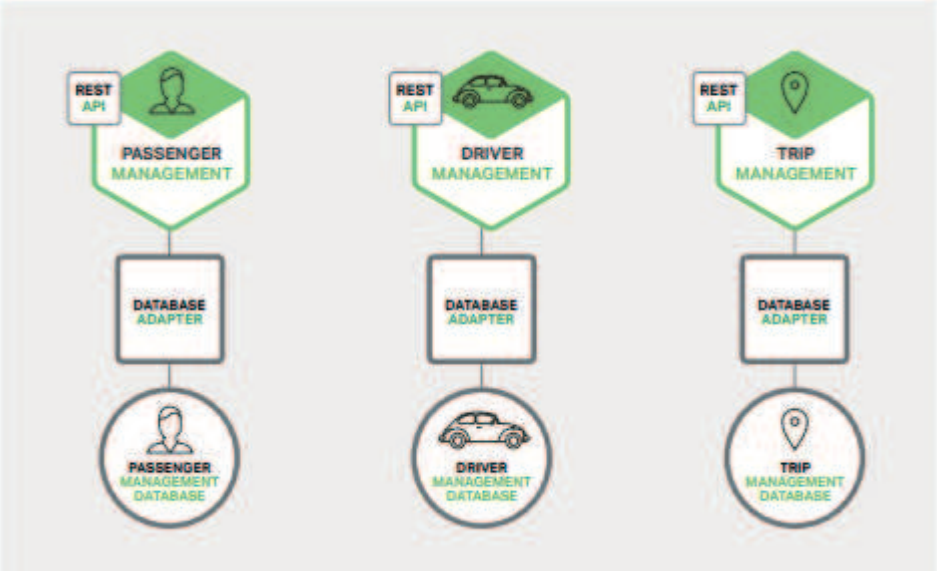

Figura 2. Representação da implementação da persistência de dados na arquitetura de *microservices*

Fonte: Richardson (2016)

A arquitetura de microservices é bastante similar ao SOA porém na questão de interação entre os serviços que representam o software como todo (TORRES, 2017).

Esta arquitetura de software tem vários benefícios em relação ao monolítico iniciando pela remoção da resolução da complexidade do problema separando em várias partes distintas. Assim pode-se continuar a desenvolver novas funcionalidades independentemente de um contexto de outro pois não afeta o software como um todo (RICHARDSON, 2016).

Pode-se utilizar tecnologias diferentes para cada serviço conforme a necessidade das regras de negócios que cada contexto irá resolver, ou seja, para determinadas funcionalidades a melhor tecnologia pode ser a utilização de NodeJS assim como em outro serviço o melhor seria a utilização de Java. Nesta linha de raciocínio pode-se aplicar o mesmo para a persistência dos dados onde um serviço utiliza um banco de dados relacional onde outro NoSQL. Com isso um software que iniciou a algum tempo pode continuar, de certa maneira, atualizada pois a cada novo serviço pode-se utilizar a tecnologia mais recente além de ser mais fácil de realizar um *refactoring* de algum serviço existente (NADAREISHVILI, 2016).

Segundo Richardson (2016), permite que cada serviço pode ser desenvolvido por equipes distintas e separadamente assim proporcionando uma entrega para cada uma especificamente na qual pode ter o software como um todo sendo desenvolvido paralelamente por cada equipe reduzindo o tempo de implementação. Isto aplica-se, também, para as questões de atualizações do serviço devido questões de melhorias ou correções permitindo que este processo seja executado mais rapidamente pelo desenvolvimento e impactando pela equipe de testes em testar apenas o que foi alterado sem a necessidade de realizar testes de regressão no software como um todo.

Em termos de escalabilidade cada serviço pode ser escalado independentemente e conforme a necessidade de consumo de cada um reduzindo os custos para a empresa em termos de investimento de infraestrutura tanto na nuvem como própria. Também

pode-se separar fisicamente onde cada serviço irá ser executado, ou seja, dependendo do serviço um pode ser disponibilidade em servidores com maior poder de processamento comparado com outro serviço que não necessita de um servidor com tanto processamento (RICHARDSON, 2016).

Porém também tem-se os seus problemas nesta adoção de arquitetura como em realizar uma constante crítica em relação ao tamanho do serviço em si, ou seja, de quanto em quanto tempo um serviço deve ser atualizado implementando novas funcionalidades no mesmo em si ou ser separado em outros serviços para não se tornar parecido com uma arquitetura monolítica e seus problemas já conhecido (TORRES, 2017).

Outro problema considerado complexo é pelo fato de a própria arquitetura ser distribuída assim os desenvolvedores e suas equipes devem escolher e implementar qual o processo de comunicação e consumo de cada serviço entre si. Realizando com isso todo o tratamento no caso de falhas e atrasos de comunicação entre os serviços que estão separados, algum serviço estar indisponível (fora do ar) em algum momento especifico (RICHARDSON, 2016).

Na questão da persistência dos dados onde cada serviço deve ter a sua própria implementação e acesso a essa persistência tem-se os problemas de coordenação de backup, onde na arquitetura monolítica normalmente tem-se apenas um backup da base de dados para realizar no *microservices* tem-se a necessidade de realizar mais de um backup conforme a quantidade de bases de dados. Assim como toda a coordenação de atualização e replicação dos dados em diferentes bases de dados (TORRES, 2017).

Ao testar um software nesta arquitetura torna-se mais onerosa devido a complexidade que a mesma está separada pelos serviços. Pois tem-se uma SPA que em determinada tela interage com mais de serviço assim sendo necessário que todos estejam operando e disponíveis normalmente para a execução de algum teste (RICHARDSON, 2016).

A coordenação de equipes de desenvolvimento por cada serviço caso alguma atualização de um serviço especifico necessite ser atualizado e este tem impacto na iteração com outros que dependem deste também necessitam se atualizar no mesmo tempo para que não ocorra problemas de versões diferentes ocasionando na incompatibilidade e erro no fluxo do software como um todo (TORRES, 2017).

Na gestão da disponibilidade do serviço onde cada um pode estar sendo executado em uns mais servidores diferentes ou não tem-se esta necessidade de configuração de balanceamento de cargo individualmente por cada serviço (RICHARDSON, 2016).

2.2. API Gateway

Segundo Richardon (2016) na arquitetura monolítica, mesmo que se tenha um balanceador de carga, os acessos às informações são realizados através de requisições aos *endpoints* que a API do software forneça. No entanto, na arquitetura de *microservices*, estes *endpoints* estão separados em outros serviços executados independentemente uns dos outros onde cada um tem o seu próprio endereço como no exemplo da Figura 3. Neste caso temos um dos problemas a serem resolvidos quando se adota esta arquitetura que é qual a estratégia de acesso aos serviços que será utilizada.

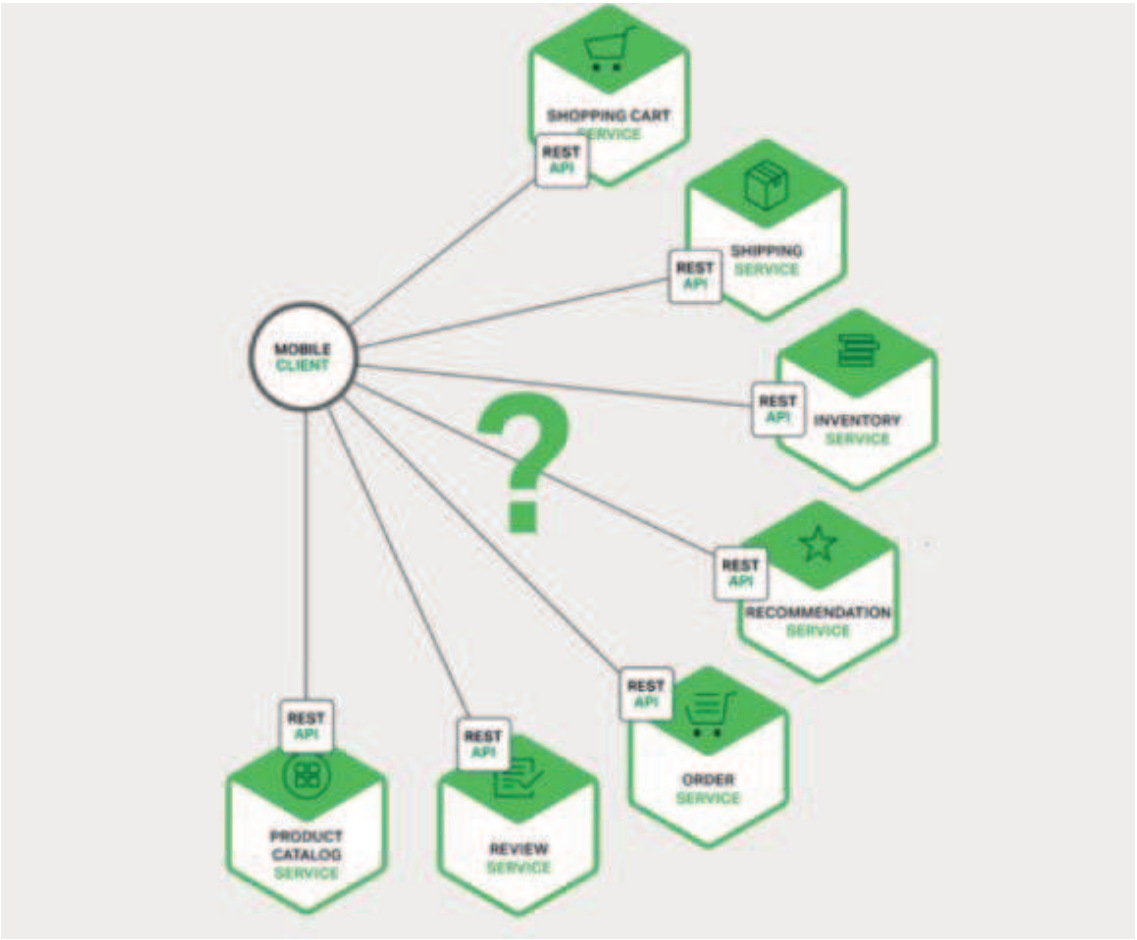

Figura 3. Exemplo de uma aplicação cliente que utiliza uma aplicação em *microservices* que não utiliza API *Gateway*

Fonte: Richardson (2016)

Estes serviços poderiam ser acessados pelo aplicativo cliente e entre si diretamente assim como é realizado na arquitetura monolítica pois cada um deles tem poderia disponibilizar o seu *endpoint* publicamente. Porém neste modelo de acesso tem-se limitações quanto ao escalonamento dos serviços além que a aplicação cliente deve acessar cada um dos *endpoints* para interagir conforme o fluxo de cada tela ao invés de apenas um *endpoint* que possa realizar esse trabalho de acessar mais *endpoints* e entregar uma informação já completa ao aplicativo cliente, ou seja, acessando diretamente o *endpoint* de cada serviço torna mais complexo o desenvolvimento do aplicativo cliente que consume os serviços. Além disso nem todo serviço necessariamente pode utilizar uma tecnologia de comunicação amigável ao protocolo web, como exemplo, utilizando um serviço de mensagens AMQP para persistência de troca de requisições via mensagens assim tendo problemas desde com *firewalls* até como a sua disponibilização via HTTP. Continuando em problemas ao adotar esta estratégia de acesso direto aos serviços é a dificuldade para realizar algum *refactor* destes pois a cada momento que um serviço é novamente subdividido em mais serviços tem-se, nesta mesma proporção, outro endereço de *endpoint* para a aplicação cliente interagir. Além das questões de segurança ao acesso a estes pois alguns *endpoints* podem não precisar de nenhuma autenticação de usuário para serem consumidas enquanto que outras sim assim cada API deve implementar o mesmo método de validação de acesso por usuário ao invés de centralizar esta

funcionalidade em apenas uma API disponibilizada pelo seu *endpoint* (RICHARDSON, 2016).

Para suprir estes problemas de acesso diretamente a cada *endpoint* da arquitetura de *microservices* tem-se a utilização da estratégia de API *Gateway*. Esta estratégia é um serviço com a finalidade de ser o único ponto de entrada, *entry point,* entre a comunicação de um aplicativo cliente com os demais *endpoints* disponibilizados pelas APIs de cada serviço da arquitetura de *microservices* através do encapsulamento dos mesmos. Porém ainda pode-se adicionar mais funcionalidades para esta estratégia como ser responsável em realizar o balanceamento de carga, *caching,* monitoração dos serviços, centralizar a estratégia de verificação de usuários auntenticados, centralização de requisições para o aplicativo cliente e separando estas para cada serviço especifico e posteriormente unificando as respostas para realizar apenas uma entrega simples (RICHARDSON, 2016).

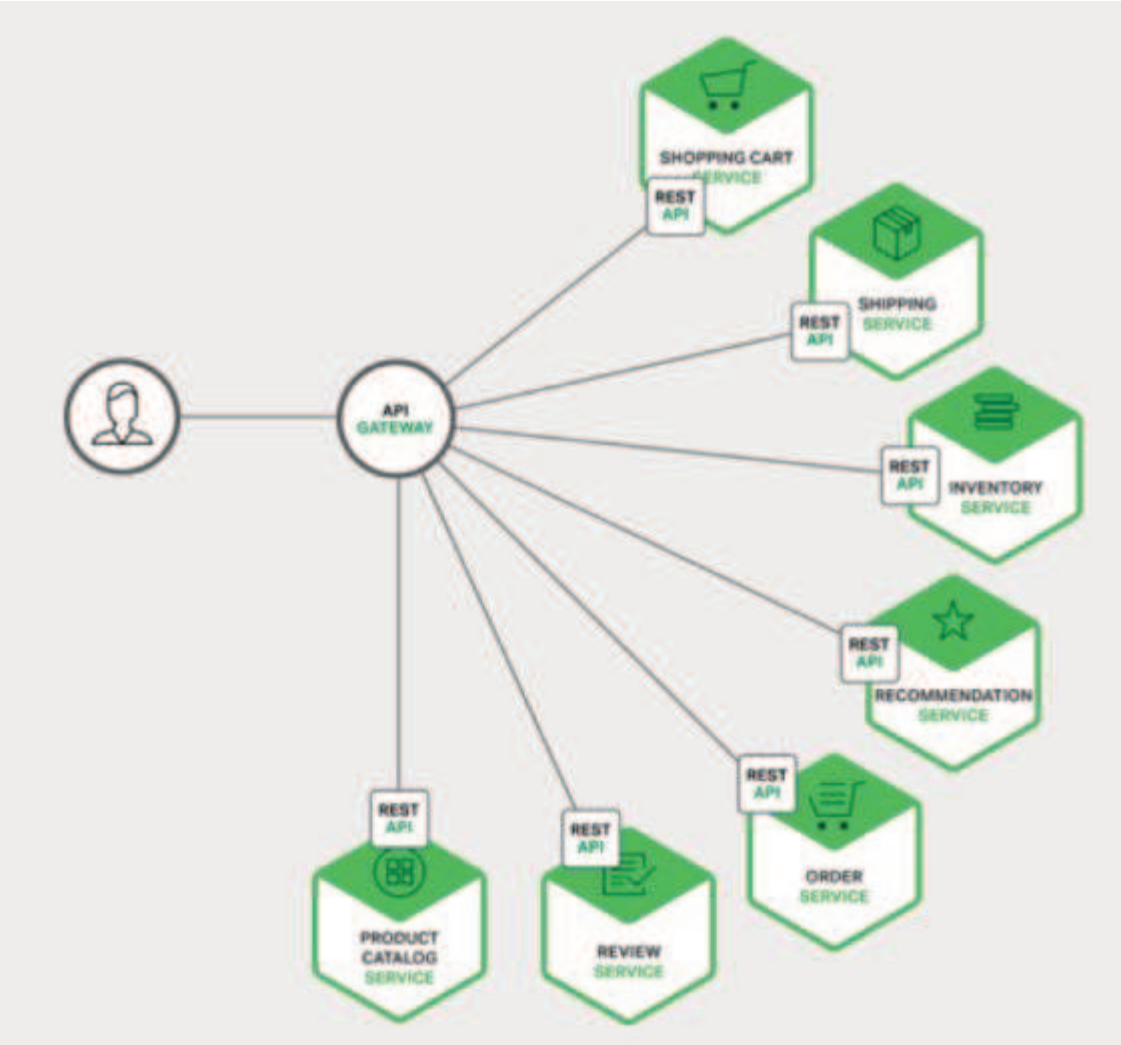

Figura 4. Demonstrando a estratégia de adoção da API *Gateway*

Fonte: Richardson (2016)

A API *Gateway* pode prover para cada aplicativo cliente APIs customizadas de acordo com a necessidade de cada um, por exemplo, um SPA pode acessar mais *endpoints* que o aplicativo *mobile* conforme o exemplo disponibilizado vide Figura 4.

Os benefícios desta abordagem foram citados acima, porém, tem-se também problemas nesta abordagem onde o maior deles é o problema de *bottleneck* pois conforme o software cresce em quantidade de serviços a API *Gateway* pode acabar acumulando muitas funcionalidades tornando o desempenho como um todo lento pois é ela que encapsula o acesso aos *endpoints*. Por isso deve-se tratar a API *Gateway* como de fato ela é, ou seja, como outro *microservices* qualquer onde sempre que necessário deve sofrer um *refactoring* dividindo-a em outras API *Gateway* que trabalham em conjunto ou que são acessadas separadamente (RICHARDSON, 2016).

Ao implementar esta estratégia algumas considerações em termos de *design* devem ser consideradas nas quais são apresentadas pelas subseções a seguir.

2.2.1 Performance e Escalabilidade

Para a maioria das aplicações, independentemente do tamanho da mesma e seu público, a questão de performance e escalabilidade da API *Gateway* é muito importante e, por isso, é extrema importância e por isso deve construir a arquitetura dos *microservices* em ambientes utilizando recursos/tecnologias que possam ser escalonados conforme a necessidade (RICHARDSON, 2016). Existem várias tecnologias e fornecedores destas na qual este artigo não irá abordar neste momento.

2.2.2 Utilizando *Reactive Programming Model*

A API *Gateway* é responsável em ser o *endpoint* publicamente para o consumo dos serviços e sendo o centralizador das requisições. Onde estas podem ser apenas realizando requisições simples para outro serviço trabalhando, deste modo, como apenas um roteador de respostas assim como pode realizar mais de uma requisição para serviços diferentes e armazenando, unificando, as respostas em apenas uma devidamente pronta para ser entregue a quem requisitou o seu *endpoint*. Para suprir esta questão de performance essas múltiplas requisições que são realizadas devem ser independentes uma das outras para minimizar o tempo estas são executadas de forma assíncrona. Em alguns casos podem haver dependências entre essas requisições onde a API *Gateway* deve realizar mais operações de modo assíncrono com a necessidade de realizar em etapas devido a uma regra de negócio. Deste modo a arquitetura será cada vez mais difícil de entender, seja para alguma rotina de atualização ou manutenção, devido a suas chamadas assíncronas que chamam outras chamadas assíncronos. Devido a essa dificuldade deve-se adotar o recurso de programação reativa (*reactive programming*) como, por exemplo, segundo Mozilla (2017) as *promises* do JavaScript. Existem outras ferramentas/bibliotecas para auxiliar ao desenvolvimento do *Reactive Programming Model* (RICHARDSON, 2016).

2.2.3 *Service Invocation*

Uma aplicação que segue a arquitetura de *microservices* segue o modelo de sistema distribuído e, como tal, deve-se utilizar algum mecanismo de interprocesso. Para tal comunicação existem dois modelos onde um adota o de processos assíncronos e o outro o síncrono. No modelo assíncrono é utilizado o de chamado de troca de mensagens onde pode-se utilizar um servidor de mensagens, também conhecido como *message broker*, como exemplo conforme AMQP (2017), o AMQP ou *brokerless*m sem servidor de mensagens segundo ZeroMQ (2017), como o ZeroMQ . No modelo síncrono tem-se como exemplo o HTTP ou Apache Thrift. (APACHE, 2017).

Normalmente na arquitetura de *microservices* é comum utilizar ambos modelos para realizar o acesso aos serviços (*service invocation*) disponíveis juntamente com as regras de negócios a serem implementadas. Neste caso a API *Gateway* tem essa característica de em certos momentos trabalhar de forma assíncrona, onde se busca uma melhor performance, porém em alguns casos é necessário trabalhar em forma síncrona, como exemplo o recebimento de sinais vitais de um paciente, onde a informação em si é mais importante que a performance de uma parte dos serviços (RICHARDSON, 2016).

2.2.4 *Service Discovery*

Em uma aplicação normal, quando se necessita saber o endereço IP qualquer, normalmente essa informação é armazenada juntamente com a própria aplicação. Porém na arquitetura de *microservices* tem-se um conceito diferente devido a sua natureza distribuída, ou seja, a API *Gateway* precisa conhecer todos os endereços IP dos serviços que serão requisitados através dela. Alguns serviços como os de mensagens possuem um endereço IP fixo para facilitar o processo de envio e recebimento das mensagens, porém, os outros serviços, em sua maioria, não possuem o seu endereço IP fixado trabalhando deste modo com endereços IP dinâmicos. Estes serviços, da arquitetura de *microservices,* utiliza-se deste modelo de endereçamento de IP dinâmicos devido aos fatos de poder utilizar o escalonamento do mesmo e, também, caso tenha-se uma nova versão a ser disponibilizada basta apenas disponibilizada com este novo endereço IP ao invés de necessariamente atualizar o executável do serviço em si (RICHARDSON, 2016).

Este trabalho de conhecer os endereços de IP onde cada serviço que compõe a aplicação pode ser realizado de duas maneiras que são *service discorery* ou *client discovery*. Caso opta-se pela utilização do *cliente discovery* então é papel da API *Gateway* pesquisar onde está o endereço IP do serviço em uma base de dados de endereços IP dos serviços onde cada serviço deve registrar o seu IP e status nesta base de dados. Esta forma de pesquisa é chamada, segundo Microservices (2017), de *service registry* (RICHARDSON, 2016).

2.2.5 *Contornando Falhas Parciais (Handling Partial Failures)*

Um problema que a arquitetura de *microservices* pode enfrentar mais comumente, se comparado a arquitetura monolítica, é o problema de falhas parciais. Onde parte dos serviços está operacional e outra parte está com algum tipo de problema, como exemplo, processamento lento ou até mesmo indisponível. Neste caso de falhas parciais é de responsabilidade da API *Gateway* contornar de alguma maneira esta situação para que o aplicativo como um todo continue funcionando, mesmo que parcialmente, mesmo que trazendo alguma informação em *cache*, caso se tenha e ainda seja relevante, ou consultar outro serviço relevante de alguma maneira com o que foi requisito que está com problema neste momento até mesmo informar um aviso padrão de problema nesta parte a quem realizou a requisição e/ou continuar tentando por algum tempo acessar o serviço para entregar o que foi solicitado. Qual o método ou procedimento a ser adotado depende conforme a regra de negócio para estes problemas, mas de fato, onde se deve implementar esta rotina é na API *Gateway* (RICHARDSON, 2016).

2.3. *Inter-Process Communication* (IPC)

Um aplicativo que utiliza uma arquitetura monolítica invoca um componente ao outro através de chamadas realizadas a nível de codificação. Em comparação a estrutura de

microservices que é um aplicativo que utiliza o sistema distribuído sendo estes serviços executados de forma independente e em ambientes que podem ser diferentes um dos outros, temos então, que interagir por cada instância destes através, ao menos, de algum tipo de mecanismo de comunicação entre os serviços (DRAGONI, 2017.

Ao escolher qual o tipo de mecanismo de comunicação entre serviços escolher deve-se saber qual o grau de importância de como estes serviços devem interagir como um todo pois dependendo do escolhido pode afetar o propósito que o aplicativo deve satisfazer.

Existem dois tipos de categorização de mecanismos de comunicação a serem utilizados na qual estão categorizados em *one-to-one* ou em *one-to-many*. Onde o primeiro cada requisição é processada exatamente por uma instância de um serviço enquanto o segundo cada requisição é processada por, pelo menos, uma das múltiplas instâncias do mesmo serviço na qual, em ambos tipos, estas podem ser realizadas de forma síncrona ou assíncrona. No caso da síncrona o cliente que realizou uma requisição a algum serviço aguardo o retorno do mesmo em um tempo definido onde as demais instruções do cliente param de ser processadas enquanto está no aguardo ou ao esgotar esse tempo (*time out)* de resposta. Em comparação ao assíncrono o cliente realiza a requisição a algum serviço, porém este continua a processar outras instruções enquanto ainda não recebe a resposta solicitada e, ao receber a resposta, realiza uma ou mais instruções especificas de acordo com alguma regra de negócio da aplicação. A Figura 5 mostra uma tabela comparativa entre a categorização do mecanismo de comunicação e qual forma esta pode ser utilizada de acordo com a forma síncrona e assíncrona (RICHARDSON, 2016).

	ONE-TO-ONE	ONE-TO-MANY
SYNCHRONOUS	Request/response	—
ASYNCHRONOUS	Notification	Publish/subscribe
	Request/async response	Publish/async responses

Figura 5. Estilos de mecanismos de processos de comunicação e suas formas

Fonte: Richardson (2016)

No caso do IPC escolhido for o *one-to-one* ocorre somente de forma síncrona através de um *request/response* ou *request/async reponse*. Onde o primeiro é o modelo padrão de execução na qual foi descrito anteriormente onde o cliente faz uma requisição (*request*) e aguarda, parando os demais processos, até receber a resposta (*reponse*) e o segundo modelo é um aperfeiçoamento do primeiro onde o cliente realiza a requisição mas colocando o aguardo da resposta em modo assíncrono assim evitando o congelamento dos demais processos até que receba a resposta. Ao escolher o *one-to-many* todos os modelos são executados de modo assíncrono podendo ser do tipo *publish/subscribe* ou *publish/async response*. Onde o primeiro publica (*publish*) uma mensagem de notificação que pode ser consumida por nenhum ou vários serviços que tenham com finalidade responder esta mensagem sendo os *subscribes* da mesma. O segundo modelo o cliente publica (*publish)* uma mensagem de requisição e aguarda por algum tempo pelas respostas (*responses*) dos serviços que tenham como finalidade responder a esta requisição. É comum em uma arquitetura de *microservices* implementar não apenas um destes mecanismos de IPC assim como ambos conforme a necessidade encontrada pelo aplicativo (RICHARDSON, 2016).

2.3.1 Comunicação baseada em mensagens assíncronas

Um dos mecanismos de comunicação entre os serviços é a comunicação baseada em mensagens que ocorrem de forma assíncronas. A estrutura de uma mensagem consiste em cabeçalho (*headers*), que são metadatas sobre quem a enviou), e o corpo (*body*). Quaisquer quantidades de mensagens podem ser enviadas por qualquer número de serviços que realizam o envio assim como qualquer número de serviços podem responder a estas conforme a estratégia escolhida de replicação de um mesmo serviço.

Existem atualmente uma variedade de sistemas de mensagens que se pode escolher sendo que o recomendado aquele que tenha o suporte para mais de uma linguagem de programação e sistema operacional, caso contrário, perde-se uma das vantagens da arquitetura *microservices* que é poder utilizar mais de uma tecnologia de desenvolvimento de software e ambiente na implementação dos serviços. Entre os disponíveis que suportam mais de uma linguagem de programação e sistema operacional e que utilizar um dos principais protocolos, que é o AMQP, são segundo RabbitMQ (2017) RabbitMQ, segundo Apache (2017) Apache Kafka, Apache ActiveMQ e, confome NSQ (2017) NSQ (RICHARDSON, 2016).

As principais vantagens em utilizar este tipo de comunicação são desacoplar a comunicação direta entre o cliente e o serviço, sendo assim, o cliente realizar uma requisição através do envio de uma mensagem sem se preocupar onde está sendo executado o serviço que a responderá enviando uma mensagem de resposta para um cliente que também não sabe onde está pois ambos estão usando a comunicação de mensagens. As mensagens, por padrão, possuem buffer, assim permitindo que em algum momento uma das partes pode estar indisponível no momento que envia ou recebe a mensagem e que posteriormente, quando estiverem disponíveis, poderão processar tal informação. Assim como se tem as suas vantagens, também, tem as suas desvantagens que são uma adição a mais na arquitetura aumentando a complexidade da mesma e seu gerenciamento, complexidade na implementação do mecanismo de envio e resposta das mensagens (RICHARDSON, 2016).

2.3.2 Comunicação síncronas baseadas em *Request/Response*

Outro mecanismo de comunicação existente na arquitetura de *microservices* é a comunicação síncrona de *request/response.* Neste mecanismo um cliente envia uma requisição para um ou mais serviços e este processa o solicitado e envia a resposta ao solicitante. Em muitos clientes que utilizam deste mecanismo acabam executando este processo unicamente, colocando os demais em estado parado, até que recebam a resposta para então continuar os processos normalmente levando ao aplicativo cliente ao estado de congelamento se este fluxo demorar demais. Existem mais de um protocolo para a utilização deste mecanismo de comunicação onde os mais populares são o REST e o Apache Thrift (RICHARDSON, 2016).

No protocolo REST é desenvolvido APIs, segundo Wikipedia (2017), no estilo RESTful utilizado principalmente sobre o protocolo HTTP. Um dos seus principais conceitos é a representação de recursos que representam objetos referentes as regras de negócios que a aplicação resolva. Onde fornece um conjunto de API que promovem, de certa forma, a escalabilidade de componentes e suas interações entre si através de *interfaces*, que representam contratos representados, segundo TECHTARGET (2017) por XML ou, de acordo JSON (2017), JSON, de como devem ser realizadas as requisições e

as respostas e, ao mesmo tempo, fornecendo meios para a redução da latência da comunicação e segurança através de autenticação e autorização de acesso aos serviços (RICHARDSON, 2016).

As principais vantagens da utilização do protocolo REST, além de funcionar sobre o HTTP, são o acesso simplificado para o consumo da API pois, normalmente, não se tem problemas com *firewalls*, é familiar ao desenvolvedores implementar estas APIs pois utiliza-se através de chamadas HTTP, ou seja, pode-se requisitar qualquer API utilizando qualquer navegador de internet ou alguma ferramenta, segundo curl (2017) como curl, nem necessita de nenhum mecanismo entre a aplicação que realiza a requisição e a algum serviço que realiza o resposta, assim, tem-se menos complexidade na configuração dos ambientes onde a aplicação e os serviços estão disponibilizados. Porém, também, tem-se as suas desvantagens como apenas suportar o direcionamento direto entre a requisição e a resposta, assim ambos devem estar disponíveis para a utilização onde algum problema, independente de qual seja, interfere no desempenho do outro. O cliente deve saber, de alguma forma, o endereço IP do serviço que pretende interagir assim trazendo a complexidade deste conhecimento para a arquitetura de *microservices* onde, geralmente, resolvida pela API *Gateway* (RICHARDSON, 2016).

O Apache Thrift é uma alternativa a utilização do protocolo REST pois é um *framework* para o desenvolvimento de clientes e serviços, independentes da linguagem, para a utilização de RPC. Este define o estilo de definição das API através de arquivos de metadatas semelhantes a IDL do C onde estes são compilados para gerar as *interfaces* de comunicação entre o cliente e os serviços. Esta *interface* é semelhante a uma *interface* de alguma linguagem orientada a objetos, como exemplo o Java, que disponibiliza uma coleção de métodos fortemente definidos (*typed*). Onde estes podem retorna algum valor ou nenhum se forem definidos como serviços que apenas recebam requisições. Suporta vários estilos de formatos de mensagens como o JSON, binário e binário compacto. Pode-se utilizar os protocolos de transporte HTTP e TCP que é mais eficiente que o anterior, porém tem-se implicações em redes que utilizam *Firewalls* (RICHARDSON, 2016).

2.4. *Service Discovery*

Na arquitetura monolítica ou de *microservices,* que utiliza a comunicação REST*,* ao realizar uma requisição a alguma API o endereço IP dos serviços são de modo estático onde o cliente, que irá realizar alguma requisição, conhece previamente o endereçamento IP para realizar a chamada de algum recurso. Porém, atualmente, os serviços estão disponibilizados na nuvem, segundo Microsoft (2017) como *cloud computing*, além da utilização do escalonamento dos mesmos armazenar estaticamente cada endereço IP de cada serviço torna-se um trabalho intangível e ineficiente. Para suprir essa necessidade utiliza-se a metodologia de *Service Discovery* na qual se dividi em dois padrões denominadas de *Client-Side Discovery* ou *Service-Side Discovery* (RICHARDSON, 2016).

2.4.1 *Client-Side Discovery Pattern*

Ao utilizar este padrão de descoberta do endereço IP dos serviços é de responsabilidade do cliente que irá requisitar os serviços determinar onde e quais são as instâncias disponíveis assim como realizar o balanceamento das requisições. O cliente deve consultar o *Service Registry* onde contém a lista atualizada das instâncias disponíveis dos serviços juntamente com algum algoritmo para realizar o balanceamento e, após estas

etapas, finalmente realizar a requisição para o serviço e devolver a resposta ao cliente que a requisitou. Para que esse padrão funcione cada serviço deve se registrar no *Service Registry* quando é instanciado para que o cliente, através da API *Gateway*, possa realizar a sua descoberta e ser requisitado. Este padrão tem suas vantagens, como exemplo, ser relativamente menos complexo de administrar pois a parte dinâmica, ou seja, que mais sofre alterações é o *Service Registry* e, após o cliente saber onde está localizado o serviço o mesmo pode ir diretamente ao mesmo na próxima requisição assim diminuindo as etapas e, por consequência, o tempo de resposta. Porém uma das suas desvantagens é que o cliente, ou seja, neste caso a API *Gateway* está fortemente acoplada com o *Service Registry* assim a cada nova implementação de um novo serviço a API *Gateway* deverá saber se comunicar com o mesmo assim como o novo serviço deve conseguir se comunicar com a tecnologia do *Service Registry* (RICHARDSON, 2016).

2.4.2 *Server-Side Discovery Pattern*

Outro padrão de descoberta dos serviços funciona com o cliente realizando uma requisição para um balanceador de carga que tem a responsabilidade de consultar o *Service Registry* e realizar o roteamento do cliente a cada serviço que responderá a sua requisição. Este padrão tem como grande benefício a abstração por parte do cliente ter que interagir com o *Service Registry* assim retira-se esta necessidade não sobrecarregando o mesmo. Porém também tem o problema de o balanceador de carga ser mais um serviço para administrar independentemente dos demais e, como um, deve sofrer manutenções e atualizações conforme a necessidade além de ser mais uma configuração a parte a ser realizado (RICHARDSON, 2016).

2.4.3 *Service Registry*

O *Service Registry* é uma parte fundamental do processo de *Service Discovery,* que é uma base de dados que contêm os registros dos endereços IP das instâncias dos serviços na qual devem sempre atualizarem o mesmo. Na qual este deve estar sempre disponível ao máximo dentro do contexto da arquitetura de *microservices*, mesmo que o cliente possa realizar *cache* destes endereços IP, em algum momento estas informações podem ficar obsoletas e tendo a necessidade de novamente consultar o *Service Registry.* Existem dois padrões de implementação de *Service Registry* que são *Self-Registration* e *Third-Party Registration*. Onde o funcionamento de ambos já foram mencionados anteriormente.

Uma abordagem para tornar mais seguro esta parte fundamental é disponibiliza-lo através de, segundo Wikipedia (2017), *clusters* de servidores usando o protocolo de replicação para manter a consistência deste recurso (RICHARDSON, 2016).

2.5. Administração de dados

Em uma aplicação monolítica tem-se, comumente, apenas a utilização de um banco de dados que geralmente é do padrão relacional usufruindo de os seus benefícios. Porém na arquitetura de *microservices,* quando se faz necessário, por padrão não utiliza-se apenas um único banco de dados e, sim, um banco de dados individual por cada parte da aplicação que está separada em vários serviços diferentes. Seguindo este padrão tem o encapsulamento de acesso às informações pelo domínio do serviço diretamente e qualquer outro acesso deve ser realizado através dos recursos que este disponibiliza mantendo assim o baixo acoplamento entre os serviços e a sua independência. Além dessa separação física de acesso aos dados, também, pode-se escolher diferentes tipos de banco

de dados para serem utilizados, por exemplo, um serviço pode utilizar a estrutura relacional enquanto outro pode utilizar a estrutura NoSQL (DRAGONI, 2017).

Com este padrão de separação da persistência de dados por serviços tem suas vantagens, como a já citada independência, assim como facilita o trabalho de escalonamento e performance do mesmo. Porém tem como uma grande complexidade a administração e manutenção dos dados pois trata-se da separação física dos dados que são compreendidos de forma única logicamente. As maiores dificuldades encontradas são como ter o mesmo recurso de uma transação do bando de dados relacional único, para manter consistência, se nesta arquitetura a base de dados está separada e independente uma das outras sendo apenas possível o acesso através das API que cada serviço implementou. Assim como implementar consultas que devolvam dados de múltiplos serviços que se completam logicamente, como exemplo, uma ordem de compra que mostra os itens comprados provenientes de um serviço juntamente com as informações de entrega e do comprador que estão disponibilizados em outros serviços específicos (RICHARDSON, 2016).

2.5.1 *Event-Driven Architecture*

Uma das soluções encontradas para ser seguida como exemplo é o *Event-Driven Architecture* onde um serviço publica um evento quando tem sua base de dados modificada de alguma maneira assim como outro ou outros serviços que respondem a este evento o recebem e realizam outras operações com suas bases de dados. Onde pode ser desde uma atualização, inclusão de novas informações ou o retorno de alguns dados de sua base que complemente ao solicitante. Mesmo utilizando esta solução as operações são realizadas atomicamente apenas dentro do contexto de cada banco de dados e não como um todo sendo assim importante implementar uma rotina de repetir as ações novamente caso aconteça uma falha em algum ponto ou de reportar a todos os serviços envolvidos para realizarem a mesma execução de *rollback* de uma base de dados relacional ou cada serviço deve implementar a sua própria rotina de inconsistência à falhas de dados (Richardson, Chris, Floyd Smith). Existem algumas técnicas que ajudam a implementar a atomicidade das operações entre os bancos de dados dos serviços como um todo que são publicando eventos de transações locais, mineração dos registros de transações do banco de dados ou utilizando *event sourcing* (RICHARDSON, 2016).

2.6. *Microservices Deployment Strategy*

Segundo Richardson (2016), o processo de publicar e disponibilizar um aplicativo seguindo a arquitetura monolítica consiste basicamente em executar um ou mais processos idênticos tornando assim, de certa forma, em um processo sem maiores complexidades. Entretanto na arquitetura de *microservices* existem mais maneiras de se realizar este processo onde os principais padrões estão descritos a seguir.

2.6.1 Múltiplas instâncias de serviços por *host*

Neste padrão é utilizado, fisicamente ou virtualmente, um ou mais *hosts* onde são executadas múltiplas instâncias em cada um. Os benefícios deste padrão são que essas instâncias dividem o mesmo servidor e sua capacidade, sendo assim, mais eficiente em termos de consumo, administração do servidor em si. Outro grande benefício é a rapidez para disponibilizar algum serviço pois tem-se apenas a necessidade de duplicar em si o serviço já que é o mesmo servidor sem a necessidade de realizar mais essa configuração

juntamente com a disponibilização do serviço. Porém em contraponto tem-se a baixa ou a falta de isolamento do servidor, ou seja, caso o servidor tenha algum problema afeta diretamente a arquitetura dos serviços. Assim como se um serviço possa consumir mais recursos do servidor afetará as instâncias restantes. Como cada servidor tem o eu sistema operacional será necessário configurá-los de acordo com cada serviço, assim, necessita que seja aprisionado um servidor com o sistema operacional que execute os determinados serviços compatíveis com sua configuração. Este padrão comumente é o primeiro a ser visto pela equipe de desenvolvimento pois é semelhante ao processo de disponibilização de um aplicativo monolítico (RICHARDSON, 2016).

2.6.2 Instância de Serviço por *host*

Neste padrão tem-se uma instância do serviço por cada *host* assim tem-se naturalmente um isolamento de cada serviço pois cada um utiliza-se apenas um servidor próprio. Este padrão é utilizado através de uma instância por máquina virtual, ou seja, mesmo tendo apenas um servidor físico o mesmo está sendo utilizado por mais de uma máquina virtual contendo apenas a execução de uma instância do mesmo serviço. Os benefícios são o controle de utilização dos recursos do servidor definidos por cada máquina virtual, sendo assim, um serviço não pode ocupar todo os recursos tornando o ambiente de execução isolado um do outro. Através deste encapsulamento pode-se optar por atualizar algum recurso ou não que não afetará outra máquina virtual. Além de poder facilmente migrar entre servidores diferentes ou até mesmo fornecedores destes através da *cloud computing*. Também possuem seus contrapontos onde um destes é menor eficiência em utilização dos recursos devido a essa camada entre o servidor físico e a máquina virtual. Caso não se utilize um software para verificar a saúde da máquina virtual para realizar o escalonamento este processo, de forma manual, é impraticável. Além de que o servidor físico, para disponibilizar o serviço, precisa primeiro executar uma instância da máquina virtual, sendo assim, esse processo é mais custoso e demorado (RICHARDSON, 2016).

2.6.3 Instância de Serviço por *Container*

Neste padrão tem-se uma instância do serviço por cada, segundo TechTerms (2017) *container* que são um mecanismo de virtualização ao nível do sistema operacional. Este padrão consiste em séries de processos sendo executados, segundo TechTerms (2017), dentro de um *sandbox* na qual cada uma destas não interfere na execução de outra. A utilização deste padrão é semelhante à utilização de máquinas virtuais, porém é muito mais rápido e eficaz e, por isso, tem-se os mesmos benefícios além destes. Também, se comparado às máquinas virtuais, são mais leves e rápidos de serem criados pois o serviço é montado dentro da imagem que o *container* irá executar posteriormente assim, a equipe de desenvolvimento, não necessita configurar o sistema operacional da máquina virtual sendo este processo abstraído. Porém os contrapontos deste padrão são que ainda não tão seguros quanto a utilização de máquinas virtuais devido a sua maturidade em relação a tecnologia de virtualização. Também diminuem a segurança e estabilidade do *host* pois cada *container* divide esse mesmo kernel, ou seja, um serviço pode acabar afetando o outro em algum momento caso aconteça alguma anomalia. Necessário a utilização de uma ferramenta para administração de *containers* pois o trabalho manual deste é impraticável (RICHARDSON, 2016).

2.6.4 *Serveless (FaaS)*

Neste padrão tem-se a utilização de algum serviço sem a necessidade de utilização de um sistema operacional contido em um servidor qualquer. Este consiste em uma combinação de serviços, como o *frontent*, *backend* sendo utilizados como *hosts* remotos e sendo executados como funções abstraindo totalmente a questão de qualquer tipo de configuração, manutenção e administração da equipe de desenvolvimento, ou seja, o próprio fornecedor do serviço de *serveless* que tem essa responsabilidade (JANAKIRAMAN, 2016).

Este padrão é como aplicar a arquitetura de *microservices* ao conceito de disponibilização dos mesmos, ou seja, abstraindo totalmente o servidor que executa o serviço em si e levando este a utilização ideal do conceito de *cloud computing,* onde um serviço pode estar sendo executado no Brasil e outro nos Estados Unidos sem qualquer intervenção da equipe de desenvolvedores responsáveis pelo aplicativo. Como vantagens principais temos apenas a necessidade de focar no principal, ou seja, na implementação do serviço e enviar o mesmo para ser executado neste padrão.

O escalonamento se faz automaticamente pelos provedores deste padrão, ou seja, sem intervenção de configuração pelos desenvolvedores, a disponibilização e execução do serviço acontece de maneira mais rápida pois não se tem um *container* ou uma máquina virtual ocupando processamento do servidor, redução de custos pois paga-se apenas por cada execução em si do serviço e não mais juntamente a infraestrutura. Como contrapontos principais tem-se o próprio amadurecimento deste padrão e sua tecnologia, mesmo que os principais provedores de *cloud computing* já o ofereçam, ainda é um padrão relativamente novo atualmente.

Como mencionando, o provedor deste padrão é que será responsável em como gerenciar o mesmo assim o proprietário do aplicativo perde certos controles de customizações caso queira realizar estes. Aplicativos que são desenvolvidos para serem executados como, segundo Wikipedia (2017), multitenant devem implementar este mecanismo pois o serviço em si é fornecido como se fosse uma função. A transição de um serviço neste padrão para outro provedor ainda não encontra a mesma facilidade encontrada nos *containers* ou máquinas virtuais (ROBERTS, 2016).

3. Principais estratégias de migração de um aplicativo monolítico para *microservices*

O processo de atualizar um aplicativo, seja através de correções, novas funcionalidades ou novas tecnologias e arquitetura, que ocorre de tempos em tempos é algo rotineiro no contexto de empresas e profissionais que trabalham com desenvolvimento de software.

Ao escolher por atualizar a tecnologia e arquitetura de um aplicativo é de extrema importância que se adote algumas estratégias que aumentem o sucesso deste processo trabalhoso e que tomará bastante tempo e dinheiro investido pela empresa. No contexto de migrar uma aplicação desenvolvida utilizando a arquitetura monolítica para a arquitetura de *microservices* tem-se algumas estratégias sugeridas que auxiliam nesse processo (SMITH, 2017).

Uma delas é não migrar totalmente a aplicação monolítica e, sim, migrar de forma incremental iniciando pelas novas funcionalidades em *microservices* que interagem com o aplicativo monolítico. Para que, então, inicia-se o processo de migrar as funcionalidades

que estão na arquitetura antiga para a nova levando em consideração a prioridade de cada uma como inicio desta transição. Esta estratégia, segundo Fowler (2004), é denominada de *Strangler Application* (RICHARDSON, 2016).

3.1 *Stop Digging*

Quando a aplicação monolítica é bastante complexa e grande, tornando-a dificilmente gerencial, qualquer atualização que seja torna-se algo complexo então é recomendável que estas novas implementações devam ocorrer em serviços separados e independentes, tanto da aplicação monolítica quanto de outros serviços, assim evitando-se de adicionar maiores complexidades à uma aplicação já considerada complexa (RICHARDSON, 2016).

Para seguir esta estratégia é necessário que se tenha dois componentes importantes para que os serviços consigam interagir com a aplicação monolítica de forma que trabalhem juntas. O primeiro componente necessário é um roteador que será responsável em, justamente, receber as requisições seguindo a mesma abordagem da API *Gateway,* porém, este envia a requisição para as novas funcionalidades assim retirando estas requisições da aplicação monolítica. O outro componente, segundo Wikipedia (2017), é chamado de *glue code* que tem a responsabilidade de integrar o serviço novo ao aplicativo legado na questão de transferência de dados. Este recurso é conhecido também, segundo Microsoft (2017), como *anti-corruption layer* pois preveni que o serviço seja poluído, de alguma forma, pelos conceitos contidos no monolítico. Esta prevenção ocorre pelo *glue code* que implementa a transferência de dados entre ambos aplicativos implementando os modelos de cada um e realizando assim a compreensão e transferência (RICHARDSON, 2016).

Através desta estratégia ganha-se tempo para implementar as funcionalidades que estão no monolítico conforme a necessidade e mantendo a complexidade do mesmo no mesmo nível ao invés de aumentá-lo ou tornando o processo de migração de uma arquitetura para outra complexa e perigosa a certo ponto (RICHARDSON, 2016).

3.2 Dividir *Frontend* e *Backend*

Uma aplicação monolítica comumente está implementada utilizando as camadas de apresentação, regras de negócios e acesso a dados que estão separadas entre si logicamente ou até fisicamente, porém altamente acopladas. Ao separar a camada de apresentação (*Frontend*) das camadas de regras de negócios e acesso a dados em uma outra camada (*Backend*) onde cada uma comunica-se entre si através de requisições remotas. Esta separação traz os benefícios de poder separar o processo de desenvolvimento por camada além de poder escalonar uma ou outra conforme a necessidade.

Esta estratégia é uma solução parcial para a migração para a arquitetura de *microservices* onde auxilia a equipe de desenvolvedores a iniciarem o processo de separação das responsabilidades da aplicação assim como a separar as funcionalidades em serviços que funcionarão futuramente independentes (RICHARDSON, 2016).

3.3 Extraindo serviços

Esta é a mais importante das estratégias a serem pensadas e colocada em prática pela equipe de desenvolvimento responsável em realizar a migração de arquitetura monolítica

para de *microservices*. Esta estratégia consiste em priorizar qual módulo tem a maior necessidade de ser separada do monolítico e disponibilizada através de um serviço. Ao realizar este processo será necessário seguir a primeira estratégia apresentada, ou seja, este módulo importante extraído do monolítico para o de um serviço ainda deve continuar se comunicando com os outros módulos que dependem do mesmo.

Este processo é o mais trabalhoso e deve ser realizado até o último módulo existente no monolítico. Porém, a cada migração realizada, o trabalho de manutenção do monolítico torna-se menos complexa devido a este desacoplamento (RICHARDSON, 2016).

4. Estudo de Caso: Desenvolvimento de uma aplicação monolítica e microservices comparando ambas

Para a realização do estudo de caso foi utilizado o contexto de um site de ecommerce simples na qual foram desenvolvidas duas versões do *Backend*. Onde uma segue a arquitetura monolítica e a outro a de *microservices*. Para tanto ambas arquiteturas utilizam o mesmo *Frontend* adaptado para consultar os *endpoints* do *Backend*.

4.1 Ecommerce *Frontend*

O *Frontend* utilizado é um projeto de demonstração de um ecommerce de roupas desenvolvido utilizando o conceito de SPA e a tecnologia de Web componentes através da ferramenta chamada de Polymer Project sendo a versão para Windows 10. Este é um projeto *open-source* liderado por uma equipe de desenvolvedores *Frontend* da organização do Chrome dentro do Google. Onde tem o foco em tornar o desenvolvimento de interfaces para a Web mais fácil e rápido através da utilização dos componentes da plataforma Web já existentes e disponibilizados assim como outros que venham a ser criados pela equipe ou pela a comunidade. Esta ferramenta auxilia aos desenvolvedores a criarem *Frontend* nativamente seguindo o conceito, segundo Google (2017), de *Progressive Web Apps* (GOOGLE, 2017).

A estrutura do ecommerce está dividido nas áreas da página principal (*home)*, lista de produtos para a compra, detalhes do produto para a compra, carrinho de compras e finalização da compra. Estes, na versão de demonstração, estão totalmente utilizando dados estáticos e cada parte foi alterada para consultar uma API de *Backend* para dinamizar o mesmo (GOOGLE, 2017).

4.2 Ecommerce *Backend*

O *Backend* foi desenvolvido utilizando a tecnologia, segundo Microsoft (2017), ASP.NET Core versão 2.0 através da linguagem de desenvolvimento C# e como persistência de dados utilizou-se o SGDB PostgreSQL versão 10 para o sistema operacional Windows 10 (POSTGRESQL, 2017).

Neste desenvolvimento foram realizadas duas versões onde uma seguindo a arquitetura monolítica e a outra de *microservices,* porém ambas seguiram o padrão de separação entre o *Frontend* e o *Backend.* Este padrão de separação é comumente utilizado quando se desenvolve o *Backend* com o conceito de Web API, na qual também, é uma das estratégias de migração de uma aplicação monolítica para de serviços. Além de seguirem o padrão de desenvolvimento DDD através do conceito de BC (WIKIPEDIA, 2017).

4.2.1 Estrutura do *Backend* monolítico

O *Backend* monolítico segue a estrutura de separação de módulos em forma de componentes de biblioteca, que implementam as regras de negócios, sendo separados fisicamente uma das outras dentro da mesma solução, porém altamente acoplados a dependência entre as mesmas. Além dos componentes de biblioteca tem-se o componente que interage com as demais e disponibiliza parte dos recursos para serem executados externamente através de *endpoints* de requisição da Web. Os outros recursos são executados conforme as regras de negócios e entre as chamadas de uma biblioteca para a outra biblioteca conforme Figura 6.

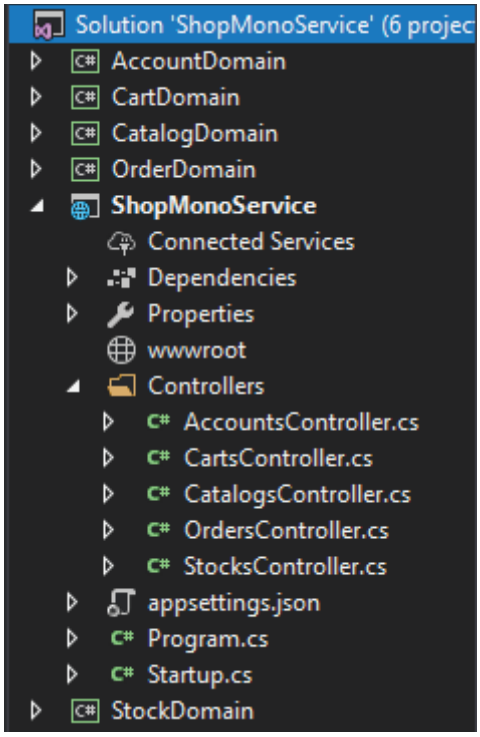

Figura 6. Estrutura do ecommerce monolítico

Fonte: Autor

Cada componente tem a sua responsabilidade conforme descrita na Tabela 1.

Tabela 1. Componentes da arquitetura e suas responsabilidades

Componente	Descrição
AccountDomain	Responsável em implementar as regras de negócios de conta/registro do usuário.
CartDomain	Responsável em implementar as regras de negócios do carrinho de compras do usuário.
CatalogDomain	Responsável em implementar as regras de negócios do catálogo de produtos disponíveis para a compra por um usuário.
OrderDomain	Responsável em implementar as regras de negócios sobre o pedido de compra do usuário.

ShopMonoService	Implementação da Web API executando cada regra de negócio e seus fluxos através da chamada das bibliotecas que ocorrem pelos *Controllers*.

Fonte: Autor

Para a persistência de dados tem-se o banco de dados denominado ShopMonoBd contendo a estrutura conforme Figura 7.

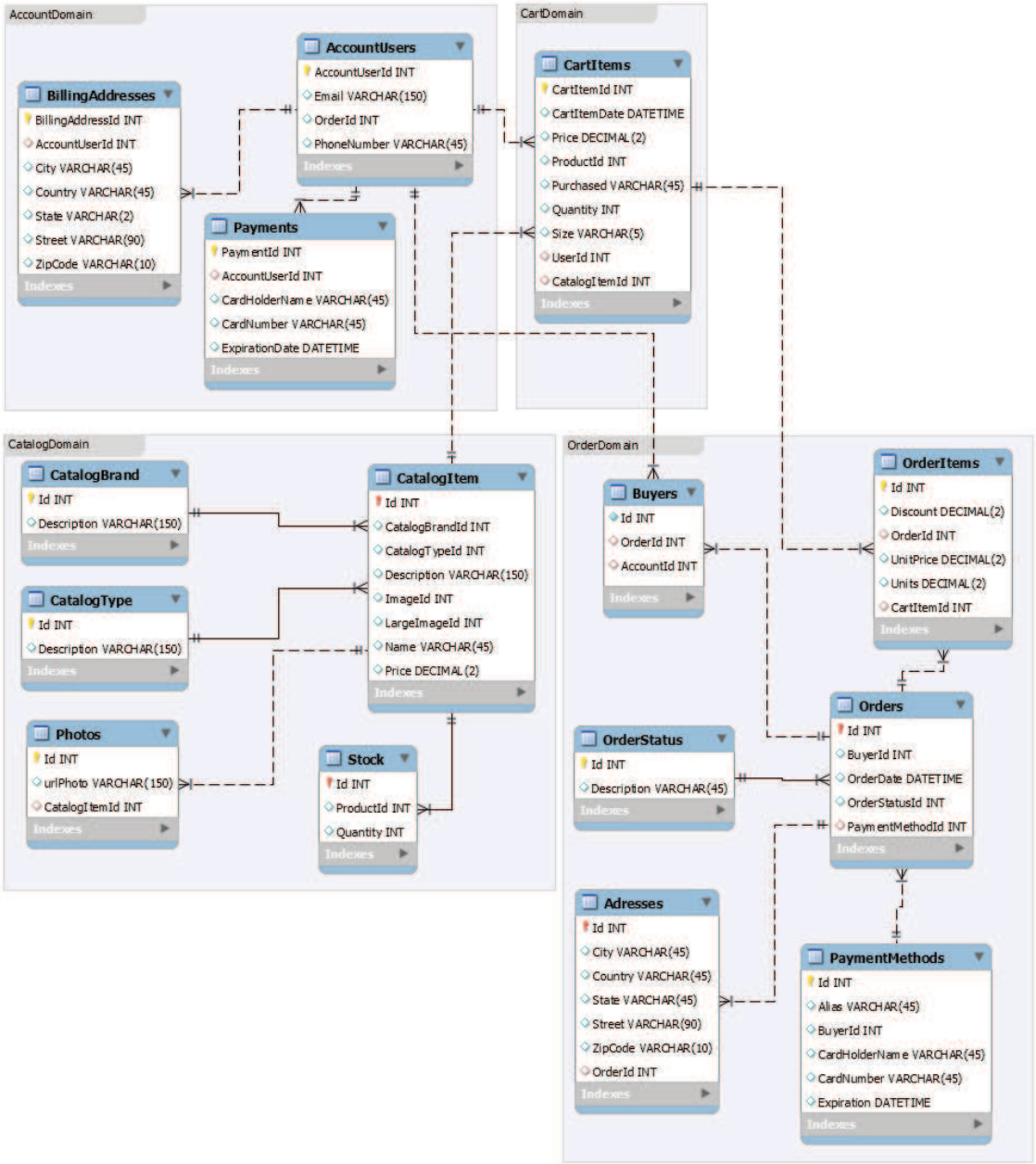

Figura 7. DER do banco de dados da arquitetura monolítico

Fonte: Autor

4.2.2 Estrutura do *Backend* em *microservices* e Impressões

O *Backend* em *microservices* segue os padrões conforme apresentados pela Seção 2. Sendo assim, antes de iniciar a estrutura em si, aplicou-se as principais estratégias de migração de uma arquitetura monolítica para uma em *microservices* na qual cada componente de biblioteca tornou-se um serviço isolado, separado fisicamente e sem acoplamento direto pois cada um implementa a sua funcionalidade na qual essa pode ser consumida por outros serviços através de *endpoints*. Pois adotou-se esse padrão de desenvolvimento Web API para cada serviço.

Além deste tipo de comunicação, também, identificou-se a necessidade de implementar outro padrão de comunicação através de mensagens AMQP para os cenários onde devam ser totalmente assíncronas as requisições. Neste contexto foi utilizado o cliente de mensagens RabbitMQ por ser multiplaforma e com suporte ao ASP.NET Core e a linguagem C#. Este recurso foi implementado no serviço de CartDomain onde pode-se interagir através dos seus *endpoints* ou via RabbitMQ prevendo assim uma flexibilidade de acordo com a operação necessária. No caso ao inserir um item no carrinho de compras essa interação ocorre com este serviço de modo a requisição ao seu *endpoint* devido a necessidade instantânea de essa operação ocorrer enquanto o usuário ainda está acessando o site. Já ao escolher remover um item do carrinho de compras este se comunica com o serviço através de mensagens AMQP onde esta remoção será processada assim que a mensagem seja recebida. Neste cenário, caso o usuário saia do site por algum motivo qualquer, esta ação será concluída igualmente.

Em relação a utilização deste tipo de comunicação através do RabbitMQ foi o processo que mais teve impacto no desenvolvimento pois foi necessário instalar, configurar e monitorar essa camada. Este possui um instalador próprio na qual instala ele em si, porém, anteriormente é necessário instalar o Erlang separadamente conforme as instruções de instalação. Após instalado, então, deve-se seguir uma série de instruções para registrá-lo como um serviço do Windows, liberar portas do *Firewall* além de instalar o cliente Web de administração via linha de comando, informado conforme as instruções. Este administrador Web, que é desenvolvido em Python, tem todas as funcionalidades que permitem realizar toda configuração necessária e monitoração deste recurso facilitando a equipe de desenvolvedores e/ou a equipe responsável por administrar e monitorar este recurso. Outra dificuldade encontrada pelo autor foi o entendimento de como seria a assinatura entre as mensagens para realizar o método de *Publish/async Response*. Está assinatura, nada mais é, do que uma classe em si na qual será instanciada duas cópias onde uma irá publicar (*Publish*) invocando a outra cópia, que responderá (*Response*). Isto acontece através de uma configuração realizada na inicialização de cada projeto do serviço, através de um drive de conexão do RabbitMQ para C# chamada, segundo RawRabbit (2017), de RawRabbit, onde um serviço tem essa assinatura da classe como *Publish* e o outro serviço tem essa assinatura de classe como *Response* configurados em suas inicializações. Para evitar a duplicação de código optou-se por criar uma biblioteca chamada de AMQP na qual é compartilhada pelos demais serviços que utilizam esse tipo de mensagens contendo as classes de mensagens e modelos que são utilizados pelas mensagens. Sem a utilização deste drive, RawRabbit, a utilização deste recurso seria muito mais complexa, como exemplo, ter que configurar até o canal de transmissão de dados. Utilizando este drive essa complexidade ficou-se abstraída pelo desenvolvimento.

Ao separar as bibliotecas em serviços isolados identificou-se que seria melhor dividir o serviço de CatalogDomain em três ao total. Neste caso, o primeiro continuou sendo chamado de CatalogDomain, tem-se as regras de negócios sobre o seu contexto e a persistência em si dos dados, através do seu banco de dados exclusivo. Para a disponibilização das imagens assim como o seu armazenamento criou-se o novo serviço, denominado de PhotoDomain, contendo as responsabilidades em disponibilizar as imagens dos produtos conforme os formatos necessários. Finalizando essa divisão tem-se o outro novo serviço, denominado de StockCatalog, que é responsável em gerir as regras de negócios relativo ao estoque de cada produto. Anteriormente o CatalogDomain estava realizando ambas funcionalidades na qual sobrecarregava o mesmo e até conflitava de certa maneira tornando-o esta biblioteca cada vez mais complexa em termos de manutenção na arquitetura antiga.

Outra necessidade encontrada para alterar foi como o *Frontend* fosse acessar o *Backend*, assim, neste sentido e seguindo as boas práticas foi criado o serviço de API *Gateway* contendo as responsabilidades de mapear os *endpoints* dos serviços para fornecer o acesso pelo *Frontend*. Desta maneira nenhum serviço é acesso diretamente por algum cliente externo. Outra funcionalidade, que não implementada nesta versão, é a verificação da autenticação do usuário para poder executar alguns recursos do ecommerce, como exemplo, adicionar um item ao carrinho de compras. A Figura 8 representa a atual arquitetura em *microservices*.

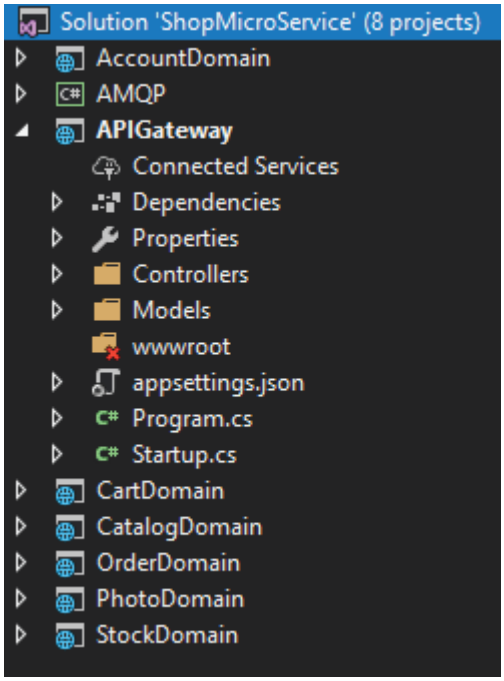

Figura 8. Projeto do *Backend* seguindo a arquitetura de *microservices*

Fonte: Autor

Em relação à persistência dos dados, quando necessário, o banco de dados da versão monolítica foi dividido conforme os serviços seguindo as boas práticas, sendo assim, cada banco de dados somente é acessível pelo serviço que o implementa e conforme as regras de negócios, como exemplo, a ordem de compra de um produto deve armazenar as informações sobre o produto comprado. Neste cenário foi escolhido a estratégia de duplicar estas informações para diminuir a latência da requisição da

informação e, também, esta informação dificilmente é modificada posteriormente a compra já que representa a mesma. Porém, nos outros cenários, como o estoque de um produto optou-se em apenas persistir o código do produto que é a sua chave de identificação para controlar a quantidade estocada e a disponibilidade do mesmo. Ao final dessa migração o banco de dados que era único dividiu-se em seis ao total sendo cada um para seu respectivo serviço, porém mantendo a mesma versão do PostgreSQL 10 para ambas instâncias assim como o utilizado para versão monolítica (Figura 9).

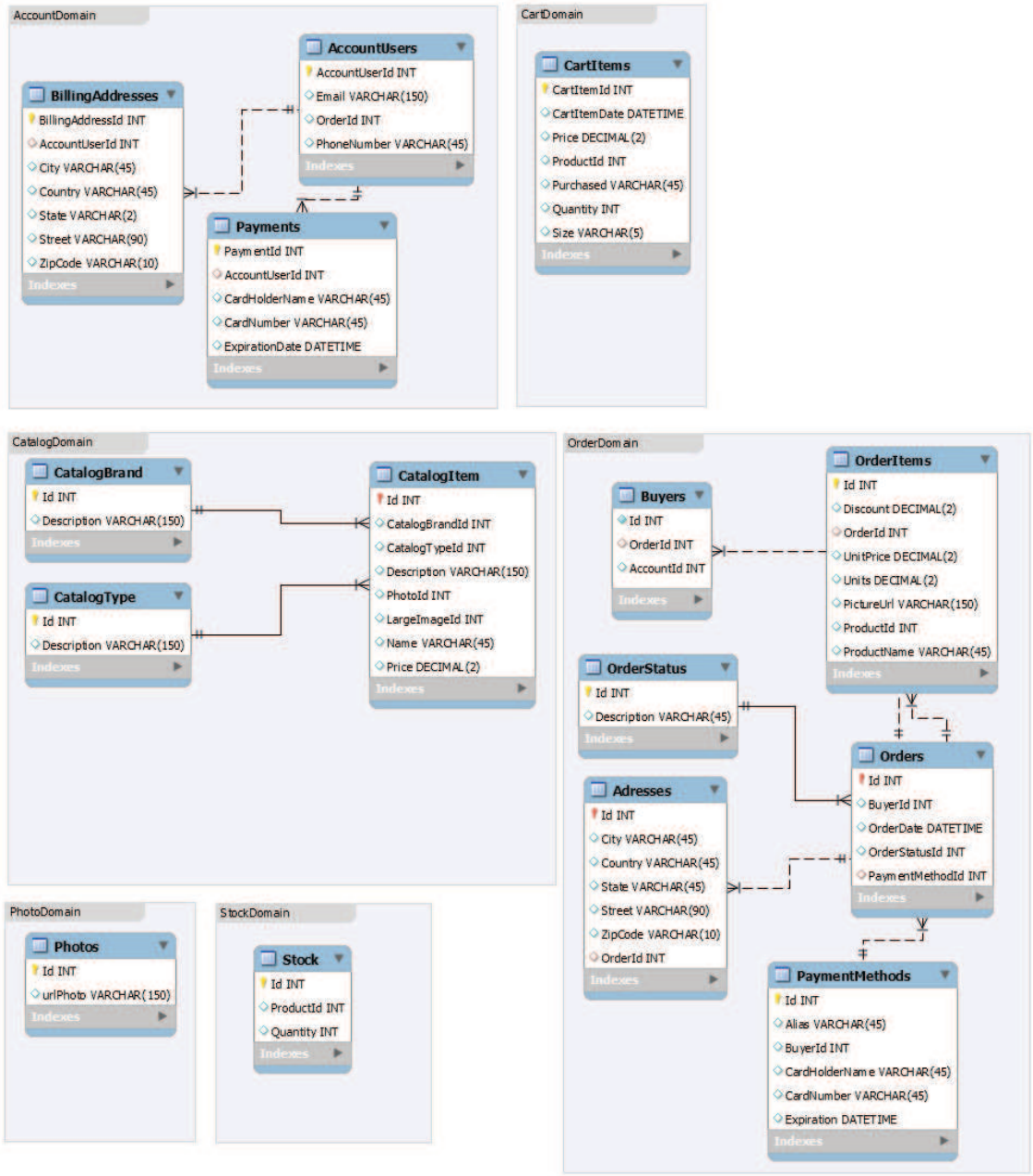

Figura 9. DER do banco de dados da arquitetura *microservices*

Fonte: Autor

5. Questionário aplicado a desenvolvedores e afins

Com o objetivo de mensurar como o mercado e a comunidade de desenvolvedores e afins estão vendo esse novo paradigma de desenvolvimento, assim como a sua aceitação e adoção, foi aplicada uma pesquisa contendo dez questões objetivas sobre o assunto. A pesquisa foi realizada durante o período de cinco dias, sendo disponibilizada *online* através de alguns canais de comunidade de tecnologia da rede social Facebook.

A pesquisa obteve um resultado total de trinta e quatro respostas, na qual o tempo de experiência dos entrevistados varia entre alguns meses até vinte e cinco anos de experiência. Destaca-se os seguintes resultados abaixo por questão:

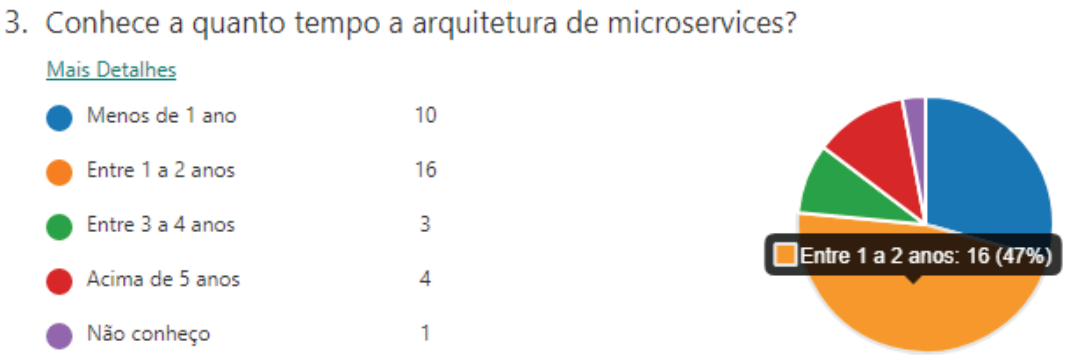

Figura 10. Resultado da pergunta sobre o tempo de conhecimento sobre *microservices*

Fonte: Autor

A Figura 10 mostra os resultados em relação ao tempo que os entrevistados conhecem a arquitetura de *microservices*. Verificou-se que o tempo médio de conhecimento sobre o assunto é entre 1 e 2 anos, sendo representando por 47% das respostas. Logo em seguida tem-se o tempo inferior a 1 ano. Com isso mostra-se que o assunto ainda é, de certa maneira, novidade entre o público da pesquisa.

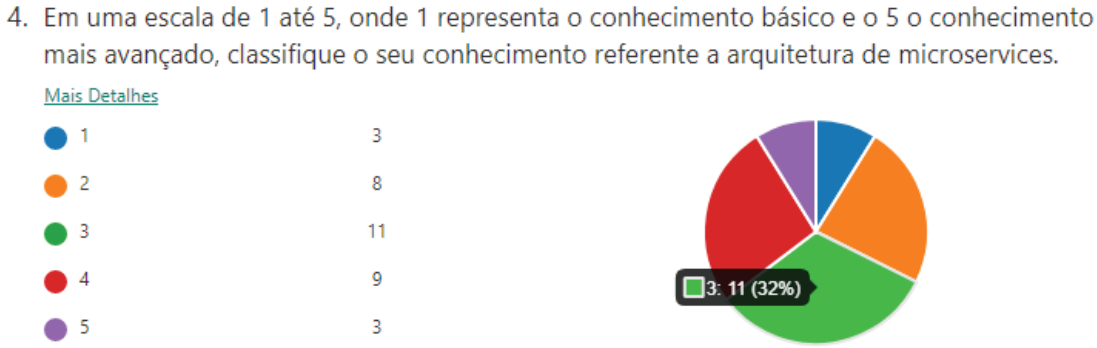

Figura 11. Resultado da pergunta relativa ao grau de conhecimento sobre *microservices*

Fonte: Autor

Contatou-se, conforme a Figura 11, que o grau de conhecimento dos entrevistados está representado em na escala de nível 3, representado por 32%, mostrando assim que a maioria acredita que o seu conhecimento seja mediano sobre a arquitetura.

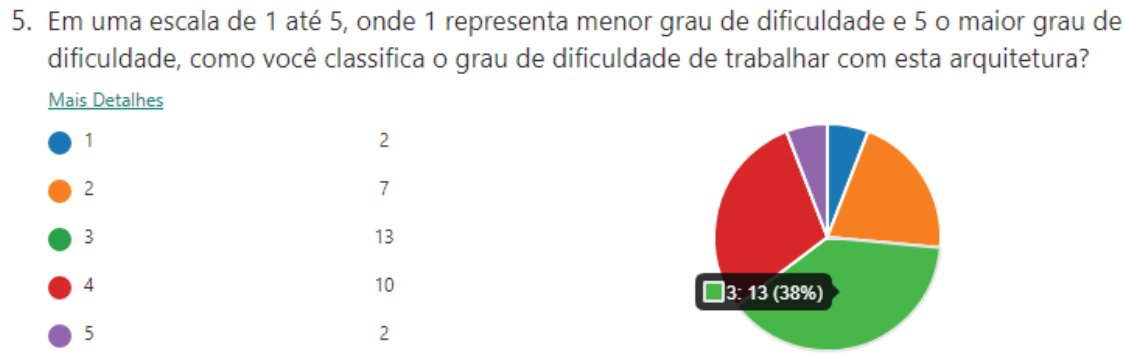

Figura 12. Resultado da pergunta relativa ao grau de dificuldade em trabalhar com *microservices*

Fonte: Autor

Contatou-se que, em relação ao grau de dificuldade em se trabalhar com *microservices*, a maioria dos entrevistados, representados por 38%, considera este mediano logo seguido por 29% acima de mediana. Deste modo tem-se uma visão sobre como, de fato, esta arquitetura tem suas complexidades em termos de dificuldades conforme demonstrado pela Figura 12.

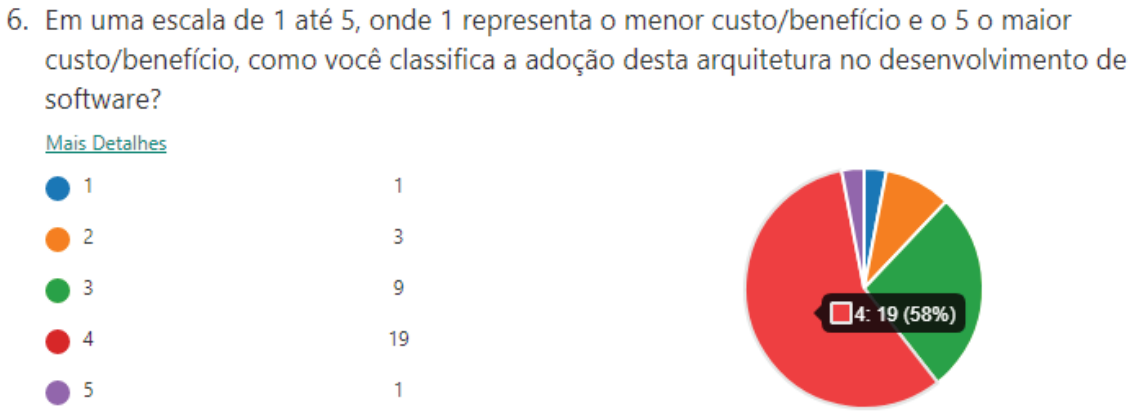

Figura 13. Resultado da pergunta relativa ao custo/benefício sobre a adoção de *microservices*

Fonte: Autor

Conforme pode-se constatar pela Figura 13, em relação ao custo/benefício na adoção desta arquitetura de *microservices*, verificou-se que, mesmo com o grau entendido de dificuldade seja elevado, os entrevistados têm como opinião que este custo/benefício seja alto sendo representado por 58% das respostas.

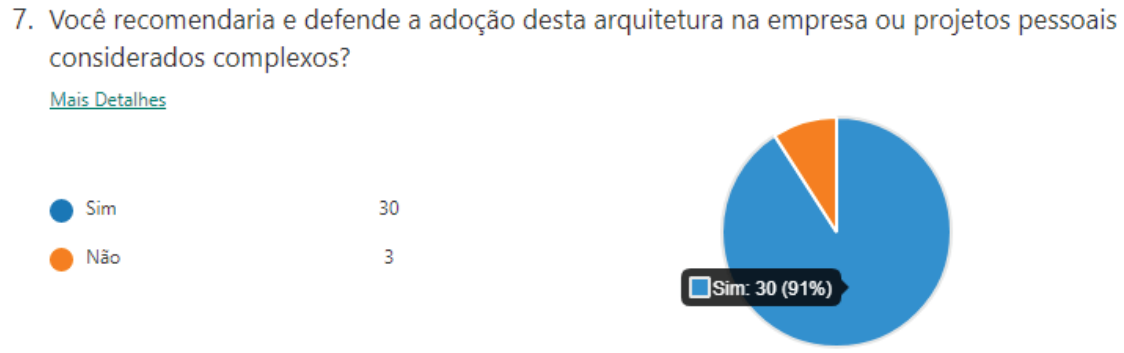

Figura 14. Resultado da pergunta relativa a adoção de *microservices*

Fonte: Autor

Ao questionar os entrevistados, como verificado pela Figura 14, se recomendariam e defenderiam a adoção da arquitetura de *microservices* em seus projetos considerados complexos teve-se como resultado afirmativo de recomendação positiva de 91% por parte do público. Esta informação foi considerada elevada e, totalmente, surpreendente por parte do autor e demonstra como uma nova realidade para o desenvolvimento de software.

O questionário completo, bem como as respostas obtidas, é apresentado no APÊNDICE A.

6. Conclusão

Com o avanço da utilização da computação em nuvem e a crescente utilização de diversos dispositivos conectados a internet visando ao aumento de produtividade tanto no quesito de processamento de dados e velocidade a atual utilização de desenvolvimento de software utilizando a arquitetura monolítica não atende à demanda atual.

Para suprir essa atual demanda, tem-se o novo paradigma no desenvolvimento de software, que é a utilização da arquitetura de *microservices*. Mostrando um desenvolvimento de aplicativos mais ágeis e flexíveis, se comparado ao monolítico, porém, assim como qualquer outra arquitetura, traz consigo seus benefícios e contrapontos.

Esse artigo apresentou argumentos para que o mercado e a comunidade de desenvolvimento de software adotem a arquitetura de *microservices*, tanto na migração dos projetos legados quanto aos novos, apresentando através do referencial teórico os pontos positivos, contrapontos e as melhores estratégias de como adotar este novo paradigma. Para auxiliar neste processo foi realizado um estudo de caso prático com uma aplicação simples desenvolvida na arquitetura monolítica para a arquitetura de *microservices* demonstrando a impressão do autor sobre esse processo juntamente com o pesquisado pelo referencial teórico.

Outro ponto levantando por este artigo foi uma pesquisa realizada com desenvolvedores e afins para mensurar sobre como o público está vendo este novo paradigma assim como a sua aceitação. Sendo que este conclui-se que mesmo sendo um assunto mais atual e independente do tempo de experiência, nível de educação e o porte da empresa a maioria conhece esta a arquitetura de *microservices* e a recomenda para adoção do mesmo.

Sendo assim, conclui-se, que o presente artigo contribuiu para o melhor esclarecimento desta arquitetura de *microservices* apresentando para a área de desenvolvimento de software um novo paradigma a ser seguido para a disponibilização de soluções que atendam a atual realidade da computação cada vez mais distribuídas e necessárias em nossas vidas.

Para trabalhas futuros tem-se as possibilidades de continuar está pesquisa em relação à utilização de *containers* demonstrando suas qualidades e desvantagens, verificação do escalonamento conforme a demanda de cada serviço, a utilização de um ambiente heterogêneo de tecnologias, ou seja, cada serviço utilizando uma linguagem de tecnologia e bases de dados diferentes dos outros para verificar esse funcionamento nesta arquitetura.

References

Richardson, C., Smith, F. (2016) "MICROSERVICES From Design to Deployment", NGINX.

Microservice.io. Microservice Architecture. Disponível em:<http://microservices.io>. Acesso em: 05 ago. 2017.

Torre, C, Wagner, B, Rousos, M. (2017) "Architecting and Developing Containerized and Microservice based .NET Applications: Early Draft", Redmond, Microsoft Corp.

Dragoni, N, Dustdar, S, Larsen, S, Mazzara, M. (2017), "Microservices: Migration of a Mission Critical System", New York, Cornell University.

Roberts, M, Fowler, M. (2016) "Serveless Architectures", http://martinfowler.com/articles/serverless.html.

Fowler, M. (2004) "Stangler Application", http://martinfowler.com/bliki/StranglerApplication.html.

Janakiraman, B, Fowler, M. (2016) "Serveless", http://martinfowler.com/bliki/Serverless.html.

Smith, S, Torre, C (2017) "Architecting and Developing Modern Web Applications with ASP.NET Core and Microsoft Core and Microsoft Azure: Early Draft", Redmond, Microsoft Corp.

Nadareishvili, I, Mitra, R, McLarty, M, Amundsen, M. (2016) "Microservice Architecture", Sebastopol, O'Reilly.

Mozilla. Promise. MDN web docs. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Promise>. Acesso em: 20 out. 2017.

AMQP. Advanced Message Queuing Protocol. Disponível em: <https://www.amqp.org>. Acesso em: 21 out. 2017.

ZeroMQ. Distributed Message Messaging. Disponível em: <http://zeromq.org>. Acesso em: 21 out. 2017.

Apache. Apache Thrift. Disponível em: <https://thrift.apache.org>. Acesso em: 21 out. 2017.

Microservices. Service Registry. Disponível em: <http://microservices.io/patterns/service-registry>. Acesso em: 21 out. 2017.

Pivotal. RabbitMQ. RabbitMQ Messaging. Disponível em: <http://www.rabbitmq.com>. Acesso em: 21 out. 2017.

Apache. Apache Kafka. A distributed streaming plataform. Disponível em: <http://kafka.apache.org>. Acesso em: 21 out. 2017.

Apache. Apache ACTIVEMQ. Disponível em: <http://activemq.apache.org>. Acesso em: 21 out. 2017.

NSQ. Realtime distributed messaging plataform. Disponível em: <http://github.com/nsqio/nsq>. Acesso em: 21 out. 2017.

Wikipedia. Buffer. Disponível em: <https://pt.wikipedia.org/wiki/Buffer_(ci%C3%AAncia_da_computa%C3%A7%C3%A3o>. Acesso em: 21 out. 2017.

Wikipedia. Representational State Transfer. Disponível em: <https://pt.wikipedia.org/wiki/Representational_state_transfer >. Acesso em: 23 out. 2017.

TECHTARGET NETWORK. XML. Extensible Markup Language. Disponível em: <http://searchmicroservices.techtarget.com/definition/XML-Extensible-Markup-Language>. Acesso em: 23 out. 2017.

JSON. Introduction JSON. Disponível em: <https://www.json.org>. Acesso em: 24 out. 2017.

curl. Command line tool and library for transferring data with URLs. Disponível em: <https://curl.haxx.se>. Acesso em: 25 out. 2017.

Microsoft. Microsoft Azure. O que é computação em nuvem. Disponível em: <https://azure.microsoft.com/pt-br/overview/what-it-cloud-computing>. Acesso em: 26 out. 2017.

Wikipedia. Computer cluster. Disponível em: <https://en.wikipedia.org/wiki/Computer_cluster>. Acesso em: 26 out. 2017.

TechTerms. Container. Disponível em: <https://techterms.com/definition/contaiber>. Acesso em: 27 out. 2017.

TechTerms. Sandboxing. Disponível em: <https://techterms.com/definition/sandboxing>. Acesso em: 29 out. 2017.

Wikipedia. Multitenancy. Disponível em: <https://en.wikipedia.org/Multitenancy>. Acesso em: 29 out. 2017.

Wikipedia. Buffer. Disponível em: <https://pt.wikipedia.org/wiki/Buffer_(ci%C3%AAncia_da_computa%C3%A7%C3%A3o>. Acesso em: 29 out. 2017.

Wikipedia. Glue code. Disponível em: <https://pt.wikipedia.org/wiki/Glue_code>. Acesso em: 29 out. 2017.

Microsoft. Microsoft Azure. Anti-Corruption Layer pattern. Disponível em: <https://docs.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer>. Acesso em: 30 out. 2017.

Google. Polymer-project. Polymer. Unlock the Power of Web Components. Disponível em: <https://www.polymer-project.org>. Acesso em: 01 dez. 2017.

Google. Progressive Web Apps.. Disponível em: <https://developers.google.com/web/progressive-web-apps>. Acesso em: 01 dez. 2017.

Microsoft. ASP.NET. Welcome to ASP.NET Core. Disponível em: <https://www.aspnet./core/overview/aspnet-vnext>. Acesso em: 02 dez. 2017.

PostgreSQL. Disponível em: <https://www.postgresql.org>. Acesso em: 02 dez. 2017.

Wikipedia. Web API. Disponível em: <https://pt.wikipedia.org/wiki/Web_API>. Acesso em: 02 dez. 2017.

RawRabbit.. Disponível em: <https://github.com/pardahlman/RawRabbit>. Acesso em: 02 dez. 2017.

APÊNDICE A

Questionário sobre a utilização de *microservices* no desenvolvimento de software.

34	02:50	Ativo
Respostas	Tempo médio para concluir	Status

1. Quanto tempo de experiência você possui?

34
Respostas

Respostas Mais Recentes
"25 anos"
"5 anos"
"Alguns meses"

2. Qual tecnologia você mais utiliza no seu dia a dia?

34
Respostas

Respostas Mais Recentes
"Java "
"Angular 2"
"C#"

3. Conhece a quanto tempo a arquitetura de microservices?

Menos de 1 ano	10
Entre 1 a 2 anos	16
Entre 3 a 4 anos	3
Acima de 5 anos	4
Não conheço	1

4. Em uma escala de 1 até 5, onde 1 representa o conhecimento básico e o 5 o conhecimento mais avançado, classifique o seu conhecimento referente a arquitetura de microservices.

1	3
2	8
3	11
4	9
5	3

5. Em uma escala de 1 até 5, onde 1 representa menor grau de dificuldade e 5 o maior grau de dificuldade, como você classifica o grau de dificuldade de trabalhar com esta arquitetura?

1	2
2	7
3	13
4	10
5	2

6. Em uma escala de 1 até 5, onde 1 representa o menor custo/benefício e o 5 o maior custo/benefício, como você classifica a adoção desta arquitetura no desenvolvimento de software?

1	1
2	3
3	9
4	19
5	1

7. Você recomendaria e defende a adoção desta arquitetura na empresa ou projetos pessoais considerados complexos?

Sim	30
Não	3

8. Qual o porte da empresa ou projeto que você trabalha ou trabalhou?

Pequena	8
Média	15
Grande	11

9. Você possui nível superior completo?

Sim	21
Não	13

10. Você possui algum curso de pós graduação?

Sim	11
Não	23