Pontifícia Universidade Católica - PUC

Arquitetura de Backend e Microsserviços
Material de Apoio para a Unidade 3

Tradução e adaptação do Artigo Microservices – A definition of this new Architectural term, de James Lewis e Martin Fowler
https://martinfowler.com/articles/microservices.html

1 Microsserviços

O termo Arquitetura de Microsserviços surgiu nos últimos anos para descrever uma forma particular de projetar aplicações de software como suítes de serviços implantáveis independentemente. Embora não exista uma definição formal deste estilo arquitetural, há certas características comuns ao redor de organização deles, em torno da capacidade de negócios, implantação automatizada, a inteligência nos endpoints e controle descentralizado das linguagens e banco de dados.

Temos visto muitos projetos utilizarem este estilo nos últimos anos, e os resultados até agora têm sido positivos, tanto assim que, para muitos de nossos colegas isso está se tornando o estilo padrão para construção de aplicativos corporativos.

Em resumo, o estilo arquitetural de microsserviços [1] é uma abordagem para o desenvolvimento de uma única aplicação como um conjunto de pequenos serviços, cada um executando em seu próprio processo e comunicando-se com mecanismos leves, muitas vezes, uma API de recursos HTTP. Estes serviços são construídos em torno de capacidades de negócios e independentemente implementados por máquinas de implantação totalmente automatizadas. Há um mínimo de gerenciamento centralizado destes serviços, que podem ser escritos em diferentes linguagens de programação e utilizarem diferentes tecnologias de armazenamento de dados.

Para começar a explicar o estilo de microsserviços é útil compará-lo com o estilo monolítico: um aplicativo monolítico construído como uma única unidade. Aplicações Corporativas são muitas vezes construídos em três partes principais: uma interface com o usuário do lado do cliente (que consiste em páginas HTML e JavaScript em execução em um navegador na máquina do usuário), um banco de dados (que consiste em muitas tabelas inseridas em um comum, e geralmente relacional, sistema de gerenciamento de banco de dados) e uma aplicação do lado do servidor. A aplicação do lado do servidor irá lidar com as solicitações HTTP, executar a lógica do domínio, recuperar e atualizar dados do banco de dados, e selecionar e preencher as views HTML para serem enviadas para o browser. Esta aplicação do lado do servidor é um monologito - um único executável lógico [2]. Quaisquer mudanças no sistema envolvem criação e implantação de uma nova versão do aplicativo do lado do servidor.

Tal servidor monolítico é uma maneira natural de se aproximar a construção de um sistema deste tipo. Toda a sua lógica para tratar um pedido é executado em um único processo, o que lhe permite usar os recursos básicos de sua linguagem para dividir a aplicação em classes, funções e namespaces. Com um pouco de cuidado, você pode executar e testar o aplicativo no computador de um desenvolvedor, e usar um pipeline de implantação para garantir que as mudanças sejam devidamente testadas e implantadas em produção. Você pode escalar horizontalmente o monoligot, executando muitos casos atrás de um balanceador de carga.

Aplicações monolíticas podem ser bem-sucedidas, mas cada vez mais as pessoas estão se sentindo frustradas com elas - especialmente à medida que um maior número de aplicações está sendo implementadas para a nuvem. Alterar ciclos que estão ligados entre si - uma alteração feita uma pequena parte da aplicação - requer que todo o monólito seja reconstruído e implementado. Ao longo do tempo, fica difícil manter uma boa estrutura modular, tornando mais difícil para manter as alterações que devem afetar apenas um módulo dentro de outro módulo. Escalonamento requer dimensionamento de toda a aplicação ao invés das partes que demandam maior de recursos.

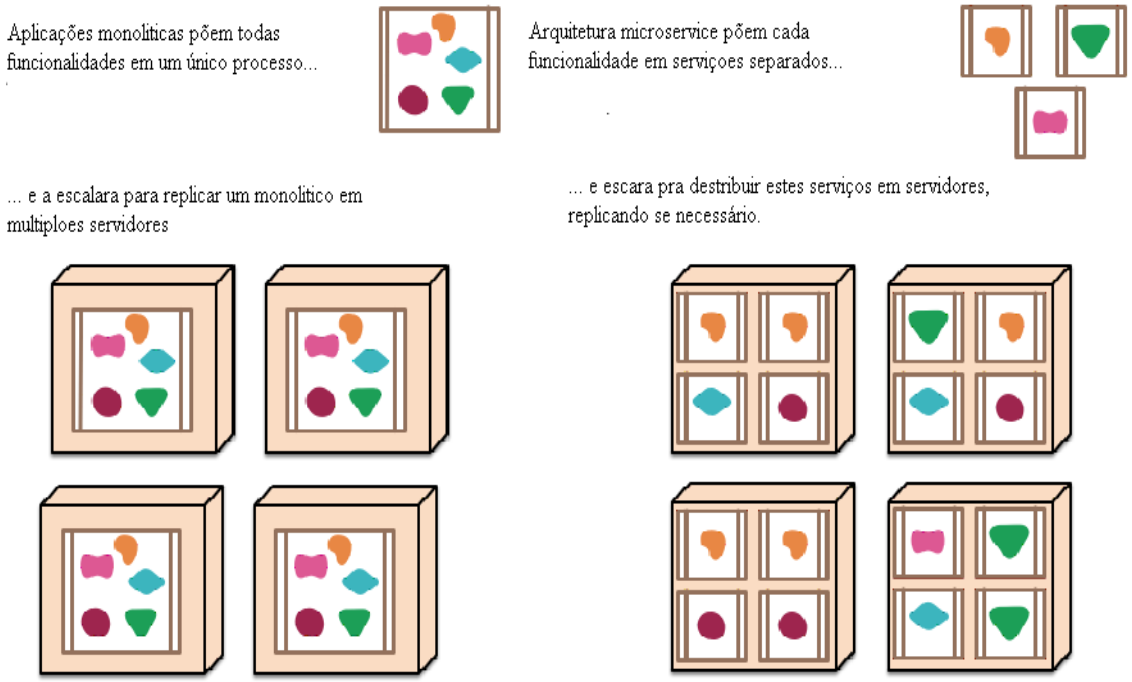

Figura 1: *monolíticos e microsserviços.*

Estas frustrações levaram ao estilo arquitetural de microsserviços, que envolve a construção de aplicações como suítes de serviços. O fato de que os serviços são implementados de forma independente e escalável permite que diferentes serviços serem escritos em diferentes linguagens de programação. Eles também podem ser geridos por equipas diferentes.

Nós não reivindicamos que o estilo de microsserviços é novidade ou inovador. As suas raízes remetem para os princípios de design do Unix. Mas achamos que a arquitetura de microsserviços não é considerada seriamente por muitas pessoas e que muitos desenvolvimentos de software poderiam ser melhorados se eles utilizassem esta arquitetura.

2 Características de uma Arquitetura de Microsserviços

Não podemos dizer que há uma definição formal do estilo arquitetural de microsserviços, mas podemos tentar descrever o que vemos como características comuns para arquiteturas que se encaixam neste rótulo. Como acontece com qualquer definição que traça características comuns, nem todas as arquiteturas de microsserviços possuem todas as características, mas esperamos que a maioria dessas arquiteturas exponham a maioria das características. Embora nós autores tenhamos sido membros ativos desta comunidade, a nossa intenção é tentar descrever da forma que vemos no nosso próprio trabalho e em esforços semelhantes por equipes que conhecemos. Em particular, não estamos estabelecendo alguma definição formal aqui.

2.1 Componentização via Serviços

Durante o tempo que estivemos envolvidos na indústria de software, houve um desejo de construir sistemas interligando componentes, da mesma forma como vemos que as coisas são feitas no mundo físico. Durante o último par de décadas temos visto um progresso considerável com grandes compêndios de bibliotecas comuns que fazem parte da maioria das plataformas de linguagem.

Ao falar sobre os componentes nos deparamos com uma difícil definição do que faz um componente. A nossa definição é que um componente é uma unidade de software que é substituível e atualizável independentemente.

Arquiteturas de microsserviços irão utilizar bibliotecas, mas a sua principal forma de componentização de seu próprio software é através da decomposição em serviços. Nós definimos bibliotecas como componentes que estão ligados em um programa e chamadas usando chamadas de funções em memória, enquanto os serviços estão fora do processo de componentes que se comunicam com um mecanismo como uma solicitação de serviço web, ou chamada de procedimento remoto. (Este é um conceito diferente do de um objeto de serviço em muitos programas OO [3]).

Uma das principais razões para a utilização de serviços como componentes (em vez de bibliotecas) é que os serviços são implementados de forma independentemente. Se você tiver uma aplicação [4] que consiste em um múltiplas bibliotecas em um único processo, uma alteração para qualquer componente resulta em ter que reimplantar o aplicativo inteiro. Mas se a aplicação é decomposta em vários serviços, você pode esperar várias alterações no serviço único para exigir que apenas aquele serviço seja reimplantado. Isso não é algo absoluto. Algumas mudanças vão mudar interfaces de serviço, resultando em algum tipo de mudança e coordenação, mas o objetivo de uma boa arquitetura de microsserviços é minimizar estas mudanças através dos limites de coesão e mecanismos de evolução nos contratos de serviços.

Outra consequência do uso de serviços como componentes é uma interface de componente mais explícita. A maioria das linguagens não tem um bom mecanismo para a definição de uma interface publicada (Published Interface) explícita. Normalmente é apenas a documentação e disciplina que impedem que os clientes quebrem o encapsulamento de um componente, levando a um acoplamento forte entre componentes. Serviços tornam mais fácil de evitar isto usando mecanismos de chamadas remotas explícitas.

A utilização de serviços desta forma tem suas desvantagens. Chamadas remotas são mais caras do que as chamadas em processo, e estas APIs remotas precisam de uma granularidade mais grossa, o que muitas vezes é mais complicado de usar. Se você precisar alterar a repartição de responsabilidades entre os componentes, tais movimentos de comportamento são mais difíceis de fazer quando se está atravessando limites de um processo.

Numa primeira abordagem, podemos observar os serviços mapeiam para processos em tempo de execução, mas isso é apenas numa primeira aproximação. Um serviço pode ser constituído por vários processos que sempre serão desenvolvidos e implantados em conjunto, como por exemplo processo de aplicação e de um banco de dados que é usado apenas por esse serviço.

2.2 Organizado em torno de Capacidades de Negócios

Ao procurar dividir um grande aplicação em partes, muitas vezes a gerência concentra-se na camada de tecnologia, levando a equipes de UI, equipes de lógica do lado do servidor, e as equipes de base de dados. Quando as equipes são separadas nestas linhas, mesmo mudanças simples podem levar um grande tempo e necessidades de aprovação do orçamento. Uma equipe inteligente irá otimizar isso e buscar o menor de dois males - apenas forçar a lógica em qualquer aplicação que eles tenham acesso. Lógica em todos os lugares em outras palavras. Este é um exemplo da Lei de Conway [5] em ação.

Qualquer empresa que projeta um sistema (definido em termos gerais) irá produzir um projeto cuja estrutura é uma cópia da estrutura de comunicação da organização.
-- Melvyn Conway, 1967

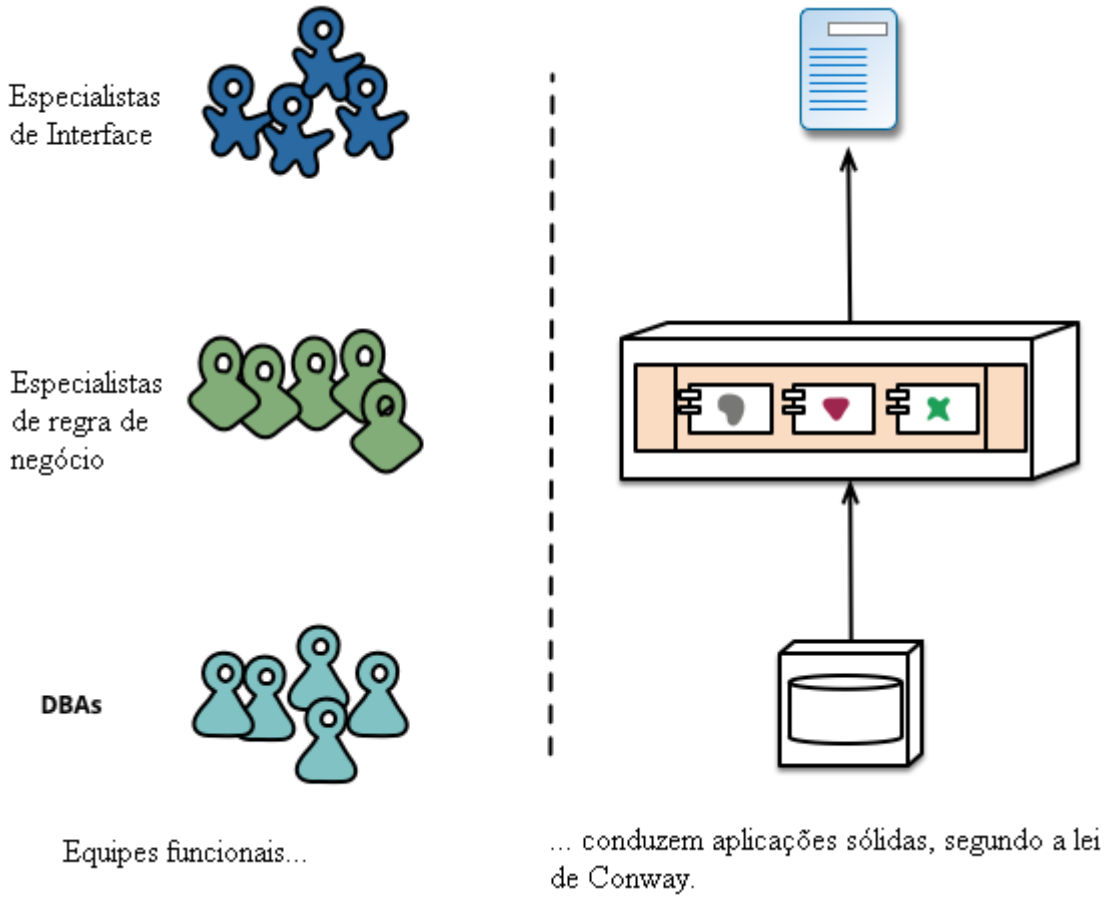

Figura 2: lei de Conway em ação.

A abordagem de microsserviços para divisão é diferente, dividindo-se em serviços organizados em torno da capacidade de negócio. Tais serviços têm uma implementação de uma ampla pilha de software para essa área de negócio, incluindo a interface do usuário, armazenamento persistente e quaisquer colaborações externas. Consequentemente, as equipes são cross-funcionais, incluindo a gama de habilidades necessárias para o desenvolvimento: experiência do usuário, banco de dados e gerenciamento de projetos.

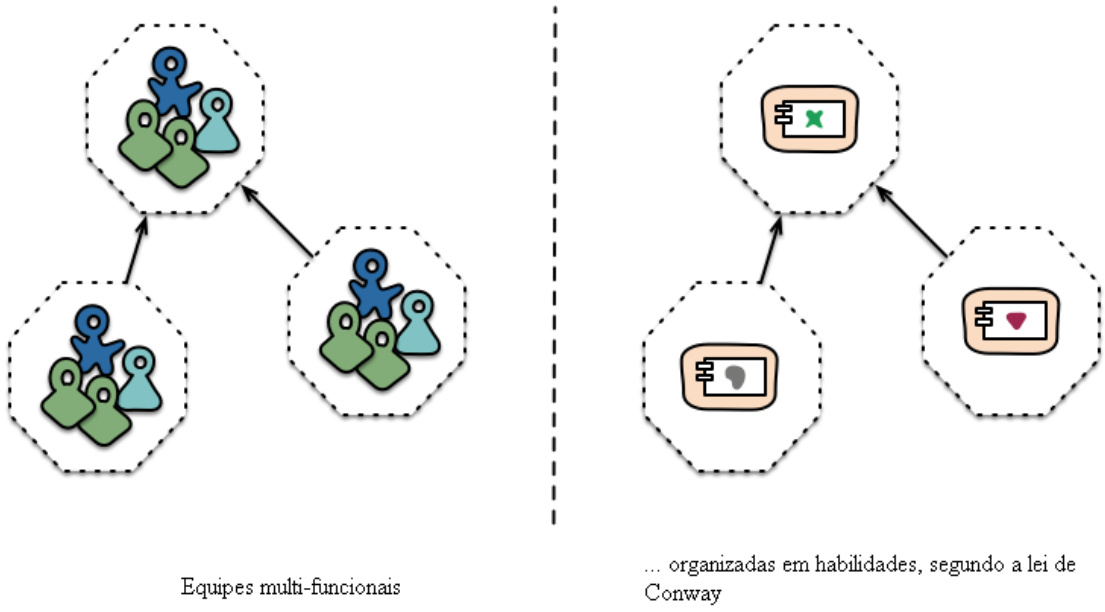

Figura 3: limites do serviço reforçado pelas fronteiras da equipe.

Uma empresa organizada dessa forma é a www.comparethemarket.com. Equipes cross-funcionais são responsáveis pela construção e operação de cada produto e cada produto é separado em uma série de serviços individuais que comunicam através de um barramento de mensagens.

Grandes aplicações monolíticas também podem ser modularizadas em torno de capacidades de negócios, apesar de que não é o caso comum. Certamente gostaríamos de exortar uma grande equipe construindo uma aplicação monolítica dividir-se ao longo das linhas de negócios. O principal problema que temos visto aqui é que elas tendem a ser organizadas em torno de muitos contextos. Se o monolito se estende por muitos desses limites modulares pode ser difícil para os membros de uma equipe encaixá-los em sua memória de curto prazo. Além disso, vemos que as linhas modulares exigem uma grande dose de disciplina para serem aplicadas. A separação necessariamente mais explícita exigida por componentes de serviço faz com que seja mais fácil de manter os limites de equipe clara.

2.3 Produtos e não Projetos

A maioria dos esforços de desenvolvimento de aplicações que vemos utilizam um modelo de projeto onde o objetivo é entregar alguma parte de um software que é então considerado concluído. Após a conclusão do software é entregue a uma organização de manutenção e a equipe do projeto que construiu é dissolvida.

A abordagem de microsserviços tende a evitar este modelo, preferindo a noção de que uma equipe deve possuir um produto durante a sua vida inteira. A inspiração comum para isso é o conceito da Amazon chamado "se você constrói, você executa ". Aqui uma equipe de desenvolvimento assume toda a responsabilidade de um software em produção. Isso traz desenvolvedores em contato no dia-a-dia com a forma como o seu software se comporta de produção e aumenta o contato com seus usuários, pois eles têm de assumir, pelo menos, parte da carga de apoio.

A mentalidade de produto faz ligação direta com recursos de negócios. Em vez de olhar para o software como um conjunto de funcionalidades para ser concluída, há uma relação em curso, onde a questão é como software pode ajudar seus usuários a aumentar a capacidade de negócios.

Não há nenhuma razão para que essa mesma abordagem não possa ser realizada com aplicações monolíticas, mas a menor granularidade de serviços pode tornar mais fácil criar as relações pessoais entre os desenvolvedores de serviços e seus usuários.

2.4 Pontos de entrada inteligentes e canais burros

Quanto à construção de estruturas de comunicação entre os diferentes processos, temos visto muitos produtos e abordagens que que buscam trazer a inteligência para próprio mecanismo de comunicação (canais inteligentes). Um bom exemplo disso é o Enterprise Service Bus (ESB), onde os ESB incluem muitas vezes sofisticadas instalações para o roteamento de mensagens, coreografia, transformação, e aplicação das regras de negócios.

A comunidade de microsserviços favorece uma abordagem alternativa: pontos de entrada inteligentes e canais burros. Aplicações construídas a partir de microsserviços buscam ser tão dissociadas e coesas quanto possível - eles possuem sua própria lógica de domínio e atuam mais como filtros no sentido Unix clássico - receber uma solicitação, aplicar a lógica adequada e produzir uma resposta. Estas são coreografadas utilizando protocolos REST simples em vez de protocolos complexos, tais como WS-Choreography ou BPEL ou orquestração realizada por uma ferramenta central como um ESB.

Os dois protocolos mais frequentemente utilizados são a solicitação-resposta HTTP com API de recursos e mensagens leves [6]. A melhor expressão para isso é:

Seja da web, não por trás da web
-- Ian Robinson

Equipes de microsserviços adotam os princípios e protocolos que a rede mundial de computadores (e, em grande parte, Unix) é construído. Muitas vezes, os recursos utilizados podem ser armazenados em cache com muito pouco esforço por parte dos desenvolvedores ou operações comuns.

A segunda abordagem de uso comum é uma mensagem através de um barramento de mensagens leve. A infraestrutura escolhida é tipicamente burra (no sentido em que atos como um roteador de mensagens somente) com implementações simples, como RabbitMQ ou ZeroMQ. O objetivo aqui é fornecer uma camada assíncrona confiável para que os pontos de entrada inteligentes consumam e produzam informações.

Em um monolito, os componentes estão executando *in-process* e a comunicação entre eles é através de qualquer método de invocação ou chamada de função. O maior problema na mudança de um monolito para microsserviços está em mudar o padrão de comunicação. A conversão ingênua do método in-memory para o RPC que gera comunicações verbosas que gere um bom desempenho. Em vez disso, você precisa substituir a comunicação de granularidade fina por uma abordagem de granulação mais grossa.

2.5 Governança Descentralizada

Uma das consequências da governança centralizada é a tendência de padronização em plataformas tecnológicas individuais. A experiência mostra que esta abordagem é restritiva - nem todo problema é um prego e nem toda solução um martelo. Nós preferimos usar a ferramenta certa para o trabalho e enquanto aplicações monolíticas podem tirar proveito de diferentes linguagens, em certa medida, não é tão comum.

Ao dividir os componentes do monolito em serviços temos uma escolha durante a criação de cada um deles. Você quer usar Node.js para exibir uma página de relatórios simples? Vá em frente. C ++ para um componente em tempo quase real? Tudo bem. Você deseja trocar em um tipo diferente de banco de dados que melhor se adequa ao comportamento de leitura de um componente? Haverá a tecnologia para reconstruir ele.

É claro que, só porque você pode fazer algo, não significa que você deve - mas particionando seu sistema desta maneira significa que você tem a opção.

Equipes que constroem microsserviços preferem uma abordagem diferente para os padrões também. Ao invés de usar um conjunto de padrões pré-definidos escrito em algum lugar no papel eles preferem a idéia de produzir ferramentas úteis que outros programadores possam utilizar para resolver problemas semelhantes aos que estão enfrentando. Estas ferramentas geralmente são colhidas de implementações e compartilhadas com um grupo mais amplo, por vezes, mas não exclusivamente, utilizando um modelo interno de código aberto. Agora que git e github tornaram-se de fato o sistema de controle de versão escolhido, práticas de código aberto estão se tornando cada vez mais comum nas empresas

A Netflix é um bom exemplo de uma organização que segue esta filosofia. Compartilhando utilitários e, acima de tudo, códigos testados em produção como bibliotecas ela encoraja outros programadores para resolver problemas semelhantes de formas semelhantes e ainda deixam a porta aberta para escolherem uma abordagem diferente, se necessário.

Para a comunidade de microsserviços, custos extras são coisas a serem evitadas. Isso não quer dizer que a comunidade não valoriza contratos de serviços. Muito pelo contrário, já que tendem a ser muito mais deles. É que eles estão olhando para diferentes formas de gerir esses contratos. Padrões como Tolerant Reader e Contratos voltados para o consumidor muitas vezes são aplicados a microsserviços. Esses contratos de serviços auxiliam na evolução deles de forma independente. A execução de contratos de consumo impulsionado como parte de sua construção aumenta a confiança e fornece feedback rápido se os seus serviços estão funcionando. Na verdade, nós conhecemos uma equipe na Austrália que dirige a construção de novos serviços com contratos voltados para o consumidor. Eles usam ferramentas simples que lhes permitam definir o contrato de um serviço. Isto se torna parte da compilação automatizada antes que o código para o novo serviço seja mesmo escrito. O serviço é então construído apenas para o ponto onde ele cumpre o contrato - uma abordagem elegante para evitar o dilema YAGNI [9] durante a criação um novo software. Estas técnicas e as ferramentas crescendo ao redor deles limita a necessidade de gestão de contratos centrais diminuindo o acoplamento temporal entre serviços.

Talvez o apogeu da governança descentralizada seja a ética “você constrói, você roda” popularizada pela Amazon. As equipes são responsáveis por todos os aspectos do software que constroem incluindo operar o software 24/7. A devolução deste nível de

responsabilidade não é, definitivamente, a norma, mas nós vemos mais e mais empresas empurrando esta responsabilidade para as equipes de desenvolvimento. Netflix é outra organização que tenha adoptado este ética [11]. Sendo acordado às 3 da manhã todas as noites pelo seu celular é certamente um poderoso incentivo para se concentrar em qualidade ao escrever o seu código. Essas ideias são tão distantes do modelo de governança centralizada tradicional como é possível ser.

2.6 Gestão de Dados Descentralizada

Descentralização da gestão de dados se apresenta em um número de maneiras diferentes. No nível mais abstrato, isso significa que o modelo conceitual do mundo será diferente entre os sistemas. Este é um problema comum na integração através de uma grande empresa: o ponto de vista de vendas de um cliente será diferente da visão do suporte. Algumas coisas que são chamadas de clientes na visão de vendas pode não se assemelhar em todos os aspectos na visão do suporte. Aqueles que fazem podem ter diferentes atributos e (pior) atributos comuns com sutis diferenças semânticas.

Este problema é comum entre as aplicações, mas também pode ocorrer dentro de aplicações, em particular quando essa aplicação é dividida em componentes separados. Uma maneira útil de pensar sobre isso é a noção Domain-Driven Design de contexto limitado. DDD divide um domínio complexo em múltiplos contextos delimitados e mapeia as relações entre eles. Este processo é útil para ambas as arquiteturas monolítica e de microsserviços, mas há uma correlação natural entre serviços e limites de contexto que ajuda a esclarecer e, como descrevemos na seção de recursos de negócios, reforça as separações.

Assim como a descentralização das decisões sobre modelos conceituais, os microsserviços também descentralizam as decisões de armazenamento de dados. Enquanto as aplicações monolíticas preferem um único banco de dados lógico para dados persistentes, as empresas muitas vezes preferem um único banco de dados em uma variedade de aplicações - muitas dessas decisões impulsionadas por meio de modelos comerciais de fornecedores em torno de licenciamento. Os microsserviços preferem deixar cada serviço gerir o seu próprio banco de dados. Diferentes instâncias da mesma tecnologia de banco de dados, ou sistemas completamente diferentes de banco de dados - uma abordagem chamada Persistência Poliglota. Você pode usar persistência poliglota em um monolito, mas isso aparece mais frequentemente com microsserviços.

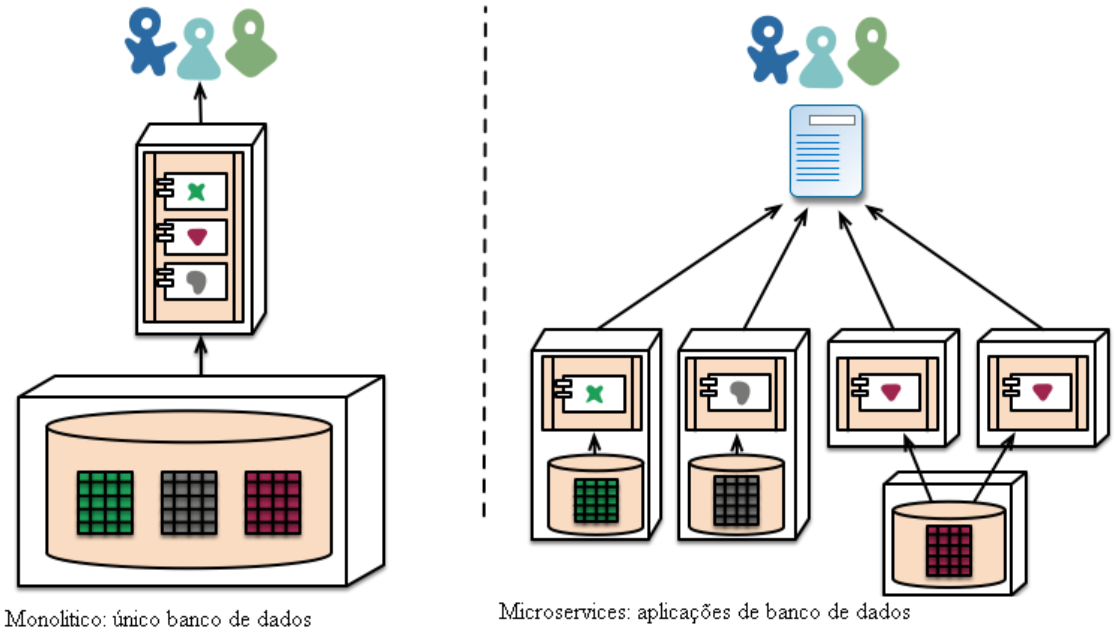

Figura 4: Descentralização de dados

Descentralizando a responsabilidade de dados através de microsserviços tem implicações para gerenciar atualizações. A abordagem comum para tratar as atualizações tem sido a de utilizar transações para garantir a consistência ao atualizar vários recursos. Esta abordagem é muitas vezes usada dentro dos monolitos.

Usando transações como estas contribui com consistência, mas impõe acoplamentos temporais significativos, o que é problemático em vários serviços. Transações distribuídas são notoriamente difíceis de implementar e, como consequência as arquiteturas de microsserviços enfatizam a coordenação transacional entre os serviços, com reconhecimento explícito de que a consistência pode ser apenas uma consistência eventual. Problemas são tratados por meio de operações de compensação, caso a caso.

Escolhendo gerenciar inconsistências desta forma é um novo desafio para muitas equipes de desenvolvimento, mas isso é o que muitas vezes coincide com a prática empresarial. Muitas vezes as empresas lidam com certo grau de incoerência a fim de responder rapidamente a demanda, apesar de terem algum tipo de processo de reversão para lidar com os erros. O trade-off vale a pena, desde que o custo dos erros de correção seja menor do que o custo da perda de negócios sob uma maior coerência.

2.7 Automação de Infraestrutura

Técnicas de automação Infraestruturas têm evoluído muito nos últimos anos - a evolução da nuvem e plataformas como a AWS, em especial, tem reduzido a complexidade de construção, implantação e operação de serviços.

Muitos dos produtos ou sistemas que estão sendo construídos com microsserviços estão sendo construídos por equipes com vasta experiência de Entrega Contínua e seu precursor, Integração Contínua. As equipes que constroem software desta forma fazem uso extensivo de técnicas de automação de infraestrutura. Isto é ilustrado na figura a seguir.

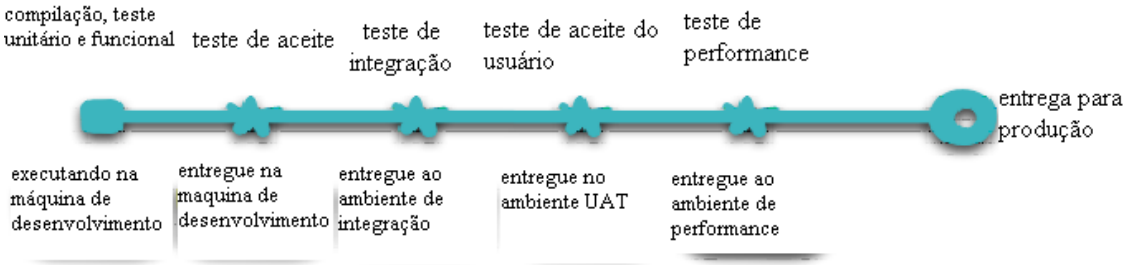

Figura 5: pipeline de construção de microsserviços.

Uma vez que este não é um artigo sobre a Entrega Contínua, vamos chamar a atenção para apenas algumas características-chave aqui. Queremos tanta confiança quanto possível de que nosso software está funcionando, então executamos muitos testes automatizados. A promoção de um software trabalhando para a direita do pipeline significa que automatizamos a implantação de um novo ambiente.

A aplicação monolítica será construída, testada e empurrada através destes ambientes. Acontece que depois de ter investido tempo em automatizar o caminho para a produção de um monolito, a implantação de novas aplicações não parecerá mais tão assustador. Lembre-se, um dos objetivos da entrega contínua é tornar a implantação simples. [12].

Outra área em que vemos equipes usando automação extensiva de infraestrutura é na gestão de microsserviços na produção.

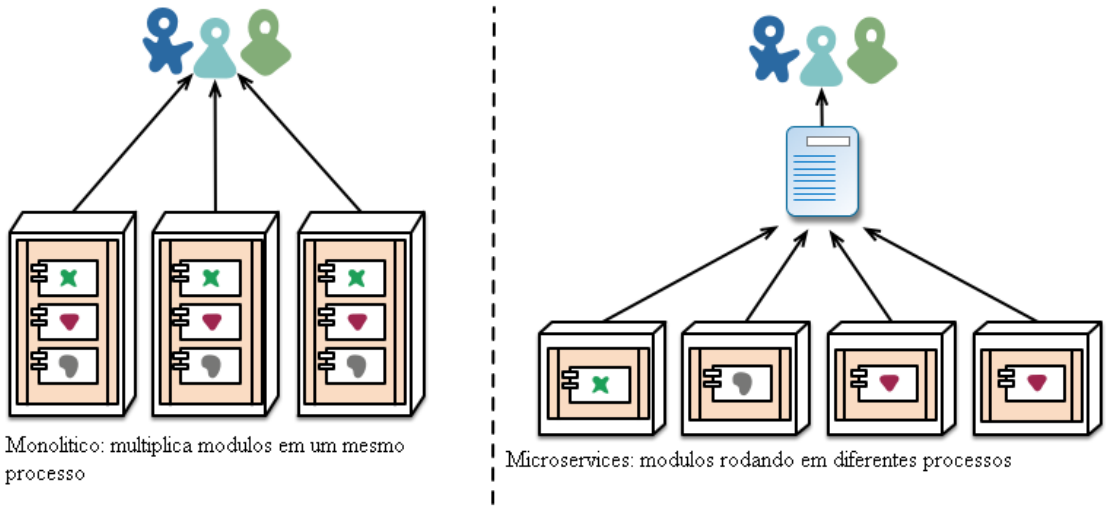

Figura 6: Implantação do Módulo muitas vezes se diferem.

2.8 Desenho para falhas

Uma consequência do uso de serviços como componentes é que aplicações precisam ser projetadas de maneira que elas possam tolerar as falhas dos serviços. Qualquer chamada de serviço poderia falhar devido à indisponibilidade do fornecedor e o cliente tem que responder a este da forma mais limpa possível. Esta é uma desvantagem em comparação

com um design monolítico, uma vez que aumenta a complexidade de lidar com isso. A consequência é que as equipes de microsserviços precisam refletir constantemente sobre como falhas de serviço podem afetar a experiência do usuário. As ferramentas Simian Army do Netflix induzem falhas de serviços e até mesmo datacenters durante a jornada de trabalho para testar a resiliência e a monitoração da aplicação.

Este tipo de teste automatizado em produção seria suficiente para dar a maioria dos grupos de funcionamento o tipo de tremores que geralmente precedem férias. Isso não quer dizer que os estilos arquitetônicos monolíticos não são capazes de setups de monitoramento sofisticados - é apenas menos comum em nossas experiências.

Uma vez que os serviços podem falhar a qualquer momento, é importante ser capaz de detectar as falhas rapidamente e, se possível, restabelecer automaticamente o serviço. Aplicações de microsserviços colocam muita ênfase no monitoramento em tempo real da aplicação, verificando os elementos arquiteturais (quantos pedidos por segundo o banco de dados está fornecido) e métricas relevantes de negócio (tais como quantos pedidos por minuto são recebidos). O monitoramento semântico pode fornecer um sistema de alerta de algo dar errado que aciona as equipes de desenvolvimento para acompanhar e investigar.

Isto é particularmente importante para uma arquitetura de microsserviços porque a preferência por coreografia de microsserviços e colaboração de eventos leva a um comportamento emergente. Enquanto muitos especialistas elogiam o valor do surgimento casual, a verdade é que o comportamento emergente às vezes pode ser uma coisa ruim. O monitoramento é vital para detectar e corrigir um comportamento emergente ruim.

Monolitos podem ser construídos para serem tão transparentes quanto os microsserviços - na verdade, eles deveriam ser. A diferença é que é absolutamente necessário saber quando os serviços que funcionam em diferentes processos estão desconectados. Com bibliotecas dentro do mesmo processo este tipo de transparência é menos provável de ser útil.

Equipes de microsserviços esperariam ver monitoramento sofisticados e registros de configurações para cada serviço de forma individual, tais como painéis mostrando os status up/down e uma variedade de métricas relevantes operacionais e de negócios.

2.9 Design evolucionário

Os praticantes de microsserviços geralmente vêm de um passado de design evolucionário e vêem a decomposição serviço como uma ferramenta adicional para permitir que os desenvolvedores de aplicações possam controlar as mudanças em sua aplicação. Com as atitudes e as ferramentas certas, você pode fazer mudanças frequentes, rápidas, e bem controlados para o software.

Sempre que você tenta quebrar um sistema de software em componentes, você se depara com a decisão de como dividir as peças - quais são os princípios em que optamos para fatiar nossa aplicação? A propriedade chave de um componente é a noção de substituição independente e a capacidade de atualização [13] - o que implica em olharmos para os pontos onde podemos imaginar reescrever um componente sem afetar seus colaboradores. Na verdade muitos grupos de microsserviços levam isto um passo

além, esperando explicitamente muitos que alguns serviços sejam abandonados, em vez de evoluírem no longo prazo.

O site The Guardian é um bom exemplo de uma aplicação que foi projetada e construída como um monolito, mas tem evoluído em direção ao padrão de microsserviços. O monolito ainda é o núcleo do site, mas eles preferem adicionar novos recursos através da construção de microsserviços que usam a API do monólito. Esta abordagem é particularmente útil para recursos que são inerentes à atividades temporárias, como páginas especializadas para lidar com um evento esportivo. Essa parte do site pode ser organizada rapidamente utilizando linguagens rápidas de desenvolvimento, e removida uma vez que o evento acabou. Vimos abordagens semelhantes em uma instituição financeira, onde novos serviços são adicionados para uma oportunidade de mercado e descartados depois de alguns meses ou até mesmo semanas.

Essa ênfase na substituição é um caso especial de um princípio do design modular, o que é conduzir modularidade através do padrão de mudança [14]. Você quer manter as coisas que mudam ao mesmo tempo no mesmo módulo. Partes de um sistema que mudam raramente deve estar em diferentes serviços para as pessoas que estão passando por muitas mudanças. Se você necessitar repetidamente mudar dois serviços em conjunto, isso é um sinal de que eles deveriam ser fundidos.

Colocar componentes em serviços acrescenta uma oportunidade para o planejamento de liberação mais granular. Com um monolito quaisquer mudanças exigem uma compilação completa e implantação de toda a aplicação. Com microsserviços, no entanto, você só precisa reimplantar o(s) serviço(s) que você modificou. Isto pode simplificar e acelerar o processo de liberação. A desvantagem é que você tem que se preocupar com que as mudanças de um serviço não quebrem seus consumidores. A abordagem de integração tradicional é tentar lidar com esse problema usando o controle de versão, mas a preferência no mundo da microsserviços é usar apenas o controle de versão como um último recurso. Podemos evitar um monte de controle de versão através da concepção de serviços tão tolerantes quanto possível às mudanças em seus fornecedores.

3 Os microsserviços são o futuro?

Nosso principal objetivo ao escrever este artigo é para explicar as principais ideias e princípios de microsserviços. Ao tomar esta iniciativa pensamos claramente que o estilo arquitetural de microsserviços é uma ideia importante - digna de uma séria consideração para aplicações empresariais. Nós construímos recentemente vários sistemas que usam este estilo e conhecemos outras pessoas que também utilizaram e favorecem esta abordagem.

Aqueles que sabemos quem são, de alguma forma pioneiros neste estilo arquitetural incluem Amazon, Netflix, The Guardian, the UK Government Digital Service, realestate.com.au, Forward and comparethemarket.com. O circuito de conferência em 2013 foi cheio de exemplos de empresas que estão se movendo para algo que gostaria de classificar como microsserviços - incluindo o Travis CI. Além disso, há uma abundância de organizações que feito muito o que gostaria de classificar como microsserviços, mas sem nunca usar este nome. (Muitas vezes isso é rotulado como SOA -., Embora, como já dissemos, SOA vem em muitas formas contraditórias [15]).

Apesar dessas experiências positivas, no entanto, não estamos defendendo que estamos certos de que microsserviços são a futura direção para arquiteturas de software. Por enquanto nossas experiências até agora foram positivas em comparação com aplicações monolíticas. Mas estamos conscientes do fato de que não se passou tempo suficiente para fazermos um julgamento completo.

Muitas vezes, as verdadeiras consequências de suas decisões de arquitetura são apenas evidentes vários anos após você ter tomado elas. Vimos projetos onde uma boa equipe, com um forte desejo de modularidade, construiu uma arquitetura monolítica que decaiu ao longo dos anos. Muitas pessoas acreditam que essa decadência é menos provável com microsserviços, uma vez que os limites de serviço são explícitos e difíceis de corrigir ao redor. No entanto, até que vejamos um número considerável de sistemas com idades suficientes, não podemos realmente avaliar como arquiteturas microsserviços amadurecem.

Existem, certamente, razões pelas quais se poderiam esperar um pobre amadurecimento dos microsserviços. Em qualquer esforço de componentização, o sucesso depende de quão bem o software se encaixa em componentes. É difícil descobrir exatamente onde os limites de componentes devem ir. O design evolucionário reconhece as dificuldades de obtenção de um correto limite e, portanto, a importância de que seja fácil de refatora-los. Mas quando os seus componentes são serviços com comunicações remotas, então refactoring é muito mais difícil do que com bibliotecas in-process. Mover código é difícil além das fronteiras dos serviços, todas as mudanças de interface devem ser coordenadas entre os participantes, as camadas de compatibilidade com versões anteriores precisam ser adicionados e o teste é realizado de maneira mais complicada.

Outra questão é se os componentes não compõem de forma limpa, então tudo que você está fazendo está mudando a complexidade dentro de um componente para as conexões entre os componentes. Não é só fazer a mudança de complexidade para o redor, move-la para um lugar que é menos explícita e mais difícil de controlar. É fácil pensar que as coisas são melhores quando você está olhando para o interior, um componente simples pequeno, enquanto faltando conexões confusas entre os serviços.

Finalmente, existe o fator de habilidade da equipe. Novas técnicas tendem a ser adotadas pelas equipes mais hábeis. Mas uma técnica que é mais eficaz para uma equipe mais hábil não irá necessariamente funcionar para as equipes menos hábeis. Temos visto muitos casos de equipes menos hábeis construindo arquiteturas monolíticas bagunçadas, mas é preciso de tempo para ver o que acontece quando esse tipo de confusão ocorre com microsserviços. Uma equipe ruim vai sempre criar um sistema fraco - é muito difícil dizer se os microsserviços reduziriam a bagunça neste caso ou se iriam torná-lo ainda pior.

Um argumento razoável que ouvimos é que você não se deve começar com uma arquitetura de microsserviços. Em vez disso, comece com um monólito, mantendo-o modular, e dividindo-o em microsserviços quando o monolito se tornar um problema. (Embora este conselho não é o ideal, uma vez que uma boa interface in-process geralmente não é uma boa interface de serviço.)

Então nós escrevemos este artigo com um otimismo cauteloso. Até agora, temos visto o suficiente sobre o estilo de microsserviços para sentir que ele pode ser um caminho que vale a pena trilhar. Nós não podemos dizer com certeza onde vamos terminar, mas um dos desafios do desenvolvimento de software é que você só pode tomar decisões com base na informação imperfeita que atualmente você tem que carregar.

4 Referencias e leituras futuras

Há uma série de fontes que defendem uma filosofia semelhante ao descrito neste artigo. Algumas destas fontes estão colocadas abaixo.

- Clemens Vasters' blog on cloud at microsoft
- David Morgantini's introduction to the topic on his blog
- 12 factor apps from Heroku
- UK Government Digital Service design principles
- Jimmy Nilsson's blogand article on infoq about Cloud Chunk Computing
- Alistair Cockburn on Hexagonal architectures

Livros

- Release it
- Rest in practice
- Web API Design (free ebook). Brian Mulloy, Apigee.
- Enterprise Integration Patterns
- Art of unix programming
- Growing Object Oriented Software, Guided by Tests
- The Modern Firm: Organizational Design for Performance and Growth
- Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation
- Domain-Driven Design: Tackling Complexity in the Heart of Software

Apresentações

- Architecture without Architects. Erik Doernenburg.
- Does my bus look big in this?. Jim Webber and Martin Fowler, QCon 2008
- Guerilla SOA. Jim Webber, 2006
- Patterns of Effective Delivery. Dan North, 2011.
- Adrian Cockcroft's slideshare channel.
- Hydras and Hypermedia. Ian Robinson, JavaZone 2010
- Justice will take a million intricate moves. Leonard Richardson, Qcon 2008.
- Java, the UNIX way. James Lewis, JavaZone 2012
- Micro services architecture. Fred George, YOW! 2012
- Democratising attention data at guardian.co.uk. Graham Tackley, GOTO Aarhus 2013
- Functional Reactive Programming with RxJava. Ben Christensen, GOTO Aarhus 2013 (registration required).
- Breaking the Monolith. Stefan Tilkov, May 2012.

Artigos

- L. Lamport, "The Implementation of Reliable Distributed Multiprocess Systems", 1978 http://research.microsoft.com/en-us/um/people/lamport/pubs/implementation.pdf
- L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problem", 1982 (available at) http://www.cs.cornell.edu/courses/cs614/2004sp/papers/lsp82.pdf
- R.T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures", 2000 http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- E. A. Brewer, "Towards Robust Distributed Systems", 2000 http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf
- E. Brewer, "CAP Twelve Years Later: How the 'Rules' Have Changed", 2012, http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed