

# Entendendo **MICROSERVICES**



**LuizTools**

# SUMÁRIO

<b>SOBRE O AUTOR</b>	4
<b>ANTES DE COMEÇAR</b>	8
Para quem é este livro	9
<b>ARQUITETURA MICROSERVICES</b>	10
A Motivação	11
Vantagens e Desvantagens	14
Tenha em mente	15
O quão “micro”?	16
Como começar?	16
<b>ARQUITETURA ORIENTADA A SERVIÇOS</b>	18
O que é SOA?	19
E os microservices?	20
Entendendo as diferenças	21
Governança descentralizada	22
Gerenciamento descentralizado de dados	23
<b>GESTÃO E SEGURANÇA</b>	25
API Gateway	26
Segurança	30
API Keys	30
JSON Web Tokens (JWT)	30
OAuth	31
JSON Web Token	31
O risco do JWT	32
Entendendo o JWT	33
Assinatura assimétrica do JWT	35

<b>AVANÇANDO COM MICROSERVICES .....</b>	38
Boas práticas com micro serviços .....	39
Tente alcançar a Glória do REST .....	39
Use um serviço de configuração .....	40
Geração automática de código cliente .....	41
Continuous Delivery .....	41
Monitoramento e Logging .....	41
API Gateway .....	42
Refatoração de monolitos .....	42
Estratégia #1 - Pare de Cavar .....	43
Estratégia #2 - Separe front-end de back-end .....	45
Estratégia #3 - Extrair Serviços .....	47
Microservices e Agile: o futuro da programação? .....	49
<b>SEGUINDO EM FRENTE .....</b>	52

# **SOBRE O AUTOR**

---

Luiz Fernando Duarte Júnior é Bacharel em Ciência da Computação pela Universidade Luterana do Brasil (ULBRA, 2010) e Especialista em Desenvolvimento de Aplicações para Dispositivos Móveis pela Universidade do Vale do Rio dos Sinos (UNISINOS, 2013).

Carrega ainda um diploma de Reparador de Equipamentos Eletrônicos (SENAI, 2005), nove certificações em Métodos Ágeis de desenvolvimento de software por diferentes certificadoras (PSM-I, PSD-I, PACC-AIB, IPOF, ISMF, IKMF, CLF, DEPC, SFPC) e três certificações de coach profissional pelo IBC (Professional & Self Coach, Life Coach e Leader Coach).

Atuando na área de TI desde 2006, na maior parte do tempo como desenvolvedor, é apaixonado por desenvolvimento de software desde que teve o primeiro contato com a linguagem Assembly no curso de eletrônica. De lá para cá teve oportunidade de utilizar diferentes linguagens em diferentes sistemas, mas principalmente com tecnologias web, incluindo ASP.NET, JSP e, nos últimos tempos, Node.js.

### **Foi amor à primeira vista e a paixão continua a crescer!**

Trabalhando com Node.js desenvolveu diversos projetos para empresas de todos os tamanhos, desde grandes empresas como Softplan até startups como Busca Acelerada e Só Famosos, além de ministrar palestras e cursos de Node.js para alunos do curso superior de várias universidades e eventos de tecnologia.

Um grande entusiasta da plataforma, espera que com esse livro possa ajudar ainda mais pessoas a aprenderem a desenvolver softwares com Node.js e aumentar a competitividade das empresas brasileiras e a empregabilidade dos profissionais de TI.

Além de viciado em desenvolvimento, como consultor em sua própria empresa, a DLZ Tecnologia e é autor do blog [www.luiztools.com.br](http://www.luiztools.com.br), onde escreve semanalmente sobre métodos ágeis e desenvolvimento de software, bem como mantenedor do canal [LuizTools](#), com o mesmo propósito.

Entre em contato, o autor está sempre disposto a ouvir e ajudar seus leitores.

#### **SOBRE O AUTOR**



# MEUS CURSOS

## Curso online NODE.JS e MONGODB

[SAIBA MAIS...](#)

## Curso online Scrum e métodos Ágeis

[SAIBA MAIS...](#)

## Curso online Jira

[SAIBA MAIS...](#)

## Curso online Web Full Stack JavaScript

[SAIBA MAIS...](#)

## Curso online React Native com Firebase

[SAIBA MAIS...](#)

Conheça todos os meus cursos

# MEUS LIVROS



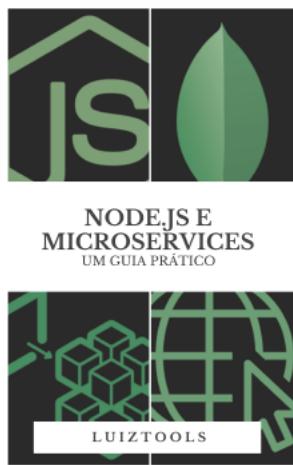
Programação  
Web com Node.js

[SAIBA MAIS...](#)



Programação  
Web com Node.js

[SAIBA MAIS...](#)



Node.js e  
Microservices

[SAIBA MAIS...](#)



MongoDB  
para Iniciantes

[SAIBA MAIS...](#)



Scrum e  
Métodos Ágeis

[SAIBA MAIS...](#)



Agile Coaching

[SAIBA MAIS...](#)



Criando apps  
para empresas  
com Android

[SAIBA MAIS...](#)



Java para  
iniciantes

[SAIBA MAIS...](#)

## Conheça todos os meus livros

Aproveita e segue nas redes sociais:



# ANTES DE COMEÇAR

---

“

Without requirements and design, programming is  
the art of adding bugs to an empty text file.

- Louis Srygley

”

Antes de começarmos, é bom você ler esta seção para evitar surpresas e até para saber se este livro é para você.

## Para quem é este livro

Primeiramente, este ebook não vai lhe ensinar programação, ele exige que você já saiba isso, ao menos em um nível básico para a web.

Parto do pressuposto que você é ou já foi um estudante de Técnico em informática, Ciência da Computação, Sistemas de Informação, Análise e Desenvolvimento de Sistemas ou algum curso semelhante. Usarei diversos termos técnicos ao longo do ebook que são comumente aprendidos nestes cursos e que não tenho o intuito de explicar aqui.

O foco deste livro é ensinar diversos aspectos da programação de micro serviços, partindo da sua definição, principais características e indo para algumas pautas mais práticas, mas sem foco no código, mas sim no entendimento da arquitetura.

Ao término deste ebook você estará apto a desenhar soluções envolvendo micro serviços na linguagem da sua escolha, mas para desenvolvê-las vai precisar de um material mais prático, como um dos livros e cursos que citei na seção anterior.

*Quer fazer um curso online de Node.js e MySQL com o autor deste livro?*

Acesse <https://www.luiztools.com.br/curso-fullstack>

---

# ARQUITETURA MICROSERVICES

“

Recognizing the need is the primary  
condition for design  
- *Charles Eames*

”

Acho que foi em 2015 que passei a ouvir falar de micro serviços (microservices) mas foi em só 2016 que passei a me interessar realmente pelo tema, uma vez que foi quando comecei a estudar Node.js para uso em meus projetos.

Micro serviços é uma maneira particular de desenvolver aplicações de maneira que cada módulo do software é um serviço standalone cujo deploy e escala acontecem de maneira independentes da “aplicação principal” (não confundir com SOA, como explicarei mais adiante). Enquanto na arquitetura tradicional de software, chamada monolítica, quebramos uma grande aplicação em bibliotecas, cujos objetos são utilizados in-process, em uma aplicação modular como proposta na arquitetura de microservices cada módulo recebe requisições, as processa e devolve ao seu requerente o resultado, geralmente via HTTP.

A ideia não é exatamente nova, é usada em ambientes Unix desde a década de 60, mas recentemente se tornou o epicentro de uma grande revolução na forma como as empresas estão desenvolvendo software ágil baseado em equipes enxutas responsáveis por componentes auto-suficientes.

Neste capítulo irei abordar os conceitos, detalhes, vantagens, desvantagens e principais dúvidas dessa arquitetura, bem como porque ela está sendo tão utilizada atualmente e como começar a organizar suas aplicações orientadas dessa maneira.

## A Motivação

Muitas são as buzzwords do mundo de desenvolvimento de software e esta me pareceu mais uma quando a ouvi pela primeira vez. Apesar do seu nome ser auto-explicativo, é interessante estudarmos os reais impactos que uma arquitetura composta por diferentes módulos conversando via um canal lightweight como HTTP pode causar na forma como programamos e nos resultados que obtemos com software.

Resumidamente, o estilo de arquitetura em microservices é uma abordagem de desenvolver uma única aplicação como uma suíte de pequenos serviços, cada um rodando o seu próprio processo e se comunicando através de protocolos leves, geralmente com APIs HTTP.

Estes serviços são construídos em torno de necessidades de negócio e são implantados de maneira independente geralmente através de deploy automatizado (pelo menos em um cenário ideal deveria ser assim). Existe um gerenciamento centralizado mínimo destes serviços e cada um deles pode ser escrito em uma linguagem diferente e utilizando persistências de dados diferentes também.

Para entender melhor a motivação por trás de microservices vale relembrar como os projetos são programados hoje, geralmente em três grandes partes: um client-side com a interface do usuário, uma base de dados com as informações do sistema, e uma camada server-side com a lógica da aplicação. A camada server-side lida com as requisições do usuário, executa a lógica de negócio, retornar e atualiza dados da base e disponibiliza informações prontas para o client-side exibir. Isto é um monólito ou aplicação monolítica, uma vez que gera uma única e grande aplicação com tudo junto. Qualquer mudança no sistema envolve em compilar tudo novamente e implantar uma nova versão do server-side inteiro no servidor.

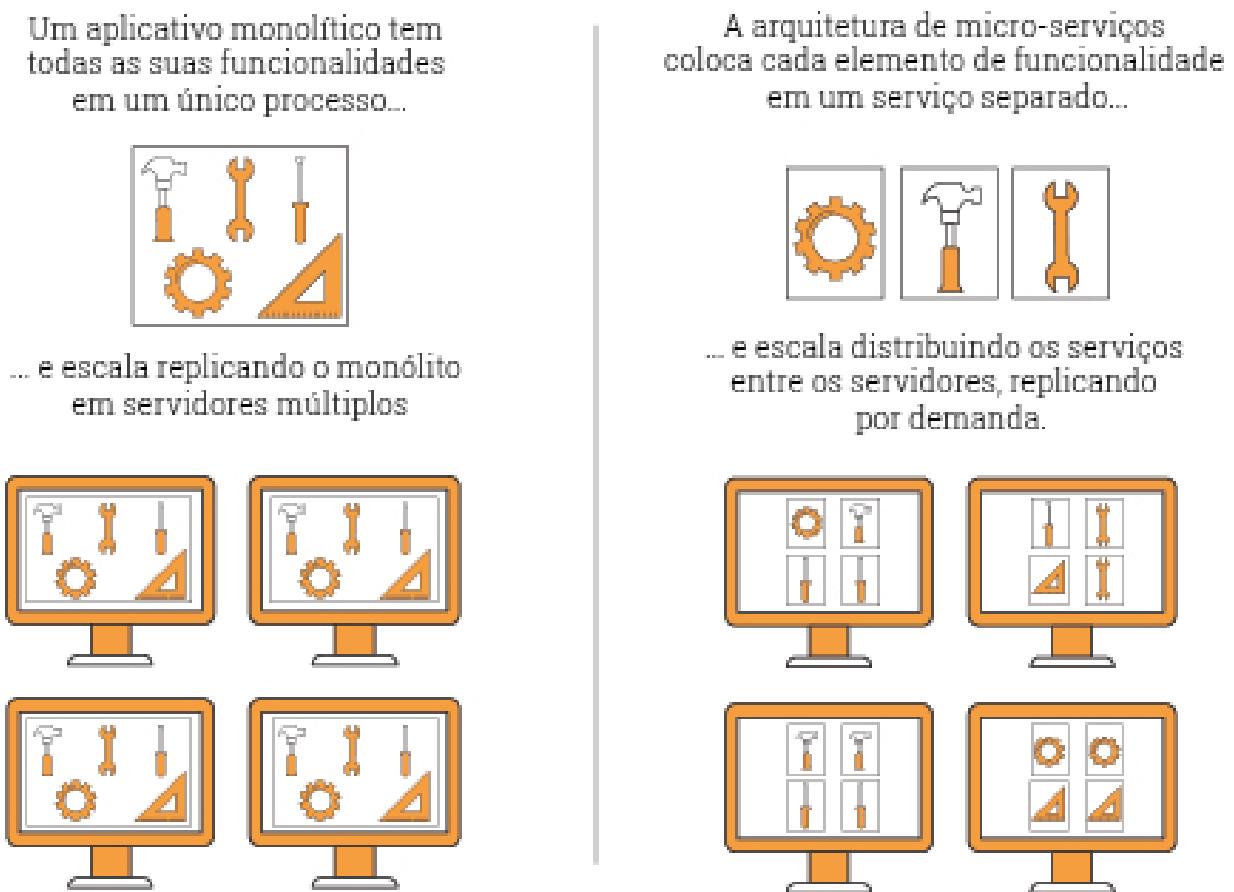
Toda a aplicação roda em um único processo, usando uma única linguagem server-side e geralmente uma única tecnologia de persistência de dados. Para escalar esse tipo de aplicação você pode adicionar mais recursos no mesmo servidor (escala vertical) ou fazer cópias desse servidor e colocá-las atrás de um load balancer (escala horizontal).

Aplicações monolíticas não são ruins e podem ser muitíssimo bem sucedidas. No entanto, cada vez mais equipes estão frustradas com suas limitações, principalmente quando estão implantando projetos na nuvem. Nestes cenários, os ciclos de mudanças, geralmente rápidos e pontuais como os propostos no Lean Startup, acabam afetando toda a aplicação pois o deploy é monolítico, assim como a aplicação. A escala horizontal requer um custo alto, pois sempre deve ser duplicada a aplicação inteira, e não apenas a parte que necessita de mais desempenho.

Vale lembrar também que aplicações monolíticas exigem que todo o codebase server-side seja escrito na mesma linguagem de programação, o que impede que você tire o máximo proveito de cada cenário usando a ferramenta mais apropriada à ele. Também impede que cada time desenvolva com a maior velocidade possível uma vez que a base de código é a mesma entre todos times.

Estas frustrações levam à arquitetura microservices de construir aplicações como um conjunto de serviços. Como os serviços são implantados de maneira independente, a escala se dá individualmente, tanto na vertical quanto na horizontal, para os serviços que estão precisando de mais desempenho. Usando protocolos comuns e contratos (interfaces) bem definidos, você consegue usar linguagens de programação diferentes de cada serviço, bem como mecanismos de persistência auxiliares que possam ser necessários para cada um deles, como mecanismos de cache, filas, índices, etc.

Para finalizar esta seção, a imagem abaixo do blog do Martin Fowler ilustra bem a diferença da escala entre as duas arquiteturas: monolítica e de micro serviços:



## Vantagens e Desvantagens

Não existe uma definição formal de como uma aplicação baseada em micro serviços deve ser construída, mas depois de muito ler a respeito e estudar o assunto notam-se algumas particularidades frequentes que podemos denominar como um padrão comum para microservices.

Sempre existiu o desejo dentro da indústria de software de construir programas apenas plugando componentes. Os primeiros esforços neste sentido foram as bibliotecas de funções, mais tarde as bibliotecas de classes e atualmente está em voga os serviços. Note que a componentização sempre existiu, mas o que propõe-se com microservices é que cada componente seja uma aplicação separada e especializada em apenas uma parte da aplicação “completa”. A principal razão por trás dessa escolha, de serviços ao invés de bibliotecas, é que serviços podem ser implantados de maneira separada. Você já tentou atualizar seu software apenas subindo para produção uma DLL ao invés de todas (ou ao menos todas as suas)? É um risco que não vale a pena correr, pois o acoplamento entre as DLLs é muito alto e o ideal é sempre a recompilação do sistema como um todo.

Claro, existe uma desvantagem clara no uso de serviços ao invés de bibliotecas: performance. Bibliotecas rodam no mesmo processo da aplicação, usam memória compartilhada, etc. Serviços dependem de canais de comunicação, como HTTP, para conseguirem tratar e responder as requisições. Exigem também uma coordenação entre os contratos de serviço para que os consumidores consigam “conversar” com os serviço da maneira que eles esperam, bem como receber as respostas que estão preparados.

Esse problema é especialmente preocupante conforme você tenha muitas chamadas síncronas entre seus serviços, pois o tempo de espera total para seu sistema responder será igual à soma de todos os tempos de espera das chamadas síncronas. Neste momento você tem duas opções: mudar para uma abordagem assíncrona ou reduzir o máximo que puder o tempo de espera (e a quantidade) das requisições síncronas. Node.js é uma tecnologia que tenho estudado bastante recentemente e trabalha muito forte com o conceito de chamada assíncrona, muito usado pelo Netflix para não bloquear a experiência do usuário. No entanto, quando isso não é possível, como foi o caso do site do jornal

The Guardian, tente limitar o número de chamadas síncronas que você vai precisar, o que no caso deles é a regra de apenas uma chamada síncrona por requisição do usuário.

Apesar desses problemas citados, os benefícios parecem superar os downsides dessa abordagem, pois cada vez mais empresas estão adotando-na.

## Tenha em mente

Os serviços irão falhar. Talvez não todos juntos, talvez não rapidamente, mas vai acontecer. Sendo assim, você deve estar sempre preparado para a falha de um ou mais serviços.

Essa é uma consequência de usar serviços como componentes e sua aplicação deve ser projetada de maneira que possa tolerar a falha de serviços. Qualquer chamado a um serviço pode falhar devido à indisponibilidade de um fornecedor e você tem de saber lidar com isso de maneira amigável com o restante do sistema. Esta talvez seja a maior desvantagem dessa arquitetura se comparada ao jeito monolítico tradicional, uma vez que adiciona uma complexidade adicional significativa. Times que trabalham com micro serviços devem constantemente refletir como a falha de cada serviço afetará a experiência do usuário. No Netflix por exemplo, partes da bateria de testes diária deles inclui derrubar servidores e até datacenters para ver como a aplicação se comporta nestas situações.

Uma aplicação em micro serviços deve ser monitorada em um nível muito superior ao de uma aplicação monolítica tradicional, uma vez que os serviços podem falhar a qualquer momento e se possível, restaurar o funcionamento completo automaticamente.

Monitoramento em tempo real deve ser enfático no processo de desenvolvimento, implantação e operação dos serviços. Não que você não tenha de ter esse mesmo cuidado com aplicações monolíticas, mas apenas que com micro serviços isso não é uma opção.

## O quão “micro”?

Uma pergunta bem comum e que cai como uma luva para o próprio nome da “arquitetura” é: o quão grande deve ser um micro serviço?

Infelizmente o nome micro serviço nos leva a perder tempo demais pensando no que exatamente micro quer dizer. Diversas empresas, de Amazon a Netflix, usam micro serviços de variados tamanhos e não há um consenso sobre eles. Especificamente na Amazon, considerando que cada micro serviço é (e deve) ser tratado como um produto separado e tem seu próprio time (squad, na verdade), eles usam a regra Two Pizza Team: se o time precisa de mais de duas pizzas por refeição para se manter alimentado é porque está grande demais, o que significa não mais de que 12 pessoas. No outro extremo, os menores times recomendados pela Amazon possuem 6 pessoas, o que eu pessoalmente, em minhas experiências como Scrum Master, acredite ser o ideal (você ainda pode chamar meus times de Two Pizza Team considerando que todo mundo gosta de comer vários pedaços de pizza!).

Resumindo, embora não exista uma regra, podemos assumir que se você precisa de uma equipe de mais de 12 pessoas para desenvolver e manter um serviço (considerando tudo: programação, infra, testes, etc), ele está grande demais e você deveria quebrá-lo em serviços menores.

## Como começar?

Projetos usando micro serviços geralmente não nascem assim do dia para a noite. O que nota-se na maioria das implementações bem sucedidas é que elas vieram de um design evolucionários, vendo a decomposição em serviços como uma ferramenta que vai aos poucos ajudando a quebrar uma aplicação monolítica que está com problemas de escala, qualidade, etc em diversos componentes menores.

Neste cenário, o maior desafio é saber como que o monólito irá ser quebrado em serviços. O quê deve ser agrupado em conjunto? O que deve estar separado? Quais bases de dados serão usadas por quais serviços?

Um dos princípios-chave por trás dessas decisões é o de deploy e escala independentes entre os serviços. Se dois serviços sempre tem de ser

colocados em produção em conjunto, eles deveriam ser um serviço só. Agora se eu tenho um serviço cujas publicações acontecem sempre por causa de alterações pequenas em um dos seus módulos internos, esse módulo interno deveria ser transformado em um serviço per se.

O site do jornal britânico The Guardian é um bom exemplo de aplicação que foi projetada e construída como um monólito e que vem evoluindo na direção de micro serviços. O monólito ainda é o core do site, mas eles têm adicionado novas funcionalidades usando micro serviços que consomem a API do site principal. Esta abordagem é particularmente interessante pra eles principalmente nos casos de módulos temporários, como um hotsite para cobrir um evento esportivo. Estes hotsites podem ser programados rapidamente na linguagem que for mais conveniente e jogados fora uma vez que não fazem mais sentido.

---

# ARQUITETURA ORIENTADA A SERVIÇOS

2

“

*Truth can only be found in one place: the code.*

*- Robert C. Martin*

”

SOA ou Arquitetura Orientada a Serviços, na tradução literal, é um padrão que já existe tem algum tempo, usado principalmente por grandes corporações. Mais recentemente o mercado voltou a falar de outra arquitetura igualmente distribuída: microservices. Também não há nada de exatamente inovador em microservices, que estão aí desde a década de 60, mas que recentemente voltou a ganhar força como padrão a ser adotado em diversos sistemas, principalmente aqueles que usam de persistência poliglota. Então porque eu venho escrever hoje sobre microservices vs SOA?

Durante meus estudos recentes sobre micro serviços me deparei com a seguinte pergunta: qual a diferença entre microservices e SOA? Não estaríamos todos falando mais do mesmo? Não vou entrar no mérito de CORBA, uma vez que este é um conceito mais consolidado e que acredito tenha caído em desuso na última década, principalmente a versão “Microsoftniana” DCOM/COM+.

Para entender melhor as diferenças, vamos relembrar alguns conceitos importantes e esclarecer algumas coisas.

## O que é SOA?

Uma arquitetura orientada à serviços não é algo exatamente difícil de entender quando você lê sua definição na Wikipedia: “funcionalidades implementadas pelas aplicações devem ser disponibilizadas através de serviços”. Esses serviços podem ser consumidos por outros serviços para que o todo forme um sistema complexo. É o clássico “dividir para conquistar”. No entanto, SOA não é tão simples assim, e foi nós desenvolvedores que o tornamos complicado demais.

Para algumas empresas, usar SOA é apenas expor software através de webservices. Dentro dessa concepção, há grupos mais conservadores que acreditam que isso inclui apenas os vários padrões WS-\* e outros mais liberais que aceitam qualquer tipo de dado sobre HTTP (não necessariamente apenas XML). O que você acha, isso é SOA?

Para outras empresas, SOA implica em uma arquitetura em que não há UMA aplicação única que consome alguns serviços, mas sim que TUDO são serviços que fornecem desde dados a regras de negócio e que são agregados através de uma UI que consome todos eles. O que você acha, isso é SOA?

Não obstante, outras empresas acreditam que SOA é um jeito de permitir que diferentes sistemas se comuniquem através uma estrutura padrão com outras aplicações, assim como CORBA, mas usando XML. Isso geralmente inclui um barramento central de comunicação e dados que todos serviços usam para se organizar e centralizar a informação. Não necessariamente sobre HTTP, inclusive. O que você acha, isso é SOA?

Muitas empresas e desenvolvedores dizem coisas diferentes a respeito do que é SOA. Dizem que serve para separar dados de processos, dizem que é bom para combinar dados e processos, dizem que deve usar web standards, dizem que é independente de web standards, que síncrono, que é assíncrono, que sincronicidade não importa... muitas coisas diferentes e conflitantes!

Isto não é um problema exclusivo de SOA, temos esse mesmo problema de definição entre as diferentes tribos que usam POO. Você deve conhecer as velhas discussões sobre classes apenas com atributos ou apenas com métodos, serem ou não orientadas à objetos, só para citar um exemplo.

## E os microservices?

Quando estamos construindo arquitetura de software focadas na comunicação entre diferentes serviços é comum acabarmos colocando muita responsabilidade e inteligência no mecanismo de comunicação propriamente dito. Um bom exemplo disso é o Enterprise Service Bus (ESB ou Barramento de Serviço Corporativo). Esse tipo de “produto” geralmente inclui recursos sofisticados como roteamento de mensagens, coreografia, transformação e aplicação de regras de negócio. E olha que ele deveria ser apenas um barramento de comunicação...

Mas e os microservices? O que eles propõem de diferente que valha a pena entender? Microservices focam em endpoints inteligentes (os serviços) e canais de comunicação burros (protocolos simples, como REST). Aplicações construídas a partir de microservices possuem o objetivo (e a regra) de serem tão desacopladas e coesas quanto for possível. A “coreografia” entre elas se dá através de simples protocolos

como REST ao invés de protocolos complicados como WS-Choreography ou BPEL ou ainda sendo orquestrado por uma ferramenta para isso.

Times trabalhando com microservices usam princípios e protocolos que a própria web usa, simplificando muito a curva de aprendizado e a arquitetura como um todo. Mesmo que se queira (ou se necessite) de um barramento de comunicação, ao invés de se adotar um ESB, usam-se barramentos leves de mensageria como RabbitMQ ou ZeroMQ, servindo apenas como filas confiáveis assíncronas para não se perder mensagens. Toda a inteligência fica no serviço, não no barramento.

O grande desafio dos times que querem trocar de padrão de arquitetura: monolítica para microservices é que o “normal” na indústria é que os componentes se comuniquem através de invocação de métodos ou chamadas de funções e é mudar esse padrão de comunicação que mora o grande desafio.

Não seriam os microservices a mesma coisa que SOA, mas algumas décadas atrás quando o padrão nasceu? Inclusive algumas empresas que dizem usar SOA programam-na da mesma forma que hoje chamamos de microservices e isso não é ruim. O grande problema de usar o termo SOA é o que mencionei no início desse tópico, onde SOA significa muitas coisas diferentes e até contraditórias para diferentes pessoas. Via de regra, geralmente o que se vê é SOA sendo usado para integrar diferentes aplicações monolíticas através de um ESB. Pronto, falei.

Sem julgamentos aqui, mas não dá pra comparar isso com microservices, principalmente considerando a complexidade do canal de comunicação. E não julgo principalmente porque quem advoga das melhores práticas de microservices são pessoas que trabalharam muitos anos em empresas com sistemas SOA e sabem do que estão falando, principalmente sobre a complexidade que isso se tornou.

Sendo assim, dizer que você usa microservices não é o mesmo que dizer que você usa SOA e vice-versa, para a maioria das pessoas. Sinceramente não me importo com o rótulo, e considerando a definição mais abstrata e geral de SOA, eu afirmaria que microservices é uma

forma de SOA. Outros mais extremistas diriam que microservices é SOA feita do jeito certo, mas não sou tão arrogante assim.

## Governança descentralizada

Tanto em microservices quanto em SOA temos algum nível de governança dos serviços. No entanto, em SOA, ela é mais centralizada.

Uma das consequências de governança centralizada é a tendência a se criar padrões em torno de uma única plataforma de tecnologia. A experiência mostra que esta abordagem é restritiva em demasia, uma vez que “nem todo problema é um prego e nem toda solução é um martelo”, como diria o ditado. Grandes empresas como ThoughtWorks acreditam que devemos sempre utilizar a ferramenta certa para cada trabalho, mas os grandes problemas são:

que as aplicações monolíticas não nos dão toda a liberdade que queremos para fazer isso; e que o SOA tem padrões demais a serem seguidos, o que acaba engessando os serviços; Ao quebrar os componentes de uma aplicação monolítica em serviços nós permitimos que cada um deles seja implementado em tecnologias diferentes e use persistências diferentes e talvez essa seja o maior atrativo de microservices. Claro, não é porque podemos fazer um sistema cheio de serviços poliglotas que você deve fazê-lo, mas você tem essa opção, quando necessário.

Times trabalhando com microservices preferem uma abordagem diferente para padrões: ao invés de definir um documento com padrões (como no caso do SOA), eles descobrem uma maneira de resolver um problema e o transformam em um componente ou serviço, que pode inclusive ser compartilhado com o mundo, como o rápido crescimento do repositório do NPM pode confirmar. Esta colaboração cria padrões emergentes como o uso de determinado ORM ou framework web, por exemplo. Com a transformação do Git em padrão de facto para controle de versão, práticas open-source tem se tornado mais e mais comuns nas empresas.

O Netflix é um bom exemplo de organização que segue esta filosofia (e que o faz com Node.js). Compartilhando componentes úteis e testados na “linha de frente” eles encorajam outros desenvolvedores a usarem os

mesmos padrões ao mesmo tempo em que deixam a porta aberta para outras possibilidades.

O mindset de microservices não aceita overheads, como por exemplo, os contratos de serviço exigidos no SOA. Padrões como Tolerant Reader e Consumer-Driven Contracts são comuns de serem aplicados a microservices e muitas ferramentas foram criadas para automatizar o processo de testes dos serviços para se certificar que tudo está funcionando mesmo sem contratos - uma abordagem elegante para resolver o dilema YAGNI.

Talvez o apogeu da governança descentralizada seja o éto popularizado pela Amazon de “você cria, você mantém” (tradução livre de “you build it/you run it”). Os times são responsáveis por todos os aspectos do software que eles constroem incluindo mantê-lo no ar 24/7. Este nível de responsabilidade é definitivamente a norma que mais e mais empresas estão aplicando sobre seus times como o Netflix tem feito também. Afinal, nada pode ser mais motivador para fazer você focar na qualidade do projeto do que ser acordado às 03h da manhã por causa de bugs em produção, certo?

## Gerenciamento descentralizado de dados

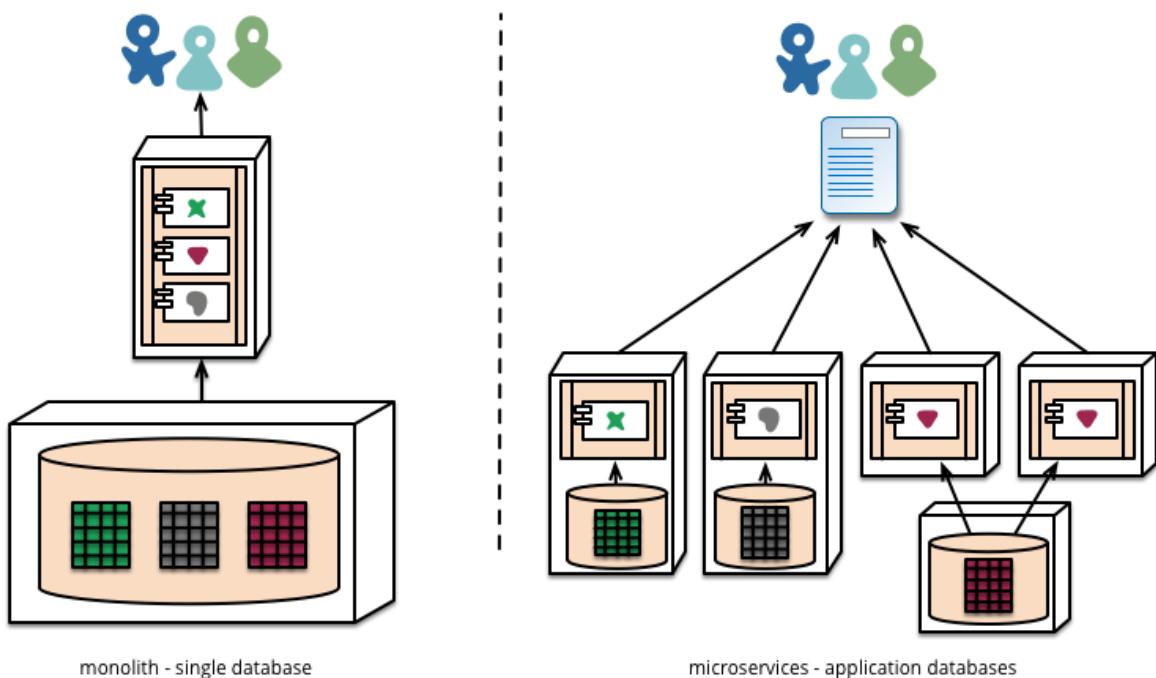
Novamente, é muito comum vermos implementações de SOA onde temos uma única base de dados centralizadora de tudo. Uma grande premissa é a de que todos os dados são da mesma empresa, logo são os mesmos em todos serviços. No entanto, isso não é 100% verdade.

A descentralização do gerenciamento de dados apresenta-se de diferentes maneiras e, em um nível mais abstrato, significa que o modelo conceitual de mundo é diferente entre os sistemas, logo não há porque juntá-los em um único banco. Quer um exemplo? Os dados de um cliente que um vendedor precisa saber são completamente diferentes do que alguém do suporte precisa enxergar, além disso, até mesmo o nome ‘cliente’ não faz sentido ser o mesmo em ambas visões do sistema.

Este problema é comum entre aplicações diferentes mas pode ocorrer mesmo dentro de uma única aplicação, principalmente quando ela é dividida em componentes separados. Um jeito útil de se pensar sobre

isso é a noção de Bounded Context do DDD, que eu não vou entrar em detalhes aqui.

Assim como pensar sobre os modelos conceituais de dados nos leva a pensar que eles deveriam ser separados, usar microservices reforça, e muito, esse mindset. Enquanto aplicações monolíticas preferem bases únicas e centralizadoras por padrão, microservices preferem deixar que cada serviço escolha a melhor maneira de persistir os seus dados, seja com a mesma base de dados, a mesma tecnologia mas bases diferentes ou tecnologias completamente diferentes. Claro, você pode usar persistência poliglota em aplicações monolíticas (Abstract Factory está aí para isso), mas ela é mais comum em projetos com microservices.



A descentralização da responsabilidade pelos dados através dos micro serviços tem seus reveses, como gerenciar atualizações. A abordagem mais comum é criar transações, assim como em aplicações monolíticas, embora isso gere uma complexidade significativamente.

Bom, o meu intuito nunca foi dizer como você deve programar suas aplicações, se com SOA ou com microservices, mas espero ter jogado alguma luz ao assunto e ajudá-lo a entender as diferenças.

---

# GESTÃO E SEGURANÇA

3

“

*It takes 20 years to build a reputation and few minutes  
of cyber-incident to ruin it.*

- Stephane Nappo

”

Após você começar a praticar e até a colocar micro serviços em produção você notará que esta arquitetura tem algumas complexidades que não existiam nos webservices monolíticos de antigamente.  
Perguntas como:

- » terei de trabalhar com múltiplos endpoints de múltiplos serviços para uma única aplicação?
- » será que o(s) client(s) deveriam conhecer as especificidades de cada microservice?
- » E no caso de algoritmos que são padrões a todos os microservices (como autenticação e autorização), devem ser repetidas em todos microservices?

Como resolver estas e outras questões sem gambiarra, de maneira profissional?

Usar um API Gateway e/ou um API Manager é um bom começo.

## API Gateway

Resumidamente um API Gateway fornece um ponto de acesso único à sua arquitetura de microservices. Não importa quantos microservices você tenha, colocando um API Gateway à frente deles você terá uma única URL para se preocupar. O API Gateway por sua vez roteia e gerencia o tráfego de requisições para os microservices de destino.

De maneira bem simplista, é isso que ele faz.

Já um API Manager é um API Gateway mais ‘bombado’, que além de atuar como proxy realiza toda uma governança dos serviços, das chamadas realizadas aos microservices, analytics, versionamento de APIs, caching, dashboards, segurança, transformação de dados, agregação de dados, etc.

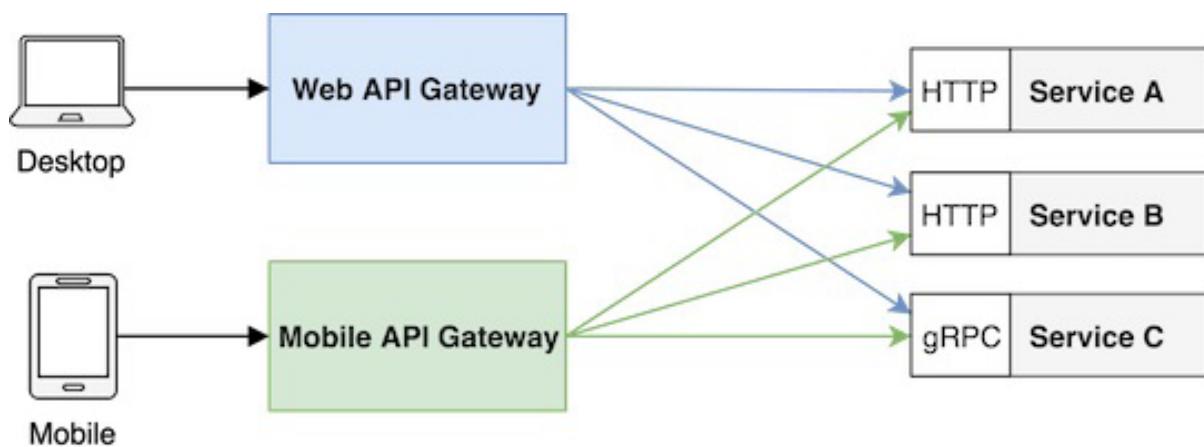
A diferença entre API Gateway e API Manager é tênue, uma vez que há diversos fabricantes e nomenclaturas diferentes no mercado. O termo mais comumente utilizado é de API Gateway, geralmente sendo chamados de API Managers as soluções comerciais corporativas.

Alguns gateways e/ou managers famosos:

- » Sensedia
- » Traeffik
- » WSO2
- » Kong
- » Akamai
- » Amazon API Gateway
- » Eureka + Zuul (Netflix OSS)

E embora em linhas gerais isso é tudo que se tenha para dizer sobre “o que é um API Gateway”, existem diversas abordagens de uso, dependendo do seu objetivo, necessidades e recursos.

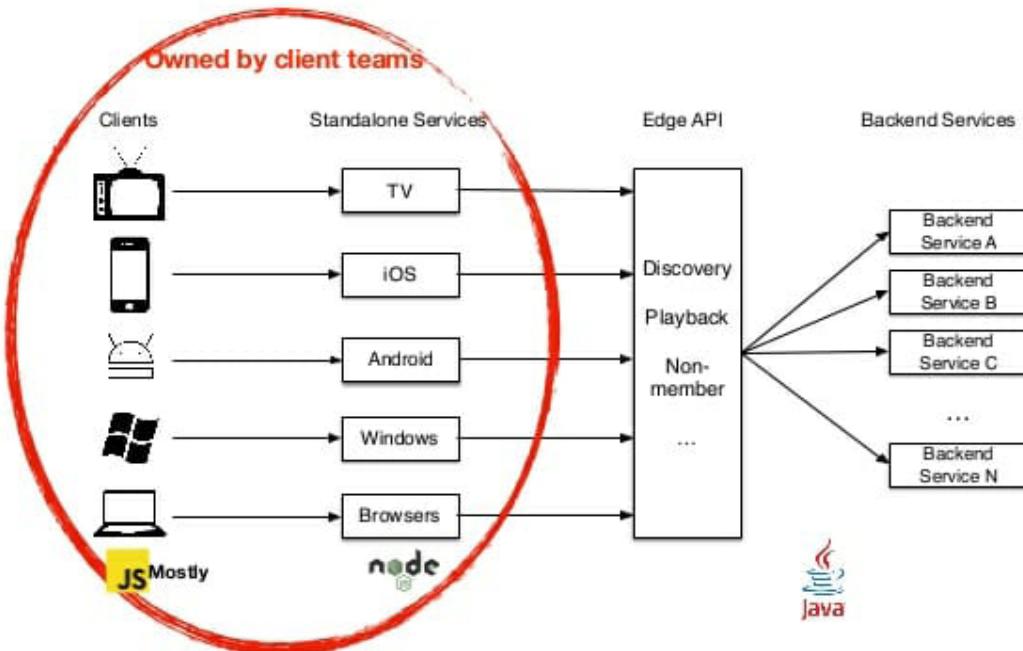
Uma abordagem possível é ter um API Gateway por diferente client, como na imagem abaixo:



A vantagem desta abordagem é realizar transformações diferentes para cada cliente, tratando erros de forma personalizada por exemplo, além de corrigir deficiências que alguns dos clients possua, como os dispositivos móveis, por exemplo, cuja conectividade é inferior aos desktops.

Outra solução bem comum no mercado é usar um proxy reverso como Apache ou Nginx para atuar como API Gateway, mas Node.js também é um excelente opção para construção de API Gateways quando se deseja um controle maior das requisições do que com webservers, permitindo fazer algo mais próximo de um API Manager sob medida.

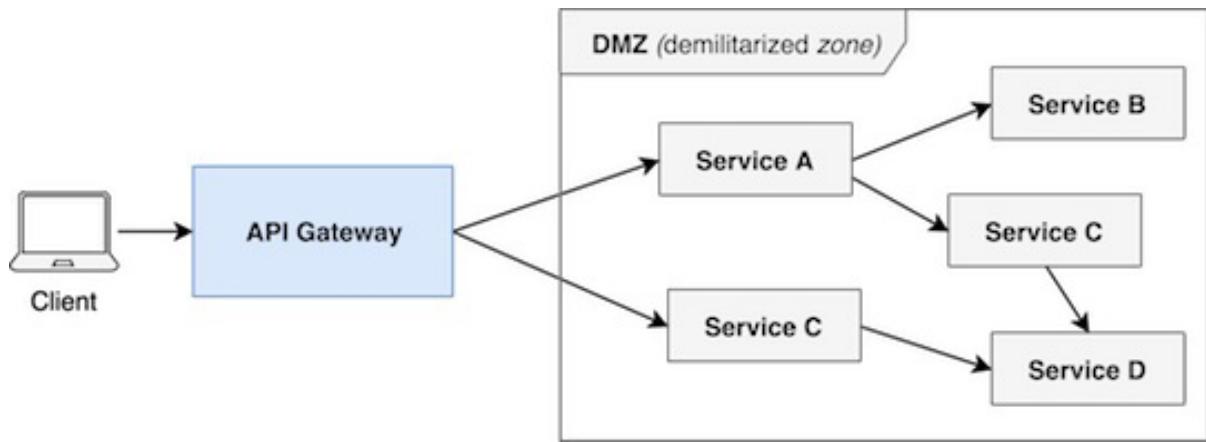
É exatamente o que fez a Netflix, segundo um artigo de engenharia deles, que construiu um API Gateway em Node.js para ficar na frente das suas centenas de APIs Java, como mostra o diagrama abaixo:



Especialmente para clientes mobile, que possuem severas limitações de latência e qualidade de conexão à Internet, um API Gateway pode simplificar em uma única chamada mobile, diversas requisições aos microservices que, acontecendo na rede local do datacenter, serão muito mais velozes do que se o dispositivo móvel tivesse de fazer diversas chamadas.

No entanto, abordagens desenvolvidas em Node.js devem cuidar para não onerar demais a plataforma com operações de computação intensiva, que são um “calcanhar de Aquiles” para o Node. Assim, nada de colocar mecanismos de compressão e/ou SSL diretamente no Node, hein!

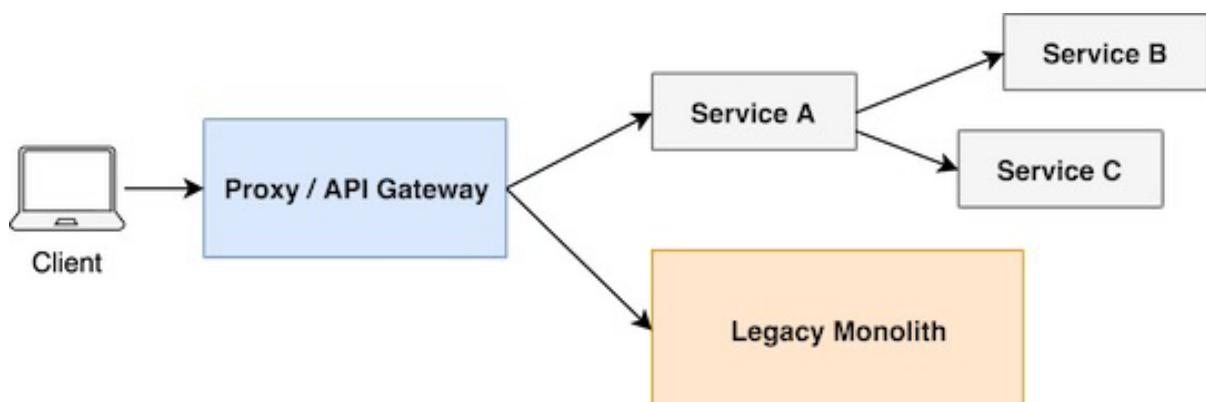
Outra arquitetura bem comum para uso de API Gateway é o desenho abaixo:



Nesta arquitetura, o API Gateway recebe e roteia as requisições para microservices dentro de uma DMZ (Zona Desmilitarizada ou Zona de Perímetro), um conceito de segurança da informação para proteger a rede da empresa dos serviços e vice-versa. Nela, os serviços não estão expostos na Internet, estão em uma rede virtual privada que somente permite conexões vindas do API Gateway.

Assim, nosso cliente somente conhecerá uma API, que é a API Gateway, podendo existir muitos outros microservices por trás dele. Essa abordagem é especialmente interessante quando estamos refatorando serviços monolíticos para microservices: podemos substituir as chamadas ao serviço antigo para chamadas ao API Gateway e rotearmos para o serviço antigo. Conforme o time for ‘escamando’ o serviço antigo em microservices, vamos ajustando no API Gateway para chamar ora o serviço antigo, ora os novos microservices.

O diagrama abaixo mostra isso:



Mais pra frente podemos aproveitar esta centralização das requisições para adicionar segurança à todas as chamadas como autenticação e rate limit (controle de uso das APIs, para evitar exageros).

## Segurança

Quando o assunto são web APIs, certamente você não irá querer permitir clientes anônimos, certo?

Identificar corretamente quem está fazendo as chamadas não apenas impede o acesso anônimo como restringe a autorização a determinadas ações e permite tomar medidas administrativas como controle de volumetria, estatísticas, revogação de acesso e muito mais.

Com tudo isso em mente, existem diversas formas de autenticação disponíveis para Web APIs, sendo API Keys uma delas, JWT outra e OAuth uma terceira igualmente popular.

### API KEYS

São uma forma fácil e simples de fornecer acesso seguro de um sistema a uma Web API. Elas identificam o projeto ou sistema que está fazendo a requisição e sua duração é geralmente até ser revogada. Ou seja, se eu vou criar uma API de previsão do tempo que vai ser usada por outros sistemas, eu posso controlar o acesso por uma API Key por sistema (independente do número de usuários em cada um). Não importa nesse caso qual usuário está chamando, mas qual sistema sim.

### JSON WEB TOKENS (JWT)

São uma forma de complexidade intermediária para fornecer acesso seguro de um usuário a uma Web API. Eles identificam o usuário ou cliente que está fazendo a requisição e sua duração é de poucos minutos, podendo ser atualizado (refresh). Ou seja, se eu vou criar uma API para o backend da aplicação que meus usuários/clientes vão usar (um mobile app ou SPA por exemplo), eu posso controlar o acesso via JWT, sabendo exatamente qual usuário está usando o backend.

## OAUTH

É um protocolo para delegar acesso seguro de um usuário em um sistema para outro sistema que permite várias configurações. Eles identificam que um terceiro pode fazer chamadas em seu nome para um sistema no qual você é cadastrado, como quando você usa login via Facebook, por exemplo. A duração é até o acesso ser revogado pelo detentor original das credenciais, é como se fosse uma procuração que o usuário assina autorizando ao sistema fazer chamadas em seu nome.

Em todos estes casos você pode ajustar a granularidade de permissões, duração, etc mas em linhas gerais, o resumo acima ilustra bem e deve ser útil para você tomar decisões futuras.

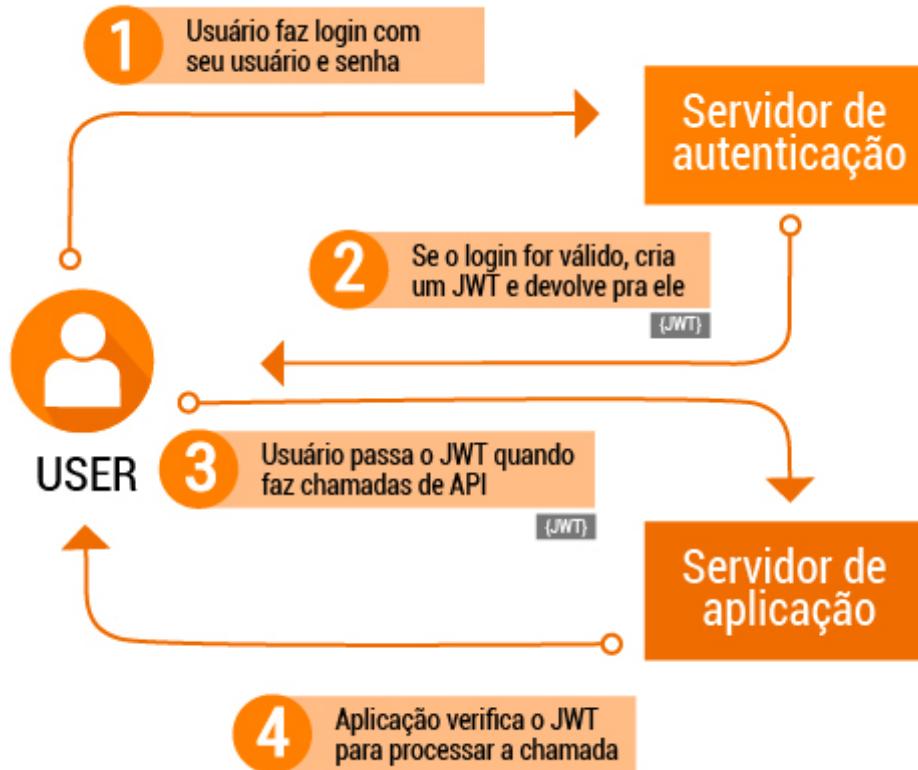
A seguir, vou explorar em mais detalhes o JWT, que é muito popular atualmente.

## JSON WEB TOKEN

JWT, resumidamente, é uma string de caracteres codificados que, caso cliente e servidor estejam sob HTTPS, permite que somente o servidor que conhece o ‘segredo’ possa ler o conteúdo do token e assim confirmar a autenticidade do cliente.

Ou seja, quando um usuário se autentica no sistema (com usuário e senha), o servidor gera um token com data de expiração pra ele. Durante as requisições seguintes do cliente, o JWT é enviado no cabeçalho da requisição e, caso esteja válido, a API irá permitir acesso aos recursos solicitados, sem a necessidade de se autenticar novamente.

O diagrama abaixo mostra este fluxo, passo-a-passo:



O conteúdo do JWT é um payload JSON que pode conter a informação que você desejar, que lhe permita mais tarde conceder autorização a determinados recursos para determinados usuários. Minimamente ele terá o ID do usuário autenticado, mas pode conter muito mais do que isso, no entanto, jamais coloque dados sensíveis dentro do token.

Dentre as dúvidas mais comuns acerca desta técnica está a segurança do token, afinal ele é o ponto mais exposto na técnica e sequestro de tokens é a forma mais comum de tentar burlar este mecanismo.

Claro que usando SSL (sugiro Lets Encrypt que é gratuito) esse risco de captura diminui consideravelmente uma vez que a conexão está criptografada e encorajo fortemente você a não aceitar requisições usando HTTP. No entanto, sabemos que para tudo existem brechas a serem exploradas principalmente por parte de pessoas maliciosas dentro da sua própria empresa muitas vezes...

## O RISCO DO JWT

Mas o token não é criptografado? Eu olhei ele e vi um monte de letras e números aleatórios...

Não, o token apenas é codificado em base64, uma representação textual de um conjunto de bytes, se você jogá-lo em qualquer decodificador online, verá as três partes que o compõem sendo que a única ilegível é a terceira onde temos a assinatura digital do servidor, atestando que aquele token foi gerado corretamente pelo seu servidor, o que impede que tokens fake se passem por tokens reais.

No entanto, para essa assinatura é necessário um segredo/secret. Esse segredo é usado tanto para assinar quanto para verificar a assinatura (e autenticidade) de um token. Se você tem apenas uma webapi que usa o JWT, isso é bem tranquilo. Agora, se você possui diferentes microservices e todos eles precisam de autenticação/autorização via JWT, então você tem um problema pois:

- » ou você faz com que o cliente se autentique em cada um dos microservices que vai usar;
- » ou você compartilha o secret entre todos eles.;

Uma solução possível para este problema é usar um API Gateway na frente de todos microserviços. Daí todos eles confiariam no gateway, sem ficar pedindo ou verificando tokens.

Antes de eu entrar em uma possível solução barata para mitigar esse risco, vale lembrar que é importante você ter uma maneira fácil e rápida de invalidar tokens em caso de fraude de chamadas, para que a área de segurança possa agir rapidamente nestas situações. Uma blacklist de tokens resolve, até porque eles possuem prazo de expiração, logo, essa lista não vai crescer pra sempre.

Mas voltando ao assunto central: como podemos adicionar mais segurança em nosso JWT?

## ENTENDENDO O JWT

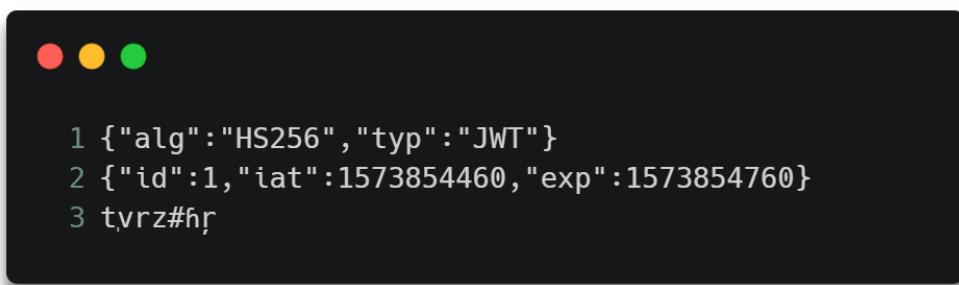
Se você executar qualquer API que gere um JWT como forma de autenticação e pegarmos o token, teremos algo como abaixo...

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.

eyJpZCI6MSwiaWF0IjoxNTczODU0NDYwLCJleHAiOjE1NzM4NTQ3NjB9.

k3SI3YTtHXb9vuaa6cjpcn-ojiNruoLjJpprwxZf3HU

Se jogarmos em um decodificador de base64 online teremos...



```
1 {"alg": "HS256", "typ": "JWT"}  
2 {"id": 1, "iat": 1573854460, "exp": 1573854760}  
3 tYrz#hj
```

A primeira parte é o header, onde temos o algoritmo utilizado na assinatura e o tipo de token. Na segunda parte, temos o payload, neste caso com a informação de um ID (do usuário provavelmente, isso é personalizável), um iat (“issued at”, timestamp de quando foi emitido) e um timestamp de expiração (exp), ou seja, tokens tem vida útil curta, geralmente de alguns minutos.

A terceira parte é a assinatura do servidor (um hash criado usando o header + payload + secret configurado no servidor).

A assinatura é o que realmente garante que este token não é forjado, mas para que ela possa ser verificada, as partes devem conhecer o mesmo secret usado na assinatura. Quando temos apenas uma API, isso não é um problema, mas se tivermos diversas APIs que desejam utilizar um único JWT como mecanismo de autenticação, teremos que espalhar o nosso secret por aí... E isso é um problema de segurança, afinal, se você tem uma senha que muita gente conhece, ela não é uma senha segura...

Mas como podemos garantir que este token possa ser usado em diferentes microservices sem compartilhar o secret com todos eles?

## ASSINATURA ASSIMÉTRICA DO JWT

Como estou falando de um token que navega entre diferentes microservices por meio inseguro (internet) o ideal é uma criptografia assimétrica, onde o emissor/servidor irá gerar o token assinado com a chave privada (de posse somente dele) e os consumidores/clientes podem verificar-lo usando a chave pública (de posse de todos microservices).

Assim, todos microservices confiam na emissão de tokens a partir um servidor central, enquanto podem validar sua assinatura para garantir que não foi forjado, sem saber o secret original.

Primeiro, vamos criar um par de chaves (uma pública e outra privada) usando o algoritmo RSA, um dos mais famosos e seguros do mundo de tipo assimétrico. O jeito mais fácil de gerar um par para fins de estudo é usando algum gerador online, enquanto que o jeito mais profissional é usando OpenSSL.

Mas quando for gerar, atenção a alguns requisitos: o format scheme exigido é PKCS #1 e o tamanho da chave varia de 256 bits a 2048 e embora chaves maiores sejam mais seguras (cada vez que você dobra o tamanho multiplica por 6x a dificuldade de quebra da chave) atente ao fato de que seu JWT deverá ser decifrado pelo servidor a cada requisição e que chaves maiores são mais demoradas para decifrar, mesmo com a chave certa.

Ainda assim, se quiser pecar pelo excesso, 2048 bits é o mais recomendado até 2030. Para fins de exemplo, segue uma chave pública que gerei e que deve ser salva em um arquivo public.key antes de ser utilizada:

```
1 -----BEGIN RSA PUBLIC KEY-----  
2 MIGJAoGBAPDA6tLiWR4mS/qj2jFygGFc1t2TYcbmbxU8JQKX4Ytz2TMKQBXd+pLK  
3 mNOzbsDrGE41gi5MET7qaSViADGyWGvn/Rc6FKPRnvMPbmySbqlmBaKR0iD/GYwL  
4 61b3pW9f90PhC08uoZJ2qcoypdnwKrb+j20FavgNz3mBG1ZTF4+TAgMBAE=
```

```
5 -----END RSA PUBLIC KEY-----
```

Enquanto que minha privada ficou assim (deve salvar em um arquivo private.key):



```
1 -----BEGIN RSA PRIVATE KEY-----
2 MIICXAIBAAKBgQDww0rS4lkeJkv6o9oxcoBhXNbdk2HG5m8VPCUJF+GLc9kzCkAV
3 3fqSypjTs27A6xhONYIuTBE+6mklYgAxslhLZ/0XOhSj0Z7zD25skm6pZgWikdIg
4 /xmMC+tW96VvX/dD4QtPLqMydqnKMqXZ8Cq2/o9tBWr4Dc95gRpWUxePkwIDAQAB
5 AoGBAM5fcGuJH39asLKPfB6G+Nv5nkJmKLUR1QzyZ2VB5oFf1Mgha8i3idNlfy
6 bbPtUo5oC8mH5xiZc7xZv3TaPi41McRpLfU0gbH2ngZpYwIUg6i6lqBC4Um7th2H
7 U0kg0r5yeD2dxE2UYLkvNHVSk0R5DsIoMregtaQ/1btbpk2BAkEA/j+Yh4+hQeao
8 6ymYzuPN4AoVmBbriilBjdxCChVEbiEJ0udE5oWRDxIRzo3nqUBKABELh+cnPDdM
9 rER656QhYQJBAPJphXbzE82CNKZS957cLYySuefl+fatSGtESSaCFbRBP4JI+CB2
10 KnAfM0Fxt0iWBG5Cwud/Dx1hNsIAcEeZMXMCQA+aa4v2PplCxJ8aAGzCAkJ7m/On
11 hHEIMyO3nr3rrDVuBaJR2yKik9Ju83TPtKXociIcq6aA:iWLiqevwj/JjWkECQEFn
12 0v025DbkVHQzml+gQDEnPUVWBTUK16kpSct50YXivNHYAiov9488u8WCej7uis98
13 770pyNgzbGesHR8UNAUCQBj4PAEpDWlGe0CXLncAaDE84r6tZKnguFaDc7h8T7Fg
14 ZbL2d55m06jwEPDlz4hcXGuroBDipTC4kuIgShWcPMA=
15 -----END RSA PRIVATE KEY-----
```

Atenção ao fato de que ambas chaves começam e terminam com comentários. Isso é parte da chave, não remova. Além disso, elas quebram linha em pontos estratégicos, não altere isto também.

Outro ponto que vale a pena salientar, quando estiver usando chaves assimétricas é que chaves muito pequenas não consegue cifrar payloads muito grandes, geralmente levando a um erro “digest too big for RSA key” ou semelhante, problema que você não deve ter com chaves a partir de 1024 bits.

Exceto se seu payload for realmente muito grande, mas aí você deveria repensá-lo.

No seu algoritmo de geração do JWT, você deverá usar a chave privada, que somente deve ser de conhecimento do servidor de autenticação. Agora, para verificar a autenticidade de uma chave, a chave pública deve ser utilizada e pode ser distribuída livremente pelos seus microservices, pois ela apenas permite fazer verificação de autenticidade mesmo.

Note que esta abordagem consome mais recursos computacionais que os tokens comuns. Esteja preparado para um aumento custo de hardware e/ou do tempo entre cada requisição que necessita deste token.

---

# AVANÇANDO COM MICROSERVICES



*One of the great beauties of architecture is that each time,  
it is like life starting all over again.*

*- Renzo Piano*

A esta altura do ebook, de duas uma: ou você está se sentindo muito mais entendido no assunto microservices ou está apavorado.

Confesso que eu tenho um misto de sentimentos a respeito deste assunto pois, quanto mais eu estudo, mais aparecem coisas para eu estudar.

Mas tecnologia é assim, não é mesmo?

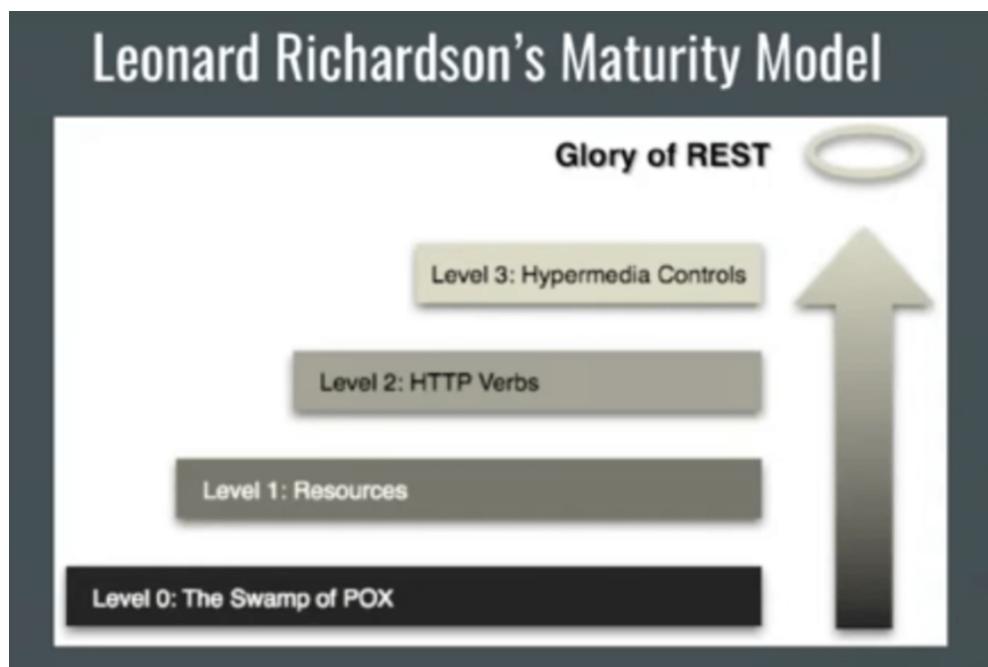
A ideia deste capítulo final é trazer conteúdos e dicas adicionais sobre a arquitetura microservices, aprendidas e extraídas de diversas fontes, tão variadas quanto mescladas, tornando difícil saber se alguma ideia aqui é original minha ou o digest de tudo que já li e ouvi a respeito.

## Boas práticas com micro serviços

A ideia deste artigo é servir como um guia de boas práticas para construção de uma arquitetura de microservices realmente profissional, seja lá qual tecnologia estiver utilizando. Apesar disso, serão dadas dicas adicionais relacionadas especificamente à plataforma Node.js que lhe serão muito úteis caso use essa tecnologia.

## TENTE ALCANÇAR A GLÓRIA DO REST

A imagem abaixo ilustra o modelo de maturidade do Leonard Richardson, um famoso desenvolvedor de APIs que programa desde os 8 anos de idade e tem vários livros publicados.



Neste modelo o objetivo é ir do Pântano do POX (Plain-Old XML) até a Glória do REST que basicamente é ter uma web API RESTful. Quanto mais RESTful for sua web API, mais próxima da Glória ela estará e consequentemente seus microservices estarão melhor desenvolvidos.

Martin Fowler resume este modelo da seguinte forma:

- » **Nível 0 (POX):** aqui usamos HTTP para fazer a comunicação e só. Não aproveitamos nada da natureza da web exceto seu túnel de comunicação, sendo que o envio e recebimento de mensagens baseia-se em XML-RPC ou SOAP (ambos POX).
- » **Nível 1 (Recursos):** aqui evoluímos a API através da segregação dos recursos em diferentes endpoints, ao contrário do nível 0 onde tínhamos apenas um endpoint com tudo dentro, respondendo de maneira diferente conforme os parâmetros no XML de envio.
- » **Nível 2 (Verbos):** aqui finalmente passamos a usar o protocolo HTTP como ele foi projetado, fazendo com que o verbo utilizado na requisição defina o comportamento esperado do recurso. Geralmente 99% das web APIs do mercado param por aqui.
- » **Nível 3 (Hipermídia):** este é o último obstáculo a ser alcançado para alcançar a Glória do REST e envolve a sigla HATEOAS que significa Hypertext As The Engine Of Application State. Aqui, o retorno de uma requisição HTTP traz, além do recurso, controles hipermídia relacionados ao mesmo, permitindo que o requisitante saiba o que fazer em seguida, ajudando principalmente os desenvolvedores clientes.

Embora este modelo não seja uma verdade universal, é um excelente ponto de partida para deixar seus microservices RESTful..

## USE UM SERVIÇO DE CONFIGURAÇÃO

Aqui a dica é usar o Consul ou serviço similar, tanto no Java quanto no Node, para lidar com dezenas de microservices, cada um com suas configurações personalizadas e com as inúmeras variações para cada ambiente (dev, homolog, prod, etc).

O Consul não serve apenas como um repositório de chave-valor para gerenciar configurações de maneira distribuída e real-time (sim, você nunca mais vai ter de fazer deploy apenas pra mudar um config), mas também para segmentação de serviços (garantindo comunicação segura

entre os serviços) e service discovery, um dos maiores desafios de arquiteturas distribuídas em containers.

## GERAÇÃO AUTOMÁTICA DE CÓDIGO CLIENTE

O código cliente para consumir microservices é sempre algo pouco criativo que faz sempre as mesmas coisas usando as mesmas bibliotecas líderes de mercado, certo?

Sendo assim, a recomendação é usar algum gerador de código client-side como o do Swagger, que inclusive suporta múltiplas linguagens de destino.

Se você usa o API Gateway da AWS, por exemplo, ele permite que você importe a sua especificação do Swagger para construir o código cliente (ou intermediário, dependendo do seu ponto de vista).

## CONTINUOUS DELIVERY

Quando você constrói uma arquitetura com dezenas de micro serviços e quer agilidade no deploy só há uma alternativa recomendada hoje em dia: containerização de serviços.

Usar Docker com Jenkins é a dobradinha mais usada atualmente para ganhar eficiência no pipeline de deploy. Realmente não há como fugir de ao menos conhecer essa dupla e não tem como falar de microservices sem falar de CD.

Pretendo fazer tutoriais no futuro de uso de Jenkins e Docker, uma vez que cada mais estas skills de DevOps tem sido necessários para se manter competitivo no mercado de trabalho hoje em dia, pelo menos em posições de liderança técnica. Sendo assim, aguarde um post futuro sobre Continuous Delivery com Jenkins e Docker.

## MONITORAMENTO E LOGGING

Quando você tem apenas uma web API para cuidar, é muito simples usar os logs do seu servidor web para saber o que aconteceu com sua aplicação e monitoramentos simples como o PerfMon do Windows

ou até um top tosco no console Unix. Monitoramentos web como StatusCake e Pingdom também são bem populares, pingando em endpoints de health-check.

Mas e quando você tem uma ou mais dezenas de microservices? Como gerenciar o monitoramento e o logging de cada um deles, principalmente considerando ambientes dinâmicas como nuvens containerizadas com auto scaling?

A questão é: independente do ‘como’, você terá de dar um jeito de ter uma solução profissional de monitoramento e logging, isso é vital para garantir a disponibilidade da sua arquitetura e responder rapidamente a incidentes. Uma dica importante é ter um bom APM, como o Zipkin, que é uma boa opção open-source.

## API GATEWAY

Já falei muito disso anteriormente neste ebook. Soluções mais profissionais e corporativas como WSO2 e Sensedia são mais indicadas nos casos de arquiteturas muito robustas, embora o Netflix tenha criado a sua própria solução em Node.js.

A questão é que você tem de ter uma camada antes dos seus microservices para acesso externo. Não apenas por uma questão de segurança mas para uma questão de agregação dos dados dos diferentes microservices que compõem a resposta de uma requisição, garantindo maior performance e usabilidade no lado do cliente.

## REFATORAÇÃO DE MONOLITOS

O processo de transformar uma aplicação monolítica em micro serviços é uma forma de modernização de aplicação, algo que os desenvolvedores já fazem há décadas. Como resultado, existem algumas ideias que você pode reutilizar quando estiver refatorando uma aplicação em microservices.

Uma estratégia é não fazer uma reescrita Big Bang, ou seja, esqueça a ideia de reescrever toda aplicação do zero, direcionando todos os esforços do time para isso. Embora muitas vezes isso soe interessante, é algo extremamente arriscado e geralmente resulta em problemas. Como

Martin Fowler já disse algumas vezes, em uma tradução livre: “a única garantia de um Big Bang é que algo vai explodir!”.

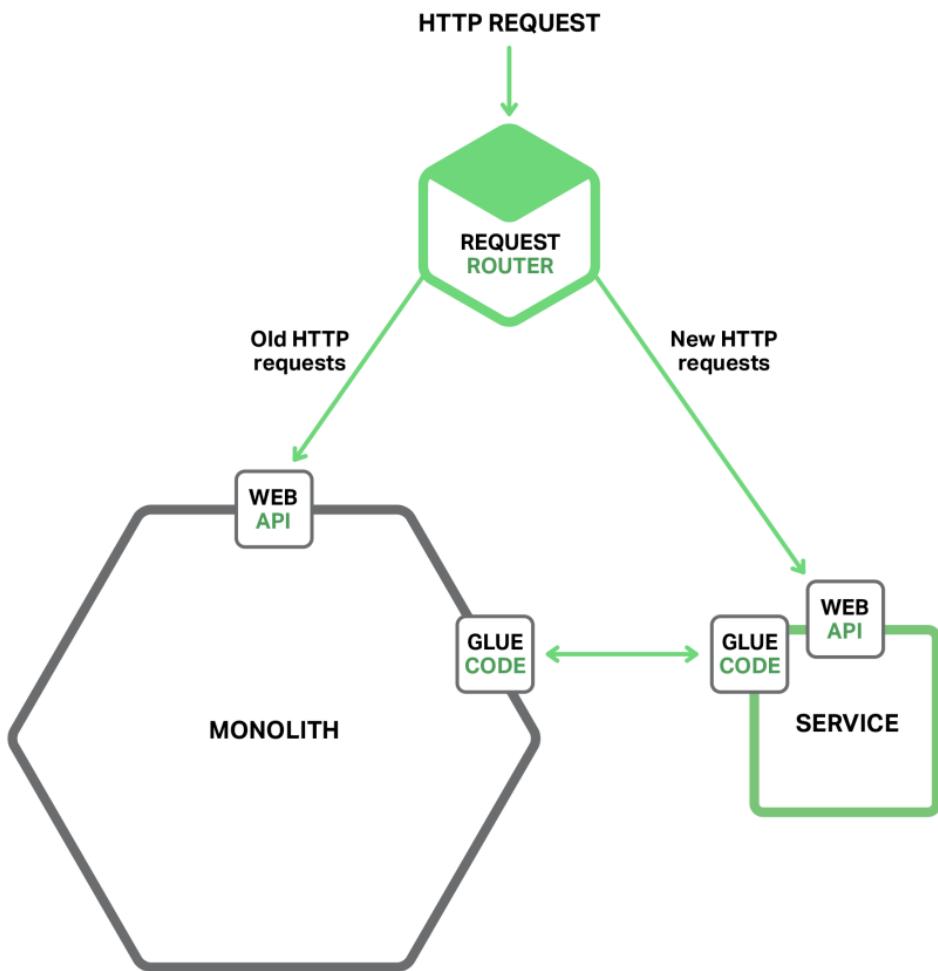
Ao invés de fazer uma reescrita Big Bang, você deve incrementalmente refatorar sua aplicação monolítica. Construa uma aplicação de microservices gradualmente, rodando cada novo microservice em conjunto do monolito original. Com o passar do tempo, a quantidade de funcionalidades implementadas em microservices versus a quantidade de funcionalidades remanescentes no monolito vai evidenciar que o mesmo encolheu ao ponto de que em algum momento ele irá sumir.

Martin Fowler refere-se à essa estratégia de modernização de Strangler Application (Aplicação Estranguladora). O nome vem da vinha estranguladora, uma espécie de cipó que cresce em árvores tropicais. Esta vinha vai se fixando na árvore original, consumindo seus recursos enquanto ela própria cresce mais rápido que sua hospedeira, não raro causando a morte da árvore original e deixando no lugar um monte de vinhas no formato de uma árvore.

## ESTRATÉGIA #1 - PARE DE CAVAR

A Lei dos Buracos (Law of Holes) diz que toda vez que você estiver em um buraco, você deve parar de cavar. Este é um ótimo conselho quando sua aplicação monolítica se tornar ingerenciável. Em outras palavras, você deve parar de tornar seu monolito maior. Se tiver de implementar uma nova funcionalidade você não deve adicionar este código ao monolito. Ao invés disso, a grande ideia com esta estratégia é colocar o novo código em um microservice próprio pra isso.

O diagrama a seguir, da Nginx, mostra a arquitetura do sistema depois de aplicar esta abordagem.



Repare no uso de um request router à frente de ambas soluções (monolito e microservice) que pode facilmente ser um API Gateway. Esta camada de software recebe as requisições originais e verifica se elas devem ser direcionadas para o legado ou para o microservice.

O outro componente que surge com esta arquitetura híbrida foi chamado de Glue Code no diagrama e nada mais é do que as dependências responsáveis por acesso a dados. Geralmente o serviço irá usar libs e componentes de acesso a dados do monolito original. Com isso em mente, esse glue code pode usar uma de três estratégias:

- » Invocar uma API remota fornecida pelo monolito;
- » Acessar a base de dados do monolito diretamente;
- » Manter sua própria cópia da parte da base de dados que lhe interessa, a qual deve estar sincronizar com a base do monolito;

Este glue code muitas vezes é chamado de camada anti-corrupção. Isto porque este glue code evita que o micro serviço seja poluído por conceitos do modelo de domínio legado. O termo camada de anti-corrupção foi introduzido pela primeira vez no livro Domain Driven Design do Eric Evans e depois foi refinando em um white paper. Desenvolver uma camada anti-corrupção não é algo exatamente trivial, mas é essencial se quiser sair do inferno do monolito.

Implementar novas funcionalidades como um micro serviço tem uma série de benefícios. Essa estratégia impede que seu monolito se torne ainda mais ingerenciável, ao mesmo tempo que permite que o serviço possa ser desenvolvido, implantado e escalado de maneira independente do monolito. A cada serviço novo que você cria, você experimenta um pouco mais dos benefícios desta arquitetura.

Entretanto, esta abordagem não endereça os problemas do monolito. Para consertar estes problemas você precisa quebrar o monolito, o que falaremos a seguir.

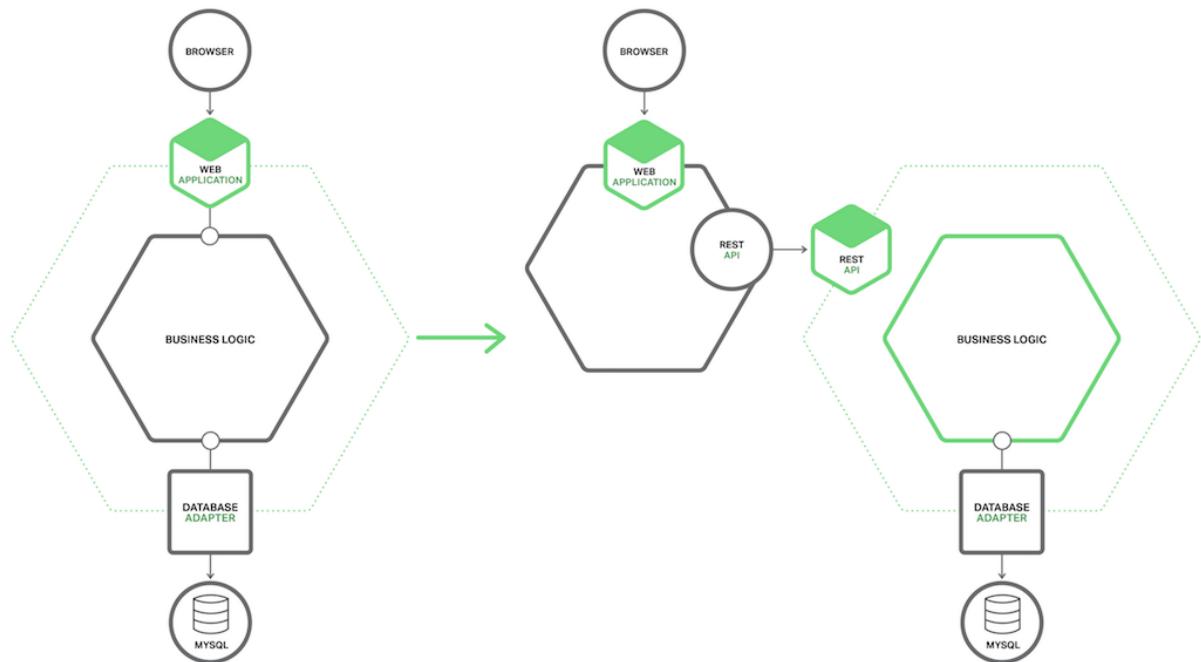
## ESTRATÉGIA #2 - SEpare FRONT-END DE BACK-END

Uma estratégia que ajuda a encolher a aplicação monolítica é separar a sua camada de apresentação da sua lógica de negócio e do acesso a dados também. Uma típica aplicação corporativa consiste de ao menos três diferentes tipos de componentes:

- » Presentation layer – Componentes que lidam com requisições HTTP e implementam APIs ou UIs. Em uma aplicação que tenha uma interface de usuário muito sofisticada terá uma quantidade de código de front-end substancialmente grande.
- » Business logic layer – Componentes que são o core da aplicação e implementam as regras de negócio.
- » Data-access layer – Componentes que acessam outros componentes de infraestrutura como bancos de dados e message brokers.

Geralmente existe uma separação muito clara entre a lógica de apresentação em um lado e as regras de negócio e de dados em outro. Geralmente a sua camada de negócio vai ter uma API consistindo de uma ou mais fachadas/interfaces que por sua vez encapsulam os

componentes de lógica de negócios. Esta API é o que permite fazer a separação do seu monolito em duas aplicações menores. Uma aplicação vai ser o front-end, a outra o back-end. Uma vez separados, o front-end irá fazer chamadas remotas ao back-end, sendo que o diagrama abaixo mostra como isso fica, antes e depois:



Separar um monolito dessa maneira tem dois benefícios principais. Ele permite que você desenvolva, implante e escale duas aplicações de maneira independente. Principalmente quando se fala de front-end e rápidas iterações, testes A/B, etc. Outro benefício é que você combina com a estratégia #1 ao fornecer uma API do monolito para ser consumida pelos novos micro services que você desenvolver.

Esta estratégia, no entanto, é somente uma solução parcial. É bem comum que você troque um problemão gigante por dois problemas menores, mas que ainda são um problema a ser resolvido. Você terá de usar uma terceira estratégia para eliminar os monolitos restantes.

## **ESTRATÉGIA #3 - EXTRAIR SERVIÇOS**

A terceira técnica de refatoração é transformar os módulos existentes dentro de um monolito em micro services standalone. Cada vez que você extrai um módulo e o transforma em um serviço, o monolito encolhe. Uma vez que você tenha convertido muitos módulos, o monolito irá parar de ser um problema. Ou ele irá sumir ou vai virar apenas um serviço por si só.

### **Priorizando quais módulos vão virar serviços**

Uma aplicação monolítica grande geralmente terá dezenas ou centenas de módulos, todos candidatos para extração. Saber quais módulos converter primeiro pode ser desafiador. Uma boa abordagem é começar com alguns módulos que são fáceis de extrair. Isto vai lhe dar experiência com microservices em geral e com o processo de extração em particular. Depois que você extrair estes módulos mais fáceis, poderá partir para os mais importantes.

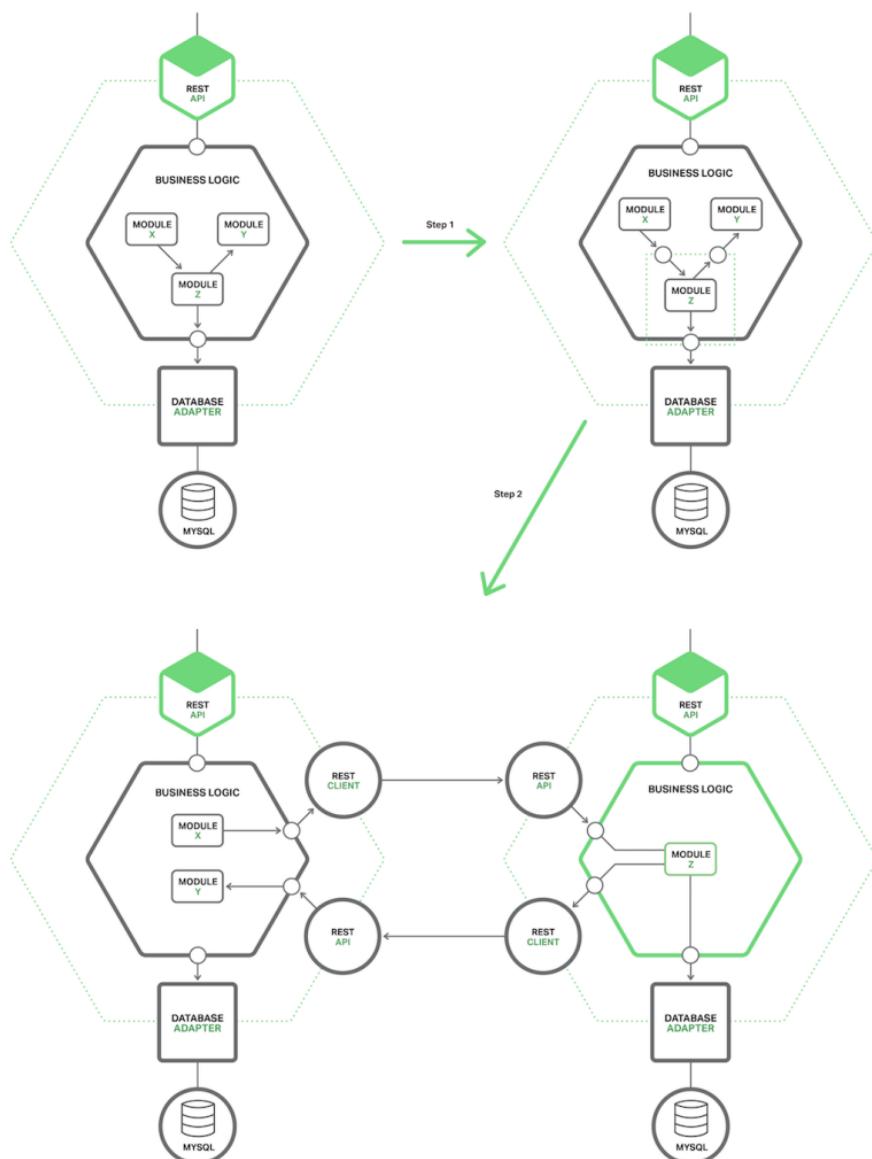
Converter um módulo em um microservice é algo que tipicamente requer tempo. Uma técnica de priorização é priorizar aqueles que mudam com frequência. Uma vez que você tenha convertido um módulo desses para um serviço, você pode desenvolver e implantar ele de maneira independente do monolito, o que vai acelerar o seu desenvolvimento.

Outra abordagem interessante é extrair os módulos que possuem requisitos significantemente diferentes do resto do monolito. Por exemplo, é útil pegar aquele módulo que usa uma base de dados in-memory e transformá-lo em um microservice, o qual poderá ser implantado em um servidor com muita memória. Da mesma forma, aquele módulo que consome muita CPU pode ser transformado em um microservice e feito deploy em um servidor com muito mais CPU do que RAM. Conforme você vai extrair estes módulos específicos, você vai ver como se tornará mais fácil escalar os mesmos.

## Como extrair um módulo

O primeiro passo ao extrair um módulo é definir a interface entre o módulo e o monolito. Geralmente será uma API bidirecional pois é comum o monolito precisar de dados do microservice e vice-versa. Se a sua lógica de negócio do monolito estiver com muitas associações entre suas classes talvez seja difícil de expor apenas o que importa para o microservice utilizar e não é raro que uma refatoração interna no monolito seja necessária para continuar avançando com essa estratégia de extração.

O diagrama abaixo mostra o passo a passo da extração de um módulo de um monolito para um microservice:



Neste exemplo, o módulo Z é o candidato a ser extraído. Seus componentes são usados pelos módulos X e Y. O primeiro passo de refatoração é definir um par de APIs de alto nível. A primeira interface é de entrada e é usada pelo módulo X para invocar o Z. A segunda é uma interface de saída usada pelo módulo Z para invocar o Y.

O segundo passo da refatoração transforma o módulo em um serviço separado. Uma vez que você tenha extraído um módulo, é mais um serviço que você tem que lhe permite desenvolver, implantar e escalar mais facilmente do restante do monolito. Você poderá inclusive reescrever este serviço do zero mais tarde, uma vez que o módulo já foi isolado.

Cada vez que você extrair um novo serviço, estará um passo mais perto de ter uma aplicação 100% em microservices. Com o passar do tempo, seu monolito vai encolher e a virada de chave entre as arquiteturas vai se tornar algo natural e inevitável.

## Microservices e Agile: o futuro da programação?

Nenhum destes termos é exatamente novo, mas por algum motivo, jamais estiveram tão na moda como atualmente. Mas o que teriam, a arquitetura microservices e os métodos ágeis de desenvolvimento de software em comum? Ou melhor: por que seriam eles o futuro da programação? Não tenho a pretensão de ser eu a dizer como as empresas de tecnologia deveriam trabalhar, mas vou tentar trazer uma luz para esse assunto e provocar uma reflexão (espero) na forma como você trabalha.

Os métodos ágeis existem desde a virada do século, mas, mais recentemente, nos últimos cinco anos, uma nova onda de “agilistas” passou a defender um modelo que seria uma evolução dos métodos ágeis originais. Essa evolução foi proposta pelo Spotify em um famoso vídeo onde é apresentado o modelo de “squads” (esquadrões) do Spotify, que, em teoria, não possui grandes diferenças em relação ao Scrum Team original mas que propõe um mindset focado em um único produto à cada um dos times/squads ao invés do foco tradicional em projetos.

O termo viralizou entre as startups e hoje empresas como Nubank, Umbler, Agibank e Conta Azul, por exemplo, organizam-se em squads

também. Organizar um squad ao redor de cada produto faz com que as pessoas “abraçem” o mesmo como se fosse “seu” e que queiram o melhor para ele em termos de tecnologia, experiência, resultados, etc. Permite que os times tenham mais controle (e mais resultados) com seus OKRs, KPIs, etc. Como bem o Scrum prega, cada time decide como seu produto deve ser melhor desenvolvido com base na estratégia da empresa e isso inclui TODOS os aspectos técnicos, como linguagem utilizada, banco de dados, etc.

Tudo isso é muito lindo e é exatamente o que as empresas querem: esquadrões focados no produto que entreguem software rapidamente (já ouviu falar de CI?) com as melhores tecnologias para resolver cada problema. Mas sabemos que na prática não é tão simples assim, certo? Mas deveria, e já, já, eu explico o porquê.

Estamos há décadas desenvolvendo software em grandes e complexos blocos de software que a própria TI chama de monolíticos, em alusão às grandes pedras (monólitos). Não importa se você organiza sua aplicação em ‘x’ camadas. Se você não pode fazer o deploy de apenas uma delas sem precisar compilar o projeto inteiro, o seu software é monolítico. Se você não pode escrever uma de suas camadas em uma linguagem diferente das demais, também. E isso não é ruim, aprendemos a construir softwares assim desde a faculdade e construímos grandes softwares ao longo das décadas que a profissão de programador existe. É apenas uma característica, sem julgamentos.

No entanto, cada vez mais o mercado nos pede softwares maiores e mais complexos, e nossos monólitos ficam ainda maiores e mais complexos. Mas ao invés de ficarem igualmente resistentes como a analogia, estão mais para um castelo de cartas altíssimo, infelizmente. Isso porque as demandas atuais estão exigindo pluralidade de tecnologias. E as aplicações monolíticas não trabalham muito bem com pluralidade, não foram concebidas desta forma. Fato.

Mas então, como podemos alinhar diferentes tecnologias, diferentes produtos, mantendo uma alta velocidade de integração, qualidade e atendendo às expectativas, internas e externas, em relação aos projetos em que trabalhamos?

Com uma outra tendência que voltou muito forte após décadas: microservices.

Microservices não é algo exatamente novo, embora tenha-se voltado a falar dessa arquitetura há poucos anos. A ideia central é que você quebre sua aplicação em serviços bem pequenos, semelhante ao que SOA e CORBA propõem, mas sem as complicações que as grandes corporações criaram no em torno destas duas excelentes ideias (ESB?). Cada um desses microservices é auto suficiente na sua responsabilidade, é independente de linguagem, de persistência e comunica-se com os demais serviços através de protocolos comuns, como HTTP.

Não é à toa que tecnologias como Node.js, Elixir, Scala e Go (sem citar as funcionais em geral) estejam tão em alta ao mesmo tempo em que só se fala de arquitetura microservices e squads (não esqueçamos a persistência poliglota!). Ao que tudo indica, essa combinação de tecnologias leves e focadas em paralelismo, em micro serviços, mantidos por squads, para compor grandes soluções; são a “bola da vez” e podem ser a “salvação” para conseguirmos construir (e manter) os sistemas do futuro. Grandes empresas como Amazon, ThoughtWorks, Uber e Netflix acreditam nisso.

Por que eu não acreditaria? :)

# SEGUINDO EM FRENTE

---

“

A code is like love, it has created with clear intentions at the beginning, but it can get complicated.

- *Gerry Geek*

”

Este ebook termina aqui.

Pois é, certamente você está agora com uma vontade louca de aprender mais e criar aplicações incríveis com arquitetura de micro services, que resolvam problemas das empresas e de quebra que o deixem cheio de dinheiro na conta bancária, não é mesmo?

Pois é, eu também! :)

Este livro é pequeno se comparado com o universo de possibilidades que esta arquitetura nos traz. Como professor, costumo dividir o aprendizado de alguma tecnologia ou conceito em duas grandes etapas: aprender o básico e executar o que foi aprendido no mercado, para alcançar os níveis intermediários e avançados. Acho que este guia atende bem ao primeiro requisito, mas o segundo só depende de você.

De nada adianta saber muita teoria se você não aplicar ela. Então agora que terminou de ler este ebook e já conhece o básico sobre construção de micro serviços, recomendo buscar algum material mais prático, envolvendo alguma linguagem de programação, como os meus livros e cursos que listo mais adiante.

Me despeço de você leitor com uma sensação de dever cumprido.

Caso tenha gostado do ebook, envia para um amigo que também deseja construir micro services. Não tenha medo da concorrência e abrace a ideia de ter um sócio que possa lhe ajudar nos projetos.

Caso não tenha gostado tanto assim, envie suas dúvidas, críticas e sugestões para  [contato@luiztools.com.br](mailto: contato@luiztools.com.br) que estou sempre disposto a melhorar.

Um abraço e até a próxima!

# MEUS CURSOS

## Curso online NODE.JS e MONGODB

[SAIBA MAIS...](#)

## Curso online Scrum e métodos Ágeis

[SAIBA MAIS...](#)

## Curso online Jira

[SAIBA MAIS...](#)

## Curso online Web Full Stack JavaScript

[SAIBA MAIS...](#)

## Curso online React Native com Firebase

[SAIBA MAIS...](#)

Conheça todos os meus cursos

# MEUS LIVROS



Programação  
Web com Node.js

[SAIBA MAIS...](#)



Programação  
Web com Node.js

[SAIBA MAIS...](#)



NODEJS E  
MICROSERVICES  
UM GUIA PRÁTICO



Node.js e  
Microservices

[SAIBA MAIS...](#)



MongoDB  
para  
Iniciantes

POR LUIZTOOLS

MongoDB  
para Iniciantes

[SAIBA MAIS...](#)



Scrum e  
Métodos Ágeis

[SAIBA MAIS...](#)



Agile Coaching

[SAIBA MAIS...](#)



Criando apps  
para empresas  
com Android

[SAIBA MAIS...](#)



Java para  
iniciantes

[SAIBA MAIS...](#)

## Conheça todos os meus livros

Aproveita e segue nas redes sociais:

