

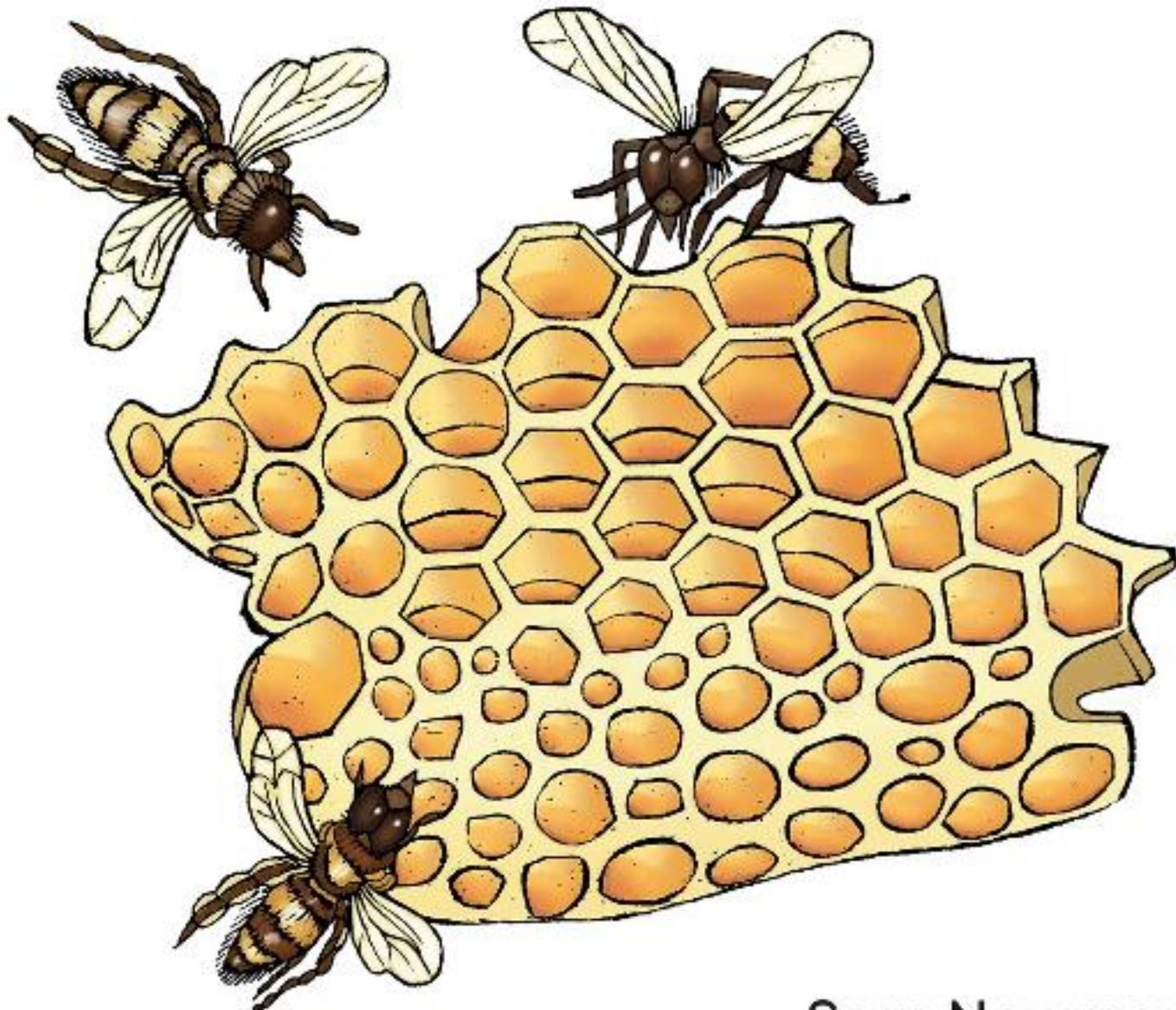
O'REILLY®

Second
Edition

Edifício

Microserviços

Projetando sistemas refinados



Sam Newman
Sam Newman

Criando microsserviços

SEGUNDA EDIÇÃO

Projetando sistemas refinados

Sam Newman

O'REILLY®

Pequim, Boston... • Farnham • Sebastopol, Tóquio

Criando microserviços

de Sam Newman

Direitos autorais: © 2021 Sam Newman. Todos os direitos reservados.

Impresso no Canadá.

Publicado pela O'Reilly Media, Inc., 1005 Gravenstein Highway North,,
Sebastopol, CA 95472.

Os livros da O'Reilly podem ser comprados para fins educacionais, comerciais ou de vendas uso promocional. As edições online também estão disponíveis para a maioria dos títulos (<http://oreilly.com>). Para mais informações, entre em contato com nosso departamento de vendas corporativo/institucional: 800-998-9938 ou corporate@oreilly.com.

Editora de aquisições: Melissa Duffield Indexadora: Judith McConville

Editora de desenvolvimento: Nicole Taché Designer de interiores: David Futato

Edição de produção: Deborah Baker Designer de capa: Karen Montgomery

Editor de texto: Arthur Johnson

Ilustradora: Kate Dullea

Revisor: Charles Roumeliotis

Fevereiro de 2015: Primeira edição

Agosto de 2021: segunda edição

Histórico de revisões da segunda edição

- 2021-07-23: Primeiro lançamento

Consulte <http://oreilly.com/catalog/errata.csp?isbn=978149203402> para ver o lançamento detalhes.

O logotipo O'Reilly é uma marca registrada da O'Reilly Media, Inc. Microserviços de construção, a imagem da capa e a imagem comercial relacionada são marcas comerciais da O'Reilly Media, Inc.

As opiniões expressas neste trabalho são do autor e não representam as opiniões do editor. Embora o editor e o autor tenham usado esforços de boa fé para garantir que as informações e instruções contidos neste trabalho são precisos, o editor e o autor negam tudo responsabilidade por erros ou omissões, incluindo, sem limitação responsabilidade por danos resultantes do uso ou da confiança neste trabalho. O uso das informações e instruções contidas neste trabalho é de sua responsabilidade. risco. Se houver alguma amostra de código ou outra tecnologia que este trabalho contenha ou descreva está sujeito a licenças de código aberto ou aos direitos de propriedade intelectual de terceiros, é sua responsabilidade garantir que seu uso esteja em conformidade com tais licenças e/ou direitos.

978-1-492-03402-5

[MBP]

Prefácio

Os microserviços são uma abordagem para sistemas distribuídos que promovem o uso de serviços refinados que podem ser alterados, implantados e lançados de forma independente. Para organizações que estão se movendo de forma mais flexível, sistemas acoplados, com equipes autônomas oferecendo funcionalidade voltada para o usuário, os microserviços funcionam incrivelmente bem. Além disso, os microserviços nos fornecem com um grande número de opções para construir sistemas, oferecendo muitas flexibilidades para garantir que nosso sistema possa mudar para atender às necessidades dos usuários.

No entanto, os microserviços têm desvantagens significativas. Como distribuído, sistema, eles trazem uma série de complexidade, muitas das quais podem ser novas até mesmo para desenvolvedores experientes.

As experiências de pessoas em todo o mundo, juntamente com o surgimento de novas tecnologias, estão tendo um efeito profundo na forma como os microserviços são usado. Este livro reúne essas ideias, juntamente com ideias concretas do mundo real, exemplos, para ajudar você a entender se os microserviços são adequados para você.

Quem deveria ler este livro

O escopo da construção de microserviços é amplo, pois as implicações de as arquiteturas de microserviços também são amplas. Como tal, este livro deve apelar para pessoas interessadas em aspectos de design, desenvolvimento, implantação, testes, e manutenção de sistemas. Aqueles de vocês que já embarcaram no jornada em direção a arquiteturas mais refinadas, seja para um campo novo, aplicação ou como parte da decomposição de um sistema existente e mais monolítico, encontrará muitos conselhos práticos para ajudá-lo. Este livro também ajudará aqueles de vocês que querem saber o motivo de todo esse alvoroço, para que você possa determinar se os microserviços são adequados para você.

Por que escrevi este livro

Em um nível, escrevi este livro porque queria ter certeza de que as informações da primeira edição permanecem atualizadas, precisas e úteis. EU escrevi a primeira edição porque havia ideias muito interessantes que eu queria compartilhar. Tive a sorte de estar em um lugar onde tive tempo e apoio para escrever a primeira edição, e que eu poderia fazer isso de uma forma justa ponto de vista imparcial porque eu não trabalhei para um grande fornecedor de tecnologia. EU não estava vendendo uma solução e também esperava não estar vendendo microsserviços -Achei as ideias fascinantes e adorei desvendar o conceito de microsserviços e encontrar maneiras de compartilhá-los de forma mais ampla.

Quando eu realmente detalho, escrevi uma segunda edição por dois motivos principais.

Em primeiro lugar, senti que poderia fazer um trabalho melhor desta vez. Eu aprendi mais, e eu sou espero que seja um pouco melhor como escritor. Mas eu também escrevi esta segunda edição porque eu desempenhei um pequeno papel em ajudar essas ideias a se tornarem populares, e como tal, tenho o dever de garantir que sejam apresentados de forma sensata, forma equilibrada. Os microsserviços se tornaram, para muitos, o padrão escolha arquitetônica. Isso é algo que eu acho difícil de justificar, e eu Eu queria uma chance de compartilhar o porquê.

Este livro não é pro microservices, nem é anti microservices. Eu só quero certifique-se de ter explorado adequadamente o contexto em que essas ideias funcionam bem e compartilharam os problemas que eles podem causar.

O que mudou desde a primeira edição?

Escrevi a primeira edição do Building Microservices em cerca de um ano, começando no início de 2014, com o livro sendo lançado em fevereiro de 2015. Isso foi no início da história dos microsserviços, pelo menos em termos mais amplos consciência da indústria sobre o termo Desde então, os microsserviços desapareceram mainstream de uma forma que eu não poderia ter previsto. Com isso, o crescimento tem vêm um conjunto muito mais amplo de experiências para aproveitar e mais tecnologia para explorar.

À medida que trabalhei com mais equipes após a primeira edição, comecei a refino meu pensamento sobre algumas das ideias associadas aos microsserviços. Em alguns casos, isso significava que ideias que estavam apenas na periferia da minha

o pensamento, como a ocultação de informações, começou a ficar mais claro à medida que conceitos fundamentais que precisavam ser melhor destacados. Em outras áreas, a nova tecnologia apresentou novas soluções e novos problemas para nossos sistemas. Ver tantas pessoas migrando para o Kubernetes na esperança de que isso acontecesse resolver todos os seus problemas com arquiteturas de microserviços certamente me deu pausa para pensar...

Além disso, escrevi a primeira edição de *Building Microservices* para fornecer não apenas uma explicação dos microserviços, mas também uma visão geral ampla de como essa abordagem arquitetônica muda aspectos do desenvolvimento de software. Então, ao examinar mais profundamente os aspectos relacionados à segurança e resiliência, descobri eu mesmo querendo voltar e expandir esses tópicos que estão cada vez mais importante para o desenvolvimento de software moderno.

Assim, nesta segunda edição, passo mais tempo compartilhando exemplos explícitos para explicar melhor as ideias. Cada capítulo foi reexaminado e cada frase revisada. Não resta muito da primeira edição em termos de concreto prosa, mas as ideias ainda estão aqui. Eu tentei ser mais claro sozinho. Opiniões, embora ainda reconheça que muitas vezes existem várias maneiras de resolver um problema. Isso significou expandir a discussão sobre interprocessos comunicação, que agora está dividida em três capítulos. Eu também gasto mais tempo analisando as implicações de tecnologias como contêineres, Kubernetes e sem servidor; como resultado, agora existem versões separadas e capítulos de implantação.

Minha esperança era criar um livro com o mesmo tamanho da primeira edição, enquanto encontra uma maneira de incluir mais ideias. Como você pode ver, eu falhei atingir meu objetivo - esta edição é maior! Mas acho que consegui ser mais claro na forma como articulo as ideias.

Navegando neste livro

Este livro é organizado principalmente em um formato baseado em tópicos. Eu estruturei e escreveu o livro como se você fosse lê-lo do começo ao fim, mas de Claro que você pode querer pular para os tópicos específicos que mais lhe interessam. Se se você decidir mergulhar direto em um capítulo específico, você pode encontrar o Glossário

no final do livro, útil para explicar termos novos ou desconhecidos. Ligado ao tópico da terminologia, eu uso microserviço e serviço de forma intercambiável ao longo do livro; você pode supor que os dois termos se referem à mesma coisa a menos que eu diga explicitamente o contrário. Também resumo algumas das chaves do livro conselhos na Bibliografia, se você realmente quiser ir direto ao fim - apenas lembre-se de que você perderá muitos detalhes se fizer isso!

O corpo principal do livro é dividido em três partes separadas: fundação, Implementação e pessoas. Vamos ver o que cada parte cobre.

Parte I, Fundação

Nesta parte, detalho algumas das principais ideias por trás dos microserviços.

Capítulo 1, O que são microserviços?

Esta é uma introdução geral aos microserviços, na qual discuto brevemente uma série de tópicos que eu expandirei mais adiante no livro.

Capítulo 2, Como modelar microserviços

Este capítulo examina a importância de conceitos como informação ocultação, acoplamento, coesão e design orientado por domínio para ajudar a encontrar os limites certos para seus microserviços.

Capítulo 3, Dividindo o monólito

Este capítulo fornece algumas orientações sobre como usar um existente aplicativo monolítico e dividi-lo em microserviços.

Capítulo 4, Estilos de comunicação de microserviços

O último capítulo desta parte oferece uma discussão sobre os diferentes tipos de comunicação por microserviços, incluindo assíncrona versus síncrona estilos de colaboração orientados por eventos e chamadas e solicitações.

Parte II, Implementação

Passando de conceitos de alto nível para detalhes de implementação, nesta parte, veja técnicas e tecnologias que podem ajudar você a aproveitar ao máximo microsserviços.

Capítulo 5. Implementando a comunicação por microsserviços

Neste capítulo, examinaremos mais profundamente as tecnologias específicas usadas para implementar comunicação entre microsserviços.

Capítulo 6, Fluxo de trabalho

Este capítulo oferece uma comparação de sagas e transações distribuídas e discute sua utilidade na modelagem de processos de negócios envolvendo vários microsserviços.

Capítulo 7. Construir

Este capítulo explica o mapeamento de um microsserviço para repositórios e constrói.

Capítulo 8, Implantação

Neste capítulo, você encontrará uma discussão sobre as inúmeras opções disponíveis para implantar um microsserviço, incluindo uma análise de contêineres, Kubernetes e.

Capítulo 9, Testando

Aqui, discuto os desafios de testar microsserviços, incluindo o problemas causados por testes de ponta a ponta e como os contratos são orientados pelo consumidor e testes em produção podem ajudar.

Capítulo 10, Do monitoramento à observabilidade

Este capítulo aborda a mudança do foco em atividades de monitoramento estático a pensar de forma mais ampla sobre como melhorar a observabilidade de arquiteturas de microsserviços, juntamente com algumas recomendações específicas em relação ao ferramental.

Capítulo 11, Segurança

As arquiteturas de microserviços podem aumentar a área de superfície do ataque, mas também nos dão mais oportunidades de nos defendermos em profundidade. Neste capítulo, nós exploraremos esse equilíbrio.

Capítulo 12, Resiliência

Este capítulo oferece uma visão mais ampla do que é resiliência e da parte em que os microserviços podem ajudar a melhorar a resiliência de seus aplicativos.

Capítulo 13, Dimensionamento

Neste capítulo, descrevo os quatro eixos de escala e mostro como eles podem ser usado em combinação para escalar uma arquitetura de microserviços.

Parte III, Pessoas

Ideias e tecnologia não significam nada sem as pessoas e organização para apoiá-los.

Capítulo 14, Interfaces de usuário

Desde a mudança de equipes de front-end dedicadas até o uso de BFFs e GraphQL, este capítulo explora como microserviços e interfaces de usuário podem trabalhar juntos.

Capítulo 15, Estruturas organizacionais

Este penúltimo capítulo se concentra em como o fluxo está alinhado e habilitado as equipes podem trabalhar no contexto de arquiteturas de microserviços.

Capítulo 16, O arquiteto evolucionário

As arquiteturas de microserviços não são estáticas, então sua visão do sistema a arquitetura pode precisar mudar - um tópico que este capítulo explora em profundidade.

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

Indica novos termos, URLs, endereços de e-mail, nomes de arquivos e arquivos extensões.

Largura constante

Usado para listagens de programas, bem como em parágrafos para se referir a elementos do programa, como nomes de variáveis ou funções, bancos de dados, dados tipos, variáveis de ambiente, declarações e palavras-chave.

GORJETA

Esse elemento significa uma dica ou sugestão.

NOTE NOTA

Esse elemento significa uma nota geral.

ADVERTÊNCIA

Esse elemento indica um aviso ou advertência.

Aprendizagem on-line da O'Reilly

NOTE NOTA

Por mais de 40 anos, a O'Reilly Media forneceu treinamento em tecnologia e negócios, conhecimento e visão para ajudar as empresas a ter sucesso.

Nossa rede exclusiva de especialistas e inovadores compartilha seus conhecimentos e experiência por meio de livros, artigos e nossa plataforma de aprendizado on-line.

A plataforma de aprendizado on-line da O'Reilly oferece acesso sob demanda ao vivo

cursos de treinamento, caminhos de aprendizado aprofundados, ambientes de codificação interativos, e uma vasta coleção de texto e vídeo de O'Reilly e mais de 200 outros editoras. Para obter mais informações, visite <http://oreilly.com/>

Como entrar em contato conosco

Por favor, envie comentários e perguntas sobre este livro para o editora:

O'Reilly Media, Inc.

1005 Rodovia Gravenstein Norte

Sebastopol, CA 95472

800-998-9938 (nos Estados Unidos ou Canadá)

707-829-0515 (internacional ou local)

707-829-0104 (fax)

Temos uma página da web para este livro, onde listamos erratas, exemplos e qualquer informação adicional. Você pode acessar esta página em
https://oreilly.ly/Building_Microservices2.

Envie um e-mail para bookquestions@oreilly.com para comentar ou fazer perguntas técnicas sobre este livro.

Para notícias e informações sobre nossos livros e cursos, visite
<http://oreilly.com/>

Encontre-nos no Facebook: <http://facebook.com/oreilly>

Siga-nos no Twitter: <http://twitter.com/oreillymedia>

Assista-nos no YouTube: <http://youtube.com/oreillymedia>

Agradecimentos

Fico continuamente impressionado com o apoio que recebo da minha família, especialmente da minha esposa, Lindy Stephens, que tolera tanta coisa com coisas incríveis graça. Dizer que este livro não existiria sem ela é um eufemismo, até se ela nem sempre acredita em mim quando eu digo isso. Isso é para ela. Também é para meu pai, Jack, Josie, Kane e todo o clã Gilmanco Staynes.

Muito deste livro foi escrito durante uma pandemia global, que, enquanto escrevo esses reconhecimentos ainda estão em andamento. Pode não significar muito, mas eu quero

estendo meus agradecimentos ao NHS aqui no Reino Unido e a todas as pessoas em todo o mundo que estão nos mantendo seguros trabalhando em vacinas, tratando os doentes, entregando nossa comida e ajudando de milhares de outras maneiras que eu não entenderia. Isso é também para todos vocês.

Esta segunda edição não existiria sem a primeira, então eu gostaria de reiterar meus agradecimentos a todos que me ajudaram durante o desafiador processo de escrevendo meu primeiro livro, inclua os revisores técnicos Ben Christensen, Martin Fowler e Venkat Subramaniam; James Lewis para nossos muitos, conversas esclarecedoras; a equipe O'Reilly de Brian MacDonald, Rachel Monaghan, Kristen Brown e Betsy Waliszewski; e excelente crítica feedback dos leitores Anand Krishnaswamy, Kent McNeil, Charles Haynes, Chris Ford, Aidy Lewis, Will Thame, Jon Eayes, Rolf Russell, Badrinath Janakiraman, Daniel Bryant, Ian Robinson, Jim Webber, Stewart Gleadow, Evan Bottcher, Eric Sword e Olivia Leonard E obrigado a Mike Loukides, eu acho, por me meter nessa confusão em primeiro lugar!

Para a segunda edição, Martin Fowler voltou mais uma vez como técnico avaliador e contou com a companhia de Daniel Bryant e Sarah Wells, que foram generosos com seu tempo e feedback. Também gostaria de agradecer a Nicky Wrightson e Alexander von Zitzerwitz por ajudar a impulsionar a revisão técnica sobre o linha de chegada. Na frente de O'Reilly, todo o processo foi supervisionado pelo meu editora incrível, Nicole Taché, sem a qual eu certamente teria ido insana; Melissa Duffield, que parece fazer um trabalho melhor gerenciando minha mais trabalho do que eu: Deb Baker, Arthur Johnson e o resto do equipe de produção (desculpe, não sei todos os seus nomes, mas obrigado!); e Mary Treseler, por guiar o navio em momentos difíceis.

Além disso, a segunda edição se beneficiou enormemente de ajuda e insights de várias pessoas, incluindo (em nenhuma ordem específica) Dave Coombes e a equipe da Tyro, Dave Halsey e a equipe do Money Supermarket, Tom Kerkhove, Erik Doernenburg, Graham Tackley, Kent Beck, Kevlin Henney, Laura Bell, Adrian Mouat, Sarah Taraporewalla, Uwe Friedrich, Liz Fong-Jones, Kane Stephens, Gilmanco Staynes, Adam Tornhill, Venkat Subramaniam, Susanne Kaiser, Jan Schaumann, Grady Booch, Pini Reznik, Nicole Forsgren, Jez Humble, Gene Kim, Manuel Pais, Matthew Skelton e

os South Sydney Rabbitohs. Finalmente, gostaria de agradecer aos incríveis leitores do a versão de acesso antecipado do livro, que forneceu um feedback inestimável; eles incluem Felipe de Moraes, Mark Gardner, David Lauzon, Assam Zafar, Michael Bleterman, Nicola Musatti, Eleonora Lester, Felipe de Moraes, Nathan DiMauro, Daniel Lemke, Soner Eker, Ripple Shah, Joel Lim e Calca. Himanshu. E, finalmente, olá para Jason Isaacs.

Escrevi a maior parte deste livro durante todo o ano de 2020 e no primeiro semestre de 2021. Eu usei o Visual Studio Code no macOS durante a maior parte da escrita, embora em raras ocasiões eu também tenha usado o Working Copy no iOS. OmniGraffle foi usado para criar todos os diagramas do livro. AsciiDoc foi usado para formatar o livro e, em geral, foi excelente, com o conjunto de ferramentas Atlas da O'Reilly fazendo a mágica de fazer um livro aparecer do outro lado.

Parte I. Fundação

Capítulo 1. O que são Microsserviços?

Os microsserviços se tornaram uma opção de arquitetura cada vez mais popular em meia década ou mais desde que escrevi a primeira edição deste livro. Eu não posso reivindicar crédito pela subsequente explosão de popularidade, mas pela pressa em ganhar o uso de arquiteturas de microsserviços significa que, embora muitas das ideias eu capturados anteriormente agora são experimentados e testados, novas ideias surgiram no miste ao mesmo tempo em que as práticas anteriores caíram em desuso. Então é mais uma vez, é hora de destilar a essência da arquitetura de microsserviços enquanto destacando os principais conceitos que fazem os microsserviços funcionarem.

Este livro como um todo foi elaborado para fornecer uma visão geral ampla do impacto que os microsserviços abordam vários aspectos da entrega de software. Para começar, este capítulo examinará as principais ideias por trás dos microsserviços, a anterior arte que nos trouxe até aqui, e algumas razões pelas quais essas arquiteturas são usadas tão amplamente.

Visão geral dos microsserviços

Microsserviços são serviços liberáveis de forma independente que são modelados em torno de um domínio comercial. Um serviço encapsula a funcionalidade e a torna acessível a outros serviços por meio de redes - você constrói um mais complexo sistema a partir desses blocos de construção. Um microsserviço pode representar estoque, outro gerenciamento de pedidos e mais um envio, mas juntos eles podem constituir um sistema completo de comércio eletrônico. Os microsserviços são um escolha de arquitetura que se concentra em oferecer muitas opções para resolver o problemas que você possa enfrentar.

Eles são um tipo de arquitetura orientada a serviços, embora seja opinativo sobre como os limites de serviço devem ser traçados, e um em

qual capacidade de implantação independente é fundamental. Eles são agnósticos em tecnologia, o que é uma das vantagens que eles oferecem.

Do lado de fora, um único microsserviço é tratado como uma caixa preta. Ele hospeda funcionalidade comercial em um ou mais endpoints de rede (por exemplo, um fila ou uma API REST, conforme mostrado na Figura 1-1), sobre quaisquer protocolos mais apropriado. Consumidores, sejam eles outros microsserviços ou outros tipos de programas, acesse essa funcionalidade, por meio desses terminais em rede. Detalhes internos da implementação (como a tecnologia em que o serviço foi criado) na forma como os dados são armazenados) estão totalmente ocultos do mundo exterior. Isso significa que as arquiteturas de microsserviços evitam o uso de bancos de dados compartilhados na maioria circunstâncias; em vez disso, cada microsserviço encapsula seu próprio banco de dados quando necessário.

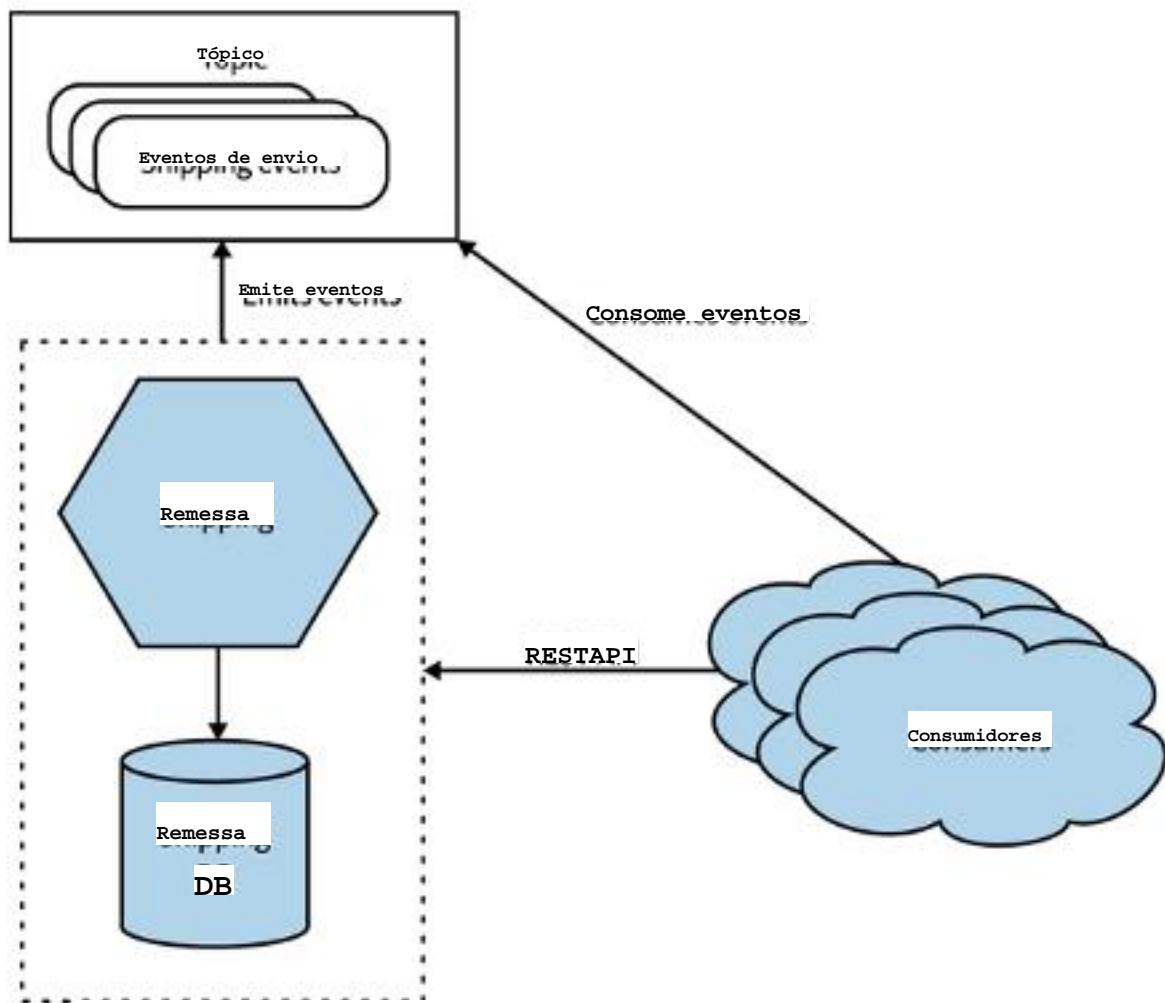


Figura 1-1. Um microsserviço expondo sua funcionalidade em uma API REST e um tópico

Os microserviços adotam o conceito de ocultação de informações. 1 Informação ocultar significa esconder o máximo de informações possível dentro de um componente e expondo o mínimo possível por meio de interfaces externas. Isso permite uma clareza separação entre o que pode mudar facilmente e o que é mais difícil de mudar. A implementação que está oculta de terceiros pode ser alterada livremente, desde que as interfaces de rede que o microserviço expõe não mudança de forma incompatível com versões anteriores. Mudanças dentro de um microserviço limite (conforme mostrado na Figura 1-1) não deve afetar um consumidor upstream, permitindo a liberação independente da funcionalidade. Isso é essencial em permitindo que nossos microserviços sejam trabalhados isoladamente e lançados em demanda. Ter limites de serviço claros e estáveis que não mudam quando o mudanças internas na implementação resultam em sistemas que têm acoplamento mais frouxo e uma coesão mais forte.

Enquanto estamos falando sobre ocultar detalhes internos de implementação, seria omita de mim, para não mencionar o padrão de arquitetura hexagonal. Primeiro detalhado por Alistair Cockburn. 2 Esse padrão descreve a importância de mantendo a implementação interna separada de suas interfaces externas, com a ideia de que talvez você queira interagir com a mesma funcionalidade diferentes tipos de interfaces. Eu desenho meus microserviços como hexágonos em parte para diferenciar os serviços "normais", mas também como uma homenagem a isso peça de arte anterior.

SÃO ARQUITETURAS ORIENTADAS A SERVIÇOS E MICROSSERVIÇOS SÃO COISAS DIFERENTES?

A arquitetura orientada a serviços (SOA) é uma abordagem de design na qual vários serviços colaboram para fornecer um determinado conjunto final de recursos. (Um serviço aqui normalmente significa um sistema operacional completamente separado processo.) A comunicação entre esses serviços ocorre por meio de chamadas em uma rede em vez de chamadas de método dentro de um limite de processo.

A SOA surgiu como uma abordagem para combater os desafios de grandes, aplicações monolíticas. Essa abordagem visa promover a reusabilidade de software; dois ou mais aplicativos de usuário final, por exemplo, poderiam usar os mesmos serviços. O SOA visa facilitar a manutenção ou a reescrita software, pois teoricamente podemos substituir um serviço por outro sem que ninguém saiba, desde que a semântica do serviço não saiba mude demais.

A SOA, em sua essência, é uma ideia sensata. No entanto, apesar de muitos esforços, há a falta de um bom consenso sobre como fazer bem o SOA. Na minha opinião, muito da indústria falhou em analisar o problema de forma holística e suficiente e apresentam uma alternativa convincente à narrativa apresentada por vários fornecedores neste espaço.

Muitos dos problemas causados pela SOA são, na verdade, problemas com coisas como protocolos de comunicação (por exemplo, SOAP), middleware de fornecedores, um falta de orientação sobre a granularidade do serviço ou orientação errada sobre escolhendo lugares para dividir seu sistema. Um cínico pode sugerir que os fornecedores cooptou (e em alguns casos impulsionou) o movimento SOA como forma de vender mais produtos, e esses mesmos produtos, no final, minaram o objetivo do SOA.

Eu vi muitos exemplos de SOA nos quais as equipes estavam se esforçando para tornaram os serviços menores, mas eles ainda tinham tudo acoplado a um banco de dados e tive que implantar tudo junto. Orientado a serviços? Sim. Mas não são microsserviços.

A abordagem de microserviços surgiu do uso no mundo real, adotando nosso melhor compreensão dos sistemas e da arquitetura para fazer bem o SOA. Você deve pensar em microserviços como sendo uma abordagem específica para SOA em da mesma forma que Extreme Programming (XP) ou Scrum é específico abordagem para desenvolvimento ágil de software.

Conceitos-chave de microserviços

Algumas ideias principais devem ser entendidas quando você está explorando microserviços. Dado que alguns aspectos são frequentemente negligenciados, é vital explorá-los conceitos adicionais para ajudar a garantir que você entenda exatamente o que faz os microserviços funcionam.

Implantabilidade independente

A capacidade de implantação independente é a ideia de que podemos fazer uma mudança em um microserviço, implante-o e libere essa alteração para nossos usuários, sem precisar para implantar qualquer outro microserviço. Mais importante ainda, não é apenas o fato de que podemos fazer isso; é que, na verdade, é assim que você gerencia as implantações em seu sistema. É uma disciplina que você adota como sua abordagem de lançamento padrão. Este é um ideia simples que, no entanto, é complexa na execução.

GORJETA

Se você extrair apenas uma coisa deste livro e do conceito de microserviços em geral, deve ser o seguinte: garanta que você adote o conceito de implantação independente de seu microserviços. Adquira o hábito de implantar e liberar mudanças em um único microserviço em produção sem precisar implantar mais nada. A partir disso, muitas boas coisas seguirão.

Para garantir uma implantação independente, precisamos garantir que nossos microserviços são fracamente acoplados: precisamos ser capazes de alterar um serviço sem precisar mudar mais nada. Isso significa que precisamos de algo explícito, bem contratos definidos e estáveis entre serviços. Alguma implementação

as escolhas dificultam isso - o compartilhamento de bancos de dados, por exemplo, é especialmente problemático.

A implantação independente por si só é claramente incrivelmente valiosa. Mas para obter uma implantação independente, há muitas outras coisas que você tem para corrigir isso, por sua vez, tem seus próprios benefícios. Então você também pode ver o foco na capacidade de implantação independente como uma função forçadora - concentrando-se nisso como uma resultado, você obterá vários benefícios auxiliares.

O desejo por serviços fracamente acoplados com interfaces estáveis orienta nosso pensando em como encontramos nossos limites de microserviços em primeiro lugar.

Modelado em torno de um domínio comercial

Técnicas como design orientado por domínio podem permitir que você estruture seu código para melhor representar o domínio do mundo real em que o software opera.³ Com arquiteturas de microserviços, usamos essa mesma ideia para definir nosso serviço limites. Ao modelar serviços em torno de domínios de negócios, podemos fazer isso mais fácil de implantar novas funcionalidades e recombinar microserviços no maneiras diferentes de oferecer novas funcionalidades aos nossos usuários.

A implantação de um recurso que exige alterações em mais de um microserviço é caro. Você precisa coordenar o trabalho em cada serviço (e potencialmente em equipes separadas) e gerencie cuidadosamente a ordem em que as novas versões desses serviços são implantadas. Isso dá muito mais trabalho do que fazer a mesma alteração dentro de um único serviço (ou dentro de um monólito, para isso importa). Portanto, queremos encontrar maneiras de fazer cruzamentos mudanças de serviço tão pouco frequentes quanto possível.

Costumo ver arquiteturas em camadas, conforme tipificado pela arquitetura de três camadas na Figura 1-2. Aqui, cada camada na arquitetura representa uma diferente limite de serviço, com cada limite de serviço baseado em técnicas relacionadas funcionalidade. Se eu precisar fazer uma alteração apenas na camada de apresentação neste exemplo, isso seria bastante eficiente. No entanto, a experiência tem mostrado que mudanças na funcionalidade normalmente abrangem várias camadas nesses tipos de arquiteturas que exigem mudanças na apresentação, nos aplicativos e nos níveis de dados.

Esse problema é exacerbado se a arquitetura for ainda mais estratificada do que a exemplo simples na Figura 1-2; geralmente cada camada é dividida em camadas adicionais.

Ao tornar nossos serviços partes completas da funcionalidade comercial, nós garantir que nossa arquitetura seja organizada para fazer mudanças nos negócios funcionalidade tão eficiente quanto possível. Indiscutivelmente, com microserviços, temos tomou a decisão de priorizar a alta coesão da funcionalidade comercial em alta coesão da funcionalidade técnica.

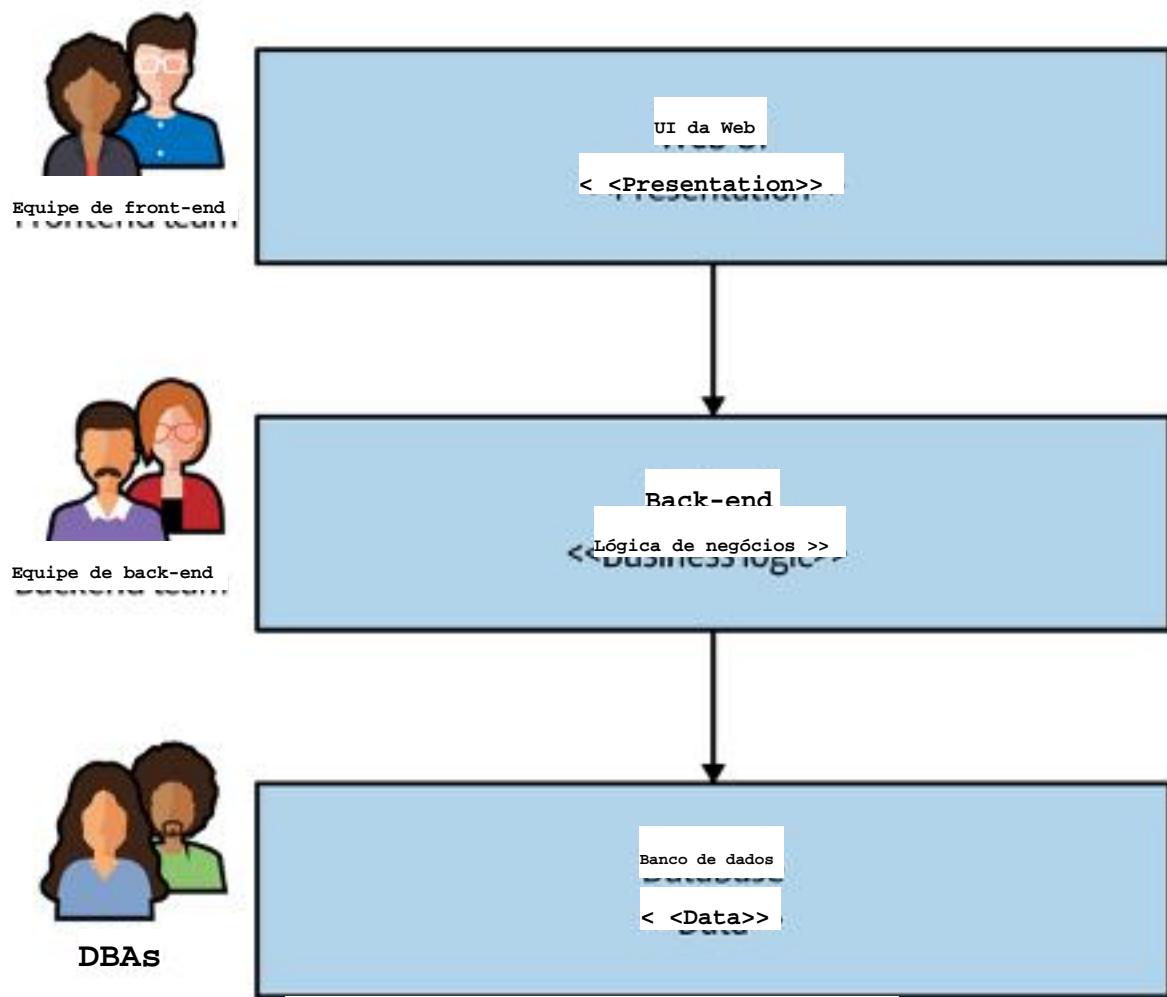


Figura 1-2. Uma arquitetura tradicional de três níveis

Voltaremos à interação do design orientado por domínio e como ele interage com o design organizacional mais adiante neste capítulo.

Possuindo seu próprio estado

Uma das coisas com as quais vejo as pessoas tendo mais dificuldade é a ideia de que os microserviços devem evitar o uso de bancos de dados compartilhados. Se for um microserviço quer acessar dados mantidos por outro microserviço, ele deve ir e perguntar isso ao segundo microserviço para os dados. Isso dá aos microserviços a capacidade de decidir o que é compartilhado e o que está oculto, o que nos permite claramente uma funcionalidade separada que pode mudar livremente (nossa implementação interna) da funcionalidade que queremos alterar com pouca frequência (a externa, o contrato que os consumidores usam).

Se quisermos tornar a implantação independente uma realidade, precisamos garantir que limitamos as alterações incompatíveis com versões anteriores em nossos microserviços. Se nós quebrando a compatibilidade com os consumidores upstream, nós os forçaremos a mudar também. Ter uma delimitação clara entre os detalhes internos da implementação e um contrato externo para um microserviço pode ajudar a reduzir a necessidade de alterações incompatíveis com versões anteriores.

Esconder o estado interno em um microserviço é análogo à prática de encapsulamento em programação orientada a objetos (OO). Encapsulamento de dados em OO sistemas são um exemplo de informações escondidas em ação.

GORJETA

Não compartilhe bancos de dados, a menos que você realmente precise. E mesmo assim, faça tudo o que puder para evite isso. Na minha opinião, compartilhar bancos de dados é uma das piores coisas que você pode fazer se estiver tentando alcançar uma capacidade de implantação independente.

Conforme discutido na seção anterior, queremos pensar em nossos serviços como fatias completas de funcionalidade de negócios que, quando apropriado, encapsular a interface do usuário (UI), a lógica de negócios e os dados. Isso ocorre porque nós querem reduzir o esforço necessário para mudar a funcionalidade relacionada aos negócios. O encapsulamento de dados e comportamento dessa maneira nos dá alta coesão de funcionalidade de negócios. Ao ocultar o banco de dados que sustenta nosso serviço, nós também garantir que reduzamos o acoplamento. Voltamos ao acoplamento e à coesão em Capítulo 2.

Tamanho

"Qual deve ser o tamanho de um microsserviço?" é uma das perguntas mais comuns I ouvir. Considerando que a palavra "micro" está logo no nome, isso vem como nenhuma surpresa. No entanto, quando você aborda o que faz os microsserviços funcionarem como um tipo de arquitetura, o conceito de tamanho é na verdade um dos menores aspectos interessantes.

Como você mede o tamanho? Contando linhas de código? Isso não faz muito sentido para mim. Algo que pode exigir 25 linhas de código em Java poderia ser escrito em 10 linhas de Clojure. Isso não quer dizer que Clojure seja melhor ou pior do que Java; algumas linguagens são simplesmente mais expressivas do que outras.

James Lewis, diretor técnico da Thoughtworks, é conhecido por dizer que "um microsserviço deve ser tão grande quanto minha cabeça." À primeira vista, isso não parecem muito úteis. Afinal, qual é o tamanho exato da cabeça de James? O A justificativa por trás dessa afirmação é que um microsserviço deve ser mantido no tamanho em que pode ser facilmente compreendido O desafio, claro, é que a capacidade de pessoas diferentes de entender algo nem sempre é a mesma, e como tal, você precisará fazer seu próprio julgamento sobre qual tamanho funciona para você. Uma equipe experiente pode gerenciar melhor uma equipe maior base de código do que outra equipe poderia. Então, talvez seja melhor ler A citação de James aqui como "um microsserviço deve ser tão grande quanto sua cabeça".

Acho que o mais próximo que chego de "tamanho" tem algum significado em termos de microsserviços é algo que Chris Richardson, autor de Padrões de microsserviços (Manning Publications), uma vez dito, o objetivo de microsserviços devem ter "a menor interface possível". Isso se alinha com o conceito de informações se escondendo novamente, mas representa uma tentativa de encontre um significado no termo "microsserviços" que não existia inicialmente. Quando o termo foi usado pela primeira vez para definir essas arquiteturas, o foco, pelo menos inicialmente, não estava especificamente no tamanho das interfaces.

Em última análise, o conceito de tamanho é altamente contextual. Fale com uma pessoa que trabalha em um sistema há 15 anos e eles sentirão que seu sistema com 100.000 linhas de código são muito fáceis de entender. Peça a opinião de alguém novato no projeto e achará que é muito grande.

Da mesma forma, pergunte a uma empresa que acabou de iniciar seu microsserviço ...
faça a transição e tenha talvez 10 ou menos microsserviços, e você obterá um
resposta diferente da que você daria de uma empresa de tamanho semelhante para a qual
os microsserviços são a norma há muitos anos e agora tem centenas.

Exorto as pessoas a não se preocuparem com o tamanho. Quando você está começando, é
muito mais importante que você se concentre em duas coisas principais. Primeiro, quantos
microsserviços com os quais você pode lidar? À medida que você tem mais serviços, a complexidade do
seu sistema aumentará e você precisará aprender novas habilidades (e talvez
adotar novas tecnologias) para lidar com isso. Uma mudança para microsserviços será
introduzir novas fontes de complexidade, com todos os desafios que isso pode trazer.
É por esse motivo que sou um forte defensor da migração incremental para um
arquitetura de microsserviços Em segundo lugar, como você define microsserviço?
limites para tirar o máximo proveito deles, sem que tudo se torne um
bagunça terrivelmente acoplada? É muito mais importante focar nesses tópicos.
quando você inicia sua jornada.

Flexibilidade

Outra citação de James Lewis é que "os microsserviços compram opções para você".
Lewis estava sendo deliberado com suas palavras: eles compram opções para você. Eles
têm um custo, e você deve decidir se o custo vale as opções que você escolheu
quero assumir. A flexibilidade resultante em vários eixos -
organizacional, técnico, de escala e robustez - podem ser incrivelmente atraentes.
Não sabemos o que o futuro reserva, então gostaríamos de uma arquitetura que possa
teoricamente, nos ajudam a resolver quaisquer problemas que possamos enfrentar no futuro.
Encontrar um equilíbrio entre manter suas opções abertas e arcar com os custos de
arquiteturas como essa podem ser uma verdadeira arte.

Pense na adoção de microsserviços como menos como apertar um botão e mais como
girando um botão. À medida que você aumenta o botão e tem mais microsserviços, você
têm maior flexibilidade. Mas você provavelmente também aumenta os pontos problemáticos. Isso é
mais uma razão pela qual defendo fortemente a adoção incremental de
microsserviços. Ao aumentar o botão gradualmente, você é mais capaz de avaliar
o impacto à medida que avança e para parar, se necessário.

Alinhamento de arquitetura e organização

A MusicCorp, uma empresa de comércio eletrônico que vende CDs on-line, usa o simples arquitetura de três camadas mostrada anteriormente na Figura 1-2. Decidimos nos mudar. A MusicCorp está entrando e gritando no século 21, e como parte disso, estamos avaliando a arquitetura do sistema existente. Temos uma interface de usuário baseada na web, uma camada lógica de negócios na forma de um back-end monolítico e armazenamento de dados em um banco de dados tradicional. Essas camadas, como é comum, são de propriedade de diferentes equipes. Voltaremos às provações e tribulações da MusicCorp ao longo do livro.

Queremos fazer uma atualização simples em nossa funcionalidade: queremos permitir que nossos clientes devem especificar seu gênero musical favorito. Esta atualização exige que altere a interface do usuário para mostrar a interface de escolha de gênero, o serviço de back-end para permitir o gênero a ser exibido na interface do usuário e para que o valor seja alterado, e o banco de dados para aceitar essa alteração. Essas mudanças precisarão ser gerenciadas por cada equipe e implantada na ordem correta, conforme descrito na Figura 1-3.

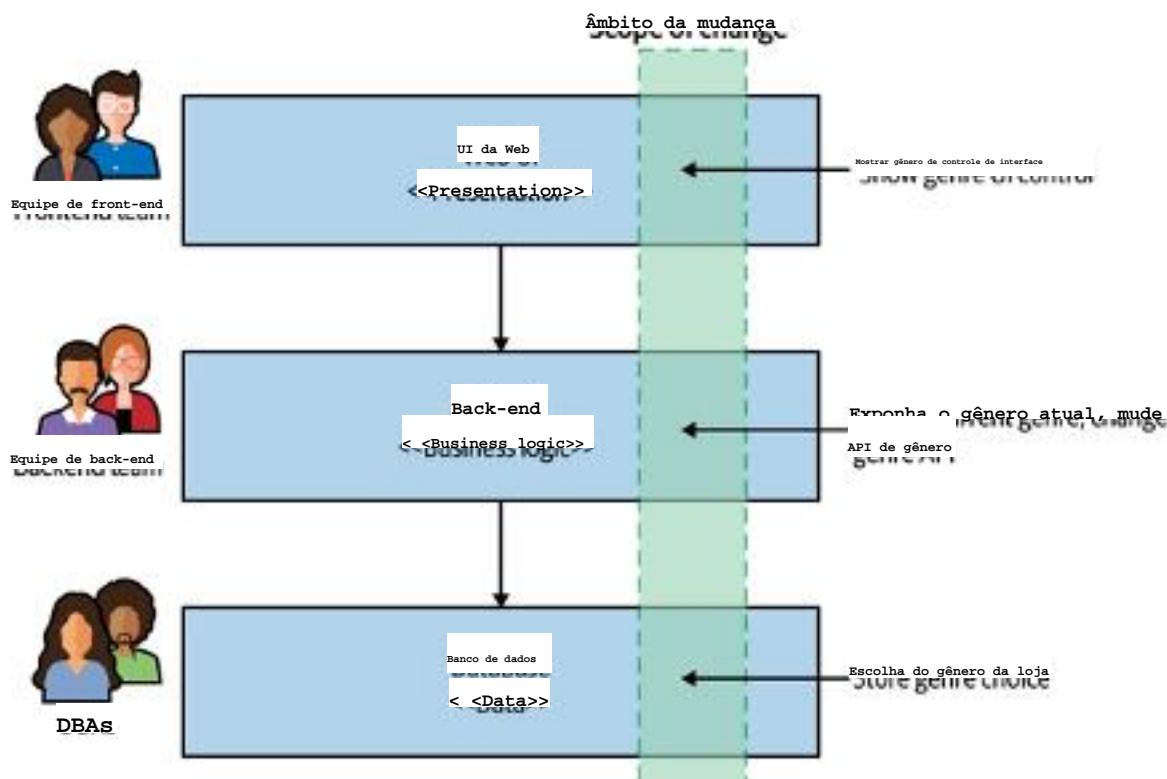


Figura 1-3. Fazer uma mudança em todos os três níveis é mais complicado.

Agora, essa arquitetura não é ruim. Toda a arquitetura acaba sendo otimizada em torno de um conjunto de metas. A arquitetura de três camadas é tão comum, em parte porque é universal - todo mundo já ouviu falar sobre isso. Então, a tendência de escolher uma arquitetura comum que você pode ter visto em outro lugar geralmente é um dos motivos continuamos vendo esse padrão. Mas acho que a maior razão pela qual vemos isso arquitetura de novo e de novo é porque é baseada em como organizamos nossas equipes.

A agora famosa lei de Conway afirma o seguinte:

Organizações que projetam sistemas são obrigadas a produzir desenhos que são cópias das estruturas de comunicação desses organizações.

-Melvin Conway, "Como os comitês inventam?"

A arquitetura de três camadas é um bom exemplo dessa lei em ação. No passado, a principal forma pela qual as organizações de TI agrupavam as pessoas era em termos de competência principal: administradores de banco de dados estavam em equipe com outro banco de dados administradores; os desenvolvedores de Java estavam em uma equipe com outros desenvolvedores de Java; e desenvolvedores front-end (que hoje em dia conhecem coisas exóticas como JavaScript e desenvolvimento de aplicativos móveis nativos) estavam em mais uma equipe. Nós agrupamos pessoas com base em sua competência principal, por isso criamos ativos de TI que podem ser alinhado a essas equipes.

Isso explica por que essa arquitetura é tão comum. Não é ruim; é só otimizado em torno de um conjunto de forças - como tradicionalmente agrupamos as pessoas, em torno da familiaridade. Mas as forças mudaram. Nossas aspirações em torno de nosso software mudou. Agora agrupamos pessoas em equipes poliqualificadas para reduzir as transferências e os silos. Queremos enviar software muito mais rapidamente do que nunca antes. Isso está nos levando a fazer escolhas diferentes sobre a maneira como organizamos nossas equipes, para que as organizemos de acordo com a forma como dividimos nossos sistemas separados.

A maioria das mudanças que somos solicitados a fazer em nosso sistema está relacionada a mudanças em funcionalidade de negócios. Mas na Figura 1-3, nossa funcionalidade comercial está em efeito espalhado por todos os três níveis, aumentando a chance de uma mudança em a funcionalidade cruzar camadas. Esta é uma arquitetura que tem alta coesão.

de tecnologia relacionada, mas baixa coesão da funcionalidade comercial. Se quisermos para facilitar as mudanças, em vez disso, precisamos mudar a forma como nos agrupamos código, escolhendo a coesão da funcionalidade comercial em vez da tecnologia. Cada serviço pode ou não acabar contendo uma mistura dessas três camadas, mas essa é uma preocupação local de implementação de serviços.

Vamos comparar isso com uma arquitetura alternativa potencial, ilustrada em Figura 1-4. Em vez de uma arquitetura e organização em camadas horizontais, em vez disso, dividimos nossa organização e arquitetura na vertical linhas de negócios. Aqui vemos uma equipe dedicada que tem tudo de ponta a ponta responsabilidade por fazer alterações em aspectos do perfil do cliente, que garante que o escopo da mudança neste exemplo seja limitado a uma equipe.

Como implementação, isso poderia ser alcançado por meio de um único microserviço de propriedade da equipe de perfil que expõe uma interface de usuário para permitir que os clientes atualizem suas informações, com o estado do cliente também armazenado dentro deste microservice A escolha de um gênero favorito está associada a um determinado cliente, então essa mudança é muito mais localizada. Na Figura 1-5, também mostramos a lista de gêneros disponíveis que está sendo obtida em um microserviço do Catalog, algo que provavelmente já estaria em vigor. Também vemos um novo Microserviço de recomendação acessando nossas informações de gênero favoritas, algo que poderia ser facilmente seguido em uma versão subsequente.

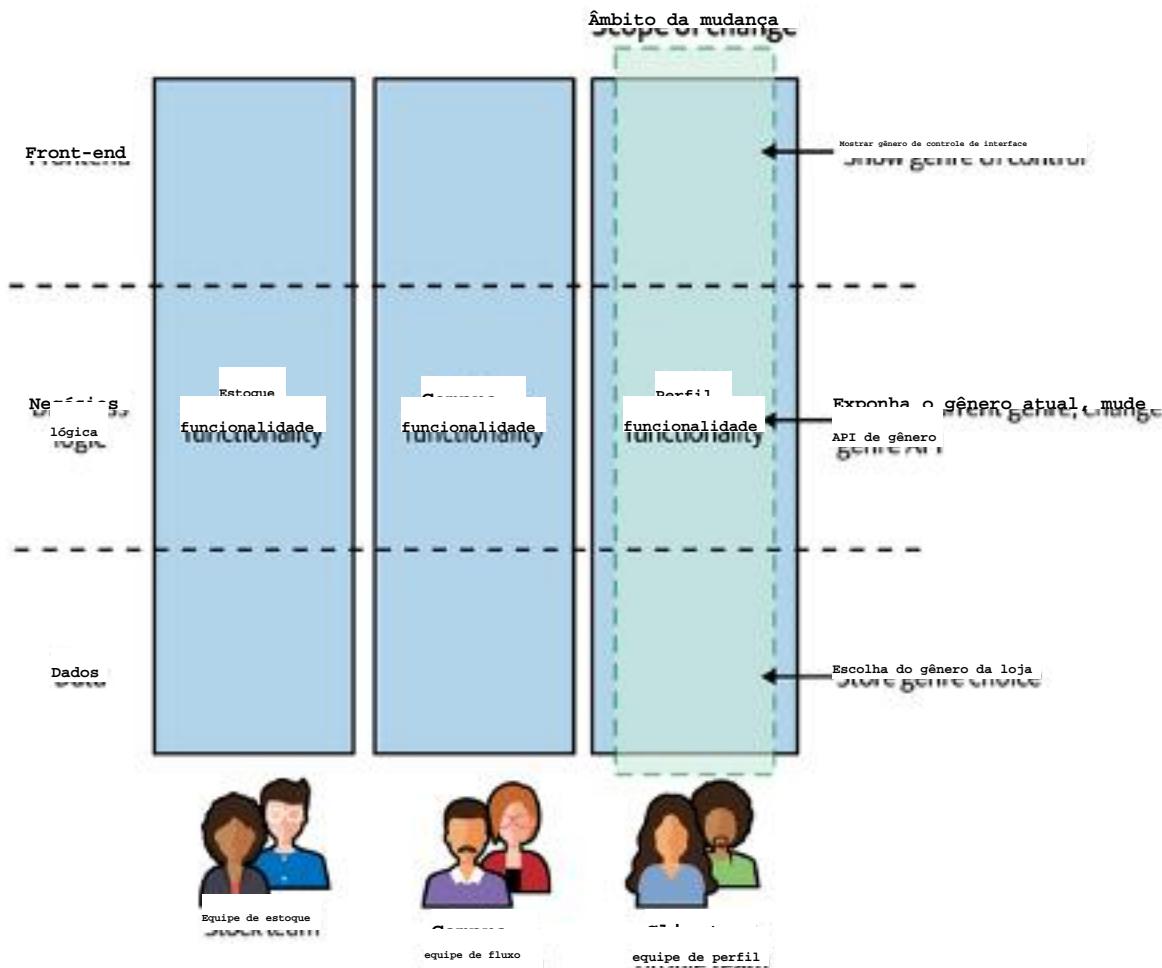


Figura 1-4. A interface do usuário está dividida e pertence a uma equipe que também gerencia o lado do servidor

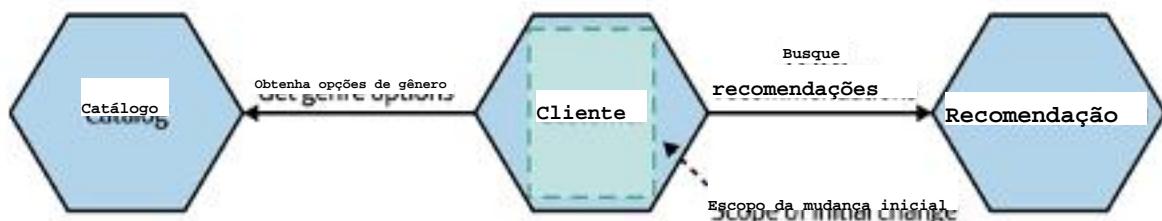


Figura 1-5. Um microsserviço dedicado ao cliente pode facilitar muito a gravação do favorito gênero musical para um cliente

Em tal situação, nosso microsserviço de cliente encapsula uma pequena fatia de cada um dos três níveis - tem um pouco de interface do usuário, um pouco de lógica de aplicação e um pouco de armazenamento de dados. Nosso domínio de negócios se torna a principal força que impulsiona nossa arquitetura do sistema, esperançosamente facilitando a realização de alterações, bem como facilitando o alinhamento de nossas equipes às linhas de negócios dentro da organização.

Muitas vezes, a interface do usuário não é fornecida diretamente pelo microsserviço, mas mesmo que seja. No caso, esperaríamos que a parte da interface do usuário relacionada a essa funcionalidade fosse ainda pertencem à equipe de perfil do cliente, conforme indica a Figura 1-4. Isso o conceito de uma equipe que possui uma fatia de funcionalidade voltada para o usuário de ponta a ponta é ganhando tração. O livro Team Topologies⁴ apresenta a ideia de um stream-equipe alinhada, que incorpora esse conceito:

Uma equipe alinhada ao fluxo é uma equipe alinhada a um único e valioso fluxo de trabalho. [A] equipe tem o poder de criar e entregar um cliente ou usuário valorize da forma mais rápida, segura e independente possível, sem exigindo transferências para outras equipes para realizar partes do trabalho.

As equipes mostradas na Figura 1-4 seriam equipes alinhadas ao fluxo, um conceito exploraremos com mais profundidade nos capítulos 14 e 15, incluindo como esses tipos de estruturas organizacionais funcionam na prática e como elas se alinham com microsserviços

UMA NOTA SOBRE EMPRESAS “FALSAS”

Ao longo do livro, em diferentes estágios, conhceremos a MusicCorp, FinanceCo, FoodCo, AdvertCo e PaymentCo.

FoodCo, AdvertCo e PaymentCo são empresas reais cujos nomes I mudaram por motivos de confidencialidade. Além disso, ao compartilhar informações sobre essas empresas, muitas vezes omiti certos detalhes para forneca mais clareza. O mundo real costuma ser confuso. Eu sempre me esforcei, no entanto, para remover apenas detalhes estranhos que não seriam úteis, enquanto ainda garantindo que a realidade subjacente da situação permaneca.

A MusicCorp, por outro lado, é uma empresa falsa composta de muitas organizações com as quais trabalhei. As histórias que eu compartilho MusicCorp são reflexos de coisas reais que eu vi, mas nem todas aconteceu com a mesma empresa!

O monólito

Falamos sobre microsserviços, mas na maioria das vezes os microsserviços são discutida como uma abordagem arquitetônica que é uma alternativa ao monolítico. Para distinguir mais claramente a arquitetura de microsserviços, e para ajudar você a entender melhor se vale a pena considerar os microsserviços, eu também deve discutir o que exatamente quero dizer com monólitos.

Quando falo sobre monólitos ao longo deste livro, estou me referindo principalmente a uma unidade de implantação. Quando todas as funcionalidades de um sistema devem ser implantadas juntas, eu o considero um monólito. Indiscutivelmente, várias arquiteturas se encaixam nisso definção, mas vou discutir aquelas que vejo com mais frequência: o processo único monólito, o monólito modular e o monólito distribuído.

O monólito de processo único

O exemplo mais comum que vem à mente ao discutir monólitos é um sistema no qual todo o código é implantado como um único processo, como em Figura 1-6. Você pode ter várias instâncias desse processo para maior robustez ou motivos de escalabilidade, mas fundamentalmente todo o código é compactado em um único processo. Na realidade, esses sistemas de processo único podem ser distribuídos de forma simples sistemas por si só porque quase sempre acabam lendo dados de ou armazenando dados em um banco de dados, ou apresentando informações para a web ou aplicativos móveis.

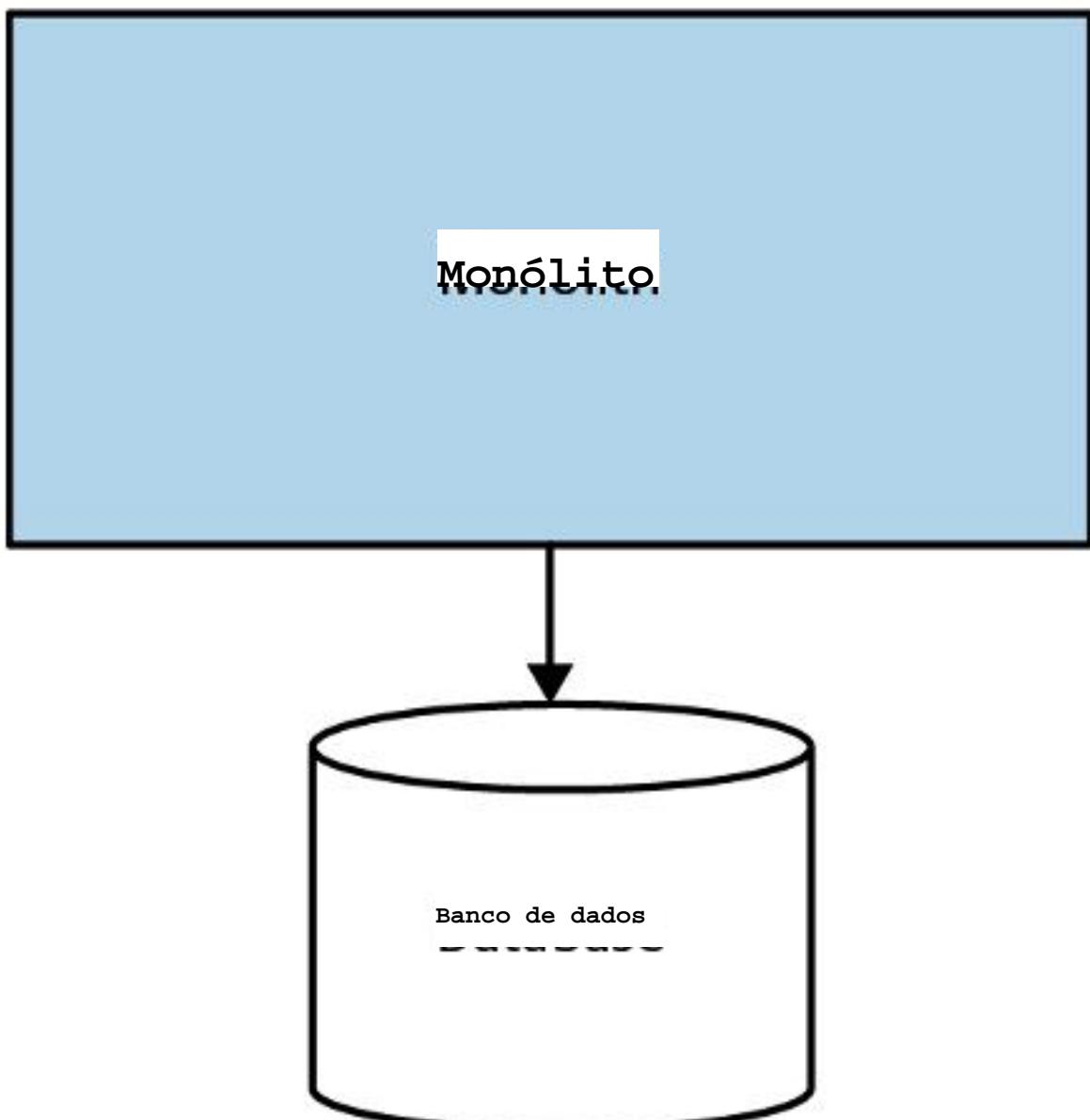


Figura 1-6. Em um monólito de processo único, todo o código é empacotado em um único processo.

Embora isso se encaixe na compreensão da maioria das pessoas sobre um monólito clássico, a maioria dos sistemas que encontro são um pouco mais complexos do que isso. Você pode ter dois ou mais monólitos que estão fortemente acoplados uns aos outros, potencialmente com alguns softwares de fornecedores na mistura.

Uma implantação monolítica clássica de processo único pode fazer sentido para muitas organizações. David Heinemeier Hansson, o criador do Ruby on Rails, tem argumentado efetivamente que essa arquitetura faz sentido para menores

~~Organizations. Even as the organization grows, however, the monolith can potentially grow with it, which leads to the monolithic modular.~~

O monólito modular

~~Como um subconjunto do monólito de processo único, o monólito modular é uma variação na qual o processo único consiste em módulos separados. Cada módulo pode ser trabalhado de forma independente, mas todos ainda precisam ser combinados juntos para implantação, conforme mostrado na Figura 1-7. O conceito de quebrar software em módulos não é novidade; o software modular tem suas raízes no trabalho feito em torno da programação estruturada na década de 1970, e ainda mais atrás isso. No entanto, essa é uma abordagem que ainda não vejo o suficiente as organizações interagem adequadamente com.~~

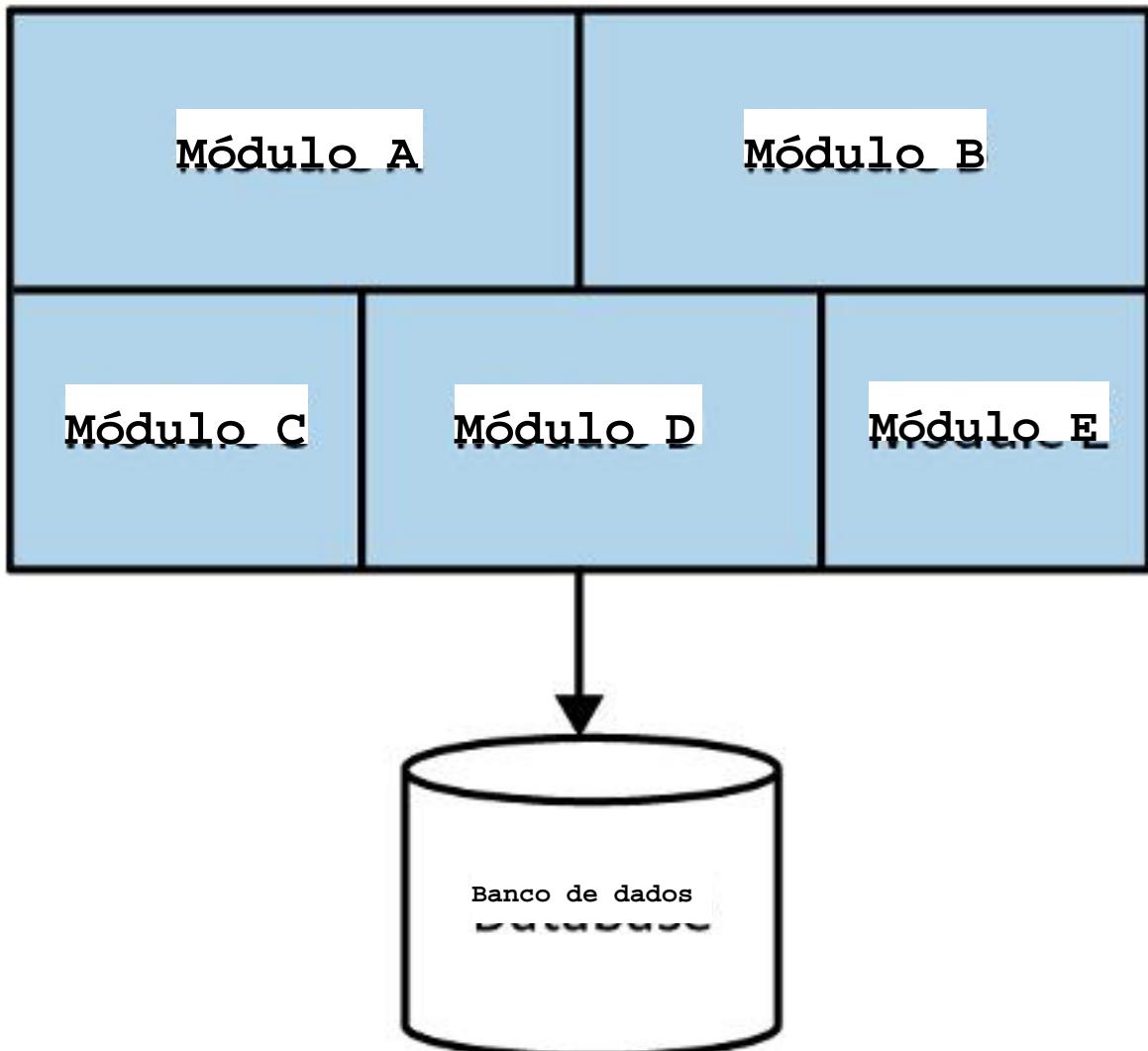


Figura 1-7. Em um monólito modular, o código dentro do processo é dividido em módulos,

Para muitas organizações, o monólito modular pode ser uma excelente escolha. Se

os limites do módulo estão bem definidos, podem permitir um alto grau de

trabalho paralelo, evitando os desafios dos mais distribuídos

arquitetura de microsserviços por ter uma topologia de implantação muito mais simples.

O Shopify é um ótimo exemplo de uma organização que usou essa técnica como

uma alternativa à decomposição de microsserviços, e parece funcionar realmente

bom para essa empresa.³

Um dos desafios de um monólito modular é que o banco de dados tende a faltar a decomposição que encontramos no nível do código, levando a desafios significativos se você quiser desmontar o monólito no futuro. Eu vi algumas equipes tentar levar a ideia do monólito modular ainda mais com o

banco de dados decomposto nas mesmas linhas dos módulos, conforme mostrado em Figura 1-8.

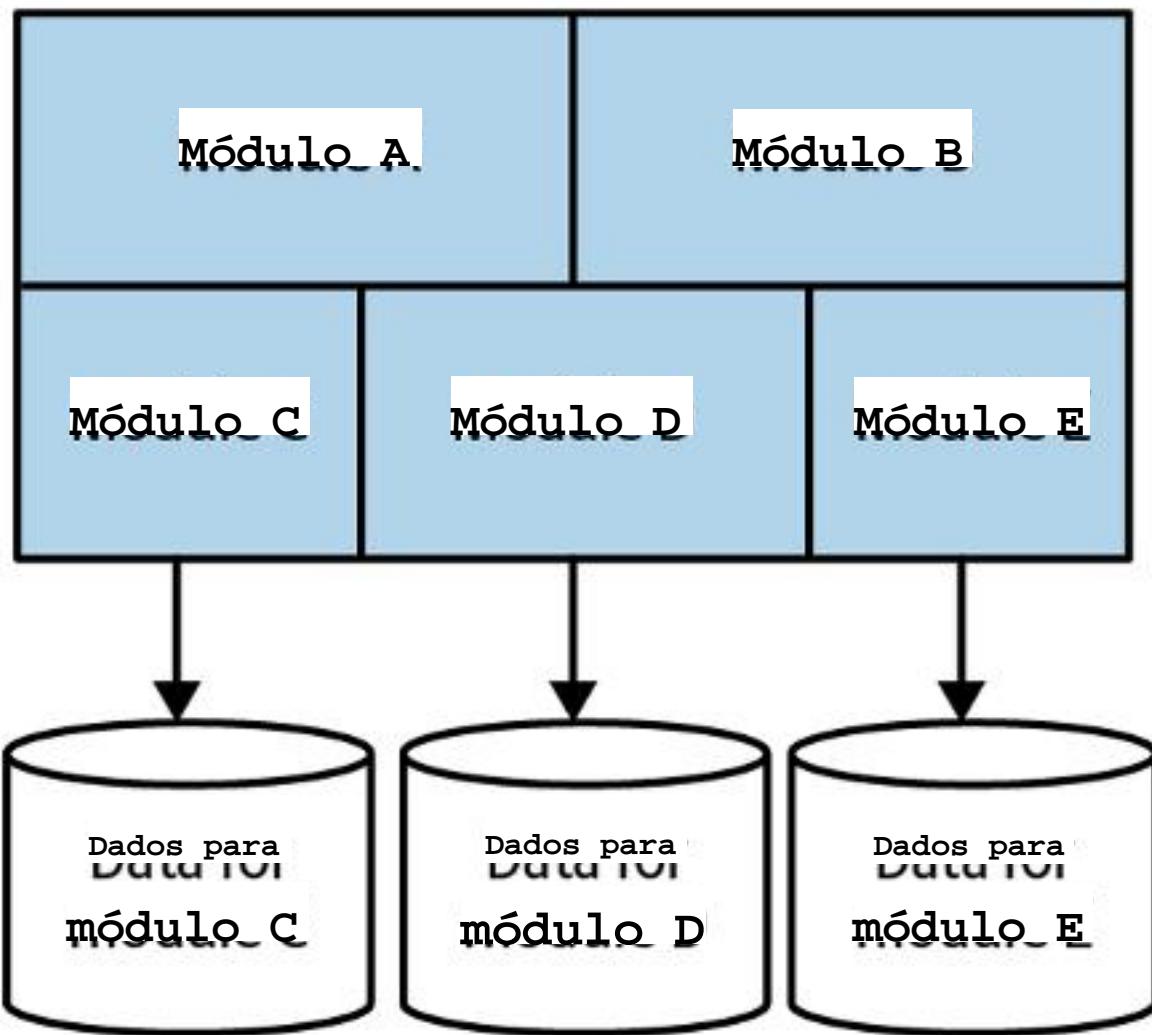


Figura 1-8. Um monólito modular com um banco de dados decomposto

O monólito distribuído

mesmo saber que existe pode tornar seu próprio computador inutilizável.
Even when you know it exists, it can render your own computer unusable.

- Leslie Lamport

Um monólito distribuído é um sistema que consiste em vários serviços, mas para seja qual for o motivo, todo o sistema deve ser implantado em conjunto. Um distribuído monólito pode muito bem atender à definição de SOA, mas, muitas vezes, falha para cumprir as promessas da SOA. Na minha experiência, um monólito distribuído

tem todas as desvantagens de um sistema distribuído e as desvantagens de um monólito de processo único, sem ter as vantagens suficientes de nenhum deles. Encontrar vários monólitos distribuídos em meu trabalho tem, em grande parte, influenciou meu próprio interesse pela arquitetura de microserviços.

Os monólitos distribuídos geralmente surgem em um ambiente no qual não foco suficiente foi colocado em conceitos como ocultação de informações e coesão de funcionalidade de negócios. Em vez disso, arquiteturas altamente acopladas causam alterações em atravessam os limites do serviço e mudam aparentemente inocentes que parecem ter um escopo local para quebrar outras partes do sistema.

Monólitos e contenção de entrega

À medida que mais e mais pessoas trabalham no mesmo lugar, elas entram umas nas outras forma - por exemplo, desenvolvedores diferentes querendo mudar a mesma parte do código, equipes diferentes querendo implantar a funcionalidade ao vivo em momentos diferentes (ou para atrasar as implantações) e confusão sobre quem é dono do quê e quem faz linhas de propriedade. Refiro-me a esse problema como contenção de entrega.

Ter um monólito não significa que você definitivamente enfrentará os desafios de contenção de entrega, assim como ter uma arquitetura de microserviços significa que você nunca enfrentará o problema. Mas uma arquitetura de microserviços faz fornecem limites mais concretos em torno dos quais as linhas de propriedade podem ser desenhado em um sistema, oferecendo muito mais flexibilidade quando se trata de reduzindo esse problema.

Vantagens dos monólitos

Alguns monólitos, como os monólitos modulares ou de processo único, têm um uma série de vantagens também. Sua topologia de implantação muito mais simples pode evite muitas das armadilhas associadas aos sistemas distribuídos. Isso pode resultam em fluxos de trabalho e monitoramento de desenvolvedores muito mais simples, solução de problemas, e atividades como testes de ponta a ponta podem ser muito importantes simplificado também.

Os monólitos também podem simplificar a reutilização de código dentro do próprio monólito. Se quisermos para reutilizar o código em um sistema distribuído, precisamos decidir se deseja copiar código, dividir bibliotecas ou inserir a funcionalidade compartilhada em um serviço. Com um monólito, nossas escolhas são muito mais simples, e muitas pessoas gosto dessa simplicidade - todo o código está lá; basta usá-lo!

Infelizmente, as pessoas passaram a ver o monólito como algo a ser evitado como algo inherentemente problemático. Eu conheci várias pessoas por a quem o termo monólito é sinônimo de legado. Isso é um problema. UMA arquitetura monolítica é uma escolha, e ainda assim válida. Eu iria mais longe e digo que, na minha opinião, é a escolha padrão sensata como arquitetura estilo. Em outras palavras, estou procurando um motivo para ser convencido a usar microserviços, em vez de procurar um motivo para não usá-los.

Se cairmos na armadilha de minar sistematicamente o monólito como um produto viável opção para entregar nosso software, corremos o risco de não fazer o que é certo por nós mesmos ou pelos usuários do nosso software.

Tecnologia habilitadora

Como mencionei anteriormente, não acho que você precise adotar muitas novidades tecnologia quando você começa a usar microserviços. Na verdade, isso pode ser contraproducente. Em vez disso, à medida que você aumenta sua arquitetura de microserviços, você deve estar constantemente procurando por problemas causados por seu sistema distribuído e, em seguida, para tecnologia que possa ajudar.

Dito isso, a tecnologia desempenhou um papel importante na adoção de microserviços como conceito. Entendendo as ferramentas que estão disponíveis para ajudar tirar o máximo proveito dessa arquitetura será uma parte fundamental da criação qualquer implementação de microserviços é um sucesso. Na verdade, eu iria tão longe quanto dizem que os microserviços exigem uma compreensão da tecnologia de suporte a tal ponto que as distinções anteriores entre lógico e físico a arquitetura pode ser problemática - se você estiver envolvido em ajudar a moldar um arquitetura de microserviços, você precisará de uma ampla compreensão desses dois mundos.

Exploraremos muito dessa tecnologia em detalhes nos capítulos subsequentes, mas antes disso, vamos apresentar brevemente algumas das tecnologias facilitadoras que pode ajudá-lo se você decidir usar microsserviços.

Agregação de registros e rastreamento distribuído

Com o aumento do número de processos que você gerencia, pode ser difícil para entender como seu sistema está se comportando em um ambiente de produção. Isso pode por sua vez, tornar a solução de problemas muito mais difícil. Estaremos explorando essas ideias mais aprofundadas no Capítulo 10, mas, no mínimo, eu fortemente defendem a implementação de um sistema de agregação de registros como pré-requisito para adotando uma arquitetura de microsserviços.

GORJETA

Tenha cuidado ao usar muitas tecnologias novas ao começar com microsserviços Dito isso, uma ferramenta de agregação de registros é tão essencial que você deve considerá-la pré-requisito para a adoção de microsserviços.

Esses sistemas permitem que você colete e agregue registros de todos os seus serviços, fornecendo a você um local central a partir do qual os registros podem ser analisados e até fez parte de um mecanismo de alerta ativo. Muitas opções neste espaço atendem a inúmeras situações. Sou um grande fã do Humio por vários motivos, mas os serviços simples de registro fornecidos pelos principais fornecedores de nuvem pública podem seja bom o suficiente para você começar.

Você pode tornar essas ferramentas de agregação de registros ainda mais úteis implementando IDs de correlação, em que uma única ID é usada para um conjunto relacionado de chamadas de serviço -por exemplo, a cadeia de chamadas que pode ser acionada devido ao usuário interação. Ao registrar esse ID como parte de cada entrada de registro, isolando os registros associado a um determinado fluxo de chamadas se torna muito mais fácil, o que, por sua vez torna a solução de problemas muito mais fácil.

À medida que seu sistema cresce em complexidade, torna-se essencial considerar as ferramentas que permitem que você explore melhor o que seu sistema está fazendo, fornecendo

capacidade de analisar rastreamentos em vários serviços, detectar gargalos e perguntar perguntas do seu sistema que você não sabia que costaria de fazer no primeiro lugar. As ferramentas de código aberto podem fornecer alguns desses recursos. Um exemplo é o Jaeger, que se concentra no lado do rastreamento distribuído do equação.

Mas produtos como Lightstep e Honeycomb (mostrados na Figura 1-9) aceitam esses ideias adicionais. Eles representam uma nova geração de ferramentas que vão além abordagens tradicionais de monitoramento, facilitando muito a exploração do estado do seu sistema em execução. Talvez você já tenha ferramentas mais convencionais em lugar, mas você realmente deve analisar os recursos que esses produtos oferecem. Eles foram construídos do zero para resolver os tipos de problemas que operadores de arquiteturas de microserviços precisam lidar com isso.

Consulta em 31/05 às 15:35 PM Trace e6ee35b206elc9e5

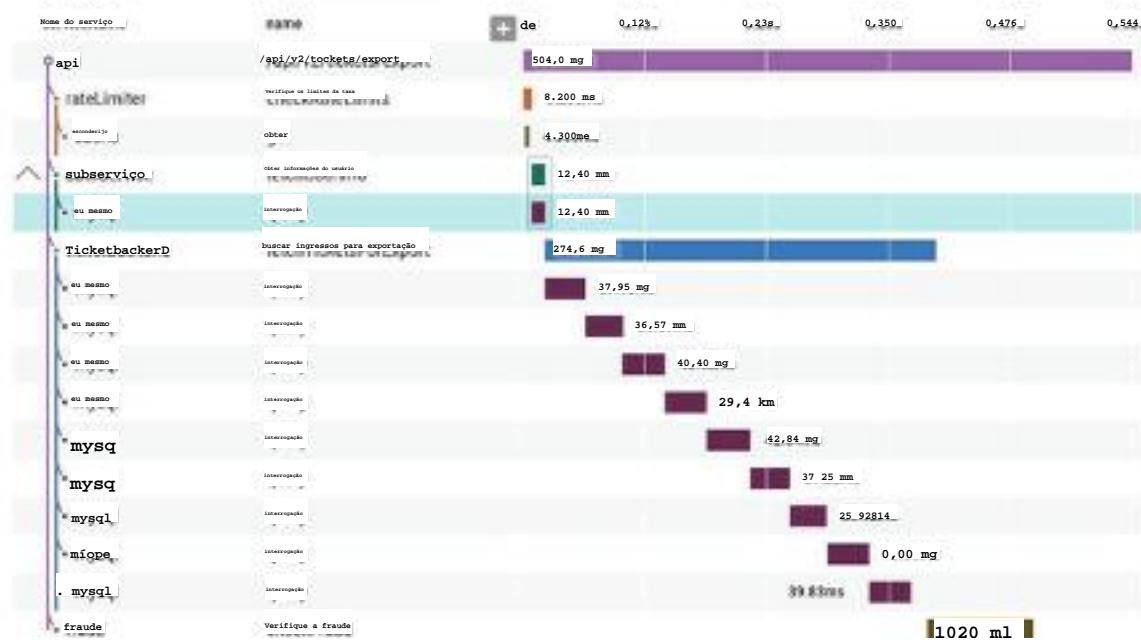


Figura 1-9: Um traco distribuído mostrado no Honeycomb, permitindo que você identifique onde o tempo está passando gasto em operações que podem abranger vários microserviços

Contêineres e Kubernetes

Idealmente, você deseja executar cada instância de microserviço de forma isolada. Isso garante que problemas em um microserviço não podem afetar outro microserviço - para exemplo, devorando toda a CPU. A virtualização é uma forma de criar

ambientes de execução isolados em hardware existente, mas normais. As técnicas de virtualização podem ser bastante pesadas quando consideramos o tamanho do nosso microsserviços. Os contêineres, por outro lado, fornecem muito mais maneira leve de provisionar execução isolada para instâncias de serviço, resultando em tempos de rotação mais rápidos para novas instâncias de contêiner, além de ser muito mais econômico para muitas arquiteturas.

Depois de começar a brincar com contêineres, você também perceberá que precisa de algo que permita que você gerencie esses contêineres em muitas máquinas subjacentes. Plataformas de orquestração de contêineres como o Kubernetes exatamente isso, permitindo que você distribua instâncias de contêiner de forma a fornecer a robustez e a produtividade de que seu serviço precisa, ao mesmo tempo em que permite você para fazer uso eficiente das máquinas subjacentes. No Capítulo 8, vamos explorar os conceitos de isolamento operacional, contêineres e Kubernetes.

Não sinta a necessidade de se apressar em adotar o Kubernetes, ou mesmo contêineres para isso importam. Eles absolutamente oferecem vantagens significativas em relação aos mais tradicionais técnicas de implantação, mas sua adoção é difícil de justificar se você tiver apenas alguns microsserviços. Após o início da sobrecarga de gerenciamento da implantação para se tornar uma dor de cabeça significativa, comece a considerar a conteinerização de seu serviço e uso do Kubernetes. Mas se você acabar fazendo isso, faça o seu melhor garantir que outra pessoa esteja executando o cluster Kubernetes para você, talvez fazendo uso de um serviço gerenciado em um provedor de nuvem pública. Executar seu próprio cluster Kubernetes pode ser uma quantidade significativa de trabalho!

Streaming

Embora com os microsserviços estejamos nos afastando do monolítico bancos de dados, ainda precisamos encontrar maneiras de compartilhar dados entre microsserviços. Isso está acontecendo ao mesmo tempo em que as organizações desejam se mudar, longe das operações de relatórios em lote e buscando mais feedback em tempo real, permitindo que eles reajam mais rapidamente. Produtos que permitem o fácil streaming e processamento de grandes volumes de dados portanto, se tornam populares entre as pessoas que usam arquiteturas de microsserviços.

Para muitas pessoas, o Apache Kafka se tornou a escolha de fato para streaming de dados em um ambiente de microsserviços, e por um bom motivo. Capacidades como permanência de mensagens, compactação e a capacidade de escalar para lidar com grandes volumes de mensagens pode ser extremamente útil. Kafka começou a adicionar recursos de processamento de fluxo na forma de KSQLDB, mas você também pode usá-lo com soluções dedicadas de processamento de fluxo, como Apache Flink. O Debezium é uma ferramenta de código aberto desenvolvida para ajudar a transmitir dados de fontes de dados existentes em Kafka, ajudando a garantir que o tradicional as fontes de dados podem se tornar parte de uma arquitetura baseada em fluxo. No capítulo 4 veremos como a tecnologia de streaming pode desempenhar um papel no microsserviço integração.

Nuvem pública e sem servidor

Provedores de nuvem pública ou, mais especificamente, os três principais provedores- Google Cloud, Microsoft Azure e Amazon Web Services (AWS) oferecem uma grande variedade de serviços gerenciados e opções de implantação para gerenciar seu aplicação. À medida que sua arquitetura de microsserviços cresce, mais e mais trabalho serão empurrados para o espaço operacional. Os provedores de nuvem pública oferecem uma host de serviços gerenciados, de instâncias de banco de dados gerenciadas ou Kubernetes clusters para corretores de mensagens ou sistemas de arquivos distribuídos. Ao fazer uso de esses serviços gerenciados, você está transferindo uma grande quantidade desse trabalho para um terceiro que é indiscutivelmente mais capaz de lidar com essas tarefas.

De particular interesse entre as ofertas de nuvem pública são os produtos que sente-se sob a bandeira da serverless. Esses produtos escondem o subjacente máquinas, permitindo que você trabalhe em um nível mais alto de abstração. Exemplos de os produtos sem servidor incluem corretores de mensagens, soluções de armazenamento e bancos de dados. As plataformas Function as a Service (FaaS) são de especial interesse porque eles fornecem uma boa abstração sobre a implantação do código. Em vez de se preocupar com quantos servidores você precisa para executar seu serviço, você acabou de implantar seu código e deixar a plataforma subjacente lidar com a inicialização instâncias do seu código sob demanda. Examinaremos a versão sem servidor com mais detalhes em Capítulo 8.

Vantagens dos microsserviços

As vantagens dos microsserviços são muitas e variadas. Muitos desses os benefícios podem ser atribuídos a qualquer sistema distribuído. Microsserviços, no entanto, tendem a alcançar esses benefícios em maior grau, principalmente porque eles assumem uma postura mais opinativa na forma como são os limites de serviço definido. Combinando os conceitos de ocultação de informações e orientado por domínio design com o poder dos sistemas distribuídos, os microsserviços podem ajudar a oferecer ganhos significativos em relação a outras formas de arquiteturas distribuídas.

Heterogeneidade tecnológica

Com um sistema composto por vários microsserviços colaborativos, podemos decidir usar tecnologias diferentes dentro de cada um. Isso nos permite escolher a ferramenta certa para cada trabalho, em vez de precisar selecionar uma mais padronizada, abordagem única que muitas vezes acaba sendo a menos comum denominador.

Se uma parte do nosso sistema precisar melhorar seu desempenho, podemos decidir para usar uma pilha de tecnologia diferente que seja mais capaz de atingir o necessário níveis de desempenho. Também podemos decidir que a forma como armazenamos nossos dados precisa mudar para diferentes partes do nosso sistema. Por exemplo, para uma rede social rede, podemos armazenar as interações de nossos usuários em um banco de dados orientado a gráficos para refletir a natureza altamente interconectada de um gráfico social, mas talvez o as postagens feitas pelos usuários podem ser armazenadas em um armazenamento de dados orientado a documentos, dando origem a uma arquitetura heterogênea como a mostrada na Figura 1-10.

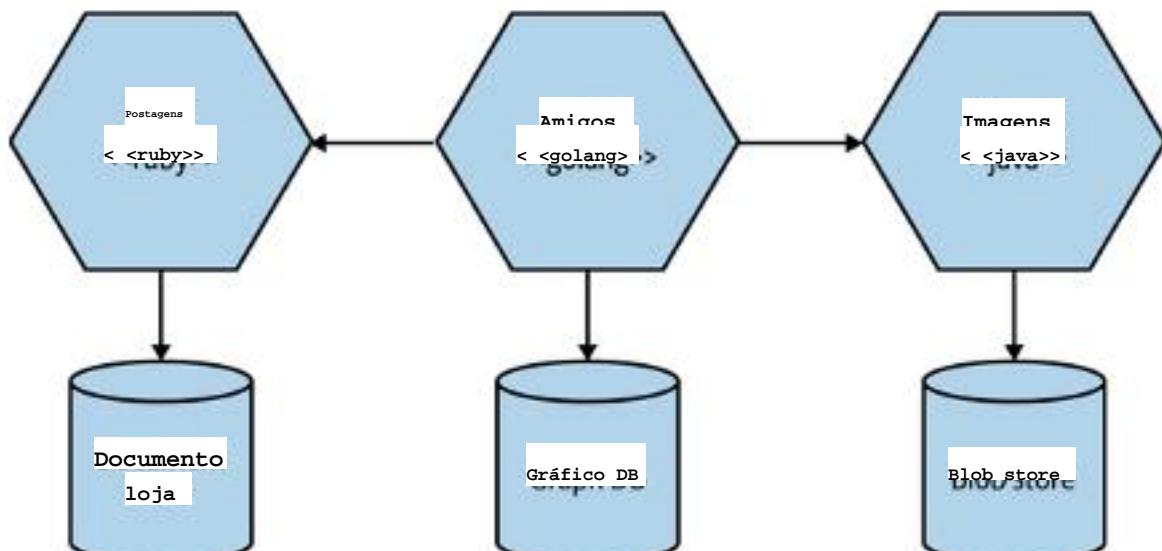


Figura 1-10. Os microserviços podem permitir que você adote mais facilmente diferentes tecnologias

Com microserviços, também podemos adotar tecnologias e para entender como os novos avanços podem nos ajudar. Um dos maiores barreiras para experimentar e adotar uma nova tecnologia são os riscos associados com isso. Com um aplicativo monolítico, se eu quiser experimentar uma nova programação linguagem, banco de dados ou estrutura, qualquer alteração afetará grande parte do meu sistema. Com um sistema que consiste em vários serviços, tenho vários lugares para experimentar uma nova peça de tecnologia. Eu posso escolher um microserviço com talvez o menor risco e uso a tecnologia lá, sabendo que posso limitar qualquer potencial impacto negativo. Muitas organizações consideram essa capacidade de mais absorver rapidamente novas tecnologias para ser uma vantagem real.

A adoção de várias tecnologias não vem sem sobrecarga, é claro. Algumas organizações optam por impor algumas restrições às escolhas de idioma. Netflix e Twitter, por exemplo, usam principalmente a Java Virtual Machine (JVM) como uma plataforma porque essas empresas têm uma compreensão muito boa da confiabilidade e desempenho desse sistema. Eles também desenvolvem bibliotecas e ferramentas para a JVM que tornam a operação em grande escala muito mais fácil, mas a confiança em bibliotecas específicas da JVM torna as coisas mais difíceis para pessoas não baseadas em Java servicos ou clientes. Mas nem o Twitter nem a Netflix usam apenas uma tecnologia pilha para todos os trabalhos.

O fato de a implementação interna da tecnologia estar oculta dos consumidores também pode facilitar a atualização de tecnologias. Todo o seu microserviço

a arquitetura pode ser baseada no Spring Boot, por exemplo, mas você poderia altere a versão da JVM ou as versões da estrutura para apenas um microsserviço, facilitando o gerenciamento do risco de atualizações.

Robustez

Um conceito-chave para melhorar a robustez de sua aplicação é o antepara. Um componente de um sistema pode falhar, mas desde que essa falha ocorra não cai em cascata, você pode isolar o problema e o resto do sistema pode continue trabalhando. Os limites de serviço se tornam seus anteparos óbvios. Em um serviço monolítico, se o serviço falhar, tudo para de funcionar. Com um sistema monolítico, podemos rodar em várias máquinas para reduzir nossa chance de falha, mas com microsserviços, podemos criar sistemas que lidam com o total falha de alguns dos serviços constituintes e degradação da funcionalidade em conformidade.

No entanto, precisamos ter cuidado. Para garantir que nossos sistemas de microsserviços podemos abraçar adequadamente essa robustez aprimorada, precisamos entender o novas fontes de falhas com as quais os sistemas distribuídos precisam lidar. Redes podem e falharão, assim como as máquinas. Precisamos saber como lidar com isso falhas e o impacto (se houver) que essas falhas terão sobre os usuários finais do nosso software. Certamente trabalhei com equipes que acabaram com um sistema menos robusto após a migração para microsserviços devido ao fato de não levando essas preocupações a sério o suficiente.

Dimensionamento

Com um serviço grande e monolítico, precisamos escalar tudo em conjunto. Talvez uma pequena parte do nosso sistema geral tenha desempenho limitado, mas se esse comportamento estiver bloqueado em um aplicativo monolítico gigante, precisamos manuseie o dimensionamento de tudo como uma peça. Com serviços menores, podemos escalar apenas aqueles serviços que precisam ser escalados, o que nos permite executar outras partes do sistema em hardware menor e menos potente, conforme ilustrado na Figura 1-11.

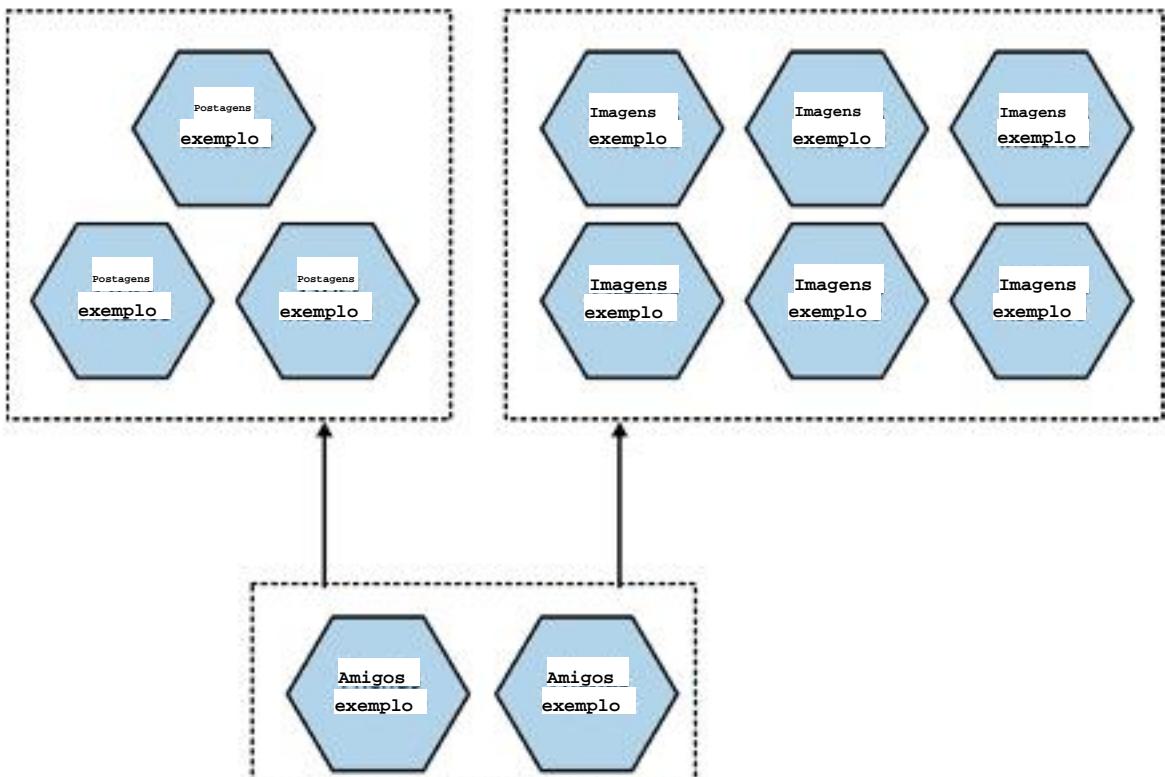


Figura 1-11. Você pode direcionar o dimensionamento apenas para os microsserviços que precisam dele.

A Gilt, uma varejista de moda on-line, adotou microsserviços exatamente por esse motivo. Tendo começado em 2007 com um aplicativo Rails monolítico, em 2009 Gilt's sistema foi incapaz de lidar com a carga que estava sendo colocada sobre ele. Ao se separar partes centrais de seu sistema, a Gilt foi mais capaz de lidar com seus picos de tráfego, e hoje ele tem mais de 450 microsserviços, cada um executado em vários máquinas separadas.

Ao adotar sistemas de provisionamento sob demanda, como os fornecidos pela AWS, podemos até mesmo aplicar esse escalonamento sob demanda para as peças que precisam dele. Isso nos permite controlar nossos custos com mais eficiência. Não é sempre que uma abordagem arquitetônica pode ser tão estreitamente correlacionada a uma abordagem quase imediata economia de custos.

Em última análise, podemos escalar nossos aplicativos de várias maneiras, e os microsserviços podem ser uma parte efetiva disso. Analisaremos a escala de microsserviços com mais detalhes no Capítulo 13.

Facilidade de implantação

Uma mudança de uma linha para um aplicativo monolítico de um milhão de linhas requer todo o aplicativo a ser implantado para liberar a alteração. Isso poderia ser um implantação de grande impacto e alto risco. Na prática, implantações como essas acabam acontecendo com pouca frequência por causa de um medo compreensível. Infelizmente, isso significa que nossas mudanças continuam aumentando entre os lançamentos, até o A nova versão do nosso aplicativo que está entrando em produção tem muitas mudanças. E quanto maior o delta entre os lançamentos, maior o risco que teremos algo errado!

Com microserviços, podemos fazer uma alteração em um único serviço e implantá-lo independentemente do resto do sistema. Isso nos permite obter nosso código implantado mais rapidamente. Se ocorrer um problema, ele pode ser rapidamente isolado para um serviço individual, facilitando a reversão rápida. Isso também significa que podemos disponibilizar nossa nova funcionalidade aos clientes mais rapidamente. Este é um dos principais motivos pelos quais organizações como Amazon e Netflix os usam arquiteturas - para garantir que elas removam o maior número possível de impedimentos para lançar o software.

Alinhamento organizacional

Muitos de nós já enfrentamos os problemas associados a grandes equipes e grandes bases de código. Esses problemas podem ser exacerbados quando a equipe está distribuído. Também sabemos que equipes menores trabalham em bases de código menores tendem a ser mais produtivos.

Os microserviços nos permitem alinhar melhor nossa arquitetura à nossa organização, nos ajudando a minimizar o número de pessoas trabalhando em qualquer base de código para atingiu o ponto ideal do tamanho e da produtividade da equipe. Os microserviços também nos permitem para mudar a propriedade dos serviços à medida que a organização muda, o que nos permite manter o alinhamento entre arquitetura e organização no futuro.

Composabilidade

Uma das principais promessas de sistemas distribuídos e orientados a serviços arquiteturas é que abrimos oportunidades para reutilização de funcionalidades. Com microserviços, permitimos que nossa funcionalidade seja consumida em diferentes

formas para diferentes propósitos. Isso pode ser especialmente importante quando pensamos sobre como nossos consumidores usam nosso software.

Já se foi o tempo em que podíamos pensar de forma restrita sobre qualquer um dos nossos desktops site ou nosso aplicativo móvel. Agora precisamos pensar nas inúmeras maneiras que talvez queiramos unir recursos para a web, nativos aplicativo, web móvel, aplicativo para tablet ou dispositivo vestível. Como organizações deixe de pensar em termos de canais estreitos para abraçar mais conceitos holísticos de engajamento do cliente, precisamos de arquiteturas que possam continue.

Com microsserviços, pense em abrirmos emendas em nosso sistema que são endereçável por terceiros. À medida que as circunstâncias mudam, podemos construir aplicações de maneiras diferentes. Com uma aplicação monolítica, muitas vezes tenho uma costura de granulação grossa que pode ser usada do lado de fora. Se eu quiser quebrar Para conseguir algo mais útil, vou precisar de um martelo!

Pontos problemáticos do microsserviço

As arquiteturas de microsserviços trazem uma série de benefícios, como já vimos. Mas eles também trazem uma série de complexidade. Se você está pensando em adotar um arquitetura de microsserviços, é importante que você possa comparar o bom com o ruim. Na realidade, a maioria dos pontos de microsserviço pode ser estabelecida no porta de sistemas distribuídos e, portanto, provavelmente seria evidente em um monólito distribuído como em uma arquitetura de microsserviços.

Abordaremos muitas dessas questões em profundidade durante o resto do livro - na verdade, eu diria que a maior parte deste livro é sobre como lidar com o dor, sofrimento e horror de possuir uma arquitetura de microsserviços.

Experiência de desenvolvedor

À medida que você tem mais e mais serviços, a experiência do desenvolvedor pode começar a sofrer. Tempos de execução que consomem mais recursos, como a JVM, podem limitar o número de microsserviços que podem ser executados em uma única máquina de desenvolvedor. Eu poderia provavelmente executar quatro ou cinco microsserviços baseados em JVM como processos separados em

meu laptop, mas eu poderia rodar 10 ou 20? Provavelmente não. Mesmo com menos tributação de tempos de execução, há um limite para o número de coisas que você pode executar localmente, o que inevitavelmente iniciará conversas sobre o que fazer quando você não conseguir executar o sistema inteiro em uma máquina. Isso pode se tornar ainda mais complicado se você está usando serviços em nuvem que não podem ser executados localmente.

As soluções extremas podem envolver o "desenvolvimento na nuvem", onde os desenvolvedores deixe de ser capaz de se desenvolver localmente. Eu não sou fã disso, porque os ciclos de feedback podem sofrer muito. Em vez disso, acho que limitar o escopo de quais partes de um sistema em que um desenvolvedor precisa trabalhar provavelmente será uma abordagem muito mais direta. No entanto, isso pode ser problemático se você deseja adotar mais um modelo de "propriedade coletiva" no qual qualquer espera-se que o desenvolvedor trabalhe em qualquer parte do sistema.

Sobrecarga de tecnologia

O grande peso da nova tecnologia que surgiu para permitir a adoção das arquiteturas de microsserviços pode ser avassaladora, vou ser honesto e dizer que grande parte dessa tecnologia acaba de ser renomeada como "microsserviço amigável", mas alguns avanços ajudaram legitimamente a lidar com a complexidade desses tipos de arquiteturas. Existe o perigo, porém, de que isso A riqueza de brinquedos novos pode levar a uma forma de fetichismo tecnológico que eu já vi muitas empresas adotando a arquitetura de microsserviços que decidiram que era também é o melhor momento para introduzir vastas variedades de tecnologia nova e, muitas vezes, alienígena.

Os microsserviços podem muito bem oferecer a opção de cada microsserviço ser escrito em uma linguagem de programação diferente, para ser executado em um tempo de execução diferente, ou usar um banco de dados diferente, mas essas são opções, não requisitos. Você precisa equilibrar cuidadosamente a amplitude e a complexidade da tecnologia que você use contra os custos que uma variedade diversificada de tecnologia pode acarretar.

Quando você começa a adotar microsserviços, alguns desafios fundamentais são inevitável: você precisará gastar muito tempo entendendo os problemas consistência de dados, latência, modelagem de serviços e similares. Se você está tentando entender como essas ideias mudam a maneira como você pensa sobre software desenvolvimento ao mesmo tempo em que você está abraçando uma grande quantidade de novidades

tecnologia, você terá dificuldades com isso. Também vale ressaltar que o a largura de banda ocupada ao tentar entender toda essa nova tecnologia será reduza o tempo que você tem para realmente enviar recursos aos seus usuários.

À medida que você aumenta (gradualmente) a complexidade do seu microserviço arquitetura, procure introduzir novas tecnologias conforme necessário. Você não precisa um cluster Kubernetes quando você tem três serviços! Além de garantir que você não está sobrecarregado com a complexidade dessas novas ferramentas, isso o aumento gradual tem o benefício adicional de permitir que você ganhe coisas novas e melhores maneiras de fazer coisas que, sem dúvida, surgirão com o tempo.

Custo

É altamente provável que, no curto prazo, pelo menos você veja um aumento nos custos de vários fatores. Em primeiro lugar, você provavelmente precisará executar mais coisas - mais processos, mais computadores, mais rede, mais armazenamento e muito mais software de suporte (que incorrerá em taxas adicionais de licença).

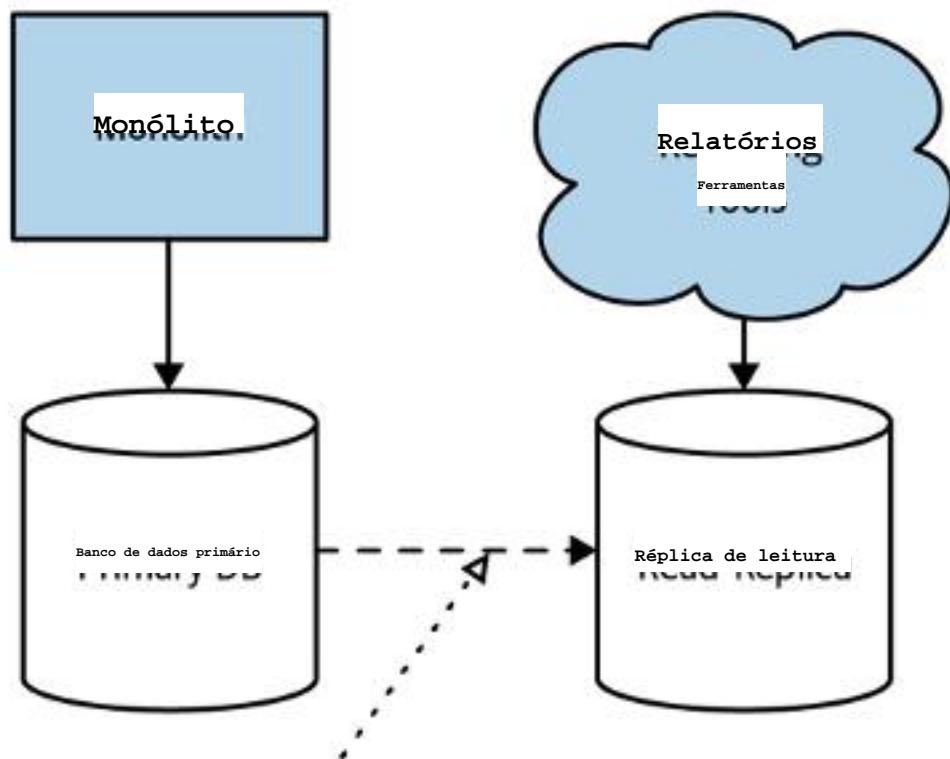
Em segundo lugar, qualquer mudança que você introduzir em uma equipe ou organização diminuirá. você cai no curto prazo. É preciso tempo para aprender novas ideias e para se exercitar como usá-los de forma eficaz. Enquanto isso estiver acontecendo, outras atividades serão impactado. Isso resultará em uma desaceleração direta na entrega de novos funcionalidade ou a necessidade de adicionar mais pessoas para compensar esse custo.

Na minha experiência, os microserviços são uma escolha ruim para uma organização preocupado principalmente com a redução de custos, como uma mentalidade de redução de custos - onde A TI é vista como um centro de custos e não como um centro de lucros - será constantemente um arraste para tirar o máximo proveito dessa arquitetura. Por outro lado, microserviços podem ajudar você a ganhar mais dinheiro se você puder usá-los arquiteturas para alcançar mais clientes ou desenvolver mais funcionalidades no paralelo. Então, os microserviços são uma forma de gerar mais lucros? Talvez. São microserviços são uma forma de reduzir custos? Nem tanto.

Relatórios

Com um sistema monolítico, você normalmente tem um banco de dados monolítico. Isso significa que as partes interessadas que desejam analisar todos os dados em conjunto, muitas vezes

envolvendo grandes operações de junção entre dados, tenha um esquema pronto contra os quais executar seus relatórios. Eles podem simplesmente colocá-los diretamente contra o banco de dados monolítico, talvez contra uma réplica de leitura, conforme mostrado na Figura 1 - 12.



Os dados são replicados de forma assíncrona em uma réplica
Data is replicated asynchronously to a replica
com o mesmo esquema.

Figura 1-12 Relatórios realizados diretamente no banco de dados de um monólito

Com uma arquitetura de microserviços, quebramos esse esquema monolítico. Isso não significa que a necessidade de gerar relatórios sobre todos os nossos dados tenha desaparecido. longe; acabamos de tornar tudo muito mais difícil, porque agora nossos dados são espalhados por vários esquemas logicamente isolados.

Abordagens mais modernas de geração de relatórios, como o uso de streaming para permitir relatórios em tempo real sobre grandes volumes de dados, podem funcionar bem com um arquitetura de microserviços, mas normalmente exige a adoção de novas ideias e tecnologia associada. Como alternativa, talvez você simplesmente precise publicar dados de seus microservices para bancos de dados centrais de relatórios (ou talvez menos lagos de dados estruturados) para permitir a geração de relatórios de casos de uso.

Monitoramento e solução de problemas

Com um aplicativo monolítico padrão, podemos ter um aplicativo bastante simplista abordagem ao monitoramento. Temos um pequeno número de máquinas com as quais se preocupar, e o modo de falha do aplicativo é um pouco binário geralmente está tudo para cima ou para baixo. Com uma arquitetura de microsserviços, compreende o impacto se apenas uma única instância de um serviço cair? Com um sistema monolítico, se nossa CPU ficar presa em 100% por um longo tempo, nós sei que é um grande problema. Com uma arquitetura de microsserviços com dezenas ou centenas de processos, podemos dizer a mesma coisa? Precisamos acordar alguém acorda às 3 da manhã quando apenas um processo está preso em 100% da CPU?

Felizmente, há uma série de ideias nesse espaço que podem ajudar. Se você tivesse gostaria de explorar esse conceito com mais detalhes, eu recomendo Sistemas Distribuídos A observabilidade de Cindy Sridharan (O'Reilly) como um excelente ponto de partida, embora também analisemos nosso próprio monitoramento e observabilidade em Capítulo 10.

Segurança

Com um sistema monolítico de processo único, muitas das nossas informações fluíram dentro desse processo. Agora, mais informações fluem pelas redes entre nossos serviços. Isso pode tornar nossos dados mais vulneráveis à observação em trânsito e também para ser potencialmente manipulado como parte do man-in-the-middle ataques. Isso significa que talvez você precise dedicar mais cuidado à proteção dos dados. em trânsito e para garantir que seus endpoints de microsserviços estejam protegidos de forma que somente partes autorizadas podem fazer uso delas. O capítulo 11 é dedicado inteiramente a analisar os desafios neste espaço.

Testando

Com qualquer tipo de teste funcional automatizado, você tem um equilíbrio delicado. Quanto mais funcionalidades um teste executar, ou seja, mais amplo será o escopo do teste — quanto mais confiança você tiver em sua inscrição. Por outro lado, o quanto maior o escopo do teste, mais difícil é configurar dados de teste e suporte

luminárias, quanto mais tempo o teste pode levar para ser executado e mais difícil pode ser trabalhar descubra o que está quebrado quando falha. No Capítulo 9, compartilharei uma série de técnicas para fazer com que os testes funcionem nesse ambiente mais desafiador.

Os testes de ponta a ponta para qualquer tipo de sistema estão no extremo da escala em termos da funcionalidade que eles abrangem, e estamos acostumados com eles sendo mais problemático de escrever e manter do que testes unitários de menor escopo. Muitas vezes isso é vale a pena, porém, porque queremos a confiança que vem de ter um teste de ponta a ponta usam nossos sistemas da mesma forma que um usuário faria.

Mas com uma arquitetura de microsserviços, o escopo de nossos testes de ponta a ponta se torna muito grande. Agora precisaríamos executar testes em vários processos, todos os quais precisam ser implantados e configurados adequadamente para os cenários de teste. Também precisamos estar preparados para os falsos negativos que ocorrem quando problemas ambientais, como instâncias de serviço morrendo ou rede o tempo limite de implantações fracassadas faz com que nossos testes falhem.

Essas forças significam que, à medida que sua arquitetura de microsserviços cresce, você obterá um retorno sobre o investimento decrescente quando se trata de testes de ponta a ponta. O teste custará mais, mas não conseguirá fornecer o mesmo nível de confiança do que aconteceu no passado. Isso o levará a novas formas de testes, como testes baseados em contratos ou testes em produção, bem como o exploracão de técnicas de entrega progressiva, como execuções paralelas ou lançamentos canários, que veremos no Capítulo 8.

Latência

Com uma arquitetura de microsserviços, processamento que antes poderia ter sido feito localmente em um processador agora pode acabar sendo dividido em vários microsserviços separados. Informações que anteriormente fluíam em apenas um único processo agora precisa ser serializado, transmitido e desserializado por redes que você pode estar exercitando mais do que nunca. Tudo isso pode resultar na piora da latência do seu sistema.

Embora possa ser difícil medir o impacto exato na latência de operações na fase de design ou codificação, esse é outro motivo pelo qual é importante para realizar qualquer migração de microsserviços de forma incremental. Faça um

pequena mudança e, em seguida, meça o impacto. Isso pressupõe que você tenha alguma maneira de medir a latência de ponta a ponta das operações que lhe interessam. Ferramentas de rastreamento distribuído, como a Jaeger, podem ajudar aqui. Mas você também precisa ter uma compreensão de qual latência é aceitável para essas operações.

Às vezes, tornar uma operação mais lenta é perfeitamente aceitável, desde que ainda é rápido o suficiente!

Consistência de dados

Mudando de um sistema monolítico, no qual os dados são armazenados e gerenciados em um banco de dados único, para um sistema muito mais distribuído, no qual vários processos gerenciam o estado em diferentes bancos de dados, causando possíveis desafios com relação à consistência dos dados. Enquanto que no passado você poderia ter com base em transações de banco de dados para gerenciar mudanças de estado, você precisará entender que segurança semelhante não pode ser facilmente fornecida em um sistema distribuído. O uso de transações distribuídas na maioria dos casos prova ser altamente problemático na coordenação de mudanças de estado.

Em vez disso, talvez você precise começar a usar conceitos como sagas (algo que eu vou detalhar no Capítulo 6) e eventual consistência para gerenciar e raciocinar sobre o estado em seu sistema. Essas ideias podem exigir mudanças fundamentais em a maneira como você pensa sobre os dados em seus sistemas, algo que pode ser bastante assustador ao migrar sistemas existentes. Mais uma vez, este é outro bem motivo para ter cuidado com a rapidez com que você decompõe seu aplicativo. Adotando uma abordagem incremental para a decomposição, para que você seja capaz de avaliar o impacto das mudanças em sua arquitetura na produção, é realmente importante.

Devo usar microsserviços?

Apesar da vontade em alguns setores de criar arquiteturas de microsserviços, abordagem padrão para software, acho que, devido aos inúmeros desafios Eu descrevi que adotá-los ainda requer uma reflexão cuidadosa. Você precisa avaliar seu próprio espaço problemático, habilidades e cenário tecnológico e entenda o que você está tentando alcançar antes de decidir se os microsserviços são

certo para você. Eles são uma abordagem arquitetônica, não a arquitetônica...
abordagem. Seu próprio contexto deve desempenhar um papel importante na sua decisão de saber se
para seguir esse caminho.

Dito isso, quero descrever algumas situações que normalmente me alertariam
escolhendo microserviços de ou para frente.

Para quem eles podem não trabalhar

Dada a importância de definir limites de serviço estáveis, sinto que
arquiteturas de microserviços geralmente são uma má escolha para produtos totalmente novos ou
startups. Em ambos os casos, o domínio com o qual você está trabalhando normalmente é
passando por mudanças significativas à medida que você itera sobre os fundamentos do que você
estão tentando construir. Essa mudança nos modelos de domínio, por sua vez, resultará em mais
mudanças que estão sendo feitas nos limites de serviço e coordenando mudanças em todo o mundo.
limites de serviço são uma tarefa cara. Em geral, acho que é mais
apropriado esperar até que uma parte suficiente do modelo de domínio tenha se estabilizado antes
procurando definir limites de serviço.

Eu vejo a tentação de as startups usarem o microserviço em primeiro lugar, o raciocínio
sendo: "Se formos realmente bem-sucedidos, precisaremos escalar!" O problema é que
você não sabe necessariamente se alguém vai querer usar seu novo
produto. E mesmo que você tenha sucesso o suficiente para exigir uma
arquitetura escalável, o que você acaba entregando aos seus usuários pode ser
muito diferente do que você começou a construir em primeiro lugar. Uber inicialmente
focado em limusines, e o Flickr surgiu das tentativas de criar um multiplayer
jogo online. O processo de encontrar um produto adequado ao mercado significa que você pode
acabar com um produto muito diferente no final do que você pensava que seria
construa quando você começou.

As startups também costumam ter menos pessoas disponíveis para criar o sistema,
o que cria mais desafios em relação aos microserviços. Microserviços
trazem consigo fontes de novo trabalho e complexidade, e isso pode acabar
largura de banda valiosa. Quanto menor a equipe, mais pronunciado é esse custo
será. Ao trabalhar com equipes menores com apenas um punhado de desenvolvedores,
Estou muito hesitante em sugerir microserviços por esse motivo.

O desafio dos microsserviços para startups é agravado pelo fato de que normalmente, sua maior restrição são as pessoas. Para uma equipe pequena, um microsserviço a arquitetura pode ser difícil de justificar porque é necessário trabalho apenas para lidar com a implantação e o gerenciamento dos próprios microsserviços. Algumas pessoas descreveram isso como o "imposto sobre microsserviços". Quando isso o investimento beneficia muitas pessoas, é mais fácil de justificar. Mas se uma pessoa sair da sua equipe de cinco pessoas está gastando seu tempo com essas questões, isso é muito tempo valioso que não está sendo gasto construindo seu produto. É muito mais fácil se mover para microsserviços posteriormente, depois de entender onde estão as restrições em seu arquitetura e quais são seus pontos problemáticos - então você pode concentrar sua energia sobre o uso de microsserviços nos lugares mais sensatos.

Finalmente, organizações que criam software que será implantado e gerenciado por seus clientes podem ter dificuldades com microsserviços. Como já fizemos arquiteturas de microsserviços cobertas podem levar muita complexidade ao domínio operacional e de implantação. Se você estiver executando o software sozinho, você é capaz de compensar essa nova complexidade adotando novas tecnologias, desenvolver novas habilidades e mudar as práticas de trabalho. Isso não é algo você pode esperar que seus clientes o façam. Se eles estão acostumados a receber seu software como instalador do Windows, será um choque legal para eles quando você envia a próxima versão do seu software e diz: "Basta colocar esses 20 pods em seu cluster Kubernetes!". Com toda a probabilidade, eles terão Não tenho ideia do que é um pod, Kubernetes ou cluster.

Onde eles funcionam bem

Em minha experiência, provavelmente o maior motivo pelo qual as organizações adotam microsserviços são para permitir que mais desenvolvedores trabalhem no mesmo sistema sem atrapalhar um ao outro. Obtenha sua arquitetura e organização limites corretos, e você permite que mais pessoas trabalhem independentemente de cada um outros, reduzindo a contenção de entrega. É provável que uma startup de cinco pessoas encontre um arquitetura de microsserviços é uma chatice. Um aumento de escala de cem pessoas que está crescendo é provável que rapidamente descubra que seu crescimento é muito mais fácil de acomodar com um arquitetura de microsserviços devidamente alinhada ao desenvolvimento de seu produto esforços.

Os aplicativos de software como serviço (SaaS) também são, em geral, uma boa opção para uma arquitetura de microserviços. Normalmente, espera-se que esses produtos operem 24 horas por dia, 7 dias por semana, o que cria desafios quando se trata de implementar mudanças. O a liberabilidade independente de arquiteturas de microserviços é um grande benefício neste área. Além disso, os microserviços podem ser ampliados ou reduzidos conforme necessário. Isso significa que, ao estabelecer uma linha de base sensata para a carga do seu sistema características, você tem mais controle sobre como garantir que você possa escalar seu sistema da maneira mais econômica possível.

A natureza independente de tecnologia dos microserviços garante que você possa obter a maioria das plataformas em nuvem. Os fornecedores de nuvem pública oferecem uma ampla variedade de serviços e mecanismos de implantação para seu código. Você pode muito mais combinar facilmente os requisitos de serviços específicos com os serviços em nuvem que ajudará você a implementá-los da melhor maneira. Por exemplo, você pode decidir implantar um serviço como um conjunto de funções, outro como uma máquina virtual gerenciada (VM), e outro em uma plataforma gerenciada de Plataforma como Serviço (PaaS).

Embora seja importante notar que a adoção de uma ampla gama de tecnologias muitas vezes pode seja um problema, ser capaz de experimentar novas tecnologias com facilidade é uma boa maneira de identifique rapidamente novas abordagens que possam gerar benefícios. O crescimento a popularidade das plataformas FaaS é um desses exemplos. Para o apropriado cargas de trabalho, uma plataforma FaaS pode reduzir drasticamente a quantidade de sobrecarga operacional, mas, no momento, não é um mecanismo de implantação que seria adequado em todos os casos.

Os microserviços também apresentam benefícios claros para organizações que buscam fornecer serviços aos seus clientes por meio de uma variedade de novos canais. Muitos os esforços de transformação digital parecem envolver a tentativa de desbloquear a funcionalidade escondido nos sistemas existentes. O desejo é criar novos clientes experiências que podem atender às necessidades dos usuários por meio de qualquer interação mecanismo faz mais sentido.

Acima de tudo, uma arquitetura de microserviços é aquela que pode oferecer muitas flexibilidade à medida que você continua evoluindo seu sistema. Essa flexibilidade tem um custo, claro, mas, se você quiser manter suas opções abertas em relação às mudanças, você pode querer ganhar no futuro, pode ser um preço que valha a pena pagar.

Resumo

As arquiteturas de microserviços podem oferecer um grande grau de flexibilidade em escolher a tecnologia, lidar com robustez e escalabilidade, organizar equipes e mais. Essa flexibilidade é, em parte, o motivo pelo qual muitas pessoas estão adotando arquiteturas de microserviços: mas os microserviços trazem consigo uma significativa grau de complexidade, e você precisa garantir que essa complexidade seja garantido. Para muitos, eles se tornaram uma arquitetura de sistema padrão, para ser usado em praticamente todas as situações. No entanto, eu ainda acho que eles são um escolha arquitetônica cujo uso deve ser justificado pelos problemas que você é tentando resolver; muitas vezes, abordagens mais simples podem funcionar com muito mais facilidade.

No entanto, muitas organizações, especialmente as maiores, mostraram como microserviços eficazes podem ser. Quando os principais conceitos de microserviços são devidamente compreendidos e implementados, eles podem ajudar a criar empoderamento, arquiteturas produtivas que podem ajudar os sistemas a se tornarem mais do que a soma de suas partes.

Espero que este capítulo tenha servido como uma boa introdução a esses tópicos. Em seguida, veremos como definimos os limites dos microserviços, explorando o tópicos de programação estruturada e design orientado por domínio ao longo do caminho.

1 Esse conceito foi descrito pela primeira vez por David Parnas em "Aspectos do Design da Distribuição de Informações". Metodologia". Anais de processamento de informações do Congresso do IFIP de 1971 (Amsterdã: Holanda do Norte (1972). 1:39-44.

2 Alistair Cockburn, "Hexagonal Architecture", 4 de janeiro de 2005, <https://oreil.ly/NfvTP>.

3 Para uma introdução detalhada ao design orientado por domínio, consulte Design orientado por domínio, de Eric Evans (Addison-Wesley) - ou para uma visão geral mais resumida, consulte Domain-Driven Design Distilled by Vaughn Vernon (Addison-Wesley).

4 Matthew Skelton e Manuel Pais, Team Topologies (Portland, OR: IT Revolution, 2019).

5 David Heinemeier Hansson, "The Majestic Monolith", Signal V. Noise, 29 de fevereiro de 2016, <https://oreil.ly/WwGIC>.

6 Para obter algumas informações úteis sobre o pensamento por trás do uso de um monólito modular pela Shopify em vez de microserviços, assista "Deconstructing the Monolith", de Kirsten Westerinde.

7 Leslie Lamport, mensagem de e-mail para um quadro de avisos do DEC SRC às 12:23:29 PDT de 28 de maio de 1987.

8 A Microsoft Research realizou estudos neste espaço, e eu recomendo todos eles, mas como ponto de partida, sugiro "Não toque no meu código! Examinando os efeitos da propriedade em "Qualidade de software", de Christian Bird et al.

Capítulo 2. Como modelar Microsserviços

O raciocínio do meu oponente me lembra dos pagãos, que, sendo questionados sobre o que o mundo estava, respondeu: "Em uma tartaruga. Mas sobre o que o suporte de tartaruga? "Em outra tartaruga. ""

-Rev. Joseph Frederick Berg (1854)

Então você sabe o que são microsserviços e, espero, tenha uma noção de sua chave benefícios. Você provavelmente está ansioso agora para começar a fazê-los, certo? Mas por onde começar? Neste capítulo, veremos alguns conceitos fundamentais, como como ocultação, acoplamento e coesão de informações e entenda como elas funcionarão mude nosso pensamento sobre traçar limites em torno de nossos microsserviços. Nós vamos veja também as diferentes formas de decomposição que você pode usar, bem como focar mais profundamente no design orientado por domínio é extremamente útil técnica neste espaço.

Veremos como pensar sobre os limites de seus microsserviços para para maximizar as vantagens e evitar algumas das possíveis desvantagens. Mas primeiro, precisamos de algo com que trabalhar.

Apresentando a MusicCorp

Livros sobre ideias funcionam melhor com exemplos. Sempre que possível, eu serei compartilhar histórias de situações do mundo real, mas descobri que também é útil para tenha um cenário fictício com o qual trabalhar. Ao longo do livro, seremos voltando a esse cenário, vendo como funciona o conceito de microsserviços dentro deste mundo.

Então, vamos voltar nossa atenção para o varejista on-line de ponta MusicCorp. A MusicCorp era até recentemente apenas uma varejista física, mas depois do bottom saiu do negócio de discos de vinil, ele se concentrou cada vez mais em

seus esforços on-line. A empresa tem um site, mas acha que agora é a hora de aposte duas vezes no mundo online. Afinal, esses smartphones para música são apenas uma moda passageira (os Zunes são muito melhores, obviamente), e os fãs de música são bastante felizes em esperar que os CDs cheguem à sua porta. Qualidade acima da conveniência, certo? E embora tenha acabado de saber que o Spotify é na verdade um dispositivo digital de música em vez de algum tipo de tratamento de pele para adolescentes, A MusicCorp está muito feliz com seu próprio foco e tem certeza de tudo isso. Os negócios de streaming acabarão em breve.

Apesar de estar um pouco atrasada, a MusicCorp tem grandes ambições. Felizmente, decidiu que sua melhor chance de dominar o mundo é certifique-se de que ele possa fazer alterações o mais facilmente possível. Microsserviços para a vitória!

O que define um bom limite de microsserviços?

Antes que a equipe da MusicCorp se afaste, criando um serviço após o serviço, na tentativa de entregar fitas de oito faixas para todos, vamos pisa no freio e fale um pouco sobre a ideia subjacente mais importante: preciso ter em mente. Queremos que nossos microsserviços possam ser alterados e implantados, e sua funcionalidade liberada para nossos usuários, de forma independente de moda. A capacidade de alterar um microsserviço isoladamente de outro é vital. Então, o que precisamos ter em mente quando pensamos sobre como traçamos os limites em torno deles?

Em essência, os microsserviços são apenas outra forma de decomposição modular, embora tenha interação baseada em rede entre os modelos e todos os desafios associados que isso traz. Felizmente, isso significa que podemos confiar em muitos arte anterior no espaço de software modular e programação estruturada para ajudam a nos orientar em termos de como definir nossos limites. Com isso em mente, vamos examinar mais profundamente três conceitos-chave que abordamos resumidamente no Capítulo 1 e que são vitais para entender quando se trata de malhar o que torna um bom microsserviço a ocultação de informações sobre limites, coesão e acoplamento.

Ocultação de informações

A ocultação de informações é um conceito desenvolvido por David Parnas para analisar a maneira mais eficaz de definir os limites do módulo. A Ocultação de informações descreve o desejo de ocultar o máximo possível de detalhes por trás de um módulo (ou, no nosso caso, limite de microsserviço). Parnas analisou os benefícios que os módulos deveriam teoricamente nos fornecer, a saber:

Tempo de desenvolvimento aprimorado

Ao permitir que os módulos sejam desenvolvidos de forma independente, podemos permitir mais trabalho a ser feito em paralelo e reduzir o impacto de adicionar mais desenvolvedores de um projeto.

Compreensibilidade

Cada módulo pode ser analisado isoladamente e compreendido isoladamente. Isso, por sua vez, torna mais fácil entender o que é o sistema como um todo. faz.

Flexibilidade

Os módulos podem ser alterados independentemente um do outro, permitindo alterações a serem feitas na funcionalidade do sistema sem exigir outros módulos para mudar. Além disso, os módulos podem ser combinados em maneiras diferentes de oferecer novas funcionalidades.

Esta lista de características desejáveis complementa muito bem o que estamos tentando para alcançar com arquiteturas de microsserviços - e, de fato, agora vejo microsserviços como apenas outra forma de arquitetura modular. Adrian Colyer na verdade, analisou vários artigos de David Parnas a partir deste período e os examinou com relação aos microsserviços e seus resumos valem bem a pena ler.²

A realidade, como Parnas explorou em grande parte de seu trabalho, é que ter módulos não resultam em você realmente alcançar esses resultados. Muito depende de como os limites do módulo são formados. De sua própria pesquisa,

a ocultação de informações foi uma técnica fundamental para ajudar a tirar o máximo proveito do modular arquiteturas e, com um olhar moderno, o mesmo se aplica aos microserviços.

De outro artigo de Parnas, 3, temos esta joia:

As conexões entre os módulos são as suposições de que o os módulos se relacionam uns com os outros.

Reduzindo o número de suposições de que um módulo (ou microserviço) faz sobre outro, impactamos diretamente as conexões entre eles. Por mantendo o número de suposições pequeno, é mais fácil garantir que possamos altere um módulo sem impactar outros. Se um desenvolvedor estiver alterando um módulo tem uma compreensão clara de como o módulo é usado por outros, ele será mais fácil para o desenvolvedor fazer alterações com segurança de forma que os chamadores upstream também não precisarão mudar.

Isso também se aplica aos microserviços, exceto que também temos a oportunidade de implantar esse microserviço alterado sem precisar implantar qualquer outra coisa, possivelmente amplificando as três características desejáveis que Parnas descreve a melhoria do tempo de desenvolvimento, da compreensibilidade e flexibilidade.

As implicações da ocultação de informações se manifestam de muitas maneiras, e eu vou aprender esse tema ao longo do livro.

Coesão

Uma das definições mais sucintas que já ouvi para descrever a coesão é isso: "o código que muda junto, permanece junto." 4 Para nossos propósitos, este é uma definição muito boa. Como já discutimos, estamos otimizando nossa arquitetura de microserviços baseada na facilidade de fazer mudanças nos negócios funcionalidade - então queremos que a funcionalidade seja agrupada de forma que pode fazer alterações no menor número possível de lugares.

Queremos que o comportamento relacionado se encaixe e o comportamento não relacionado fique sentado em outro lugar. Por quê? Bem, se quisermos mudar o comportamento, queremos ser capazes de altere-o em um só lugar e libere essa alteração o mais rápido possível. Se nós tem que mudar esse comportamento em muitos lugares diferentes, teremos que liberar

muitos serviços diferentes (talvez ao mesmo tempo) para oferecer essa mudança. Fazer mudanças em vários lugares diferentes é mais lento e implantar muitas serviços ao mesmo tempo são arriscados, portanto, queremos evitar os dois.

Portanto, queremos encontrar limites em nosso domínio problemático que ajudem a garantir o comportamento relacionado está em um só lugar e se comunica com outros limites o mais vagamente possível. Se a funcionalidade relacionada estiver espalhada pelo sistema, dizemos que a coesão é fraca, enquanto que para nosso microserviço arquiteturas que buscamos uma forte coesão.

Acoplamento

Quando os serviços estão fracamente acoplados, uma mudança em um serviço não deve exigir uma mudança para outra. O objetivo principal de um microserviço é ser capaz de fazer uma alteração em um serviço e implantá-lo sem precisar alterar nenhuma outra parte do sistema. Isso é realmente muito importante.

Que tipo de coisa causa um acoplamento estreito? Um erro clássico é escolher um estilo de integração que vincula fortemente um serviço a outro, causando mudanças dentro do serviço para exigir uma mudança para os consumidores.

Um serviço fracamente acoplado sabe o mínimo necessário sobre os serviços com o qual colabora. Isso também significa que provavelmente queremos limitar o número de tipos diferentes de chamadas de um serviço para outro, porque além do potencial problema de desempenho, a comunicação por chat pode levar a um acoplamento firme.

O acoplamento, no entanto, vem em várias formas, e eu vi várias mal-entendidos sobre a natureza do acoplamento no que se refere a um serviço-arquitetura baseada. Com isso em mente, acho importante que exploremos este tópico com mais detalhes, algo que faremos em breve.

A interação de acoplamento e coesão

Como já mencionamos, os conceitos de acoplamento e coesão são obviamente relacionados. Logicamente, se a funcionalidade relacionada estiver espalhada por nosso sistema, as mudanças nessa funcionalidade ultrapassarão esses limites,

implicando um acoplamento mais estreito. Lei de Constantino, batizada em homenagem ao design estruturado

O pioneiro Larry Constantine, resume isso perfeitamente:

Uma estrutura é estável se a coesão for forte e o acoplamento for baixo. 5
A structure is stable if cohesion is strong and coupling is low.

O conceito aqui de estabilidade é importante para nós. Para nosso microserviço limites para cumprir a promessa de implantação independente, permitindo nós devemos trabalhar em microserviços em paralelo e reduzir a quantidade de coordenação entre as equipes que trabalham nesses serviços, precisamos de alguns grau de estabilidade nos próprios limites. Se o contrato for um o microserviço expõe está mudando constantemente de forma incompatível com versões anteriores moda, então isso fará com que os consumidores upstream tenham que mudar constantemente também.

Acoplamento e coesão estão fortemente relacionados e, pelo menos em algum nível, são indiscutivelmente o mesmo, pois ambos os conceitos descrevem a relação entre coisas. A coesão se aplica à relação entre coisas dentro de um limite (um microserviço em nosso contexto), enquanto o acoplamento descreve o relacionamento entre coisas além de um limite. Não existe a melhor maneira absoluta de se organizar nosso código: acoplamento e coesão são apenas uma forma de articular os vários compensações que fazemos em torno de onde agrupamos o código e por quê. Tudo o que podemos nos esforçar fazer é encontrar o equilíbrio certo entre essas duas ideias, uma que faça com que o mais adequado para seu contexto específico e para os problemas que você está enfrentando atualmente.

Lembre-se de que o mundo não é estático - é possível que, como seu sistema os requisitos mudam, você encontrará motivos para rever suas decisões. Às vezes, partes do seu sistema podem estar passando por tantas mudanças que a estabilidade pode ser impossível. Veremos um exemplo disso no Capítulo 3 quando compartilho as experiências da equipe de desenvolvimento de produtos por trás da Snap CI.

Tipos de acoplamento

Você pode inferir da visão geral anterior acima que todo acoplamento é ruim. Isso não é estritamente verdade. Em última análise, algum acoplamento em nosso sistema será inevitável. O que queremos fazer é reduzir a quantidade de acoplamento que temos.

Muito trabalho foi feito para analisar as diferentes formas de acoplamento no contexto de programação estruturada que foi amplamente considerada modular software (não distribuído, monolítico). Muitos desses modelos diferentes para avaliando a sobreposição ou conflito de acoplamentos e, em qualquer caso, eles falam principalmente sobre coisas no nível do código, em vez de considerar as coisas baseadas em serviços interações. Como os microserviços são um estilo de arquitetura modular (embora com a complexidade adicional dos sistemas distribuídos), podemos usar muitos desses conceitos originais e aplique-os no contexto de nossa base de microserviços sistemas.

ARTE ANTERIOR EM PROGRAMAÇÃO ESTRUTURADA

Muito do nosso trabalho em computação envolve desenvolver o trabalho que surgiu antes. Às vezes é impossível reconhecer tudo o que veio antes, mas com esta segunda edição eu pretendia destacar a arte anterior onde eu posso - em parte para dar crédito quando o crédito é devido, em parte como uma forma de garantindo que eu coloque algumas migalhas de pão para os leitores que desejam para explorar determinados tópicos com mais detalhes, mas também para mostrar que muitas dessas ideias são experimentadas e testadas.

Quando se trata de desenvolver o trabalho anterior, há poucos áreas temáticas deste livro que têm tanto arte anterior quanto estruturação programação. Eu já mencionei Larry Constantine: seu livro com Edward Yourdon, Design Estruturado, é considerado um dos mais textos importantes neste área. O Guia Prático do Meilleur Design para O Structured Systems Design7 também é útil. Infelizmente, uma coisa é Os livros têm em comum o quão difícil podem ser de se apossar, pois são esgotados e não estão disponíveis no formato de e-book. Mais uma razão para apoie sua biblioteca local!.

Nem todas as ideias são mapeadas de forma limpa, então fiz o meu melhor para sintetizar um trabalho modelo para os diferentes tipos de acoplamento para microserviços. Onde essas as ideias se relacionam perfeitamente com as definições anteriores. Eu continuei com esses termos. Em outros lugares em que tive que inventar novos termos ou combinar ideias em outro lugar. Então, por favor, considere o que segue para ser construído em cima de muitos

arte anterior neste espaço, que estou tentando dar mais significado no contexto de microsserviços.

Na Figura 2-1, vemos uma breve visão geral dos diferentes tipos de acoplamento, organizado de baixo (desejável) a alto (indesejável).

A seguir, examinaremos cada forma de acoplamento sucessivamente e veremos exemplos de mostre como esses formulários podem se manifestar em nosso microsserviço arquitetura.

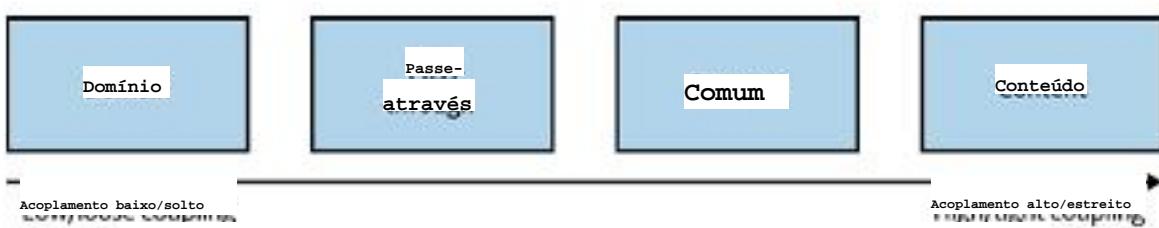


Figura 2-1. Os diferentes tipos de acoplamento, de solto (baixo) a apertado (alto).

Acoplamento de domínio

O acoplamento de domínio descreve uma situação na qual um microsserviço precisa interagir com outro microsserviço, porque o primeiro microsserviço precisa fazer uso da funcionalidade que o outro microsserviço fornece. 8

Na Figura 2-2, vemos parte de como os pedidos de CDs são gerenciados internamente na Music Corp. Neste exemplo, o Processador de Pedidos chama o Depósito microsserviço para reservar estoque e o microsserviço de pagamento para levar pagamento. O Processador de Pedidos é, portanto, dependente e acoplado aos microsserviços de Depósito e Pagamento para essa operação. Nós não vemos no entanto, tal acoplamento entre Armazém e Pagamento, pois não interagir.

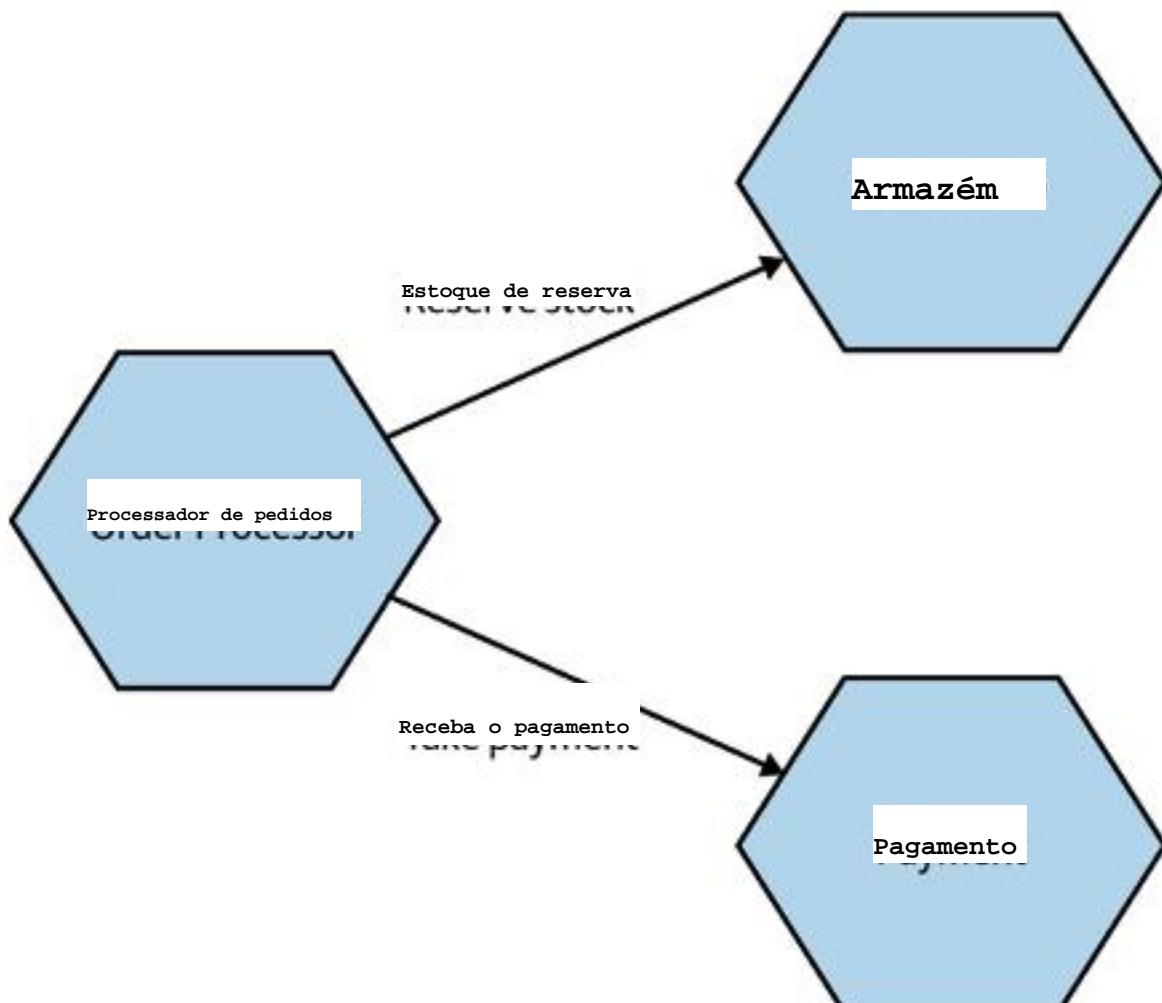


Figura 2-2. Um exemplo de acoplamento de domínios, em que o Processador de Pedidos precisa fazer uso do funcionalidade fornecida por outros microsserviços.

Em uma arquitetura de microsserviços, esse tipo de interação é praticamente inevitável. Um sistema baseado em microsserviços depende da colaboração de vários microsserviços para que ele faça seu trabalho. No entanto, ainda queremos reduzir isso ao mínimo; sempre que você vê um único microsserviço dependendo de vários downstream serviços dessa forma, podem ser motivo de preocupação - podem implicar um microsserviço que está fazendo demais.

Como regra geral, o acoplamento de domínio é considerado uma forma fraca de acoplamento, embora mesmo aqui possamos ter problemas. Um microsserviço que precisa falar com muitos microsserviços downstream pode apontar para uma situação em cuja lógica demais foi centralizada. O acoplamento de domínio também pode se tornar problemático à medida que conjuntos de dados mais complexos são enviados entre serviços

- isso geralmente pode apontar para as formas mais problemáticas de acoplamento que abordaremos explore em breve.

Basta lembrar a importância de esconder informações. Compartilhe somente o que você absolutamente necessário, e envie apenas a quantidade mínima absoluta de dados que você precisa.

UMA BREVE NOTA SOBRE ACOPLAMENTO TEMPORAL

Outra forma de acoplamento da qual você já deve ter ouvido falar é o acoplamento temporal.

De uma visão centrada no código do acoplamento, o acoplamento temporal se refere a um situação em que os conceitos são agrupados simplesmente porque acontecem ao mesmo tempo. O acoplamento temporal tem uma diferença sutil significado no contexto de um sistema distribuído, onde se refere a um situação em que um microsserviço precisa de outro microsserviço para ser executado algo ao mesmo tempo para a operação ser concluída.

Ambos os microsserviços precisam estar ativos e disponíveis para comunicação um ao outro ao mesmo tempo para que a operação seja concluída. Então, na Figura 2-3, onde o processador de pedidos da MusicCorp está fazendo um processo síncrono Chamada HTTP para o serviço Warehouse, o Warehouse precisa estar ativo e disponível ao mesmo tempo em que a chamada é feita.

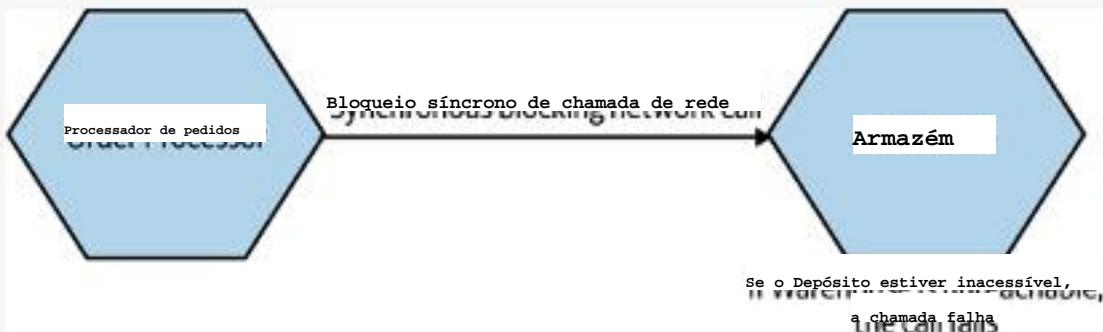


Figura 2-3. Um exemplo de acoplamento temporal, no qual o Order Processor faz um chamada HTTP síncrona para o microsserviço Warehouse.

Se, por algum motivo, o Depósito não puder ser contatado pelo Pedido Processador, então a operação falha, pois não podemos reservar os CDs para serem enviado. O processador de pedidos também terá que bloquear e esperar por um resposta do Warehouse, potencialmente causando problemas em termos de disputa de recursos.

O acoplamento temporal nem sempre é ruim; é apenas algo que você deve observar. À medida que você tem mais microsserviços, com interações mais complexas entre eles, os desafios do acoplamento temporal podem aumentar para tal ponto em que fica mais difícil escalar seu sistema e mantê-lo.

trabalhando. Uma das maneiras de evitar o acoplamento temporal é usar alguma forma de comunicação assíncrona, como um agente de mensagens.

Acoplamento de passagem

"Acoplamento de passagem" descreve uma situação em que um microsserviço passa dados para outro microsserviço simplesmente porque os dados são necessários para algum outro microsserviço mais a jusante. Em muitos aspectos, é um dos formas mais problemáticas de acoplamento de implementação, pois implica não apenas que o chamador ainda não sabe apenas que o microsserviço que está invocando chama outro microsserviço, mas também que ele potencialmente precisa saber como aquele microsserviço com etapas removidas funciona.

Como exemplo de acoplamento de passagem, vamos examinar mais de perto agora, parte de como funciona o processamento de pedidos da MusicCorp. Na Figura 2-4, temos um Processador de pedidos, que está enviando uma solicitação ao Warehouse para preparar um pedido para envio. Como parte da carga útil da solicitação, enviamos um "Manifesto de envio". Este Manifesto de Envio contém não apenas o endereço do cliente, mas também o tipo de envio. O armazém apenas passa esse manifesto para o microsserviço downstream Shipping.

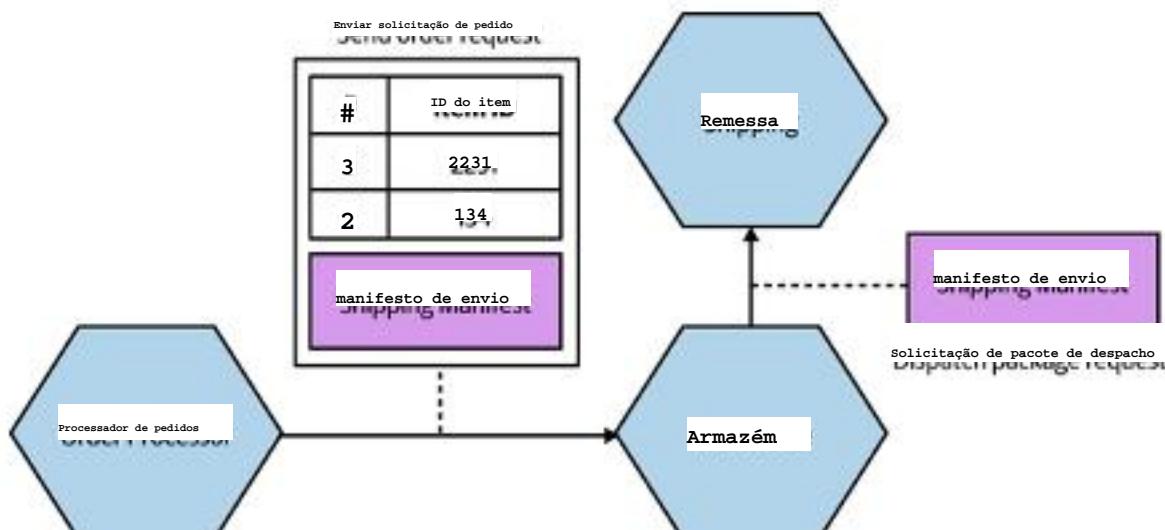


Figura 2-4. Acoplamento de passagem, no qual os dados são passados para um microsserviço simplesmente porque outro serviço downstream precisa disso

O principal problema com o acoplamento de passagem é que uma mudança no necessário os dados a jusante podem causar uma alteração mais significativa no upstream. Em nosso exemplo, se o Shipping agora precisar que o formato ou o conteúdo dos dados sejam alterado, então, tanto o Warehouse quanto o Processador de Pedidos provavelmente precisariam mudar.

Há algumas maneiras pelas quais isso pode ser corrigido. A primeira é considerar se faz sentido que o microsserviço de chamada simplesmente ignore o intermediário. Em nosso exemplo, isso pode significar que o Processador de Pedidos fala diretamente com Envio, como na Figura 2-5. No entanto, isso causa algumas outras dores de cabeça. Nossa o Processador de Pedidos está aumentando seu acoplamento de domínios, já que o Shipping ainda está outro microsserviço que ele precisa conhecer - se esse fosse o único problema, este ainda pode ser bom, já que o acoplamento de domínio é, obviamente, uma forma mais flexível de acoplamento. No entanto, essa solução fica mais complexa aqui, pois o estoque precisa ser reservado com o Warehouse antes de despacharmos o pacote usando o Shipping, e depois o envio foi feito, precisamos atualizar o estoque adequadamente. Isso introduz mais complexidade e lógica no Order Processor, ou seja, anteriormente escondido dentro do Warehouse

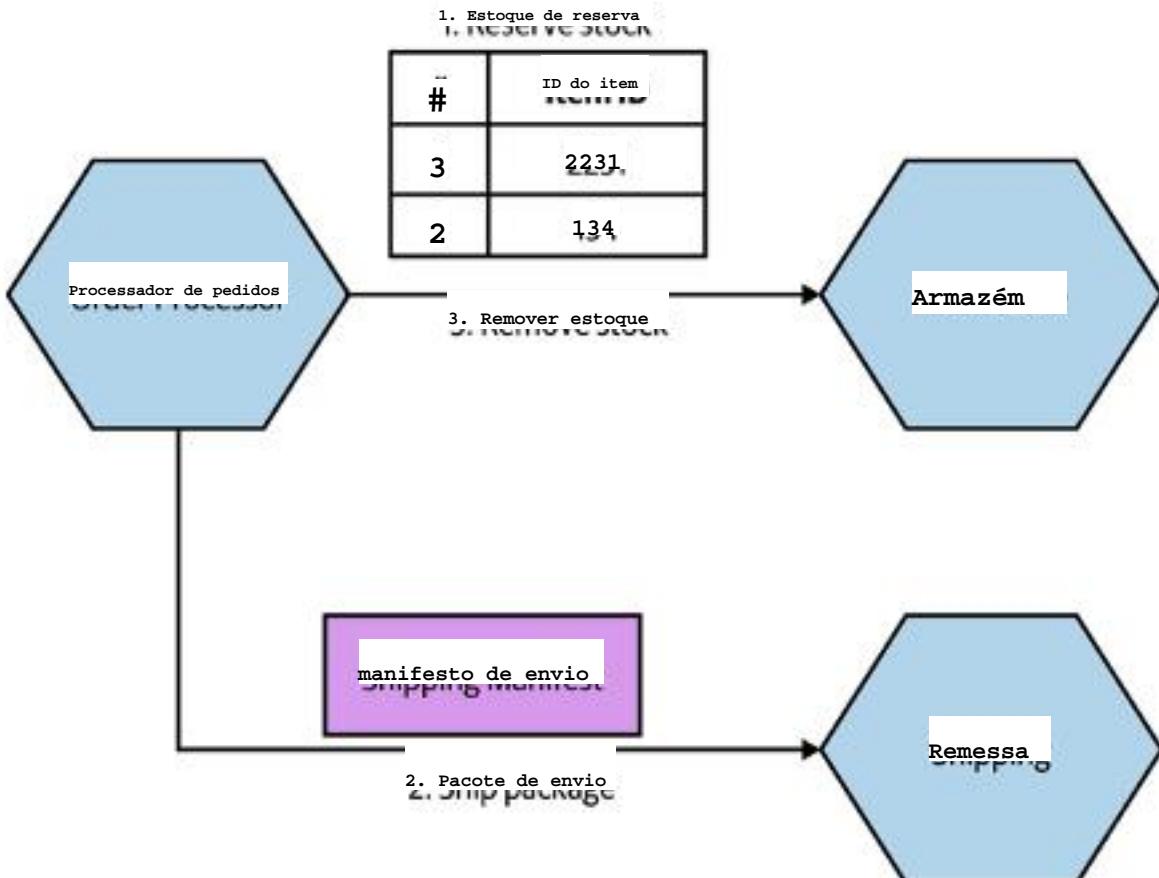


Figura 2-5. Uma forma de contornar o acoplamento de passagem envolve a comunicação direta com o serviço downstream.

Para este exemplo específico, eu poderia considerar um mais simples (embora com mais nuances) alteração, ou seja, para ocultar totalmente a exigência de um manifesto de remessa do Order Processor. A ideia de delegar o trabalho de ambos os gerentes estoque e organização do envio do pacote para o nosso serviço de armazém faz sentido, mas não gostamos do fato de termos vazado algum nível inferior de implementação, ou seja, o fato de que o microsserviço Shipping deseja um Manifesto de envio. Uma forma de esconder esse detalhe seria ter o Warehouse receber as informações necessárias como parte de seu contrato e, em seguida, faça com que ele construa o Manifesto de Envio localmente, como vemos na Figura 2-6. Isso significa que, se o serviço de transporte mudar seu contrato de serviço, a mudança será invisível do ponto de vista do Processador de Pedidos, desde que o Warehouse coleta os dados necessários.

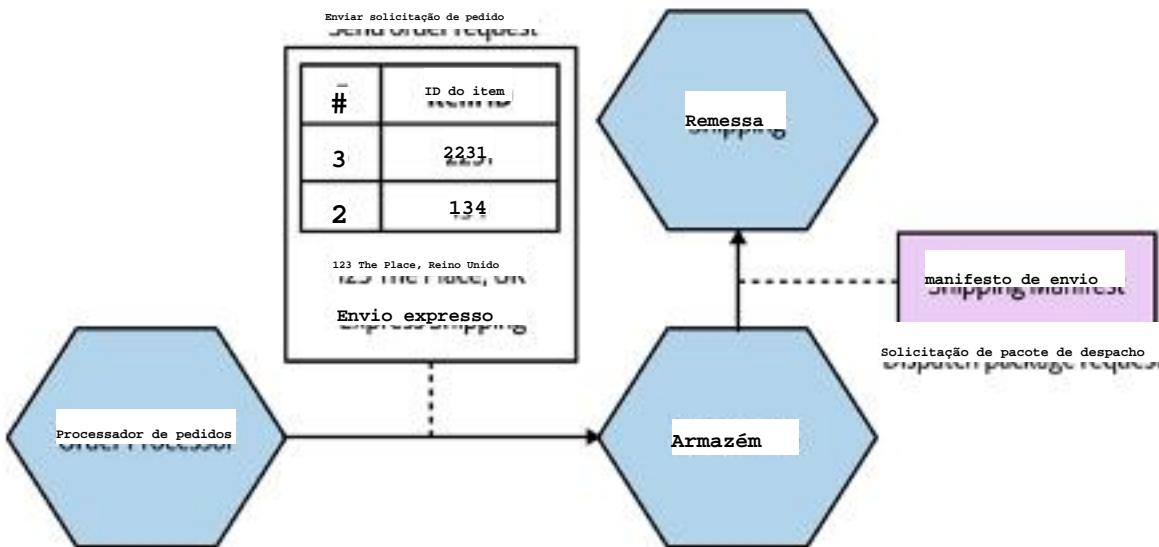


Figura 2-6. Ocultando a necessidade de um manifesto de envio do processador de pedidos

Embora isso ajude a proteger o microserviço Warehouse de algumas mudanças para o transporte marítimo, há algumas coisas que ainda exigiriam que todas as partes mudassem. Vamos considerar a ideia de que queremos começar a enviar internacionalmente. Como parte disso, o serviço de transporte precisa de uma declaração alfandegária para ser incluído no manifesto de envio. Se esse for um parâmetro opcional, então poderíamos implantar uma nova versão do microserviço Shipping sem problemas. Se esse for um parâmetro obrigatório, no entanto, o Warehouse precisaria criar um. Pode ser capaz de fazer isso com as informações existentes que possui (ou é fornecido), ou pode exigir que informações adicionais sejam passadas a ele pelo Processador de pedidos.

Embora, neste caso, não tenhamos eliminado a necessidade de fazer alterações em todos os três microserviços, recebemos muito mais poder sobre quando e como essas mudanças poderiam ser feitas. Se tivéssemos o apertado (através) do acoplamento do exemplo inicial, adicionando esta nova alfândega A declaração pode exigir uma implantação contínua de todos os três microserviços. Em pelo menos escondendo esse detalhe, poderíamos muito mais facilmente implantar em fases. Uma abordagem final que poderia ajudar a reduzir o acoplamento de passagem seria é necessário que o Processador de Pedidos ainda envie o Manifesto de Envio para o Enviando microserviço pelo Depósito, mas fazer com que o Armazém seja

totalmente inconsciente da estrutura do próprio Manifesto de Envio. A Ordem
O processador envia o manifesto como parte da solicitação do pedido, mas o
Warehouse não faz nenhuma tentativa de examinar ou processar o campo - ele apenas o trata
como uma bolha de dados e não se importa com o conteúdo. Em vez disso, ele apenas envia
junto. Uma mudança no formato do Manifesto de Envio ainda aconteceria...
exigir uma alteração no Processador de Pedidos e no Envio
microsserviço, mas como o Warehouse não se importa com o que realmente está no
manifesto, não precisa mudar.

Acoplamento comum

O acoplamento comum ocorre quando dois ou mais microsserviços fazem uso de um
conjunto comum de dados. Um exemplo simples e comum dessa forma de acoplamento
seriam vários microsserviços fazendo uso do mesmo banco de dados compartilhado,
mas também pode se manifestar no uso de memória compartilhada ou compartilhada
sistema de arquivos.

O principal problema com o acoplamento comum é que as mudanças na estrutura do
os dados podem impactar vários microsserviços ao mesmo tempo. Considere o exemplo de
alguns dos serviços da MusicCorp na Figura 2-7. Como discutimos anteriormente,
A MusicCorp opera em todo o mundo, por isso precisa de várias informações
sobre os países em que opera. Aqui, vários serviços são todos
lendo dados de referência estáticos de um banco de dados compartilhado. Se o esquema deste
banco de dados alterado de forma incompatível com versões anteriores, exigiria alterações
para cada consumidor do banco de dados. Na prática, dados compartilhados como esses tendem a
Como resultado, será muito difícil mudar.

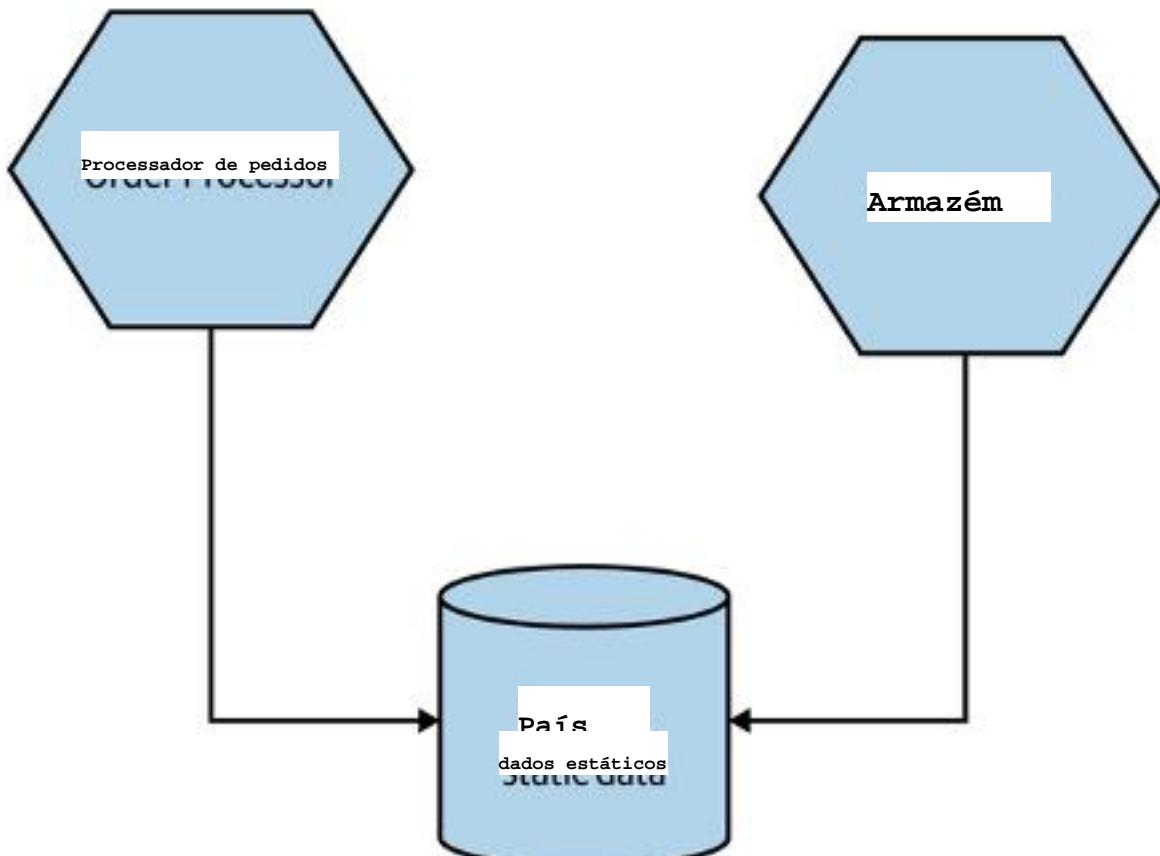


Figura 2-7. Vários serviços acessando dados de referência estáticos compartilhados relacionados a países do mesmo banco de dados

O exemplo na Figura 2-7 é, relativamente falando, bastante benigno. Isso é porque, por sua própria natureza, os dados de referência estáticos não tendem a mudar com frequência, e também porque esses dados são apenas para leitura – como resultado, costumo ficar relaxado compartilhando dados de referência estáticos dessa forma. O acoplamento comum se torna mais problemático, porém, se a estrutura dos dados comuns mudar mais frequentemente, ou se vários microsserviços estiverem lendo e gravando no mesmo dado.

A Figura 2-8 mostra uma situação em que o Processador de Pedidos e o serviço de armazém estão lendo e escrevendo a partir de uma tabela de pedidos compartilhada para ajudar a gerenciar o processo de envio de CDs aos clientes da MusicCorp. Ambos os microsserviços estão atualizando a coluna Status. O processador de pedidos pode definir os status COLOCADO, PAGO e CONCLUÍDO, enquanto o Depósito aplicará os status PICKING ou SHIPPED.

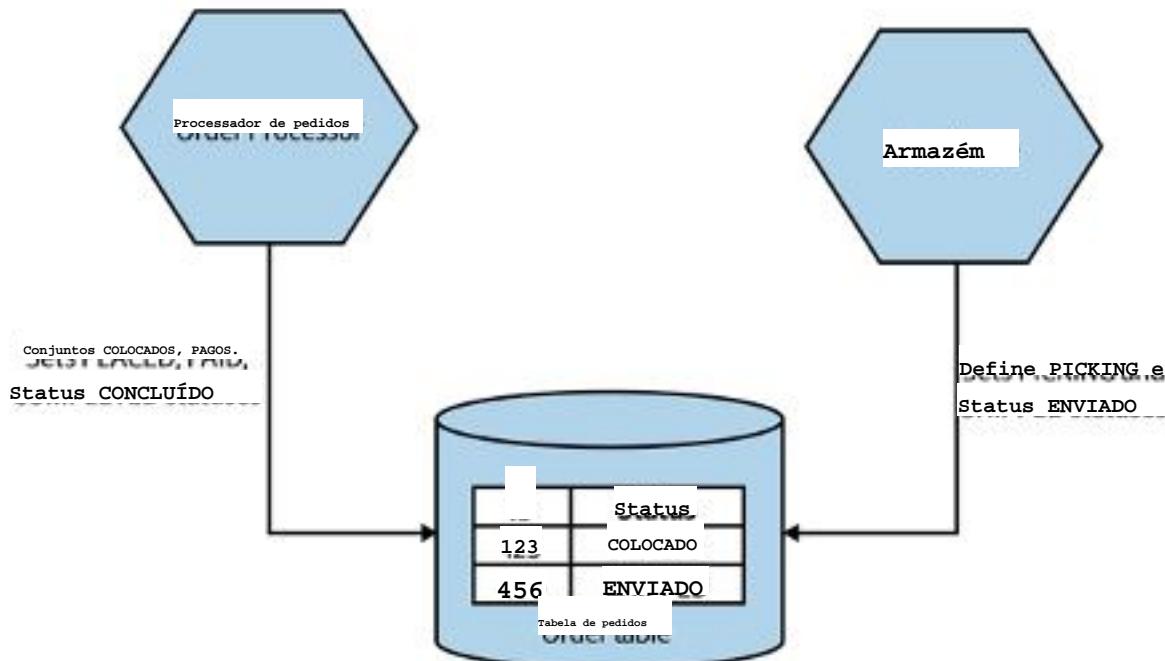


Figura 2-8. Um exemplo de acoplamento comum em que o Order Processor e o Warehouse estão atualizando o mesmo registro de pedido

Embora você possa considerar a Figura 2-8 um tanto artificial, esta no entanto, um exemplo simples de acoplamento comum ajuda a ilustrar um problema central. Conceitualmente, temos o Processador de Pedidos e Microsserviços de armazém gerenciando diferentes aspectos do ciclo de vida de um pedido. Ao fazer alterações no Processador de Pedidos, posso ter certeza de que estou não alterando os dados do pedido de forma a quebrar a visão do Warehouse sobre o mundo, ou vice-versa?

Uma forma de garantir que o estado de algo seja alterado de forma correta seria criar uma máquina de estados finitos. Uma máquina de estado pode ser usada para gerenciar a transição de alguma entidade de um estado para outro, garantindo transições de estado inválidas são proibidas. Na Figura 2-9, você pode ver o permitiu transições de estado para um pedido na MusicCorp. Um pedido pode ir diretamente de COLOCADO para PAGO, mas não diretamente de COLOCADO para PICKING (este a máquina estatal provavelmente não seria suficiente para os negócios do mundo real processos envolvidos na compra e envio completos de mercadorias de ponta a ponta, mas queria dar um exemplo simples para ilustrar a ideia).

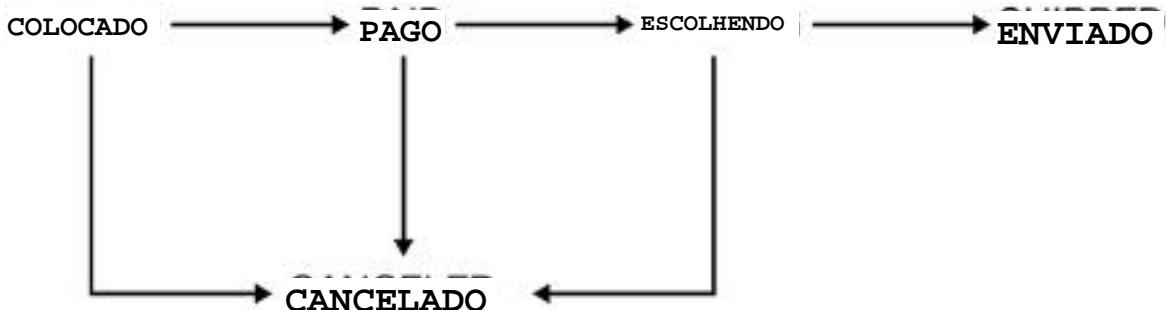
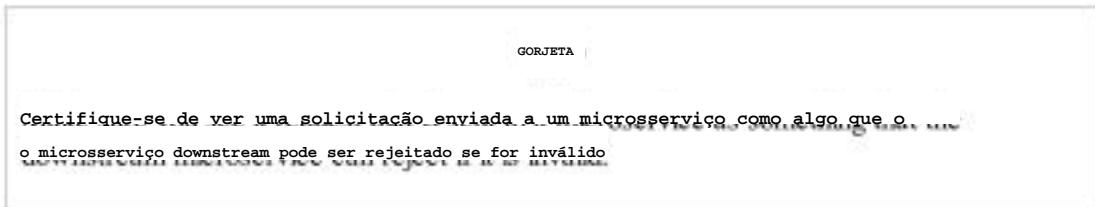


Figura 2-9. Uma visão geral das transições de estado permitidas para um pedido no MusicCorp.

O problema neste exemplo específico é que tanto o Depósito quanto o Pedido o processador compartilha as responsabilidades pelo gerenciamento dessa máquina de estado. Como fazer garantimos que eles estejam de acordo sobre quais transições são permitidas? Existem maneiras de gerenciar processos como esse em microsserviços limites; retornaremos a esse tópico quando discutirmos as sagas no Capítulo 6. Uma solução potencial aqui seria garantir que um único microsserviço gerencia o estado do pedido. Na Figura 2-10, Armazém ou Pedido. O processador pode enviar solicitações de atualização de status para o serviço de pedidos. Aqui, o microsserviço de pedidos é a fonte da verdade para qualquer pedido. Neste situação, é muito importante que vejamos as solicitações do Warehouse e o processador de pedidos é exatamente isso que solicita. Neste cenário, é o trabalho do Solicite um serviço para gerenciar as transições de estado aceitáveis associadas a um agregado de pedidos. Dessa forma, se o serviço de pedidos recebeu uma solicitação do pedido Processador para mover um status diretamente de COLOCADO para CONCLUÍDO, é gratuito para rejeite essa solicitação se for uma alteração inválida.

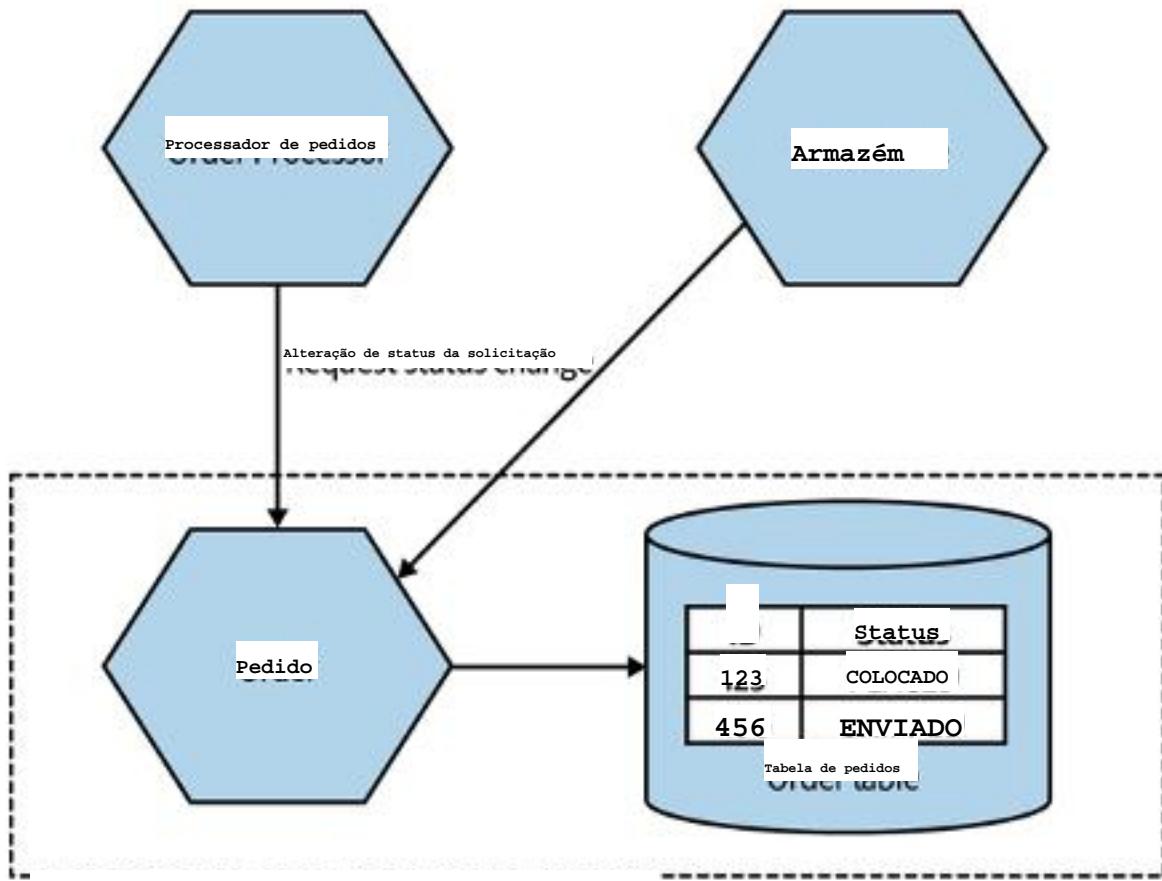


Uma abordagem alternativa que vejo nesses casos é implementar a Ordem serviço como pouco mais do que um invólucro das operações CRUD do banco de dados, onde as solicitações apenas são mapeadas diretamente para as atualizações do banco de dados. Isso é semelhante a um

objeto com campos privados, mas getters e setters públicos – o comportamento tem vazou do microsserviço para os consumidores upstream (reduzindo a coesão), e estamos de volta ao mundo do gerenciamento de transições de estado aceitáveis em vários serviços diferentes.

ADVERTÊNCIA

Se você ver um microsserviço que parece uma embalagem fina em torno do banco de dados CRUD operações, isso é um sinal de que você pode ter uma coesão fraca e um acoplamento mais estreito, como lógica que deveria estar nesse serviço para gerenciar os dados, em vez disso, está espalhado em outro lugar em seu sistema.



O serviço de pedidos pode rejeitar alterações de status inválidas

Figura 2-10. Tanto o Processador de Pedidos quanto o Depósito podem solicitar que alterações sejam feitas em um pedido, mas o microsserviço Order decide quais solicitações são aceitáveis.

Fontes de acoplamento comum também são fontes potenciais de recursos contention. Vários microserviços usando o mesmo sistema de arquivos ou o banco de dados pode sobrecarregar esse recurso compartilhado, potencialmente causando efeitos significativos problemas se o recurso compartilhado ficar lento ou até mesmo totalmente indisponível.

Um banco de dados compartilhado é especialmente propenso a esse problema, pois vários consumidores pode executar consultas arbitrárias no próprio banco de dados, o que, por sua vez, pode ter características de desempenho extremamente diferentes. Eu vi mais de um consulta SQL cara - talvez eu tenha até foi o culpado uma ou duas vezes. 10

Então, o acoplamento comum às vezes é bom, mas muitas vezes não é. Mesmo quando é benigno, significa que estamos limitados em quais mudanças podem ser feitas nos dados compartilhados, mas geralmente falam da falta de coesão em nosso código. Também pode nos causam problemas em termos de contenção operacional. É por esses motivos que consideramos o acoplamento comum uma das formas menos desejáveis de acoplamento, mas pode piorar.

Acoplamento de conteúdo

O acoplamento de conteúdo descreve uma situação na qual um serviço upstream alcança entra na parte interna de um serviço downstream e altera seu estado interno. O a manifestação mais comum disso é um serviço externo acessando outro banco de dados do microserviço e sua alteração direta. As diferenças entre o acoplamento de conteúdo e o acoplamento comum são sutis. Em ambos os casos, dois ou mais microserviços estão lendo e gravando no mesmo conjunto de dados. Com comum acoplamento, você entende que está fazendo uso de um acoplamento externo compartilhado dependência. Você sabe que não está sob seu controle. Com o acoplamento de conteúdo, as linhas de propriedade se tornam menos claras e fica mais difícil para desenvolvedores para mudar um sistema.

Vamos revisitar nosso exemplo anterior da MusicCorp. Na Figura 2-11, temos um serviço de pedidos que deve gerenciar as mudanças de estado permitidas para pedidos em nosso sistema. O Processador de Pedidos está enviando solicitações para o Solicite um serviço, delegando não apenas a mudança exata de estado que será feita mas também a responsabilidade de decidir quais transições de estado são permitidas. Ligado

Por outro lado, o serviço Warehouse está atualizando diretamente a tabela na qual os dados do pedido são armazenados, ignorando qualquer funcionalidade no serviço de pedidos que pode verificar se há alterações permitidas. Temos que esperar que o Armazém o serviço tem um conjunto consistente de lógica para garantir que somente as alterações válidas sejam feito. Na melhor das hipóteses, isso representa uma duplicação da lógica. Na pior das hipóteses, o verificar as mudanças permitidas no Warehouse é diferente daquela no Serviço de pedidos e, como resultado, podemos acabar com pedidos muito estranhos, estados confusos...

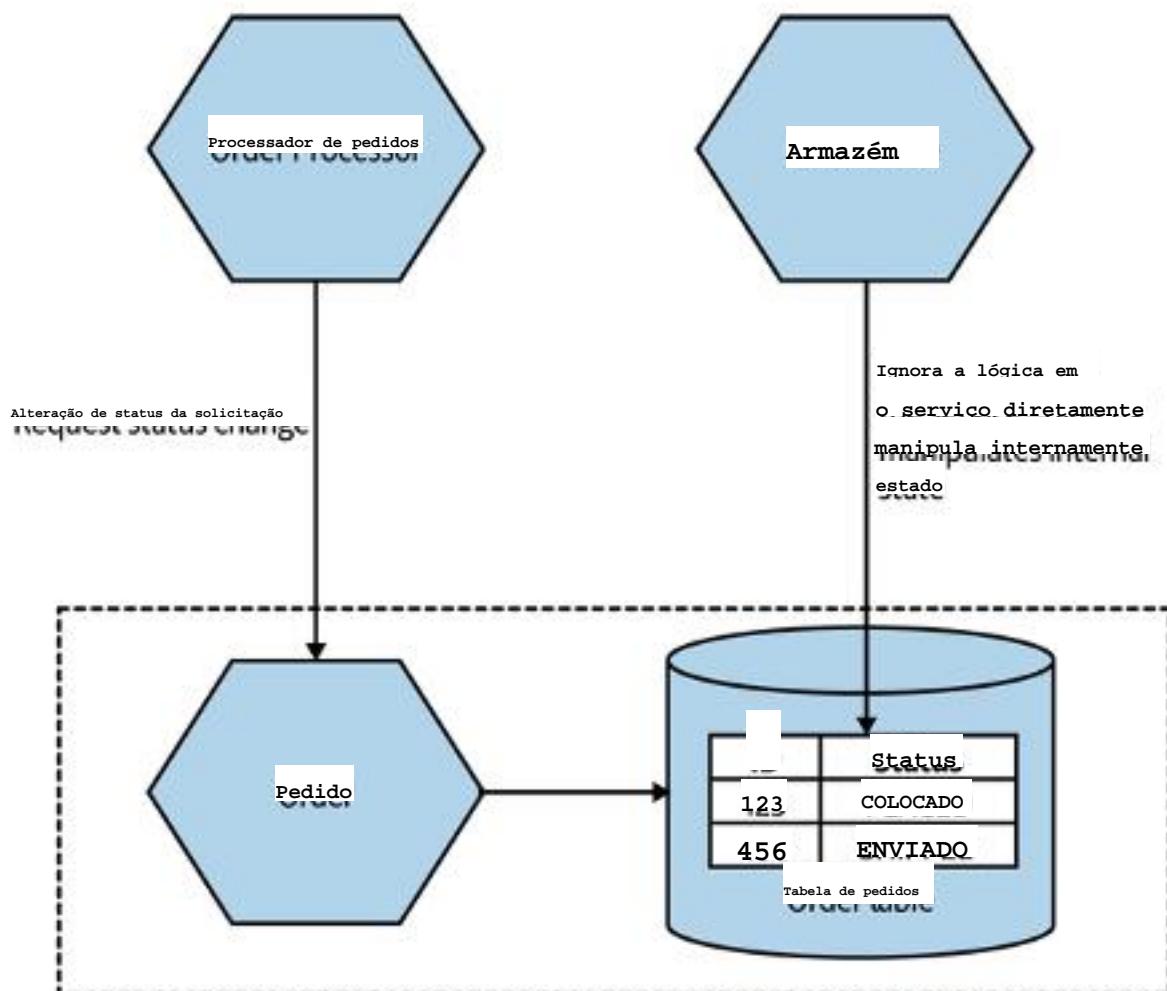


Figura 2-11. Um exemplo de acoplamento de conteúdo no qual o Depósito está acessando diretamente o dados internos do serviço de pedidos.

Nessa situação, também temos o problema de que a estrutura interna de dados do nosso a mesa de pedidos é exposta a uma pessoa externa. Ao alterar o serviço de pedidos, agora temos que ser extremamente cuidadosos ao fazer alterações nesse particular

tabela - isso mesmo assumindo que é óbvio para nós que essa tabela está sendo direta
acessado por uma parte externa. A solução mais fácil aqui é ter o Depósito
envie solicitações para o próprio serviço de pedidos, onde podemos examinar a solicitação, mas
também oculta os detalhes internos, fazendo alterações subsequentes no serviço de pedidos
muito mais fácil.

Se você está trabalhando em um microsserviço, é vital que você tenha uma visão clara
separação entre o que pode ser mudado livremente e o que não pode. Ser
explícito, como desenvolvedor, você precisa saber quando está mudando
funcionalidade que faz parte do contrato que seu serviço expõe ao exterior
mundo. Você precisa garantir que, se fizer alterações, não quebre
consumidores upstream. Funcionalidade que não afeta o contrato
as exposições de microsserviços podem ser alteradas sem preocupação.

Certamente, os problemas que ocorrem com o acoplamento comum
também se aplica ao acoplamento de conteúdo, mas o acoplamento de conteúdo tem alguns adicionais
dores de cabeça que o tornam problemático o suficiente para que algumas pessoas se refiram a ele como
acoplamento patológico.

Quando você permite que uma pessoa externa acesse diretamente seu banco de dados, o
O banco de dados em vigor se torna parte desse contrato externo, embora seja um em que
você não pode raciocinar facilmente sobre o que pode ou não ser alterado. Você perdeu
a capacidade de definir o que é compartilhado (e, portanto, não pode ser alterado facilmente),
e o que está oculto. As informações escondidas saíram pela janela.

Resumindo, evite o acoplamento de conteúdo.

Design orientado por domínio suficiente

Conforme apresentei no Capítulo 1, o mecanismo primário que usamos para encontrar
os limites do microsserviço estão em torno do próprio domínio, fazendo uso do domínio
design orientado (DDD) para ajudar a criar um modelo do nosso domínio. Vamos agora estender
nossa compreensão de como o DDD funciona no contexto de microsserviços.

O desejo de que nossos programas representem melhor o mundo real no qual
eles vão operar não é novidade. Linguagens de programação orientadas a objetos, como

Os simulados foram desenvolvidos para nos permitir modelar domínios reais. Mas é preciso mais do que recursos de linguagem de programa para que essa ideia realmente tome forma.

O Domain-Driven Design de Eric Evans apresentou uma série de ideias importantes que nos ajudaram a representar melhor o domínio do problema em nossos programas. Um completo a exploração dessas ideias está fora do escopo deste livro, mas existem alguns conceitos básicos do DDD que valem a pena destacar, incluindo:

Linguagem onipresente

Definindo e adotando uma linguagem comum para ser usada em código e em descrevendo o domínio, para auxiliar na comunicação

Agregar

Uma coleção de objetos que são gerenciados como uma única entidade, normalmente referindo-se a conceitos do mundo real.

Contexto limitado

Um limite explícito dentro de um domínio comercial que fornece funcionalidade para o sistema mais amplo, mas isso também esconde a complexidade.

Linguagem onipresente

A linguagem ubíqua se refere à ideia de que devemos nos esforçar para usar a mesma termos em nosso código conforme os usuários usam. A ideia é que ter um comum a linguagem entre a equipe de entrega e as pessoas reais facilitará as coisas para modelar o domínio do mundo real e também deve melhorar a comunicação

Como contraexemplo, lembro-me de uma situação em que trabalhava em uma grande empresa global banco. Estávamos trabalhando na área de liquidez corporativa, um termo sofisticado que basicamente se refere à capacidade de movimentar dinheiro entre diferentes contas mantidas por a mesma entidade corporativa. Foi muito bom trabalhar com o proprietário do produto, e ela tinha uma compreensão incrivelmente profunda dos vários produtos que ela queria levar ao mercado. Ao trabalhar com ela, teríamos discussões sobre coisas como cortes de cabelo e varreduras de fim de dia, todas as coisas que fazia muito sentido em seu mundo e isso tinha significado para seus clientes.

O código, por outro lado, não tinha nenhuma dessas linguagens. Em algum momento anteriormente, foi tomada a decisão de usar um modelo de dados padrão para o banco de dados. Foi amplamente referido como "o modelo bancário da IBM", mas eu vim para saber se esse era um produto padrão da IBM ou apenas a criação de um consultor da IBM. Ao definir o conceito vago de um "arranjo", a teoria era que qualquer operação bancária poderia ser modelada. Retirando um empréstimo? Isso foi um acordo. Comprando uma ação? Isso é um acordo! Solicitando um cartão de crédito? Adivinha: isso também é um acordo!

O modelo de dados poluiu o código a tal ponto que a base de código foi sem qualquer compreensão real do sistema que estávamos construindo. Nós não éramos construindo um aplicativo bancário genérico. Estábamos construindo um sistema especificamente para gerenciar a liquidez corporativa. O problema era que precisávamos mapear a rica linguagem de domínio do proprietário do produto para o código genérico conceitos, o que significa muito trabalho para ajudar a traduzir nossos analistas de negócios muitas vezes estavam apenas gastando seu tempo explicando os mesmos conceitos repetidamente mais uma vez como resultado.

Ao trabalhar a linguagem do mundo real no código, as coisas se tornaram muito mais fácil. Um desenvolvedor escolhendo uma história escrita usando os termos que surgiram diretamente do proprietário do produto, era muito mais provável que entendesse seus significado e descobrindo o que precisava ser feito.

Agregar

No DDD, um agregado é um conceito um tanto confuso, com muitos conceitos diferentes definições disponíveis. É apenas uma coleção arbitrária de objetos? O menor unidade que deve ser retirada de um banco de dados? O modelo que sempre foi funcionou para mim é primeiro considerar um agregado como uma representação de um real conceito de domínio - pense em algo como um pedido, uma fatura, um item de estoque, e assim por diante. Os agregados normalmente têm um ciclo de vida em torno deles, que abre eles até serem implementados como uma máquina de estado.

Como exemplo, no domínio MusicCorp, um agregado de pedidos pode conter vários itens de linha que representam os itens no pedido. Esses itens de linha têm significado apenas como parte do agregado geral do Pedido.

Queremos tratar os agregados como unidades independentes; queremos garantir que o código que lida com as transições de estado de um agregado é agrupado juntos, junto com o próprio estado. Portanto, um agregado deve ser gerenciado por um microserviço, embora um único microserviço possa possuir o gerenciamento de vários agregados.

Em geral, porém, você deve pensar em um agregado como algo que tem estado, identidade, um ciclo de vida que será gerenciado como parte do sistema. Os agregados geralmente se referem a conceitos do mundo real.

Um único microserviço cuidará do ciclo de vida e do armazenamento de dados de um ou tipos mais diferentes de agregados. Se a funcionalidade de outro serviço quiser altere um desses agregados, ele precisa solicitar diretamente uma alteração em esse agregado ou então fazer com que o próprio agregado reaja a outras coisas no sistema para iniciar suas próprias transições de estado, talvez assinando eventos emitido por outros microserviços.

A principal coisa a entender aqui é que, se uma parte externa solicitar um estado transição em um agregado, o agregado pode dizer não. Idealmente, você deseja implemente seus agregados de forma que as transições ilegais de estado sejam impossível.

Os agregados podem ter relacionamentos com outros agregados. Na Figura 2-12, nós ter um agregado de clientes associado a um ou mais pedidos e uma ou mais listas de desejos. Esses agregados poderiam ser gerenciados pelo mesmo microserviço ou por diferentes microserviços.

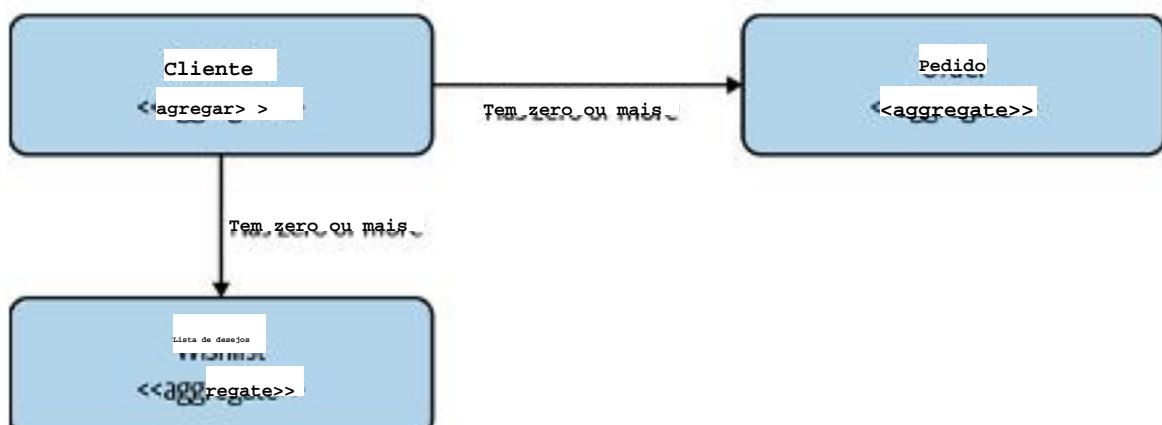


Figura 2-12. Um agregado de clientes pode estar associado a um ou mais pedidos ou listas de desejos

agregados

Se essas relações entre agregados existirem dentro do escopo de um único microserviço, eles poderiam ser facilmente armazenados usando algo como uma chave estrangeira relacionamento se estiver usando um banco de dados relacional. Se as relações entre esses os agregados abrangem os limites dos microserviços, no entanto, precisamos de alguma forma de modele os relacionamentos.

Agora, poderíamos simplesmente armazenar o ID do agregado diretamente em nosso local banco de dados. Por exemplo, considere um microserviço financeiro que gerencia um livro financeiro, que armazena transações contra um cliente. Localmente, dentro no banco de dados do microserviço financeiro, poderíamos ter uma coluna CustID que contém o ID desse cliente. Se quiséssemos obter mais informações sobre esse cliente, teríamos que fazer uma pesquisa com o microserviço do cliente usando esse ID.

O problema com esse conceito é que ele não é muito explícito - na verdade, o a relação entre a coluna CustID e o cliente remoto é inteiramente implícita. Para saber como esse ID estava sendo usado, teríamos que examinar o código do próprio microserviço financeiro. Seria bom se pudéssemos armazenar um referência a um agregado estrangeiro de uma forma mais óbvia.

Na Figura 2-13, mudamos as coisas para tornar a relação explícita. Em vez de um ID básico para a referência do cliente, armazenamos um URI, que podemos usar ao construir um sistema baseado em REST. 12

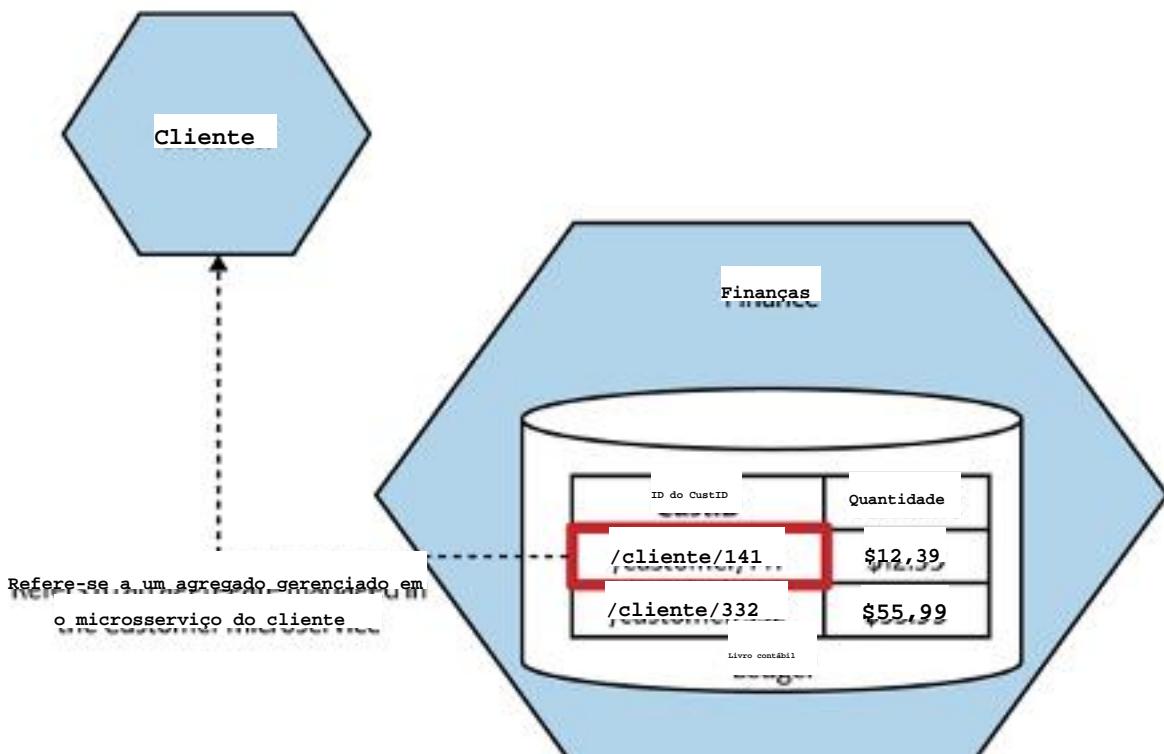


Figura 2-13. Um exemplo de como um relacionamento entre dois agregados em microserviços diferentes pode ser implementado.

Os benefícios dessa abordagem são duplos. A natureza do relacionamento é explícito e, em um sistema REST, poderíamos desreferenciar diretamente esse URI para procure o recurso associado. Mas e se você não estiver construindo um REST sistema? Phil Calcado descreve uma variação dessa abordagem em uso em SoundCloud,¹³ onde eles desenvolveram um esquema pseudo-URI para cruzamento referências de serviços. Por exemplo, faixas do soundcloud: 123 seriam um referência a uma faixa com o ID 123. Isso é muito mais explícito para um humano olhando para esse identificador, mas também é um esquema útil o suficiente para seria fácil imaginar a criação de código que pudesse facilitar o microserviço cruzado pesquisas agregadas, se necessário.

Há várias maneiras de dividir um sistema em agregados, com algumas opções sendo altamente subjetivo. Você pode decidir, por motivos de desempenho ou por facilidade de implementação, para remodelar agregados ao longo do tempo. Eu considero no entanto, a implementação deve ser secundária; começo deixando que o modelo mental dos usuários do sistema seja minha luz orientadora no design inicial até outros fatores entram em jogo.

Contexto limitado

Um contexto limitado normalmente representa um limite organizacional maior.

Dentro do escopo desse limite, responsabilidades explícitas precisam ser assumidas fora. Tudo isso é um pouco confuso, então vamos dar uma olhada em outro exemplo específico.

Na MusicCorp, nosso armazém é uma colmeia de pedidos de gerenciamento de atividades

enviado (e devolução ímpar), recebendo novo estoque, tendo empilhadeira

corridas de caminhões e assim por diante. Em outros lugares, o departamento financeiro talvez seja menos divertido-

amar, mas ainda tem uma função importante dentro de nossa organização, lidar

folha de pagamento, pagamento de remessas e afins.

Os contextos limitados ocultam os detalhes da implementação. Existem preocupações internas-

por exemplo, os tipos de empilhadeiras usados são de pouco interesse para qualquer pessoa

além das pessoas no armazém. Essas preocupações internas devem ser

escondido do mundo exterior, que não precisa saber, nem deveria

cuidado.

Do ponto de vista da implementação, os contextos delimitados contêm um ou mais

agregados. Alguns agregados podem ser expostos fora do contexto limitado.

outros podem estar ocultos internamente. Assim como acontece com os agregados, os contextos limitados podem

ter relacionamentos com outros contextos limitados, quando mapeados para serviços,

essas dependências se tornam dependências entre serviços.

Vamos voltar por um momento aos negócios da MusicCorp. Nossa domínio é o

todo o negócio em que estamos operando. Abrange tudo, desde o

armazém até a recepção, do financiamento ao pedido. Nós podemos ou podemos

não modelamos tudo isso em nosso software, mas esse ainda é o domínio em

que estamos operando. Vamos pensar em partes desse domínio que parecem

os contextos limitados aos quais Eric Evans se refere.

Modelos ocultos

Para a MusicCorp, podemos considerar o departamento financeiro e o armazém

ser dois contextos delimitados separados. Ambos têm uma interface explícita para

o mundo exterior (em termos de relatórios de inventário, recibos de pagamento, etc.), e eles

têm detalhes que só eles precisam conhecer (empilhadeiras, calculadoras).

O departamento financeiro não precisa conhecer os detalhes internos de funcionamento do armazém. No entanto, ele precisa saber algumas coisas, pois Por exemplo, ele precisa saber sobre os níveis de estoque para manter as contas atualizadas. A Figura 2-14 mostra um exemplo de diagrama de contexto. Vemos conceitos que são interno ao armazém, como um selecionador (alguém que escolhe pedidos), prateleiras que representam locais de estoque e assim por diante. Da mesma forma, entradas no geral os livros contábeis são essenciais para as finanças, mas não são compartilhados externamente aqui.

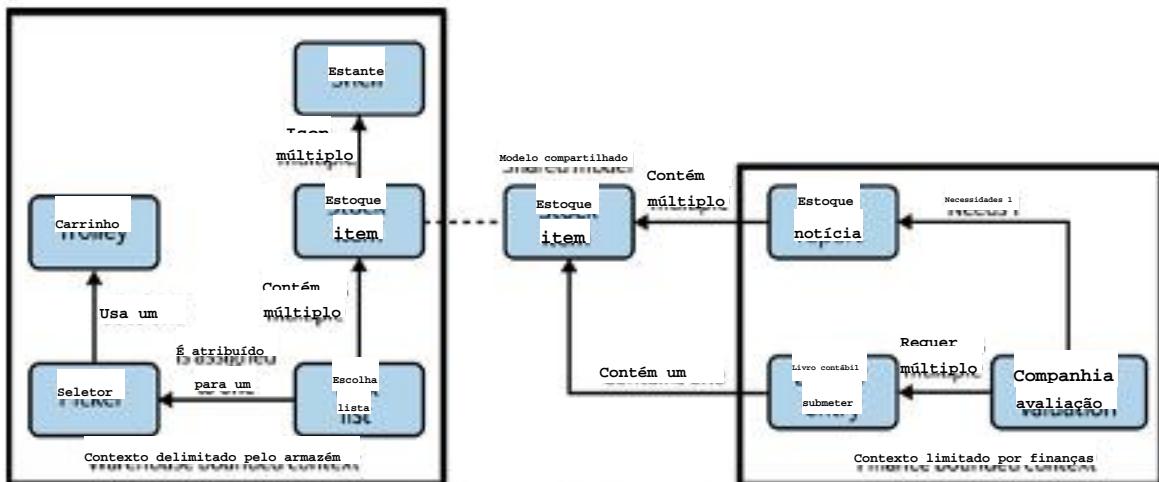


Figura 2-14. Um modelo compartilhado entre o departamento financeiro e o armazém

Para poder calcular a avaliação da empresa, porém, as finanças os funcionários precisam de informações sobre as ações que mantemos. O item de estoque então torna-se um modelo compartilhado entre os dois contextos. No entanto, observe que nós não precisa expor cegamente tudo sobre o item de estoque do contexto de armazém. Na Figura 2-15, vemos como o item de estoque está dentro do o contexto limitado do armazém contém referências aos locais das prateleiras, mas o a representação compartilhada contém apenas uma contagem. Portanto, existe apenas o interno representação e a representação externa que expomos. Muitas vezes, quando você ter representações internas e externas diferentes, pode ser benéfico para nomeie-os de forma diferente para evitar confusão - nesta situação, uma abordagem poderia ser chamar o item de estoque compartilhado de contagem de estoque.

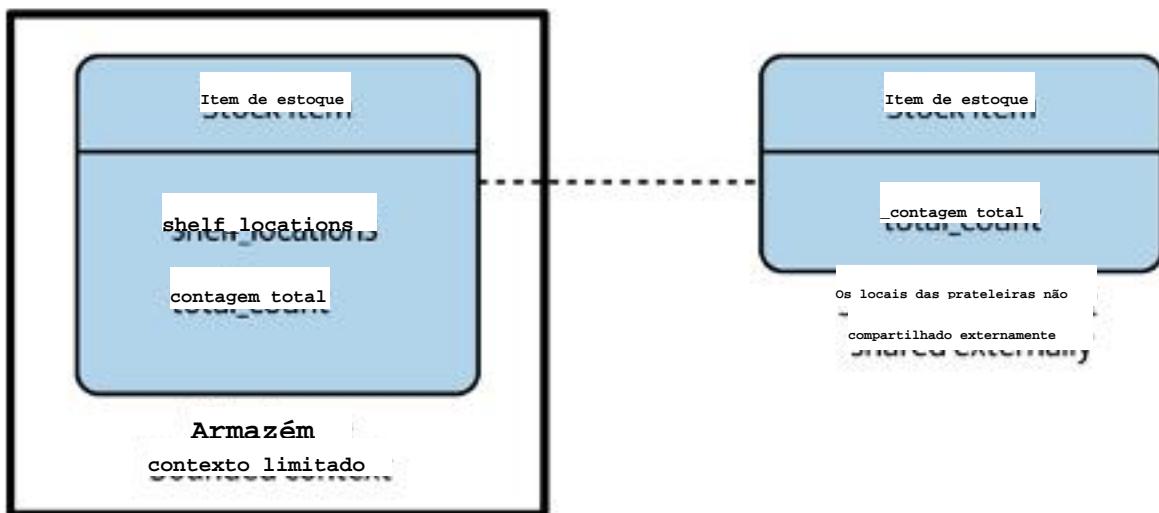


Figura 2-15. Um modelo compartilhado pode decidir ocultar informações que não devem ser compartilhadas externamente

Modelos compartilhados

Também podemos ter conceitos que aparecem em mais de um contexto limitado. Em Na Figura 2-14, vimos que existe um cliente nos dois locais. O que faz isso quer dizer? O cliente foi copiado? A maneira de pensar sobre isso é que conceitualmente, tanto as finanças quanto o armazém precisam saber algo sobre nosso cliente. As finanças precisam saber sobre os pagamentos financeiros feitos a um cliente, enquanto o armazém precisa saber sobre o cliente para o na medida em que sabe quais pacotes foram enviados para permitir entregas a serem rastreadas.

Quando você tem uma situação como essa, um modelo compartilhado como o cliente pode ter significados diferentes nos diferentes contextos limitados e, portanto, podem ser chamadas de coisas diferentes. Talvez fiquemos felizes em manter o nome "cliente" em finanças, mas no armazém podemos chamá-los de "destinatários", pois esse é o papel que desempenham nesse contexto. Armazenamos informações sobre o cliente em ambos localizações, mas as informações são diferentes. Finanças armazena informações sobre os pagamentos financeiros (ou reembolsos) do cliente; as lojas do armazém informações relacionadas às mercadorias enviadas. Talvez ainda precisemos vincular os dois locais conceitos para um cliente global, e talvez queiramos procurar conceitos comuns e compartilhados informações sobre esse cliente, como nome ou endereço de e-mail - poderíamos usar uma técnica como a mostrada na Figura 2-13 para conseguir isso.

Mapeando agregados e contextos limitados para Microsserviços

Tanto o contexto agregado quanto o limitado nos fornecem unidades de coesão com interfaces bem definidas com o sistema mais amplo. O agregado é um eu máquina de estado contida que se concentra em um único conceito de domínio em nosso sistema, com o contexto limitado representando uma coleção de associados que agrega, novamente, com uma interface explícita para o resto do mundo.

Portanto, ambos podem funcionar bem como limites de serviço. Ao começar, como já mencionei que você deseja reduzir o número de serviços em que trabalha com. Como resultado, você provavelmente deve segmentar serviços que abranjam toda a sua base de usuários em contextos limitados. Enquanto você se recupera e decide interromper esses serviços em serviços menores, é preciso lembrar que os agregados em si não querem ser divididos em um microsserviço. Um agregado pode gerenciar um ou mais agregados, mas não queremos que um agregado seja gerenciado por mais de um microsserviço.

Tartarugas até o fim

No início, você provavelmente identificará vários limites de granulação grossa de contextos. Mas esses contextos limitados, por sua vez, podem conter limites adicionais de contextos. Por exemplo, você pode decompor o armazém em capacidades associadas ao atendimento de pedidos, gerenciamento de estoque ou recebimento de mercadorias. Ao considerar os limites de seus microsserviços, pense primeiro em termos dos contextos maiores e mais grosseiros e, em seguida, subdividida ao longo desses contextos aninhados de contextos em que você está procurando os benefícios de dividir essas costuras.

Um truque aqui é que, mesmo que você decida dividir um serviço, ele modela um inteiro contexto limitado em serviços menores posteriormente, você ainda pode ocultar isso da decisão do mundo exterior, talvez apresentando uma granulação mais grossa da API para consumidores. A decisão de decompor um serviço em partes menores é indiscutivelmente uma decisão de implementação, então podemos muito bem escondê-la, se pudermos.

Na Figura 2-16, vemos um exemplo disso. Dividimos o Warehouse em Inventário e envio. No que diz respeito ao mundo exterior, há ainda é apenas o microsserviço Warehouse. Porém, internamente, temos mais.

coisas decompostas para permitir que o inventário gerencie itens de estoque e tenha Envio e gerenciamento de remessas. Lembre-se de que queremos manter a propriedade de um único agregado dentro de um único microserviço.

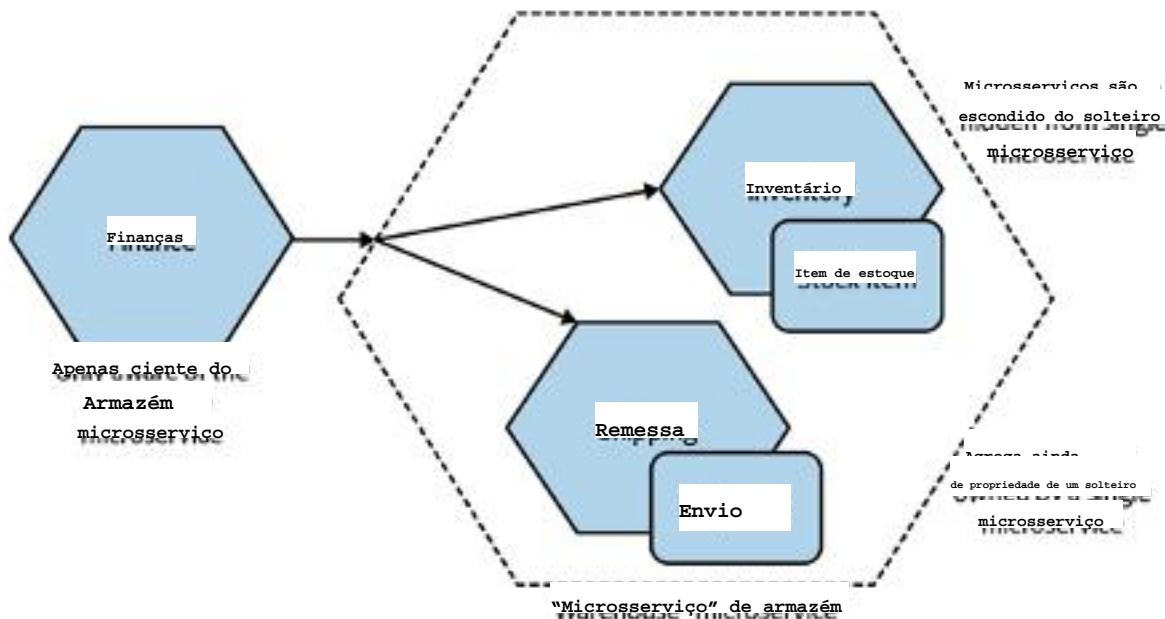


Figura 2-16. O serviço de Armazém internamente foi dividido em Estoque e Envio microserviços

Essa é outra forma de ocultação de informações - ocultamos uma decisão sobre implementação interna de tal forma que, se esse detalhe de implementação mudar novamente no futuro, nossos consumidores não saberão.

Outro motivo para preferir a abordagem aninhada pode ser dividir seu arquitetura para simplificar os testes. Por exemplo, ao testar serviços que consome o armazém, não preciso colocar cada serviço dentro do contexto de armazém - apenas a API mais abrangente. Isso também pode dar você é uma unidade de isolamento ao considerar testes de escopo maior. Eu posso, por exemplo, decidir fazer testes de ponta a ponta nos quais eu inicio todos os serviços dentro do contexto do armazém, mas para todos os outros colaboradores eu poderia substituí-los fora. Exploraremos mais sobre testes e isolamento no Capítulo 9.

Invasão de eventos

O Event Storming, uma técnica desenvolvida por Alberto Brandolini, é um exercício colaborativo de brainstorming desenvolvido para ajudar a revelar um domínio

representação própria do que é o modelo de domínio, 14 eventos de tempestade traz
reunir partes interessadas técnicas e não técnicas em um exercício conjunto. A ideia
é que, ao tornar o desenvolvimento do modelo de domínio uma atividade conjunta, você
acabam com uma visão compartilhada e conjunta do mundo.

Vale a pena mencionar neste momento que, embora os modelos de domínio sejam definidos
por meio de armazenamento de eventos pode ser usado para implementar sistemas orientados por eventos - e
de fato, o mapeamento é muito simples - você também pode usar esse domínio...
modelo para criar um sistema mais orientado à solicitação/resposta.

Logística

Alberto tem algumas opiniões muito específicas sobre como a invasão de eventos deve ser.
corro, e em alguns desses pontos estou muito de acordo. Em primeiro lugar, obtenha
todos em uma sala juntos. Geralmente, essa é a etapa mais difícil
alinhar calendários de pessoas pode ser um problema, assim como encontrar um grande o suficiente
sala. Esses problemas eram todos verdadeiros em um mundo pré-COVID, mas enquanto escrevo isso
durante o bloqueio relacionado a vírus no Reino Unido, estou ciente de que essa etapa pode
seja ainda mais problemático no futuro. A chave, porém, é ter tudo
partes interessadas presentes ao mesmo tempo. Você quer representantes de todas as partes
do domínio que você planeja modelar: usuários, especialistas no assunto, produto
proprietários - quem está em melhor posição para ajudar a representar cada parte do domínio.

Quando todos estiverem juntos em uma sala, Alberto sugere a remoção de todos...
cadeiras para garantir que todos se levantem e se envolvam. Como alguém com um problema
de volta, embora essa estratégia seja algo que eu entenda, reconheço que talvez não
trabalhe para todos. Uma coisa em que concordo com Alberto é a necessidade de
tenha um grande espaço onde a modelagem possa ser feita. Uma solução comum é
prenda grandes rolos de papel pardo nas paredes da sala, permitindo que todos os
paredes a serem usadas para capturar informações.

A principal ferramenta de modelagem são notas adesivas para capturar os vários conceitos, com
notas de cores diferentes representando conceitos diferentes.

O processo

O exercício começa com os participantes identificando os eventos do domínio. Esses representam coisas que acontecem no sistema - são os fatos de que você se importa sobre. "Pedido feito" seria um evento com o qual nos importaríamos no contexto da MusicCorp, assim como "Pagamento recebido". Eles são capturados em notas adesivas de laranja. É neste ponto que tenho outra discordância com A estrutura de Alberto, já que os eventos são, de longe, as coisas mais numerosas você estará capturando, e notas adesivas de laranja são surpreendentemente difíceis de conseguir de.¹⁵

Em seguida, os participantes identificam os comandos que fazem com que esses eventos aconteçam. UMA comando é uma decisão tomada por um humano (um usuário do software) para fazer alguma coisa. Aqui você está tentando entender os limites do sistema e identificar os principais atores humanos no sistema. Os comandos são capturados em azul notas adesivas.

Os técnicos da sessão de invasão do evento devem ouvir o que seus colegas não técnicos inventam aqui. Uma parte fundamental deste exercício não é para permitir que qualquer implementação atual distorça a percepção do que é o domínio (isso vem depois). Nesta fase, você deseja criar um espaço no qual possa tire os conceitos da cabeça das principais partes interessadas e divulgue-os abertamente.

Com eventos e comandos capturados, os agregados vêm em seguida. Os eventos que você neste estágio, não são úteis apenas para compartilhar, não apenas o que acontece no sistema, mas eles também começam a destacar quais podem ser os agregados potenciais. Pense no evento de domínio mencionado acima, "Pedido feito". O substantivo aqui -- "Pedido" - poderia muito bem ser um agregado potencial. E "Colocado" descreve algo que pode acontecer com um pedido, então isso pode muito bem fazer parte da vida ciclo do agregado. Os agregados são representados por notas adesivas amarelas, e os comandos e eventos associados a esse agregado são movidos e agrupados em torno do agregado. Isso também ajuda você a entender como os agregados estão relacionados entre si - os eventos de um agregado podem desencadear comportamento em outro.

Com os agregados identificados, eles são agrupados em contextos limitados. Os contextos limitados geralmente seguem a organização de uma empresa

estrutura, e os participantes do exercício estão bem posicionados para entender quais agregados são usados por quais partes da organização.

A invasão de eventos é mais do que o que acabei de descrever - isso foi apenas concebido como uma breve visão geral. Para uma visão mais detalhada da invasão de eventos, eu gostaria sugiro que você leia o livro (atualmente em andamento) EventStorming de Alberto Brandolini (Leanpub).¹⁶

O caso do design orientado por domínio para Microsserviços

Exploramos como o DDD pode funcionar no contexto de microsserviços, então vamos resumir como essa abordagem é útil para nós.

Em primeiro lugar, uma grande parte do que torna o DDD tão poderoso é que contextos limitados, que são tão importantes para o DDD, tratam explicitamente de esconder informações apresentando um limite claro para o sistema mais amplo, ao mesmo tempo em que oculta o interior complexidade que é capaz de mudar sem afetar outras partes do sistema. Isso significa que, quando seguimos uma abordagem de DDD, quer percebemos isso ou não, também estamos adotando a ocultação de informações - e, como vimos, isso é essencial para ajudar a encontrar limites estáveis de microsserviços.

Em segundo lugar, o foco na definição de uma linguagem comum e onipresente ajuda muito quando se trata de definir endpoints de microsserviços. Isso nos dá perfeitamente um compartilhamento vocabulário a ser usado ao criar APIs, formatos de eventos e o como. Também ajuda a resolver o problema de até que ponto a padronização das APIs está precisa ir em termos de permitir que a linguagem mude dentro de contextos limitados -mudança dentro de um limite impactando esse próprio limite.

As mudanças que implementamos em nosso sistema geralmente envolvem mudanças no as empresas querem entender como o sistema se comporta. Estamos mudando capacidades de funcionalidade - que são expostas aos nossos clientes. Se nossos sistemas são decompostos ao longo dos contextos limitados que representam nossos domínio, todas as alterações que desejamos fazer têm maior probabilidade de serem isoladas em um limite único de microsserviços. Isso reduz o número de lugares que precisamos fazer uma alteração e nos permita implantar essa mudança rapidamente.

Fundamentalmente, o DDD coloca o domínio comercial no centro do software que nós estamos construindo. O incentivo que isso nos dá para usar a linguagem do fazer negócios em nosso código e design de serviços ajuda a melhorar a experiência no domínio entre as pessoas que constroem o software. Isso, por sua vez, ajuda a construir compreensão e empatia pelos usuários de nosso software e construções maiores. comunicacão entre entrega técnica, desenvolvimento de produtos e fim usuários. Se você estiver interessado em migrar para equipes alinhadas ao fluxo, o DDD é adequado é perfeitamente um mecanismo para ajudar a alinhar a arquitetura técnica com a estrutura organizacional mais ampla. Em um mundo em que estamos cada vez mais tentando derrubar os silos entre a TI e "a empresa", isso não é ruim coisa.

Alternativas aos limites do domínio comercial

Como descrevi, o DDD pode ser incrivelmente útil ao criar microserviços arquiteturas, mas seria um erro pensar que essa é a única técnica você deve considerar ao encontrar limites de microserviços. Na verdade, eu frequentemente use vários métodos em conjunto com o DDD para ajudar a identificar como (e se) um sistema deve ser dividido. Vamos dar uma olhada em alguns dos outros fatores que podemos considerar ao encontrar limites.

Volatilidade

Cada vez mais ouço falar de resistência contra a decomposição orientada a domínios, frequentemente por defensores da volatilidade como o principal fator de decomposição. A decomposição baseada em volatilidade permite que você identifique as partes do seu sistema passando por mudanças mais frequentes e, em seguida, extrair essa funcionalidade em seus próprios serviços, onde eles podem ser trabalhados de forma mais eficaz. Conceitualmente, não tenho nenhum problema com isso, mas promovo-o como o único a maneira de fazer as coisas não ajuda, especialmente quando consideramos as diferentes drivers que podem estar nos impulsionando a adotar microserviços. Se meu maior problema é relacionado à necessidade de escalar meu aplicativo, por exemplo, um aplicativo baseado em volatilidade é improvável que a decomposição traga muitos benefícios.

A mentalidade por trás da decomposição baseada na volatilidade também é evidente em abordagens como TI bimodal. Um conceito apresentado pela Gartner, TI bimodal divide perfeitamente o mundo no alegremente chamado "Modo 1" (também conhecido como Categorias Sistemas de Registro) e "Modo 2" (também conhecida como Sistemas de Inovação) com base na velocidade (ou lentidão) que diferentes sistemas precisam operar. Sistemas de modo 1, nos dizem que não mude muito e não precise de muito envolvimento comercial. O Modo 2 é onde está a ação, com sistemas que precisam mudar rapidamente e que exigem um envolvimento próximo da empresa. Deixando de lado para um momento, a drástica simplificação excessiva inerente a tal categorização esquema, também implica uma visão muito fixa do mundo e desmente os tipos de transformações que são evidentes em todo o setor, à medida que as empresas buscam "seguir em frente" digital." Partes dos sistemas das empresas que não precisaram mudar muito no o passado de repente o faz, a fim de abrir novas oportunidades de mercado e fornecer serviços para seus clientes de maneiras que eles não imaginavam anteriormente.

Vamos voltar para a MusicCorp. É a primeira incursão no que agora chamamos de digital. estava apenas tendo uma página na web; tudo o que ela oferecia em meados dos anos noventa era um lista do que estava à venda, mas você teve que telefonar para a MusicCorp para colocar o pedido. Era pouco mais do que um anúncio em um jornal. Em seguida, pedidos on-line tornou-se uma coisa, e todo o armazém, que até aquele momento tinha acabado de foi manuseado com papel, teve que ser digitalizado. Quem sabe, talvez A MusicCorp, em algum momento, terá que considerar a disponibilização de músicas digitalmente! Embora você possa considerar que a MusicCorp está atrasada, você ainda pode apreciar a quantidade de turbulência que as empresas passaram avançando à medida que entendem como a tecnologia e o cliente estão mudando o comportamento pode exigir mudanças significativas em partes de uma empresa que não poderiam pode ser facilmente previsto.

Não gosto da TI bimodal como conceito, pois ela se torna uma forma de as pessoas abandonarem coisas que são difíceis de transformar em uma caixa bem arrumada e dizer "não precisamos lidar com os problemas aí contidos - esse é o Modo 1." É mais um modelo que um a empresa pode adotar para garantir que nada realmente tenha que mudar. Também evita o fato de que muitas vezes mudanças na funcionalidade exigem mudanças na "Sistemas de Registro" (Modo 1) para permitir mudanças em "Sistemas de

Inovação" (Modo 2). Em minha experiência, organizações adotando a TI bimodal acabam tendo duas velocidades - lenta e mais lenta.

Para ser justo com os proponentes da decomposição baseada na volatilidade, muitos deles não estão necessariamente recomendando modelos simplistas como a TI bimodal. Em Na verdade, acho que essa técnica é muito útil para ajudar a determinar limites se o principal fator for a rapidez de extração no mercado a funcionalidade que está mudando ou precisa mudar com frequência leva à perfeição sentido em tal situação. Mas, novamente, o objetivo determina o mais adequado mecanismo

Dados

A natureza dos dados que você mantém e gerencia pode levá-lo a diferentes formas de decomposição. Por exemplo, você pode querer limitar quais os serviços lidam com informações de identificação pessoal (PII), tanto para reduzir seu risco de violações de dados e para simplificar a supervisão e a implementação de coisas como o GDPR.

Para um dos meus clientes recentes, uma empresa de pagamento que chamaremos de PaymentCo, a uso de certos tipos de dados influenciou diretamente as decisões que tomamos sobre decomposição do sistema PaymentCo lida com dados de cartão de crédito, o que significa que seu sistema precisa estar em conformidade com vários requisitos estabelecidos por Padrões do setor de cartões de pagamento (PCI) sobre como esses dados precisam ser gerenciado. Como parte dessa conformidade, o sistema e os processos da empresa precisavam ser auditados. A PaymentCo precisava lidar com todo o cartão de crédito dados e em um volume que significava que seu sistema tinha que estar em conformidade com o PCI Nível 1, que é o nível mais rigoroso e que exige um externo trimestral avaliação dos sistemas e práticas relacionadas à forma como os dados são gerenciados.

Muitos dos requisitos do PCI são de bom senso, mas garantem que o todo o sistema atendeu a esses requisitos, principalmente a necessidade de o sistema ser auditado por uma parte externa, estava provando ser bastante oneroso. Como resultado, a empresa queria dividir a parte do sistema que lidava com a totalidade dados do cartão de crédito, o que significa que apenas um subconjunto do sistema exigia isso nível adicional de supervisão. Na Figura 2-17, vemos uma forma simplificada do

design que criamos. Servicos que operam na zona verde (delimitados por um
linha verde pontilhada) nunca veja nenhuma informação de cartão de crédito - esses dados são limitados
aos processos (e redes) na zona vermelha (cercados por tracos vermelhos). O
o gateway desvia as chamadas para os serviços apropriados (e para a zona apropriada);
à medida que as informações do cartão de crédito passam por esse gateway, elas também estão em vigor
na zona vermelha.

Como as informações do cartão de crédito nunca fluem para a zona verde, todos os serviços em
essa área pode ser isenta de uma auditoria completa do PCI. Os serviços na zona vermelha são
no escopo dessa supervisão. Ao trabalhar no design, fizemos
tudo o que pudemos para limitar o que tem que estar nessa zona vermelha. É fundamental anotar
que tínhamos que garantir que as informações do cartão de crédito nunca fluíssem para o
zona verde de qualquer forma - se um microsserviço na zona verde pudesse solicitar isso
informações, ou se essas informações puderem ser enviadas de volta para a zona verde por um
microsserviço na zona vermelha, então as linhas claras de separação se quebrariam
para baixo.

A segregação de dados geralmente é impulsionada por uma variedade de privacidade e segurança
preocupações; voltaremos a este tópico e ao exemplo da PaymentCo em
Capítulo 11.

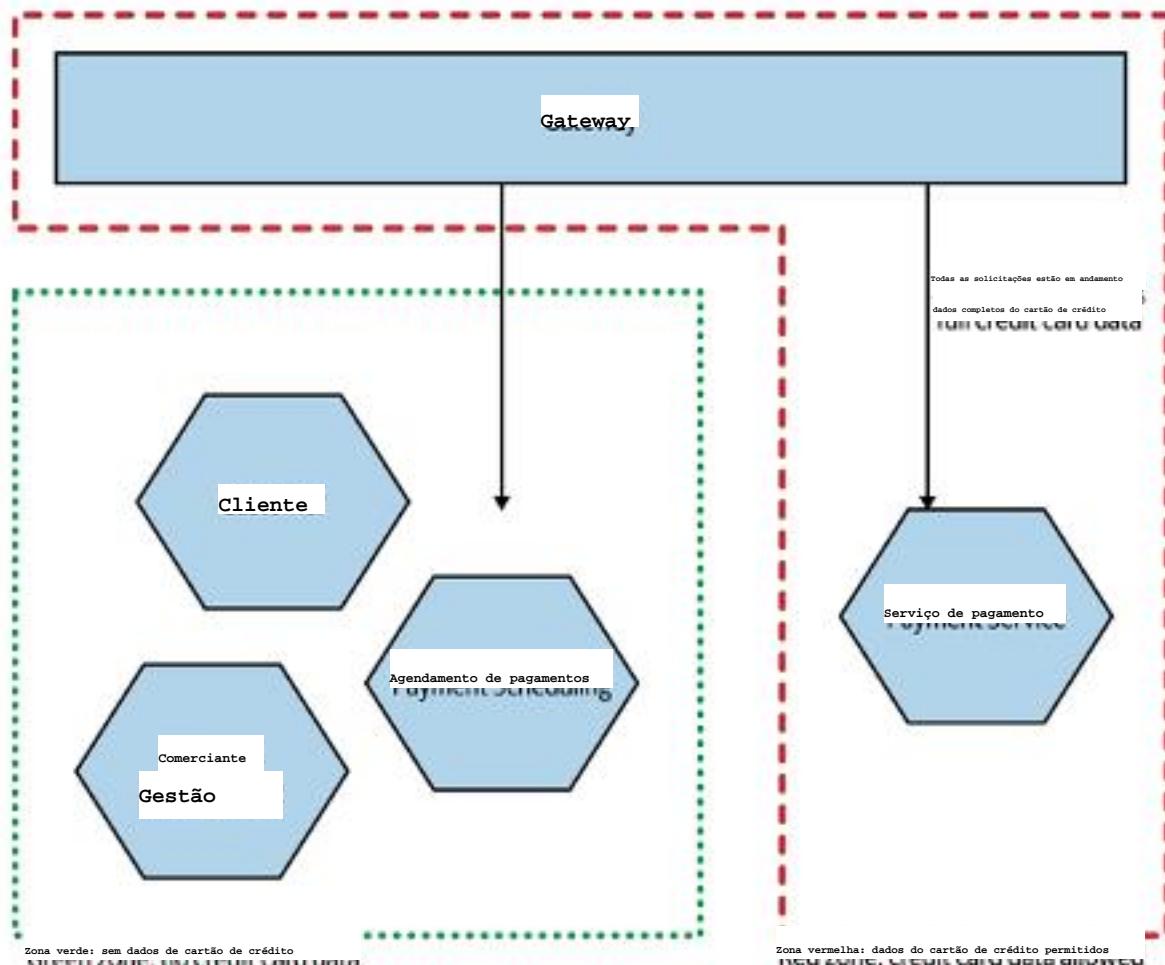


Figura 2-17 PaymentCo, que separa processos com base no uso de informações de cartão de crédito

para limitar o escopo dos requisitos do PCI

Tecnologia

A necessidade de fazer uso de diferentes tecnologias também pode ser um fator em termos de encontrando um limite. Você pode acomodar bancos de dados diferentes em um único executando microserviços, mas se você quiser misturar diferentes modelos de tempo de execução, pode enfrentar um desafio. Se você determinar que parte de sua funcionalidade precisa seja implementado em um tempo de execução como o Rust, que permite que você execute melhorias adicionais de desempenho, que acabam sendo uma grande força fator.

Obviamente, precisamos estar cientes de onde isso pode nos levar se adotado como meios gerais de decomposição. A arquitetura clássica de três camadas que discutimos no capítulo de abertura e que mostramos novamente na Figura 2-18,

é um exemplo de tecnologia relacionada sendo agrupada. Como nós temos ...
já explorada, essa geralmente é uma arquitetura menos do que ideal.

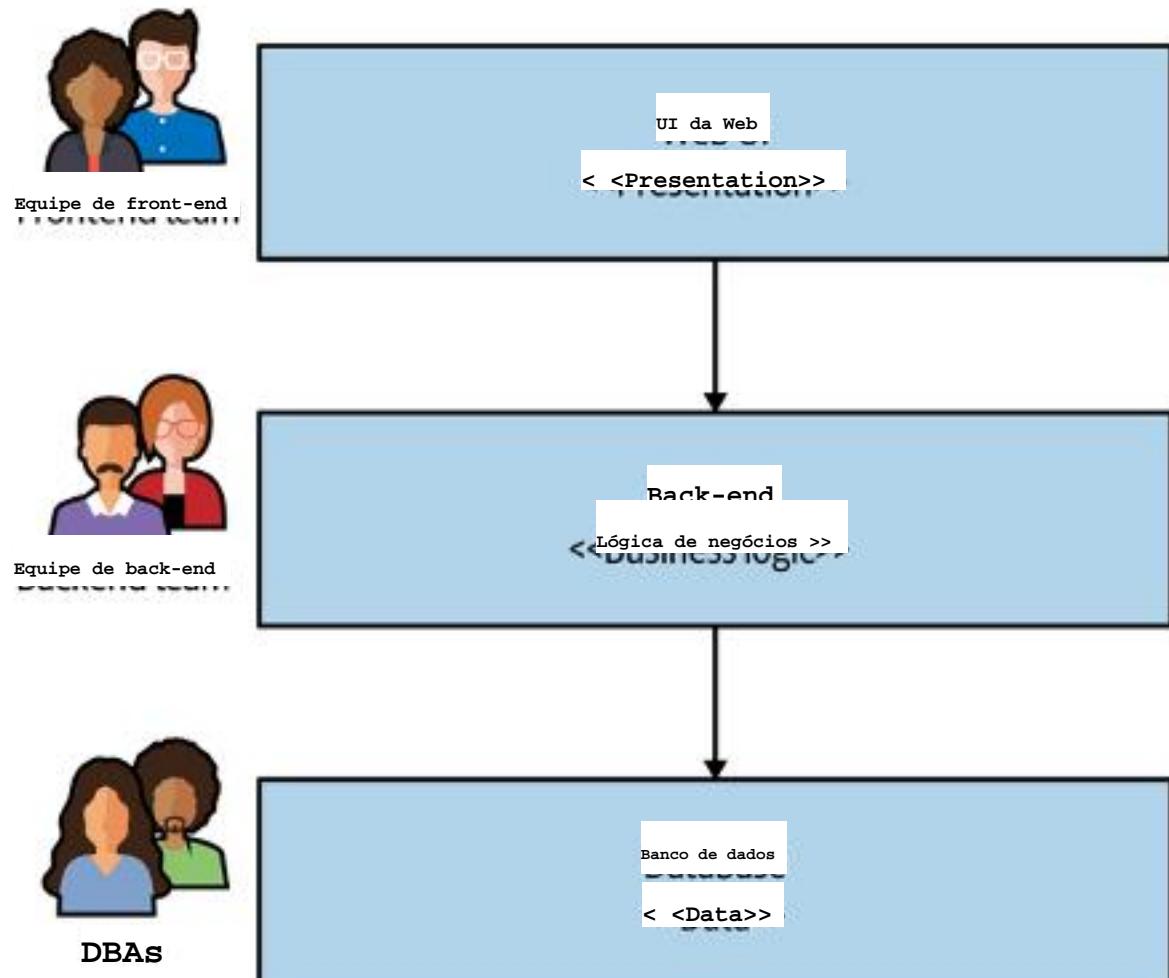


Figura 2-18. Uma arquitetura tradicional de três camadas geralmente é impulsionada por limites tecnológicos

Organizacional

Como estabelecemos quando apresentei a lei de Conway no Capítulo 1, há ...
é uma interação inerente entre a estrutura organizacional e o sistema ...
arquitetura com a qual você acaba. Além dos estudos que mostraram
neste link, em minha própria experiência anedótica, vi isso acontecer uma vez e
mais uma vez. A forma como você se organiza acaba impulsionando seus sistemas
arquitetura, para o bem ou para o mal. Quando se trata de nos ajudar a definir nossos
limites de serviço, temos que considerar isso como uma parte fundamental de nossa decisão
fazendo.

Definindo um limite de serviço cuja propriedade atravessaria vários
é improvável que equipes diferentes produzam os resultados que desejariam, assim como faremos
explore mais detalhadamente no Capítulo 15, a propriedade compartilhada de microserviços é uma questão
caso. Conclui-se, portanto, que devemos levar em consideração o existente
estrutura organizacional ao considerar onde e quando definir
limites e, em algumas situações, talvez devêssemos até considerar
mudando a estrutura organizacional para dar suporte à arquitetura que queremos.

Obviamente, também precisamos considerar o que acontece se nossa organização
a estrutura também muda. Isso significa que agora temos que rearquitetar nosso
software? Bem, na pior das hipóteses, isso pode nos levar a examinar um existente
microserviço que agora precisa ser dividido, pois contém uma funcionalidade que agora
pode ser de propriedade de duas equipes separadas, enquanto antes uma única equipe era
responsável por ambas as partes. Por outro lado, muitas vezes mudanças organizacionais
exigiria, apenas que o proprietário de um microserviço existente mudasse.
Considere uma situação em que a equipe responsável pelas operações de armazenagem
anteriormente também lidava com a funcionalidade de calcular quantos itens
deve ser encomendado aos fornecedores. Digamos que decidimos mover isso
responsabilidade para com uma equipe de previsão dedicada que deseja extrair informações
das vendas atuais e promoções planejadas para descobrir o que precisa ser
encomendado. Se a equipe de armazenagem tivesse um fornecedor dedicado ao pedido
microserviço, isso poderia ser transferido para a nova equipe de previsão. Sobre o
por outro lado, se essa funcionalidade foi previamente integrada a uma maior
sistema de escopo de propriedade da armazenagem, então ele pode precisar ser dividido.

Mesmo quando trabalhamos dentro de uma estrutura organizacional existente, há uma
perigo de não colocarmos nossos limites no lugar certo. Há muitos anos, um
alguns colegas e eu estávamos trabalhando com um cliente na Califórnia, ajudando o
a empresa adota algumas práticas de código mais limpas e avança mais para
testes automatizados. Começamos com algumas das frutas mais baratas, como
decomposição do serviço, quando notamos algo muito mais preocupante. EU
não posso entrar em muitos detalhes sobre o que o aplicativo fez, mas foi um
aplicativo voltado para o público com uma grande base de clientes global.

A equipe e o sistema haviam crescido. Originalmente a visão de uma pessoa, o sistema
havia adquirido cada vez mais recursos e cada vez mais usuários. Eventualmente,

a organização decidiu aumentar a capacidade da equipe com uma nova
Um grupo de desenvolvedores com sede no Brasil assume parte do trabalho. O sistema
foi dividido, com a metade frontal do aplicativo sendo essencialmente sem estado.
implementando o site voltado para o público, conforme mostrado na Figura 2-19. A parte de trás
metade do sistema era simplesmente uma interface de chamada de procedimento remoto (RPC) em um
armazenamento de dados. Essencialmente, imagine que você tenha escolhido uma camada de repositório em seu
base de código e transformou isso em um serviço separado.

Mudanças frequentemente precisavam ser feitas em ambos os serviços. Ambos os serviços falaram
termos de chamadas de método de baixo nível, no estilo RPC, que eram excessivamente frágeis (vamos
discuta isso mais detalhadamente no Capítulo 4). A interface do serviço era muito falante, pois
bem, resultando em problemas de desempenho. Isso levou à necessidade de elaborar
Mecanismos de dosagem de RPC. Eu chamei isso de “arquitetura da cebola”, pois tinha muitas
camadas e me fez chorar quando tivemos que cortá-las.

Agora, à primeira vista, a ideia de dividir o sistema anteriormente monolítico
ao longo de linhas geográficas/organizacionais faz todo o sentido, pois expandiremos
continuado no Capítulo 15. Aqui, no entanto, em vez de assumir uma posição vertical, o negócio...
focada em uma fatia da pilha, a equipe escolheu o que antes era um in-
processse a API e faça uma fatia horizontal. Um modelo melhor teria sido
para que a equipe da Califórnia tenha uma fatia vertical de ponta a ponta, consistindo em
as partes relacionadas do front-end e da funcionalidade de acesso a dados, com a equipe
no Brasil pegando outra fatia.

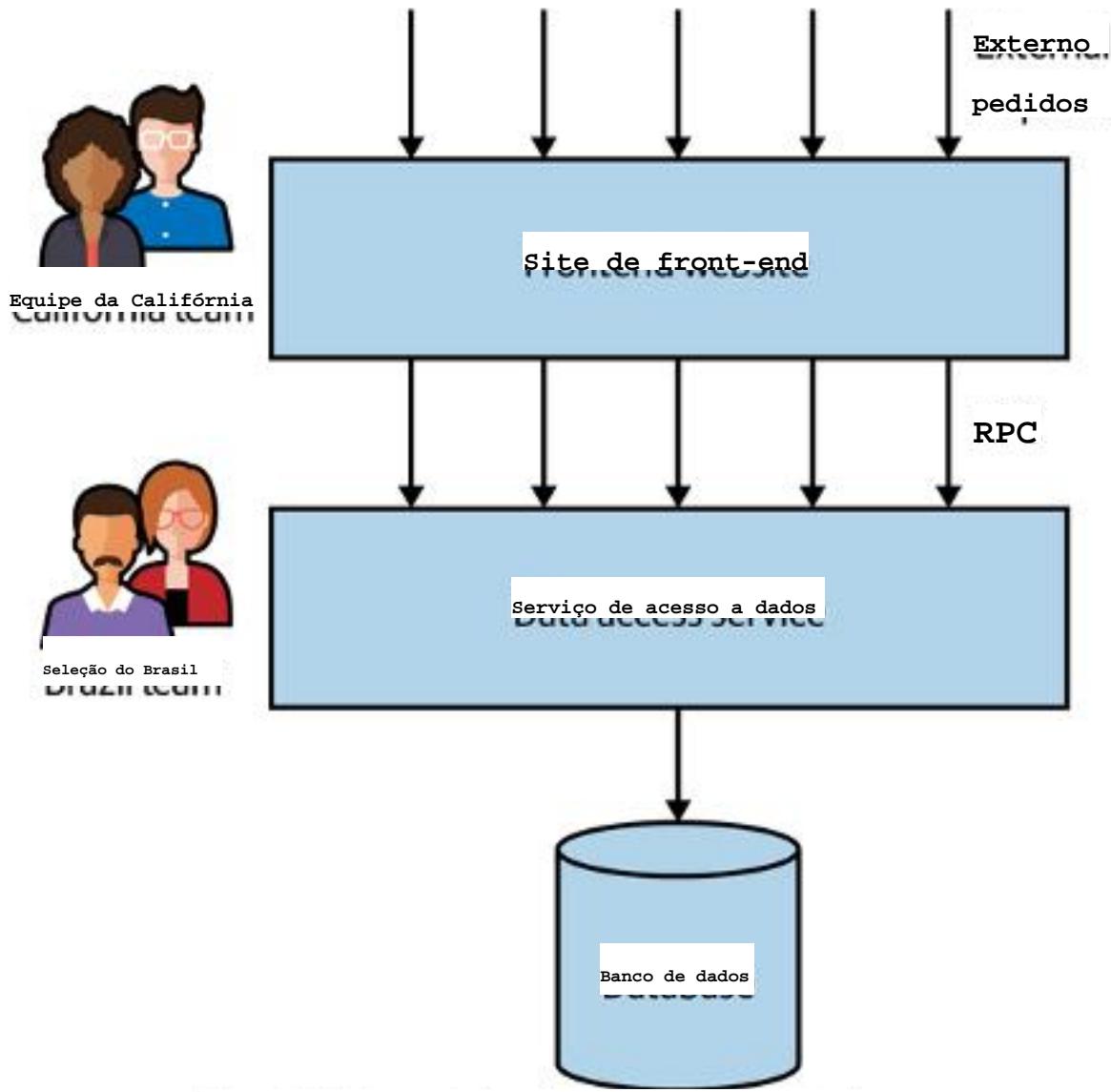


Figura 2-19. Um limite de serviço dividido entre costuras técnicas

CAMADAS INTERNAS VERSUS CAMADAS EXTERNAS

Como espero que você possa ver agora, não sou fã de camadas horizontais arquitetura. No entanto, as camadas podem ter seu lugar. Dentro de um microsserviço limite, pode ser totalmente sensato delinear entre diferentes camadas para facilitar o gerenciamento do código. O problema ocorre quando isso a estratificação se torna o mecanismo pelo qual seu microsserviço e limites de propriedade são traçados.

Modelos de mistura e exceções

Como espero que esteja claro até agora, não sou dogmático em termos de como você os encontra limites. Se você seguir as diretrizes de ocultação e apreciação de informações a interação de acoplamento e coesão, então é provável que você evite alguns dos piores armadilhas de qualquer mecanismo que você escolher. Acontece que eu acho que por concentrando-se nessas ideias, é mais provável que você acabe com um domínio arquitetura, mas isso é a propósito. O fato é, porém, que muitas vezes pode haver razões para misturar modelos, mesmo que você decida escolher "orientado ao domínio" como seu principal mecanismo para definir limites de microsserviços.

Os diferentes mecanismos que descrevemos até agora também têm muito potencial. interação entre eles. Ser muito restrito em suas escolhas aqui causará você deve seguir o dogma em vez de fazer a coisa certa. Baseado em volatilidade, a decomposição pode fazer muito sentido se seu foco for melhorar a velocidade de entrega, mas se isso fizer com que você extraia um serviço que cruza limites organizacionais, então espere que seu ritmo de mudança sofra devido a contêncio de entrega.

Eu poderia definir um bom serviço de armazém com base na minha compreensão do domínio de negócios, mas se uma parte desse sistema precisar ser implementada em C++ e outra parte em Kotlin, então você terá que se decompor mais adiante essas linhas técnicas.

Os limites de serviços organizacionais e orientados por domínios são meu próprio começo ponto. Mas essa é apenas minha abordagem padrão. Normalmente, vários fatores Eu descrevi aqui como entrar em jogo e quais influenciam a sua. as decisões serão baseadas nos problemas que você está tentando resolver. Você precisa analise suas próprias circunstâncias específicas para determinar o que funciona melhor para você - e espero ter lhe dado algumas opções diferentes a serem consideradas. Apenas lembre-se, se alguém disser "A única maneira de fazer isso é X!" eles são prováveis só estou te vendendo mais dogmas. Você pode fazer melhor do que isso.

Com tudo isso dito, vamos nos aprofundar no tópico da modelagem de domínio por explorando o design orientado por domínio com um pouco mais de detalhes.

Resumo

Neste capítulo, você aprendeu um pouco sobre o que faz um bom microserviço. Um limite e como encontrar costuras em nosso espaço problemático que nos forneçam a dupla benefícios do baixo acoplamento e da forte coesão. Ter um detalhado entendimento do nosso domínio pode ser uma ferramenta vital para nos ajudar a encontrá-los. E ao alinhar nossos microserviços a esses limites, garantimos que o sistema resultante tem todas as chances de manter essas virtudes intactas. Nós temos também recebi uma dica sobre como podemos subdividir ainda mais nossos microserviços.

As ideias apresentadas no Domain-Driven Design de Eric Evans são muito úteis para encontrarmos limites sensatos para nossos serviços, e acabei de começar a superfície Aqui - o livro de Eric entra em muito mais detalhes. Se você quiser ir mais profundamente nessa abordagem, a obra de Vaughn Vernon *Domain-Driven Design*¹⁷ para ajudar você a entender os aspectos práticos dessa abordagem, enquanto Vernon's *Domain-Driven Design Distilled*¹⁸ é um ótimo condensado de visão geral se você estiver procurando por algo mais breve.

Grande parte deste capítulo descreveu como encontrariamos o limite para nosso microserviços. Mas o que acontece se você já tiver um monolítico aplicativo e está procurando migrar para uma arquitetura de microservices? Isso é algo que exploraremos com mais detalhes no próximo capítulo.

1 David Parnas, "Sobre os critérios a serem usados na decomposição de sistemas em módulos", (revista contribuição Carnegie Mellon University, 1971), <https://oreil.ly/BnVVg>.

2 O ponto de partida óbvio é o resumo de Adrian de "Sobre os critérios, mas a cobertura de Adrian". O trabalho anterior de Parnas, "Aspectos da Distribuição de Informações da Metodologia de Design", contém algumas ótimas ideias junto com comentários do próprio Parnas.

3 Parnas, "Aspectos da distribuição de informações".

4 Irritantemente, não consigo encontrar a fonte original dessa definição.

5 Em meu livro *Monolith to Microservices* (O'Reilly), atribuí isso ao próprio Larry Constantine. Embora a declaração resuma perfeitamente grande parte do trabalho de Constantino neste espaço, a citação deve realmente deve ser atribuído a Albert Endres e Dieter Rombach, de seu livro de 2003 *A Handbook of Engenharia de Software e Sistemas* (Addison-Wesley).

6 Edward Yourdon e Larry L. Constantine, *Design Estruturado* (Nova York: Yourdon Press, 1976).

- 7 Meilir Page-Jones. *O Guia Prático para Design de Sistemas Estruturados* (Nova York: Yourdon, Computação de imprensa, 1980).
- 8 Esse conceito é semelhante ao protocolo de aplicativo de domínio, que define as regras pelas quais os componentes interagem em um sistema baseado em REST.
- 9 Acoplamento pass-through é meu nome para o que foi originalmente descrito como "acoplamento fixo" por Meilir Page-Jones em *O Guia Prático para Design de Sistemas Estruturados*. Eu escolhi usar um termo diferente aqui devido ao fato de eu ter achado o termo original um tanto problemático e não é muito significativo para um público mais amplo.
- 10 OK, mais de uma ou duas vezes. Muito mais do que uma ou duas vezes.
- 11 Eric Evans, *Design orientado por domínio: enfrentando a complexidade no coração do software* (Boston: Addison-Wesley, 2004).
- 12 Sei que algumas pessoas se opõem ao uso de URIs modelados em sistemas REST, e eu entendo. Por que eu só quero manter as coisas simples neste exemplo.
- 13 Phil Calcado, "Padrão: Usando pseudo-URIs com microserviços", <https://oreil.ly/xOYMr>.
- 14 Não quero desrespeitar se for você - eu mesmo já fiz isso mais de uma vez.
- 15 Quero dizer, por que não amarelo? É a cor mais comum!
- 16 Alberto Brandolini, *EventStorming* (Victoria, BC: Leanpub, em breve).
- 17 Vaughn Vernon, *implementando design orientado por domínio* (Upper Saddle River, NJ: Addison-Wesley, 2013).
- 18 Vaughn Vernon, *Design orientado por domínio destilado* (Boston: Addison-Wesley, 2016).

Capítulo 3. Dividindo o monólito

Muitos de vocês que estão lendo este livro provavelmente não têm uma tela em branco para projete seu sistema e, mesmo que o fizesse, começar com microsserviços poderia não é uma boa ideia, por motivos que exploramos no Capítulo 1. Muitos de vocês vão já tem um sistema existente, talvez alguma forma de monolítico arquitetura, que você deseja migrar para uma arquitetura de microsserviços.

Neste capítulo, descreverei alguns primeiros passos, padrões e dicas gerais para ajudar você navega na transição para uma arquitetura de microsserviços.

Tenha um objetivo

Os microsserviços não são o objetivo. Você não "ganha" por ter microsserviços. A adoção de uma arquitetura de microsserviços deve ser uma decisão consciente, com base na tomada de decisão racional. Você deve estar pensando em migrar para um arquitetura de microsserviços somente se você não conseguir encontrar uma maneira mais fácil de se mover em direção ao seu objetivo final com sua arquitetura atual.

Sem uma compreensão clara do que você está tentando alcançar, você poderia cair na armadilha de confundir atividade com resultado. Eu vi equipes obcecado em criar microsserviços sem nunca perguntar por quê. Isso é problemático ao extremo, dadas as novas fontes de complexidade que microsserviços podem introduzir.

Fixar-se em microsserviços em vez de no objetivo final também significa que você provavelmente pare de pensar em outras maneiras pelas quais você pode provocar a mudança você está procurando. Por exemplo, os microsserviços podem ajudar você a escalar seu sistema, mas geralmente existem várias técnicas alternativas de escalonamento que deve ser examinado primeiro. Criando mais algumas cópias do seu existente. Um sistema monolítico por trás de um平衡ador de carga pode muito bem ajudá-lo a escalar seu sistema muito mais eficaz do que passar por um complexo e demorado decomposição em microsserviços.

GORJETA

Microserviços não são fáceis. Experimente as coisas simples primeiro.

Finalmente, sem um objetivo claro, fica difícil saber por onde começar.

Qual microsserviço você deve criar primeiro? Sem uma abordagem abrangente entendendo o que você está tentando alcançar, você está voando às cegas.

Portanto, seja claro sobre qual mudança você está tentando alcançar e considere mais fáceis maneiras de atingir esse objetivo final antes de considerar os microsserviços. Se os microsserviços são realmente a melhor maneira de seguir em frente e, em seguida, rastrear seus progressos em relação a esse objetivo final e mude de rumo conforme necessário.

Migração incremental

Se você reescrever em grande escala, a única coisa garantida é uma grande estrondo.

-Martin Fowler

Se você chegar ao ponto de decidir que está separando seu monólito existente sistema é a coisa certa a fazer, eu recomendo fortemente que você elimine o monólito, extraindo um pouco de cada vez. Uma abordagem incremental ajudará você aprender sobre microsserviços à medida que avança e também limitará o impacto da obtenção algo errado (e você vai errar!). Pense em nosso monólito como bloco de mármore. Poderíamos explodir tudo, mas isso raramente acaba bem. Faz muito mais sentido simplesmente reduzir isso de forma incremental.

Divida a grande jornada em vários pequenos passos. Cada etapa pode ser realizada e aprendi com Se for um passo retrógrado, foi apenas um pequeno um. De qualquer forma, você aprende com isso, e o próximo passo será informado pelas etapas anteriores.

Quebrar as coisas em pedaços menores também permite identificar vitórias rápidas e aprenda com eles. Isso pode ajudar a tornar a próxima etapa mais fácil e pode ajudar ganhe impulso. Ao dividir os microsserviços um de cada vez, você também pode

desbloqueie o valor que eles trazem de forma incremental, em vez de ter que esperar por alguns implantação do big bang.

Tudo isso leva ao que se tornou meu conselho de ações para pessoas que procuram microsserviços: se você acha que microsserviços são uma boa ideia, comece de algum lugar pequeno. Escolha uma ou duas áreas de funcionalidade, implemente-as como microsserviços, faça com que eles sejam implantados na produção e, em seguida, reflita sobre se a criação de seus novos microsserviços ajudou você a se aproximar do seu fim gol.

ADVERTÊNCIA

Você não apreciará o verdadeiro horror, dor e sofrimento de uma arquitetura de microsserviços. pode trazer até que você esteja executando a produção.

O monólito raramente é o inimigo

Embora eu já tenha argumentado no início do livro que alguma forma de arquitetura monolítica pode ser uma escolha totalmente válida, ela merece ser repetida que uma arquitetura monolítica não é inherentemente ruim e, portanto, não deveria ser visto como o inimigo. Não se concentre em "não ter o monólito"; concentre-se em vez disso sobre os benefícios que você espera que sua mudança na arquitetura traga.

É comum que a arquitetura monolítica existente permaneça após uma mudança em direção a microsserviços, embora muitas vezes em uma capacidade reduzida. Por exemplo, um mover para melhorar a capacidade do aplicativo de lidar com mais carga pode ser satisfeito ao remover os 10% da funcionalidade que atualmente está obstruída, deixando os 90% restantes no sistema monolítico.

Muitas pessoas acham que a realidade de um monólito e microsserviços coexistem com seja "confuso" - a arquitetura de um sistema em execução no mundo real nunca é limpo ou imaculado. Se você quer uma arquitetura "limpa", por favor, lamine um impressão de uma versão idealizada da arquitetura do sistema que você possa ter tinhado, se ao menos você tivesse uma visão perfeita e fundos ilimitados. Sistema real a arquitetura é uma coisa em constante evolução que deve se adaptar às necessidades e

mudança de conhecimento. A habilidade está em me acostumar com essa ideia, algo que eu vou
volte ao Capítulo 16.

Ao tornar sua migração para microsserviços uma jornada incremental, você está
capaz de eliminar a arquitetura monolítica existente, oferecendo
melhorias ao longo do caminho, além de, principalmente, saber quando parar.

Em circunstâncias surpreendentemente raras, o fim do monólito pode ser difícil
exigência. Em minha experiência, isso geralmente se limita a situações em que o
monólito existente é baseado em tecnologia morta ou moribunda, está vinculado a
infraestrutura que precisa ser descontinuada ou talvez seja uma empresa terceirizada cara
sistema que você deseja abandonar. Mesmo nessas situações, um incremental
a abordagem da decomposição é garantida pelas razões que descrevi.

Os perigos da decomposição prematura

Há perigo em criar microsserviços quando você não está claro
compreensão do domínio. Um exemplo dos problemas que isso pode causar
vem da minha empresa anterior, a Thoughtworks. Um de seus produtos era
Snap CI, uma ferramenta hospedada de integração contínua e entrega contínua. (vamos
discuta esses conceitos no Capítulo 7). A equipe já havia trabalhado em um
ferramenta semelhante, GoCD, uma ferramenta de entrega contínua de código aberto que pode ser
implantado localmente em vez de ser hospedado na nuvem.

Embora tenha havido alguma reutilização de código muito cedo entre o Snap CI e
Projetos GoCD, no final das contas, o Snap CI se revelaram completamente novos
base de código. No entanto, a experiência anterior da equipe no domínio de
As ferramentas de CD os encorajaram a se moverem mais rapidamente na identificação de limites
e construindo seu sistema como um conjunto de microsserviços.

Depois de alguns meses, porém, ficou claro que os casos de uso do Snap CI
eram sutilmente diferentes o suficiente para que a abordagem inicial assumisse os limites do serviço
não estava certo. Isso fez com que muitas mudanças fossem feitas em todos os serviços,
e um alto custo de mudança associado. Eventualmente, a equipe fundiu o
serviços de volta em um sistema monolítico, dando aos membros da equipe tempo para
entender melhor onde os limites devem existir. Um ano depois, a equipe
foi capaz de dividir o sistema monolítico em microsserviços, cujo

os limites provaram ser muito mais estáveis. Este está longe de ser o único exemplo dessa situação que eu vi. Decompondo prematuramente um sistema em microsserviços podem ser caros, especialmente se você for novo no domínio. Em muitas maneiras, ter uma base de código existente na qual você deseja se decompor microsserviços é muito mais fácil do que tentar acessar microsserviços a partir do começo por esse mesmo motivo.

O que dividir primeiro?

Depois de entender por que você acha que os microsserviços são uma boa ideia, você pode usar esse entendimento para ajudar a priorizar quais microsserviços usar crie primeiro. Quer escalar o aplicativo? Funcionalidade que atualmente as restrições de que a capacidade do sistema de lidar com a carga estarão no topo da lista. Quer melhorar o tempo de comercialização? Veja a volatilidade do sistema para identificar aquelas peças de funcionalidade que mudam com mais frequência, e veja se elas funcionariam como microsserviços. Você pode usar ferramentas de análise estática como CodeScene para encontrar rapidamente partes voláteis da sua base de código. Você pode ver um exemplo de uma visualização do CodeScene na Figura 3-1, onde vemos pontos ativos em o projeto Apache Zookeeper de código aberto.

Mas você também precisa considerar quais decomposições são viáveis. Alguns a funcionalidade pode ser profundamente incorporada ao aplicativo monolítico existente que é impossível ver como ela pode ser desembaraçada. Ou talvez o a funcionalidade em questão é tão importante para o aplicativo que qualquer alteração é considerado de alto risco. Como alternativa, a funcionalidade que você deseja migrar pode já ser um pouco independente e, portanto, a extração parece muito direto.

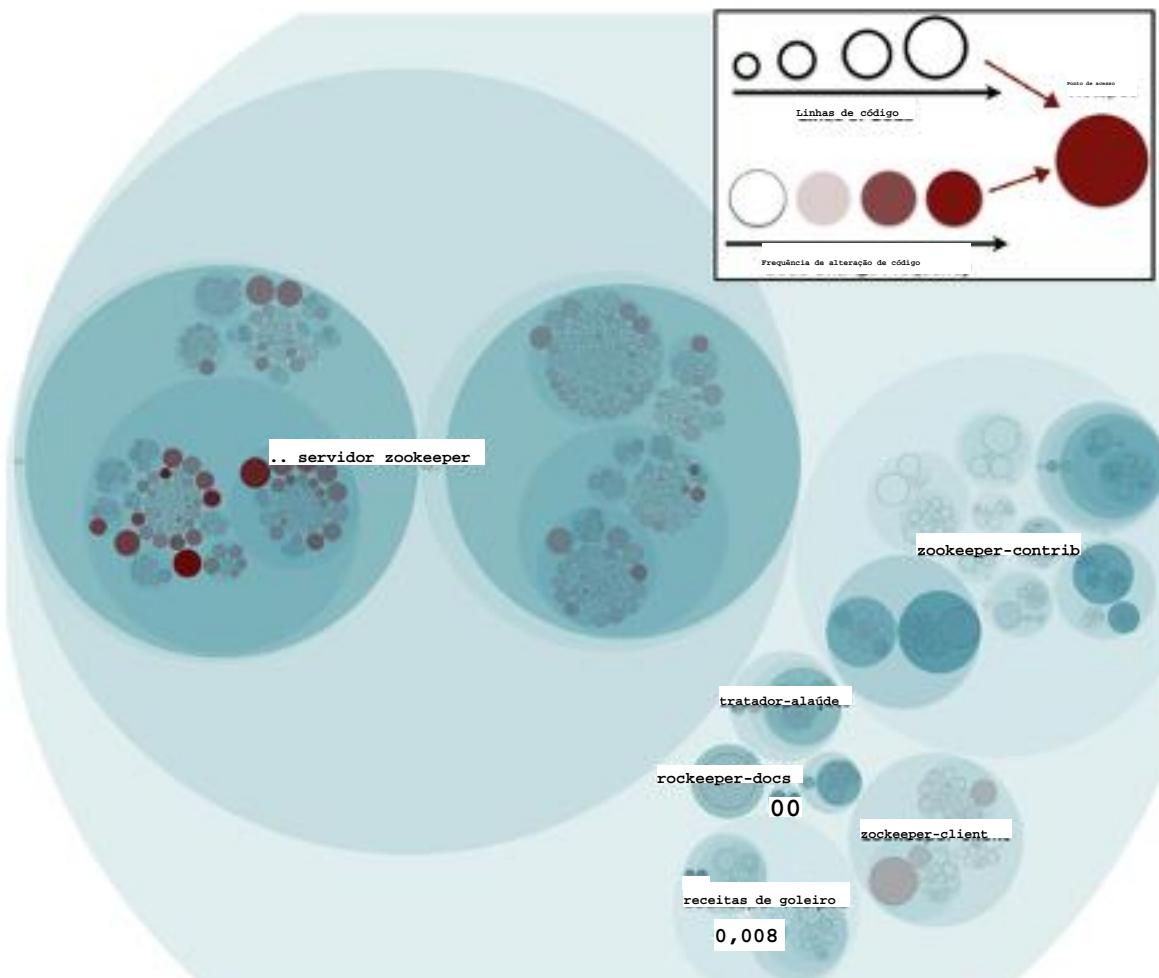


Figura 3-1. A visualização do ponto de acesso no CodeScene, ajudando a identificar partes da base de código que mudam frequentemente

Fundamentalmente, a decisão sobre qual funcionalidade dividir em o microserviço acabará sendo um equilíbrio entre essas duas forças – como fácil é a extração versus o benefício de extrair o microserviço no primeiro lugar.

Meu conselho para os primeiros dois microserviços seria escolher coisas que incline-se um pouco mais para a extremidade “fácil” do espectro – um microserviço que achamos que tem algum impacto em termos de alcançar nossa meta de ponta a ponta certamente – mas algo que consideraríamos ser uma fruta fácil. É importante

ganhe uma sensação de impulso desde o início. Então você precisa de algumas vitórias rápidas sob seu cinto.

Por outro lado, se você tentar extrair o que considera ser o mais fácil microsserviço e não conseguem fazer com que funcione, pode valer a pena reconsiderando se os microsserviços são realmente adequados para você e seus organização.

Com alguns sucessos e algumas lições aprendidas, você estará muito melhor posicionado para lidar com extrações mais complexas, que também podem estar operando em mais áreas críticas de funcionalidade.

Decomposição por camada

Então você identificou seu primeiro microsserviço a ser extraído; o que vem a seguir? Bem, nós pode dividir essa decomposição em etapas adicionais e menores.

Se considerarmos os três níveis tradicionais de uma pilha de serviços baseados na web, podemos analisar a funcionalidade que queremos extrair em termos de seu usuário interface, código do aplicativo de back-end e dados.

O mapeamento de um microsserviço para uma interface de usuário geralmente não é 1:1 (este é um tópico que exploramos com muito mais profundidade no Capítulo 14). Como tal, extraíndo o usuário a funcionalidade da interface relacionada ao microsserviço pode ser considerada uma etapa separada. Vou dar uma nota de cautela aqui sobre ignorar o usuário interface, parte da equação. Já vi muitas organizações parecerem apenas nos benefícios de decompor a funcionalidade de back-end, o que geralmente resulta em uma abordagem excessivamente isolada para qualquer reestruturação arquitetônica. Às vezes, os maiores benefícios podem vir da decomposição da interface do usuário, então ignore isso em seu perigo. Freqüentemente, a decomposição da interface do usuário tende a ficar aquém da decomposição do back-end em microsserviços, pois até os microsserviços são disponível, é difícil ver as possibilidades de decomposição da interface do usuário; apenas certifique-se de que não demore muito.

Se analisarmos o código de back-end e o armazenamento relacionado, é vital que ambos esteja no escopo ao extrair um microsserviço. Vamos considerar a Figura 3-2, onde estamos procurando extrair funcionalidades relacionadas ao gerenciamento de um lista de desejos do cliente. Há algum código de aplicativo que está no

monólito e algum armazenamento de dados relacionado no banco de dados. Então, qual parte deveria ser extraída, and some related data storage in the database. So which one should we extract first?

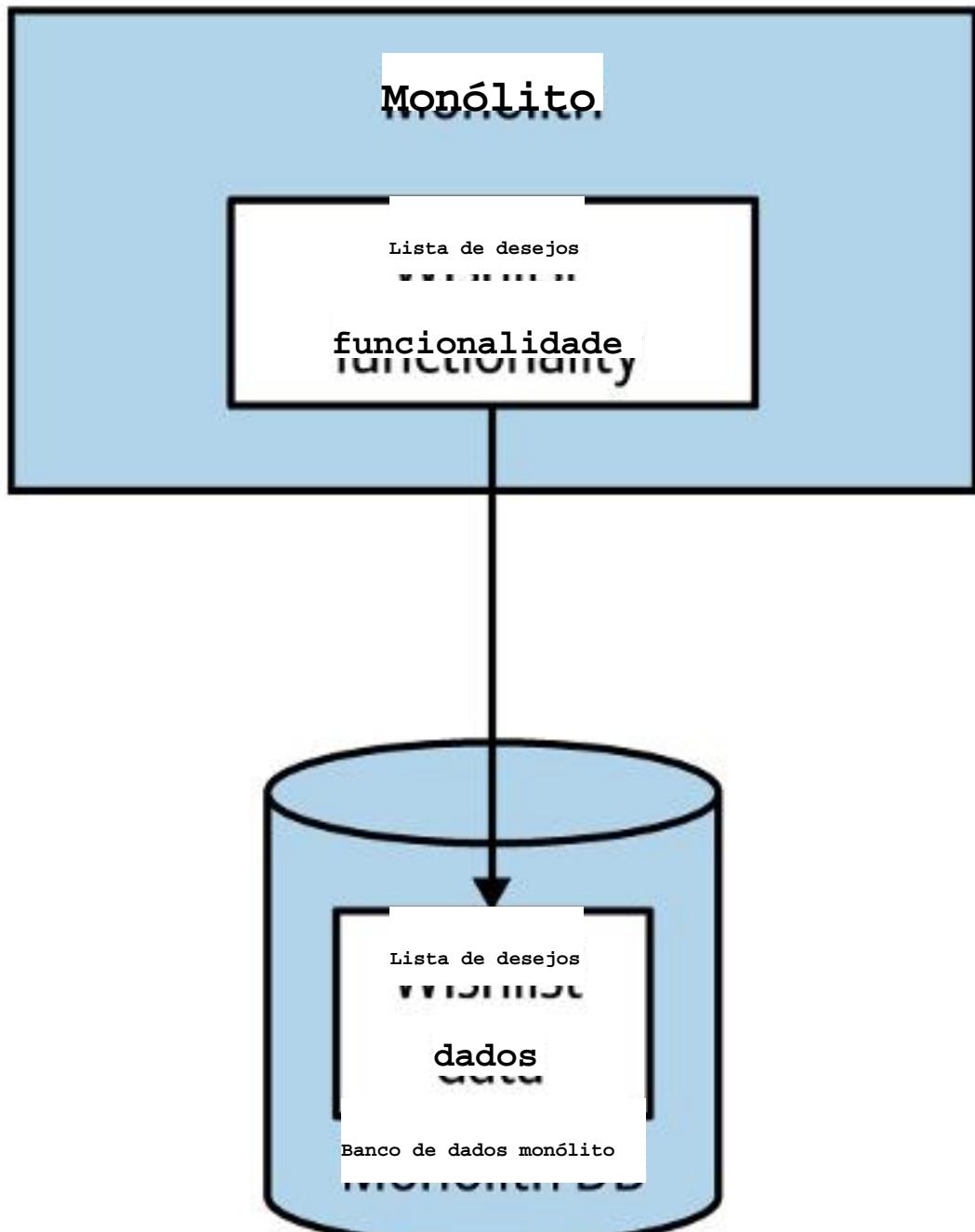


Figura 3-2. O código e os dados da lista de desejos no aplicativo monolítico existente.
Figure 3-2. The code and data of the wish list in the existing monolithic application

Codifique primeiro

Na Figura 3-3, extraímos o código associado à lista de desejos funcionalidade em um novo microsserviço. Os dados da lista de desejos permanecem em o banco de dados monolítico neste estágio - não concluímos o decomposição até que também removamos os dados relacionados ao novo..

Microserviço da lista de desejos.

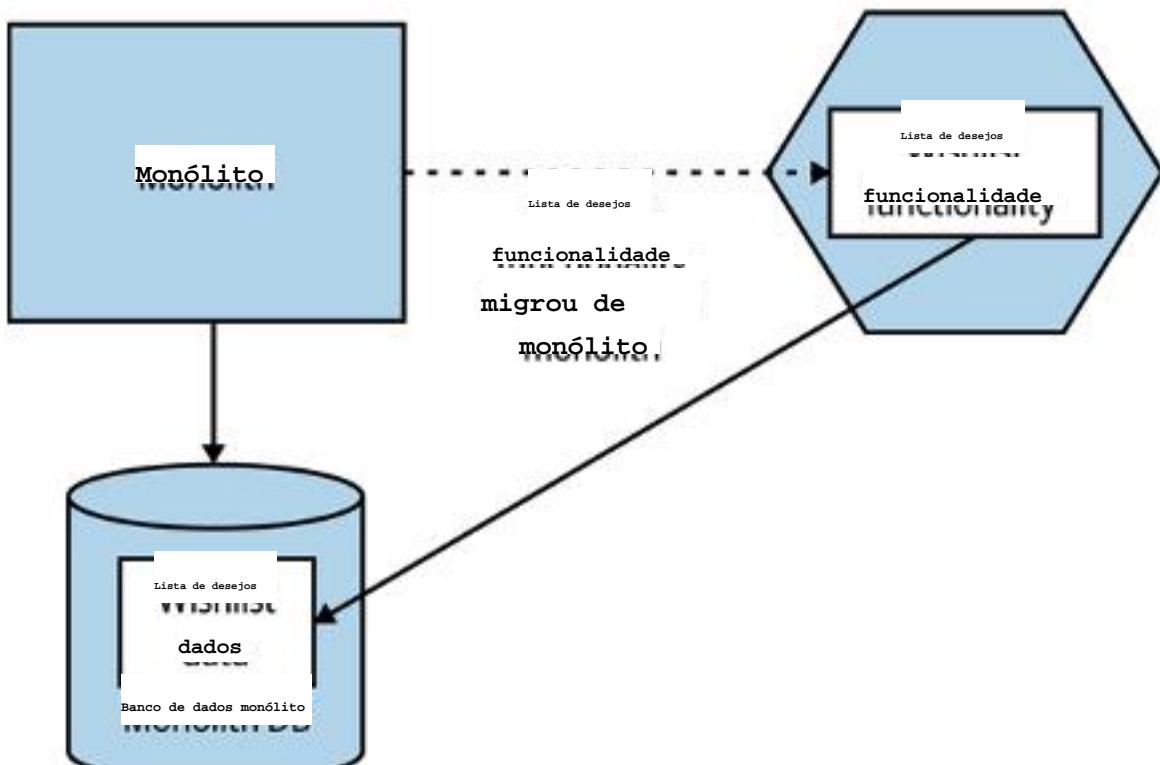


Figura 3-3. Movendo primeiro o código da lista de desejos para um novo microsserviço, deixando os dados no banco de dados monolítico

Na minha experiência, esse tende a ser o primeiro passo mais comum. O principal A razão para isso é que ele tende a oferecer mais benefícios a curto prazo. Se deixássemos o dados no banco de dados monolítico, estamos armazenando muita dor para o futuro, então isso também precisa ser resolvido, mas ganhamos muito com nosso novo microsserviço.

Extraír o código do aplicativo tende a ser mais fácil do que extraír coisas do banco de dados. Se descobrirmos que era impossível extraír o aplicativo codifique de forma limpa, poderíamos abortar qualquer trabalho adicional, evitando a necessidade de desembaraçar o banco de dados. Se, no entanto, o código do aplicativo for extraído de forma limpa, mas extrair os dados prova ser impossível, podemos estar em apuros - portanto, é

essencial que, mesmo que você decida extrair o código do aplicativo antes dos dados, você precisa ter examinado o armazenamento de dados associado e ter alguma ideia de se a extração é viável e como você fará isso. Então faça o trabalho braçal para esboçar como serão o código e os dados do aplicativo extraído antes de você começar.

Dados em primeiro lugar

Na Figura 3-4, vemos os dados sendo extraídos primeiro, antes da aplicação. Eu vejo essa abordagem com menos frequência, mas ela pode ser útil em situações em que você não tem certeza se os dados podem ser separados de forma limpa. Aqui, você prova que isso pode ser feito antes de passar para o aplicativo, esperançosamente mais fácil extração de código.

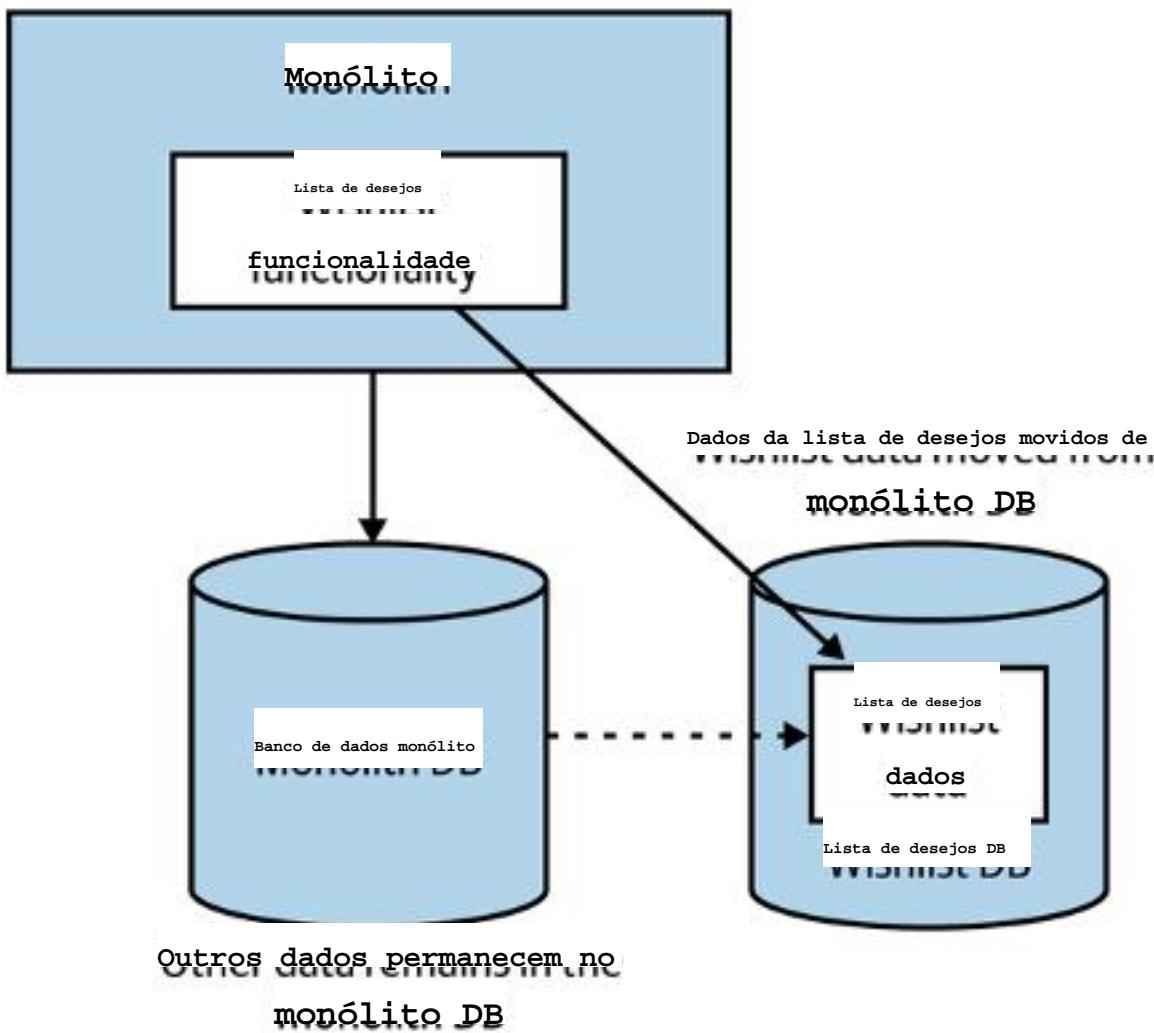


Figura 3-4. As tabelas associadas à funcionalidade da lista de desejos são extraídas primeiro.

O principal benefício dessa abordagem no curto prazo é a redução do risco total. A extração do micserviço. Isso força você a lidar com questões como perda da integridade forçada dos dados em seu banco de dados ou falta de informações transacionais operações em ambos os conjuntos de dados. Abordaremos brevemente as implicações de ambas as edições mais adiante neste capítulo.

Padrões de decomposição úteis

Vários padrões podem ser úteis para ajudar a separar um sistema existente. Muitos deles são explorados em detalhes em meu livro *Monolith to Microservices*; em vez de repetir todos eles aqui, compartilharei uma visão geral de alguns deles para dar uma ideia do que é possível.

Padrão Strangler Fig

Uma técnica que tem sido usada com frequência durante a reescrita do sistema é o estrangulador, padrão de figo, um termo cunhado por Martin Fowler. Inspirado em um tipo de planta, o padrão descreve o processo de empacotar um sistema antigo com o novo sistema com o tempo, permitindo que o novo sistema assuma cada vez mais recursos do sistema antigo de forma incremental.

A abordagem mostrada na Figura 3-5 é direta. Você intercepta chamadas para o sistema existente - no nosso caso, o aplicativo monolítico existente. Se a chamada para essa funcionalidade é implementada em nosso novo microsserviço arquitetura, ela é redirecionada para o microsserviço. Se a funcionalidade ainda estiver fornecida pelo monólito, a chamada pode continuar até o monólito em si.

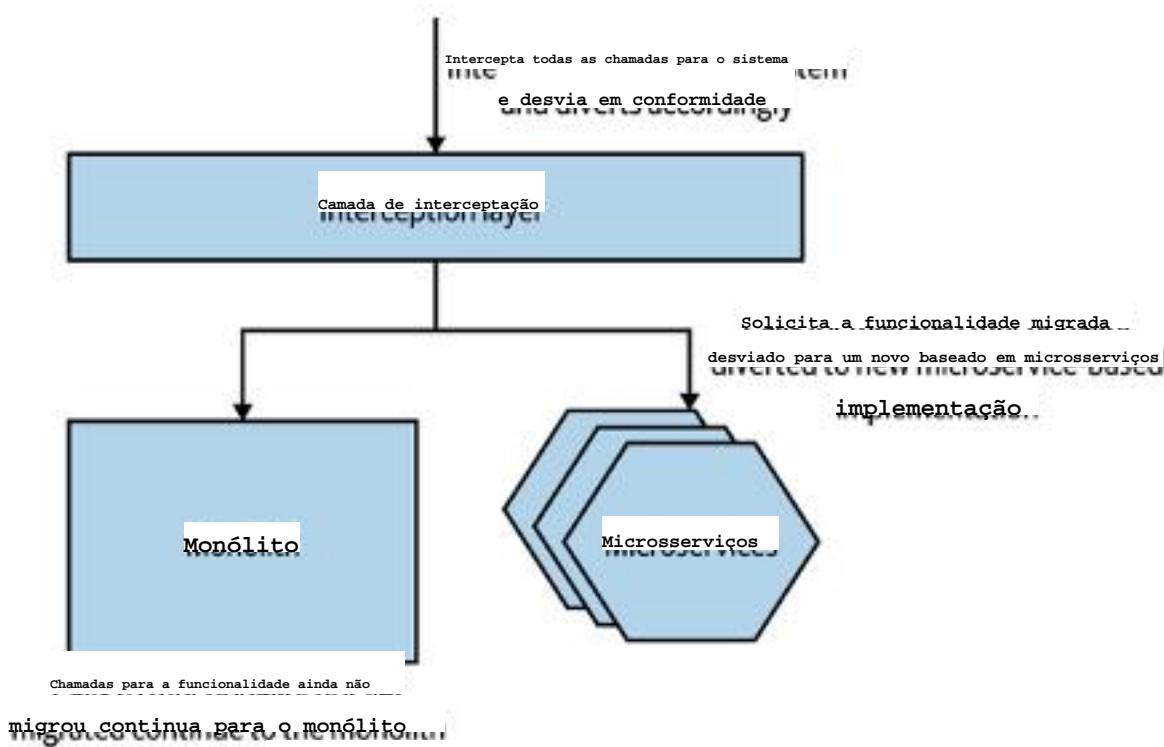


Figura 3-5. Uma visão geral do padrão do figo estrangulador

A beleza desse padrão é que muitas vezes ele pode ser feito sem fazer nenhuma mudança na aplicação monolítica subjacente. O monólito não sabe que até foi "embrulhado" com um sistema mais novo.

Execução paralela

Ao mudar da funcionalidade fornecida por uma já testada e comprovada arquitetura de aplicativos até uma nova e sofisticada baseada em microserviços, pode haver seja um pouco nervoso, especialmente se a funcionalidade que está sendo migrada for crítica para sua organização...

Uma forma de garantir que a nova funcionalidade esteja funcionando bem sem arriscar o comportamento existente do sistema é fazer uso do padrão de execução paralela: executando sua implementação monolítica da funcionalidade e da nova implementação de microserviços lado a lado, atendendo às mesmas solicitações e comparando os resultados. Exploraremos esse padrão com mais detalhes em "Paralelo Corra".

Alternar recurso

Uma alternância de recursos é um mecanismo que permite que um recurso seja desligado ou ativado ou para alternar entre duas implementações diferentes de alguma funcionalidade. A alternância de recursos é um padrão que tem boa aplicabilidade geral, mas pode seja especialmente útil como parte de uma migração de microserviços.

Como descrevi com o aplicativo Strangler Fig, geralmente deixamos o existente funcionalidade em vigor no monólito durante a transição, e queremos o capacidade de alternar entre as versões da funcionalidade - a funcionalidade em o monólito e aquele no novo microserviço. Com o padrão de figo estrangulador exemplo de uso de um proxy HTTP, poderíamos implementar o recurso toggle in a camada proxy para permitir um controle simples para alternar entre implementações.

, recomendo o de Pete Hodgson

artigo "Alternâncias de recursos (também conhecidas como sinalizadores de recursos) ."2 artigo "Alternâncias de recursos (também conhecidas como sinalizadores de recursos) ."2

Preocupações com a decomposição de dados

Quando começamos a separar os bancos de dados, podemos causar vários problemas. Aqui estão alguns dos desafios que você pode enfrentar e algumas dicas para ajudar.

Desempenho

Bancos de dados, especialmente bancos de dados relacionais, são bons em unir dados entre mesas diferentes. Muito bom. Tão bom, na verdade, que tomamos isso como certo. Muitas vezes, porém, quando separamos bancos de dados em nome de microserviços, acabamos tendo que mover as operações de junção do nível de dados para o microserviços em si. E por mais que tentemos, é improvável que seja tão rápido. Considere a Figura 3-6, que ilustra uma situação em que nos encontramos em relação à MusicCorp. Decidimos extrair a funcionalidade do nosso catálogo, algo que possa gerenciar e expor informações sobre artistas, faixas e álbuns. Atualmente, nosso código relacionado ao catálogo dentro do monólito usa um Tabela de álbuns para armazenar informações sobre os CDs que possamos ter disponível para venda. Esses álbuns acabam sendo referenciados em nosso Ledger tabela, que é onde rastreamos todas as vendas. As linhas na tabela Ledger registre a data em que algo é vendido, junto com um identificador que é chamado de SKU (estoque) unidade de manutenção), uma prática comum em sistemas de varejo. 3 keeping unit), a common practice in retail systems.

No final de cada mês, precisamos gerar um relatório descrevendo nosso melhor vendendo CDs. A tabela Ledger nos ajuda a entender qual SKU vendeu mais cópias, mas as informações sobre esse SKU estão na tabela Álbuns. Nós quero tornar os relatórios agradáveis e fáceis de ler, então, em vez de dizer: "Nós vendeu 400 cópias do SKU 123 e ganhou \$1.596", acrescentaríamos mais informações sobre o que foi vendido, dizendo em vez disso: "Vendemos 400 cópias de Now That's o que eu chamo de Death Polka e ganhei \$1.596." Para fazer isso, a consulta ao banco de dados açãonado por nosso código financeiro precisa juntar informações do Ledger tabela para a tabela Álbuns, como mostra a Figura 3-6.

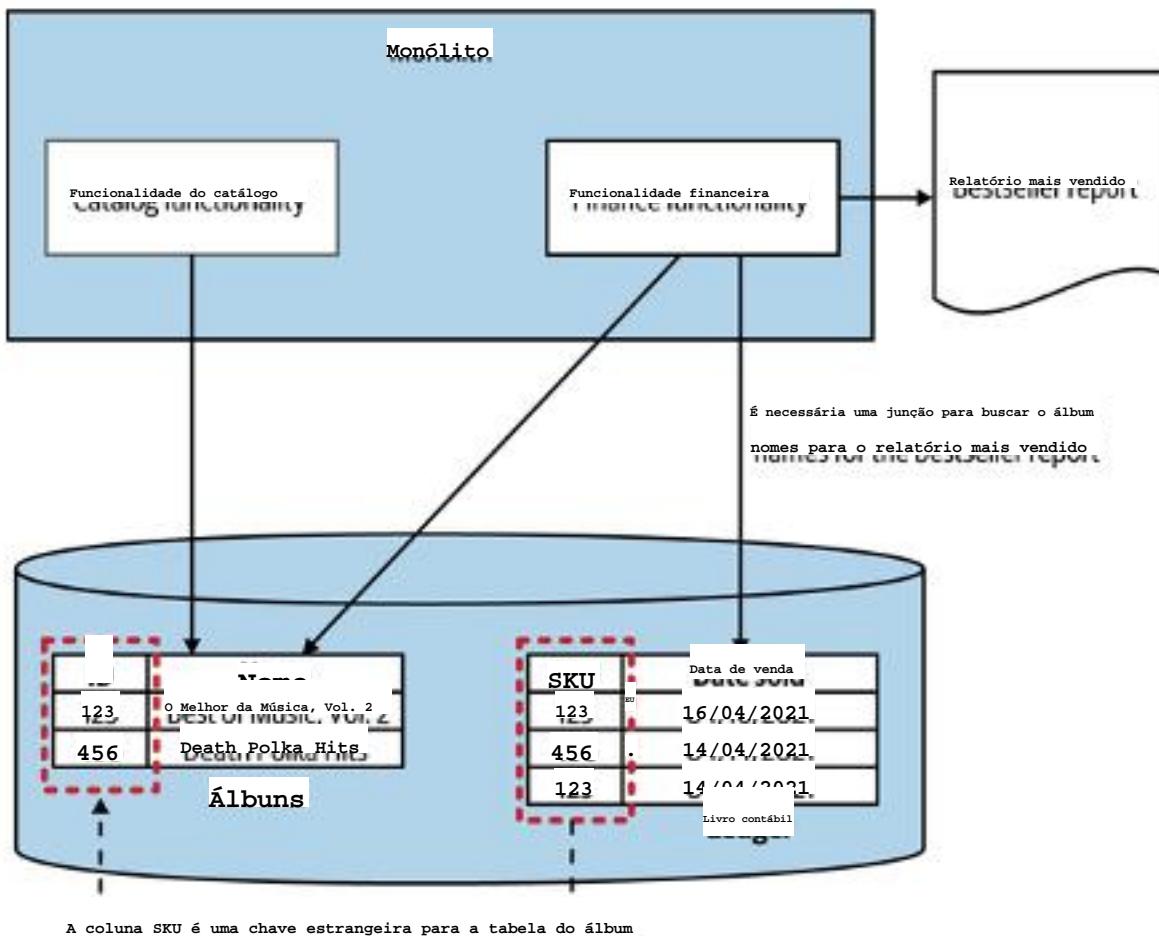


Figura 3-6. Uma operação de junção no banco de dados monolítico

Em nosso novo mundo baseado em microserviços, nosso novo microserviço financeiro tem a responsabilidade de gerar o relatório dos mais vendidos, mas não tem os dados do álbum localmente. Portanto, ele precisará buscar esses dados do nosso novo catálogo microserviço, conforme mostrado na Figura 3-7. Ao gerar o relatório, o microserviço financeiro primeiro consulta a tabela Ledger, extraíndo a lista de SKUs mais vendidos no último mês. Neste ponto, a única informação que ter localmente é uma lista de SKUs e o número de cópias vendidas para cada SKU.

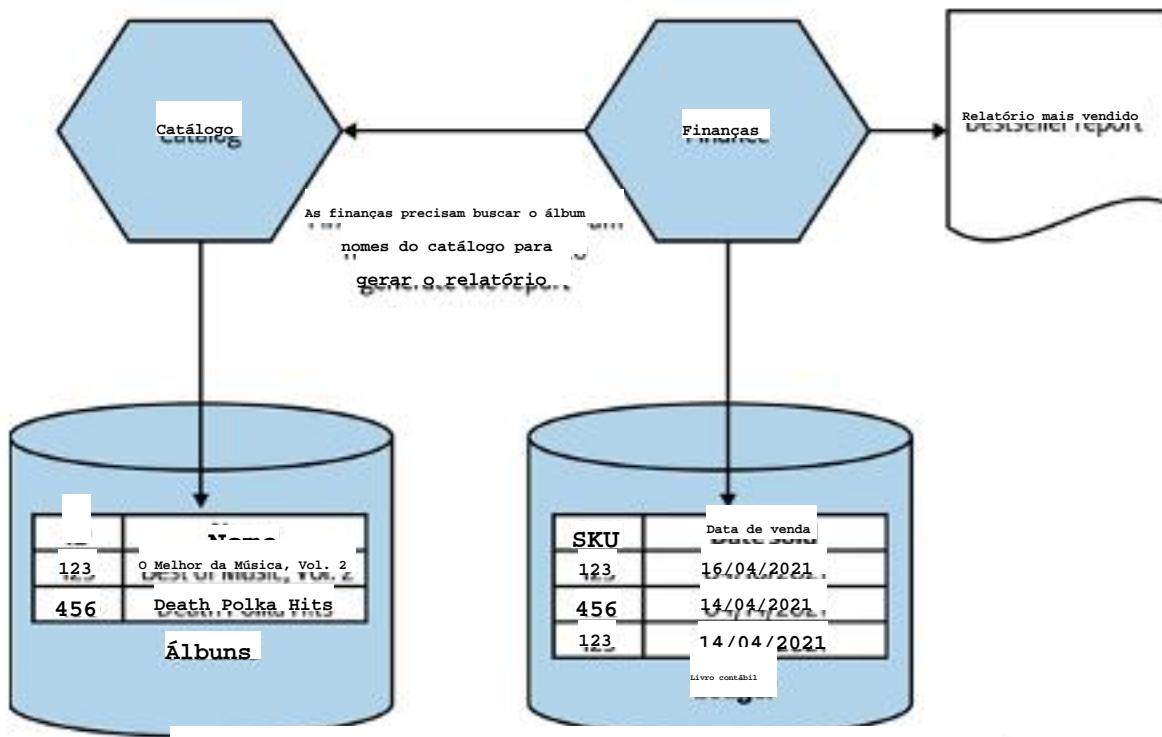


Figura 3-7 Substituindo uma operação de união de banco de dados por chamadas de serviço

Em seguida, precisamos ligar para o microsserviço Catalog, solicitando informações sobre cada um desses SKUs. Essa solicitação, por sua vez, fará com que o Catálogo microsserviço para criar seu próprio SELECT local em seu próprio banco de dados. Logicamente, a operação de junção ainda está acontecendo, mas agora está acontecendo internamente o microsserviço de finanças em vez de no banco de dados. A junção passou de da camada de dados até a camada de código do aplicativo. Infelizmente, esta operação não é tão eficiente quanto teria sido se a junção permaneceu no banco de dados. Saímos de um mundo em que temos uma instrução SELECT única, para um novo mundo no qual temos uma consulta SELECT contra a tabela Ledger, seguida por uma chamada para o microsserviço Catalog, que, por sua vez, aciona uma instrução SELECT na tabela Albums, como vemos na Figura 3-7.

Nessa situação, eu ficaria muito surpreso se a latência geral dessa operação não aumentou. Isso pode não ser um problema significativo neste caso específico, já que este relatório é gerado mensalmente e, portanto, pode ser aceitável em cache (exploraremos esse tópico com mais detalhes em "Armazenamento em cache"). Mas se isso é um

operação frequente, que poderia ser mais problemática. Nós podemos mitigar o provável impacto desse aumento na latência ao permitir que os SKUs sejam analisados no microserviço Catalog em massa, ou talvez até mesmo armazenando em cache as informações necessárias do álbum localmente.

Integridade de dados

Os bancos de dados podem ser úteis para garantir a integridade de nossos dados. Voltando para

Figura 3-6, com as tabelas Album e Ledger na mesma

banco de dados, poderíamos (e provavelmente definiríamos) uma relação de chave estrangeira entre as linhas na tabela Ledger e na tabela Álbum. Isso seria garantir que sempre possamos navegar a partir de um registro no Ledger e retorne às informações sobre o álbum vendido, pois não conseguiríamos excluir registros da tabela Álbum se eles foram referenciados no Ledger.

Com essas tabelas agora em bancos de dados diferentes, não temos mais imposição da integridade do nosso modelo de dados. Não há nada que nos pare de excluir uma linha na tabela Álbum, causando um problema quando tentamos trabalhar descubra exatamente qual item foi vendido.

Até certo ponto, você simplesmente precisará se acostumar com o fato de que não pode mais confie em seu banco de dados para garantir a integridade dos relacionamentos entre entidades. Obviamente, para dados que permanecem em um único banco de dados, isso não é um problema.

Existem várias soluções alternativas, embora os "padrões de enfrentamento" sejam um termo melhor para maneiras pelas quais podemos lidar com esse problema. Poderíamos usar um mafio excluir na tabela Álbum para que não removemos realmente um registro, mas apenas marque-o como excluído. Outra opção poderia ser copiar o nome do álbum na tabela do Ledger quando uma venda é feita, mas teríamos que resolver como queríamos lidar com as mudanças de sincronização no nome do álbum.

Transações

Muitos de nós confiamos nas garantias que obtemos do gerenciamento de dados em transações. Com base nessa certeza, criamos aplicativos em um determinado forma, sabendo que podemos confiar no banco de dados para lidar com várias coisas.

para nós. No entanto, quando começamos a dividir os dados em vários bancos de dados, perder a segurança das transações ACID às quais estamos acostumados. (Eu explico o acrônimo ACID e discuta as transações ACID com mais profundidade no Capítulo 6.)

Para pessoas que estão saindo de um sistema no qual todas as mudanças de estado podem ser gerenciada em um único limite transacional, a mudança para sistemas distribuídos pode ser um choque, e muitas vezes a reação é procurar implementar distribuído transações para recuperar as garantias que as transações ACID nos deram arquiteturas mais simples. Infelizmente, como abordaremos em profundidade em "Banco de dados Transações", As transações distribuídas não são apenas complexas de implementar, mesmo quando bem feitos, mas eles também não nos dão o mesmo garantias que esperávamos com um banco de dados com escopo mais restrito transações.

À medida que exploramos em "Sagas", existem alternativas (e preferíveis) mecanismos para transações distribuídas para gerenciar mudanças de estado em vários microsserviços, mas eles vêm com novas fontes de complexidade. Como com a integridade dos dados, temos que aceitar o fato de que, ao quebrar além de nossos bancos de dados por motivos que podem ser muito bons, encontraremos um novo conjunto de problemas.

Ferramentas

Alterar bancos de dados é difícil por vários motivos, um dos quais é limitado as ferramentas permanecem disponíveis para nos permitir fazer alterações com facilidade. Com o código, nós temos ferramentas de refatoração incorporadas em nossos IDEs e temos o benefício adicional que os sistemas que estamos mudando são fundamentalmente sem estado. Com um banco de dados, as coisas que estamos mudando têm estado e também nos faltam coisas boas ferramentas do tipo refatoração.

Existem muitas ferramentas disponíveis para ajudar você a gerenciar o processo de mudança o esquema de um banco de dados relacional, mas a maioria segue o mesmo padrão. Cada alteração do esquema é definida em um script delta controlado por versão. Esses scripts são então executados em ordem estrita de forma idempotente. As migrações do Rails funcionam dessa forma, assim como o DBDeploy, uma ferramenta que ajudei a criar há muitos anos.

Hoje em dia, eu indico as pessoas para a Flyway ou a Liquibase para conseguir o mesmo resultado, se eles ainda não tiverem uma ferramenta que funcione dessa maneira.

Banco de dados de relatórios

Como parte da extração de microserviços de nosso aplicativo monolítico, também separar nossos bancos de dados, pois queremos ocultar o acesso aos nossos dados internos armazenamento. Ao ocultar o acesso direto aos nossos bancos de dados, somos mais capazes de criar interfaces estáveis, que possibilitam a implantação independente. Infelizmente, isso nos causa problemas quando temos casos de uso legítimos para acessar dados de mais de um microserviço ou quando esses dados são melhores disponibilizado em um banco de dados, em vez de por meio de algo como uma API REST.

Com um banco de dados de relatórios, em vez disso, criamos um banco de dados dedicado que é projetado para acesso externo, e nós fazemos disso a responsabilidade do microserviço para enviar dados do armazenamento interno para o acessível externamente banco de dados de relatórios, conforme mostrado na Figura 3-8.

O banco de dados de relatórios nos permite ocultar o gerenciamento interno do estado, enquanto ainda apresentando os dados em um banco de dados - algo que pode ser muito útil. Por exemplo, talvez você queira permitir que as pessoas executem um SQL definido ad hoc consultas, execute junções em grande escala ou faça uso de conjuntos de ferramentas existentes que esperam para ter acesso a um endpoint SQL. O banco de dados de relatórios é uma boa solução para esse problema.

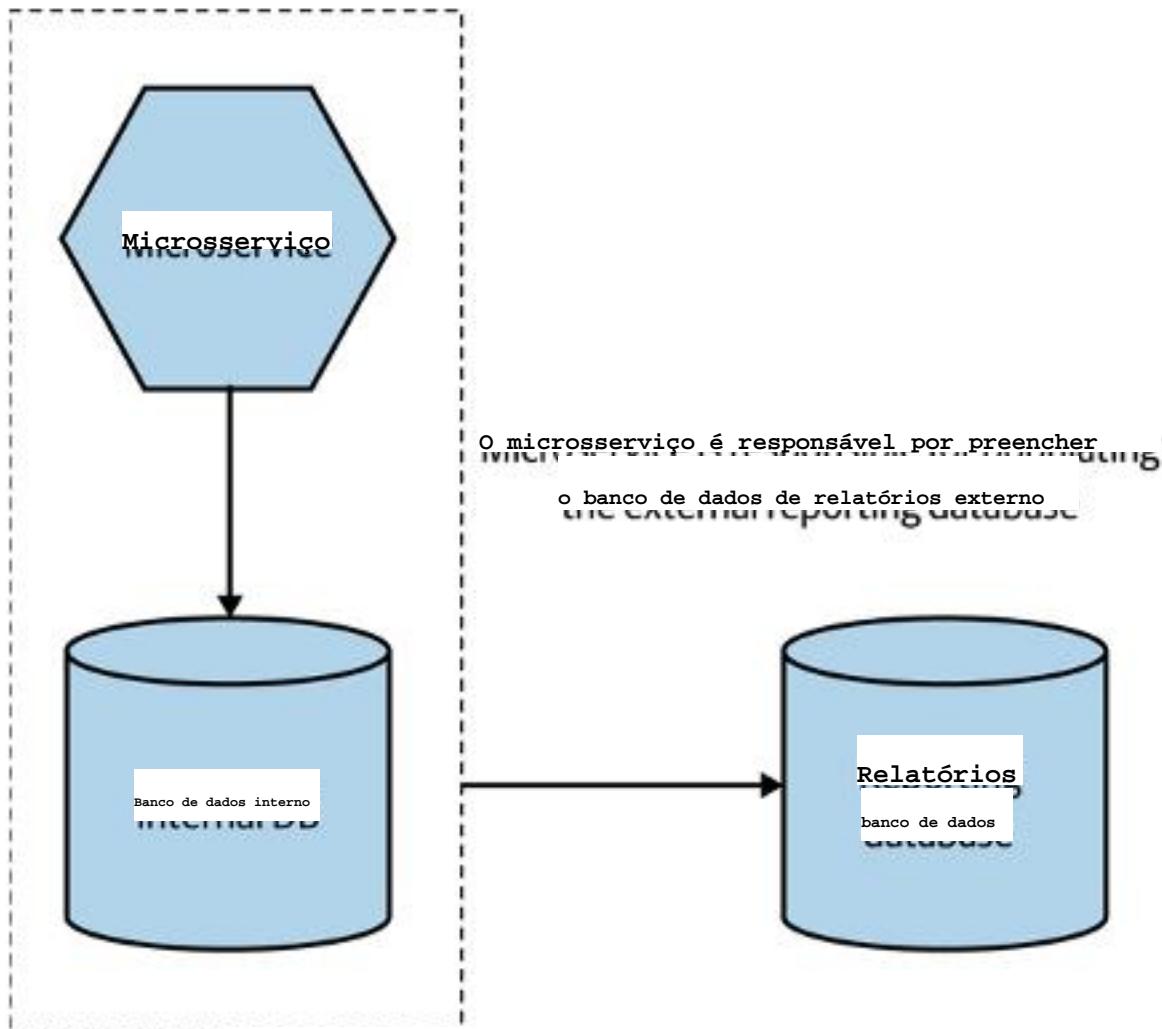


Figura 3-8. Uma visão geral do padrão do banco de dados de relatórios

Há dois pontos principais a serem destacados aqui. Em primeiro lugar, ainda queremos praticar ocultação de informações. Portanto, devemos expor apenas o mínimo de dados em o banco de dados de relatórios. Isso significa que o que está no banco de dados de relatórios pode seja apenas um subconjunto dos dados que o microsserviço armazena. No entanto, como este não é um mapeamento direto, ele cria a oportunidade de criar um design de esquema para o banco de dados de relatórios que é adaptado exatamente aos requisitos do consumidores - isso pode envolver o uso de um esquema radicalmente diferente, ou talvez até mesmo um tipo totalmente diferente de tecnologia de banco de dados.

O segundo ponto chave é que o banco de dados de relatórios deve ser tratado como qualquer outro outro endpoint de microsserviços, e esse é o trabalho do mantenedor de microsserviços para garantir que a compatibilidade desse endpoint seja mantida mesmo que o microsserviço altera seus detalhes internos de implementação. O mapeamento de

o estado interno do banco de dados de relatórios é de responsabilidade das pessoas que desenvolvem o microsserviço em si.

Resumo

Então, para simplificar as coisas, ao começar a trabalhar para migrar a funcionalidade de uma arquitetura monolítica a uma arquitetura de microsserviços, você deve ter uma compreensão clara do que você espera alcançar. Esse objetivo moldará como você realiza o trabalho e também o ajudará a entender se você está movendo-se na direção certa.

A migração deve ser incremental. Faça uma mudança, implemente essa mudança, avalie isso e vá novamente. Até mesmo o ato de dividir um microsserviço pode, por si só, ser dividido em uma série de pequenos passos.

Se você quiser explorar qualquer um dos conceitos deste capítulo com mais detalhes, outro livro que escrevi, *Monolith to Microservices* (O'Reilly), é um mergulho profundamente neste tópico.

Grande parte deste capítulo teve um nível um pouco alto em sua visão geral. No próximo capítulo, porém, começaremos a ficar mais técnicos quando veja como os microsserviços se comunicam entre si.

1 Sam Newman, *Monolith to Microservices* (Sebastopol: O'Reilly, 2019).

2 Pete Hodgson, "Feature Toggles (aka Feature Flags)", martinfowler.com, 9 de outubro de 2017, <https://oreil.ly/xiu2t>.

3 Obviamente, isso é uma simplificação de como seria um sistema do mundo real. Parece razoável, por exemplo, que registrássemos por quanto vendemos um item em uma área financeira.

Capítulo 4. Microsserviço

Estilos de comunicação

Obter a comunicação correta entre microsserviços é problemático para muitos devido em grande parte, eu sinto, ao fato de que as pessoas gravitam em torno de um escolhido abordagem tecnológica sem primeiro considerar os diferentes tipos de comunicação que eles possam querer. Neste capítulo, vou tentar separar os diferentes estilos de comunicação para ajudar você a entender os prós e os contras de cada uma, bem como qual abordagem melhor se adequará ao seu espaço problemático.

Analisaremos o bloqueio síncrono e o não bloqueio assíncrono mecanismos de comunicação, bem como comparação entre solicitação e resposta colaboração com colaboração orientada por eventos.

Ao final deste capítulo, você deve estar muito melhor preparado para entender as diferentes opções disponíveis para você e terão uma base conhecimento que ajudará quando começarmos a analisar mais detalhadamente questões de implementação nos capítulos seguintes.

Do processo ao processo entre processos

OK, vamos esclarecer as coisas mais fáceis primeiro, ou pelo menos o que eu espero que seja as coisas fáceis. Ou seja, chamadas entre diferentes processos em uma rede (entre processos) são muito diferentes das chamadas dentro de um único processo (em-processo). Em um nível, podemos ignorar essa distinção. É fácil, por exemplo, pensar em um objeto fazendo uma chamada de método em outro objeto e depois apenas mapeie essa interação para dois microsserviços se comunicando por meio de uma rede. Deixando de lado o fato de que os microsserviços não são apenas objetos, esse pensamento pode nos mete em muitos problemas.

Vejamos algumas dessas diferenças e como elas podem mudar a forma como você pensa nas interações entre seus microsserviços.

Desempenho

O desempenho de uma chamada em andamento é fundamentalmente diferente daquele de uma chamada entre processos. Quando eu faço uma chamada em andamento, o subjacente compilador e o tempo de execução podem realizar uma série de otimizações para reduzir o impacto da chamada, incluindo embutir a invocação para que seja como se estivesse lá nunca foi uma ligação em primeiro lugar. Essas otimizações não são possíveis com chamadas entre processos. Os pacotes precisam ser enviados. Espere a sobrecarga de um intermédio a chamada de processo deve ser significativa em comparação com a sobrecarga de uma chamada em andamento. O primeiro é muito mensurável - basta viajar de ida e volta com um único pacote em um dado o centro é medido em milissegundos, enquanto a sobrecarga de fazer um a chamada de método é algo com o qual você não precisa se preocupar.

Isso geralmente pode fazer com que você queira repensar as APIs. Uma API que faz sentido em o processo pode não fazer sentido em situações entre processos. Eu posso fazer um milhares de chamadas em um limite de API em andamento sem preocupação. Eu faço quer fazer mil chamadas de rede entre dois microsserviços? Talvez não..

Quando eu passo um parâmetro para um método, a estrutura de dados que eu passo normalmente não se move - o mais provável é que eu passe um ponteiro para um localização da memória. Transferência de um objeto ou estrutura de dados para outro método não exige que mais memória seja alocada para copiar os dados.

Ao fazer chamadas entre microsserviços em uma rede, por outro lado, os dados realmente precisam ser serializados em alguma forma que possa ser transmitida por meio de uma rede. Os dados então precisam ser enviados e desserializados por outro lado fim. Portanto, talvez precisemos estar mais atentos ao tamanho das cargas úteis. sendo enviado entre processos. Quando foi a última vez que você soube do tamanho de uma estrutura de dados que você estava transmitindo dentro de um processo? O A realidade é que você provavelmente não precisava saber; agora você sabe. Isso pode levar você deve reduzir a quantidade de dados enviados ou recebidos (talvez não seja ruim (se pensarmos em ocultar informações), escolha uma serialização mais eficiente mecanismos, ou até mesmo descarregar dados para um sistema de arquivos e passar uma referência em vez disso, para esse local de arquivo.

Essas diferenças podem não causar problemas imediatamente, mas você certamente preciso estar ciente deles. Eu vi muitas tentativas de me esconder do desenvolvedor o fato de que uma chamada de rede está mesmo ocorrendo. Nossa desejo de criar abstrações para ocultar detalhes é uma grande parte do que nos permite fazer mais coisas com mais eficiência, mas às vezes criamos abstrações que também se escondem muito. Um desenvolvedor precisa estar ciente de que está fazendo algo que o faça resultar em uma chamada de rede; caso contrário, você não deve se surpreender se acabar com alguns gargalos desagradáveis de desempenho mais adiante causados por interações estranhas entre serviços que não eram visíveis para o desenvolvedor que estava escrevendo o código.

Alterando interfaces

Quando consideramos mudanças em uma interface dentro de um processo, o ato de rolar. Mas a mudança é simples. O código que implementa a interface e o código que chama a interface é todo empacotado no mesmo processo.

Na verdade, se eu alterar a assinatura de um método usando um IDE com refatoração capacidade, geralmente o próprio IDE refatora automaticamente as chamadas para isso mudando o método. A implementação dessa mudança pode ser feita de forma atômica. -ambos os lados da interface são empacotados juntos em um único processo.

Com a comunicação entre microserviços, no entanto, o microserviço expondo uma interface e os microserviços consumidores usando essa interface são microserviços implantáveis separadamente. Ao fazer um retrocesso-alteração incompatível em uma interface de microserviço, ou precisamos fazer uma implantação contínua com os consumidores, garantindo que eles estejam atualizados para usar o nova interface, ou então encontre alguma maneira de implementar gradualmente a nova contrato de microserviço. Exploraremos esse conceito com mais detalhes posteriormente neste capítulo.

Tratamento de erros

Dentro de um processo, se eu chamar um método, a natureza dos erros tende a ser bonita direto. Simplisticamente, os erros são esperados e fáceis de

manipulam, ou são catastróficos a ponto de simplesmente propagarmos o erro. Isso aumenta a pilha de chamadas. Os erros, em geral, são determinísticos.

Com um sistema distribuído, a natureza dos erros pode ser diferente. Você é vulnerável a uma série de erros que estão fora de seu controle. Horário das redes está fora. Os microserviços downstream podem estar temporariamente indisponíveis. Redes são desconectadas, os contêineres são eliminados devido ao consumo excessivo de memória, e em situações extremas, partes do seu data center podem pegar fogo. 1

Em seu livro *Distributed Systems*, 2 Andrew Tanenbaum e Maarten Steen detalhe os cinco tipos de modos de falha que você pode ver ao analisar uma comunicação entre processos. Aqui está uma versão simplificada:

Falha na colisão

Tudo estava bem até o servidor falhar. Reinicie!

Falha de omissão

Você enviou algo, mas não recebeu uma resposta. Também inclui situações em que você espera que um microserviço downstream seja acionado mensagens (talvez incluindo eventos), e isso simplesmente para.

Falha de temporização

Algo aconteceu tarde demais (você não entendeu a tempo) ou algo assim aconteceu muito cedo!

Falha na resposta

Você recebeu uma resposta, mas parece errada. Por exemplo, você pediu um resumo do pedido, mas as informações necessárias estão faltando na resposta.

Falha arbitrária

Também conhecido como falha bizantina, é quando algo desaparece errado, mas os participantes não conseguem concordar se a falha ocorreu (ou por quê). Parece que esse é um momento ruim para todos.

Muitos desses erros geralmente são de natureza transitória - eles duram pouco, problemas que podem desaparecer. Considere a situação em que enviamos um solicite um microsserviço, mas não receba resposta (um tipo de falha de omissão). Isso pode significar que o microsserviço downstream nunca recebeu a solicitação o primeiro lugar, então precisamos enviá-lo novamente. Outros problemas não podem ser resolvidos com facilidade e pode precisar de um operador humano para intervir. Como resultado, pode torna-se importante ter um conjunto mais rico de semânticas para retornar erros em uma forma que possa permitir que os clientes tomem as medidas apropriadas.

O HTTP é um exemplo de protocolo que entende a importância disso. Cada resposta HTTP tem um código, com os códigos das séries 400 e 500 sendo reservados para erros. Os códigos de erro da série 400 são erros de solicitação - essencialmente, um o serviço downstream está dizendo ao cliente que há algo errado com a solicitação original. Como tal, provavelmente é algo de que você deveria desistir - vale a pena tentar novamente um 404 Not Found, por exemplo? A série 500 os códigos de resposta estão relacionados a problemas posteriores, um subconjunto dos quais indica que cliente de que o problema pode ser temporário. Um serviço 503 não disponível, para exemplo, indica que o servidor downstream não consegue lidar com a solicitação, mas pode ser um estado temporário, caso em que um cliente upstream pode decidir repetir a solicitação. Por outro lado, se um cliente receber um 501 Resposta não implementada, é improvável que uma nova tentativa ajude muito.

Se você escolhe ou não um protocolo baseado em HTTP para comunicação entre microsserviços, se você tiver um rico conjunto de semânticas em torno da natureza do erro, você facilitará que os clientes realizem a compensação ações, que, por sua vez, devem ajudá-lo a criar sistemas mais robustos.

Tecnologia para interprocessos

Comunicação: tantas opções

E em um mundo onde temos muitas opções e muito pouco tempo, o A coisa óbvia a fazer é simplesmente ignorar as coisas.

-Seth Godin

A variedade de tecnologia disponível para comunicação entre processos é vasto. Como resultado, muitas vezes podemos ficar sobrecarregados com opções. Muitas vezes acho que as pessoas gravitam em torno da tecnologia que lhes é familiar, ou talvez apenas da a mais recente tecnologia de ponta que eles aprenderam em uma conferência. O problema com isso é que, quando você compra uma opção de tecnologia específica, geralmente comprando um conjunto de ideias e restrições que surgem durante a viagem. Essas restrições podem não ser as certas para você - e a mentalidade por trás da tecnologia pode não estar de fato alinhada com o problema que você está tentando resolver.

Se você está tentando criar um site, tecnologia de aplicativo de página única, como Angular ou React não é uma boa opção. Da mesma forma, tentando usar o Kafka para solicitação-resposta realmente não é uma boa ideia, pois foi projetado para mais interações baseadas em eventos (tópicos que abordaremos em um momento). E, no entanto, vejo a tecnologia usada no lugar errado uma e outra vez. As pessoas escolhem a nova tecnologia brilhante (como microsserviços!), sem considerar se isso realmente se encaixa no problema deles.

Portanto, quando se trata da variedade desconcertante de tecnologia disponível para nós, para comunicação entre microsserviços, acho importante falar primeiro sobre o estilo de comunicação que você deseja, e só então procure o certo tecnologia para implementar esse estilo. Com isso em mente, vamos dar uma olhada em um modelo que venho usando há vários anos para ajudar a distinguir entre o diferentes abordagens para comunicação de microsserviço para microsserviço, o que, por sua vez, pode ajudá-lo a filtrar as opções de tecnologia que você deseja procurar em.

Estilos de comunicação por microsserviços

Na Figura 4-1, vemos um esboço do modelo que eu uso para pensar sobre diferentes estilos de comunicação. Este modelo não foi feito para ser inteiramente exaustivo (não estou tentando apresentar uma grande teoria unificada do interprocesso comunicação aqui), mas fornece uma boa visão geral de alto nível para considerando os diferentes estilos de comunicação que são mais amplamente utilizados para arquiteturas de microsserviços.

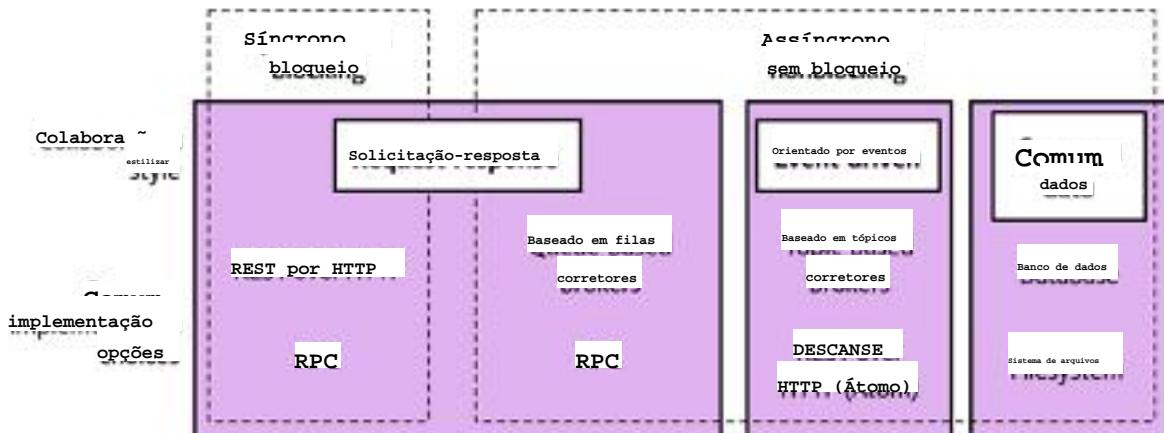


Figura 4-1. Diferentes estilos de comunicação entre microserviços, juntamente com exemplos de implementação tecnologias

Examinaremos os diferentes elementos desse modelo com mais detalhes em breve, mas primeiro, gostaria de descrevê-los brevemente:

Bloqueio síncrono

Um microserviço faz uma chamada para outro microserviço e bloqueia a operação aguardando a resposta.

Não bloqueador assíncrono

O microserviço que emite uma chamada é capaz de continuar processando se ou não, a chamada é recebida.

Solicitação-resposta

Um microserviço envia uma solicitação para outro microserviço solicitando algo a ser feito. Ele espera receber uma resposta informando-o sobre o resultado.

Orientado por eventos

Microserviços emitem eventos, que outros microserviços consomem e reagem para adequadamente. O microserviço que emite o evento não sabe qual dos microserviços, se houver, consomem os eventos que emitem.

Dados comuns

Nem sempre vistos como um estilo de comunicação, os microserviços colaboram via alguma fonte de dados compartilhada.

Ao usar esse modelo para ajudar as equipes a decidirem sobre a abordagem correta, eu gasto um muito tempo entendendo o contexto em que estão operando. Suas necessidades em termos de comunicação confiável, latência aceitável e volume de todas as comunicações desempenharão um papel na escolha de tecnologia. Mas, em geral, costumo começar decidindo se é uma solicitação-resposta ou uma o estilo de colaboração orientado por eventos é mais apropriado para o determinado situação. Se estou analisando a solicitação-resposta, então tanto síncrona quanto implementações assíncronas ainda estão disponíveis para mim, então eu tenho uma segunda escolha a fazer. Se escolher um estilo de colaboração orientado por eventos, no entanto, meu as opções de implementação serão limitadas a assíncronos sem bloqueio escolhas.

Uma série de outras considerações entram em jogo ao escolher a opção certa tecnologia que vai além do estilo de comunicação - por exemplo, a necessidade para comunicação de baixa latência, aspectos relacionados à segurança ou a capacidade de escala. É improvável que você possa fazer uma escolha tecnológica fundamentada sem levando em consideração os requisitos (e restrições) de sua especialidade espaço problemático. Quando analisarmos as opções de tecnologia no Capítulo 5, vamos discutir algumas dessas questões.

Misture e combine

É importante observar que uma arquitetura de microserviços como um todo pode ter uma mistura de estilos de colaboração, e isso normalmente é a norma. Alguns interações apenas fazem sentido como solicitação-resposta, enquanto outras fazem sentido como orientado por eventos. Na verdade, é comum que um único microserviço seja implementado mais de uma forma de colaboração. Considere um microserviço de pedido que expõe uma API de solicitação-resposta que permite que pedidos sejam feitos ou alterado e, em seguida, dispara eventos quando essas alterações são feitas.

Com isso dito, vamos analisar esses diferentes estilos de comunicação em mais detalhe.

Padrão: Bloqueio síncrono

Com uma chamada de bloqueio síncrono, um microsserviço envia uma chamada de algum tipo para um processo posterior (provavelmente outro microsserviço) e bloqueia até o a chamada foi concluída e, potencialmente, até que uma resposta seja recebida. Em Figura 4-2, o processador de pedidos envia uma chamada para o microsserviço Loyalty para informá-lo de que alguns pontos devem ser adicionados à conta do cliente.

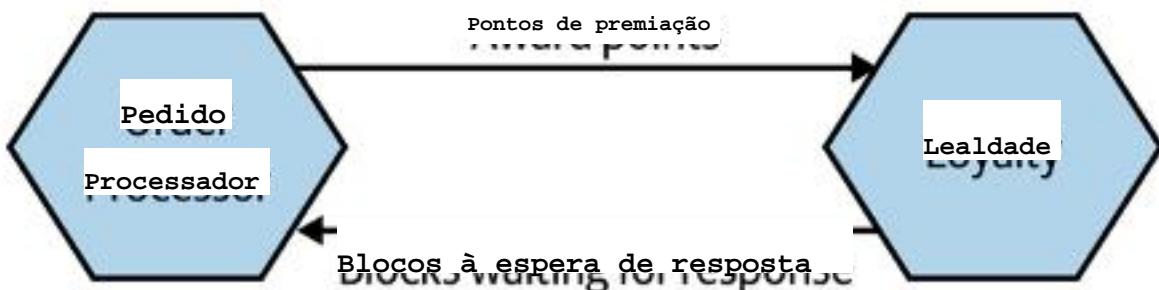


Figura 4-2. O Processador de Pedidos envia uma chamada síncrona para os blocos de microsserviços Loyalty, e espera por uma resposta.

Normalmente, uma chamada de bloqueio síncrono é aquela que está aguardando uma resposta do processo posterior. Isso pode ser porque o resultado da chamada é necessário para alguma operação adicional, ou apenas porque deseja garantir que o A chamada funcionou e, se não, realizar algum tipo de nova tentativa. Como resultado, virtualmente cada chamada de bloqueio síncrono que vejo também constituiria uma solicitação-chamada de resposta, algo que veremos em breve.

Vantagens

Há algo simples e familiar em uma chamada síncrona e bloqueada. Muitos de nós aprendemos a programar em um estilo fundamentalmente síncrono-lendo um trecho de código como um script, com cada linha sendo executada por vez, e com a próxima linha de código esperando sua vez de fazer alguma coisa. A maior parte do situções em que você teria usado chamadas entre processos provavelmente eram feito em um estilo de bloqueio síncrono, executando uma consulta SQL em um banco de dados, por exemplo, ou fazer uma solicitação HTTP de uma API downstream. Ao migrar de uma arquitetura menos distribuída, como a de uma única monólito de processo, aderindo às ideias que são familiares quando existem

muitas outras coisas novas que estão acontecendo podem fazer sentido.

Desvantagens

O principal desafio das chamadas síncronas é o acoplamento temporal inerente. Isso ocorre, um tópico que exploramos brevemente no Capítulo 2. Quando o pedido é enviado para o Loyalty, o Loyalty precisa fazer uma chamada para o Order Processor. O Order Processor precisa estar acessível para que a chamada funcione. Se a lealdade precisar de tempo para processar a chamada, o Order Processor precisará esperar. Se o Order Processor não estiver disponível, a chamada falhará e o Loyalty precisará descobrir que tipo de ação compensatória realizar - isso pode envolver uma nova tentativa imediata, armazenando a chamada para tentar novamente mais tarde ou talvez fornecendo completamente acordado.

Esse acoplamento é bidirecional. Com esse estilo de integração, a resposta é normalmente enviado pela mesma conexão de rede de entrada para o upstream microsserviço. Então, se o microsserviço Loyalty quiser enviar uma resposta de volta para o Order Processor, mas a instância upstream morreu posteriormente, a resposta será perdida. O acoplamento temporal aqui não é apenas entre dois microsserviços; está entre duas instâncias específicas desses microsserviços.

Enquanto o remetente da chamada está bloqueando e aguardando o downstream microsserviço para responder, também se segue que, se o microsserviço downstream responde lentamente ou, se houver um problema com a latência da rede, o remetente da chamada será bloqueado por um período prolongado de espera para obter uma resposta. Se o microsserviço Loyalty estiver sob carga significativa e for respondendo lentamente às solicitações, isso, por sua vez, fará com que o Processador de Pedidos para responder lentamente.

Assim, o uso de chamadas síncronas pode tornar um sistema vulnerável à cascata. Problemas causados por interrupções posteriores com mais rapidez do que o uso de chamadas assíncronas.

Onde usá-lo

Para arquiteturas de microsserviços simples, não tenho grandes problemas com o uso de chamadas síncronas e bloqueadoras. Sua familiaridade para muitas pessoas é

uma vantagem ao lidar com sistemas distribuídos.

Para mim, esses tipos de chamadas começam a ser problemáticos quando você comece a ter mais cadeias de chamadas - na Figura 4-3, por exemplo, temos um exemplo de fluxo da MusicCorp, onde verificamos um pagamento potencialmente fraudulento. Atividade. O Processador de Pedidos liga para o serviço de pagamento para receber o pagamento. O serviço de pagamento, por sua vez, deseja verificar com a Detecção de Fraude, microsserviço para saber se isso deve ou não ser permitido. A fraude microsserviço de detecção, por sua vez, precisa obter informações do cliente. microsserviço.

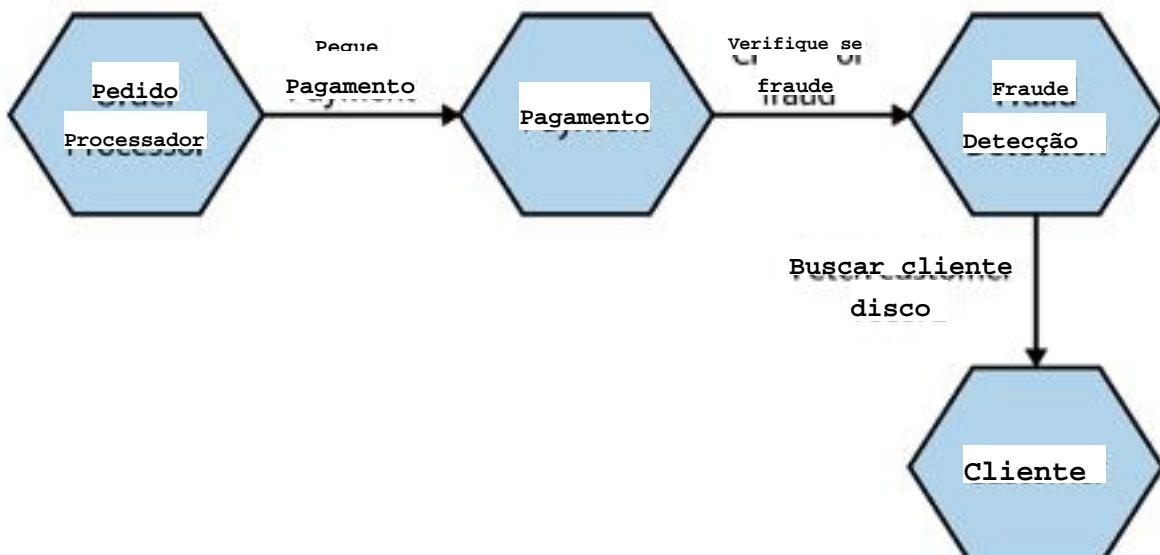


Figura 4-3. Verificação de comportamento potencialmente fraudulento como parte do fluxo de processamento de pedidos

Se todas essas chamadas forem síncronas e bloqueadas, haverá várias problemas que podemos enfrentar. Um problema em qualquer um dos quatro microserviços envolvidos, ou nas chamadas de rede entre eles, pode fazer com que toda a operação falhe. Isso além do fato de que esses tipos de cadeias longas podem causar disputa significativa de recursos. Nos bastidores, o Processador de Pedidos provavelmente tem uma conexão de rede aberta aguardando uma resposta do Payment. O pagamento, por sua vez, tem uma conexão de rede aberta, aguardando uma resposta de Detecção de fraudes e assim por diante. Ter muitas conexões que precisam ser mantidas abertas pode ter um impacto no sistema em execução - você é muito mais

~~provavelmente terá problemas nos quais você fica sem conexões disponíveis ou~~

~~Como resultado, sofrem com o aumento do congestionamento da rede.~~

~~Para melhorar essa situação, poderíamos reexaminar as interações entre o~~
~~microsserviços em primeiro lugar. Por exemplo, talvez usemos o uso de~~
~~Detectar fraude fora do fluxo principal, conforme mostrado na Figura 4-4.~~
~~e, em vez disso, faça com que seja executado em segundo plano. Se encontrar um problema com um problema específico~~
~~cliente, seus registros são atualizados adequadamente, e isso é algo que~~
~~poderia ser verificado mais cedo no processo de pagamento. Efetivamente, isso significa~~
~~estamos fazendo parte desse trabalho em paralelo. Ao reduzir a duração da chamada~~
~~cadeia, veremos a latência geral da operação melhorar e tomaremos~~
~~um de nossos microsserviços (detecção de fraude) fora do caminho crítico para o~~
~~fluxo de compra, o que nos dá uma dependência a menos com que nos preocupar com o que é um~~
~~operação crítica.~~

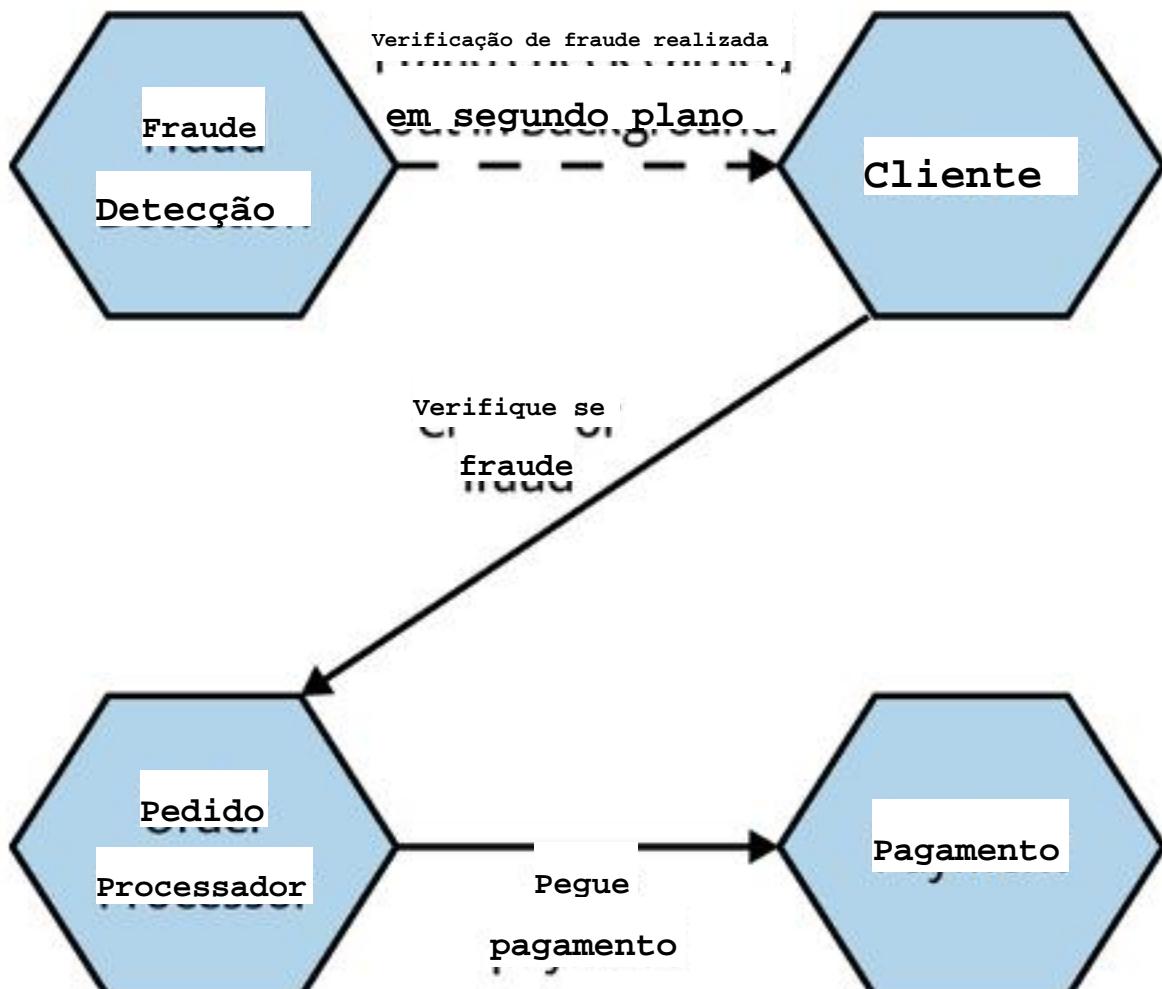


Figura 4-4. Transferir a detecção de fraudes para um processo em segundo plano pode reduzir as preocupações com o comprimento da cadeia de chamadas

Obviamente, também poderíamos substituir o uso do bloqueio de chamadas por algum estilo de interação sem bloqueio sem alterar o fluxo de trabalho aqui, uma abordagem vamos explorar a seguir.

Padrão: Assíncrono sem bloqueio

Com a comunicação assíncrona, o ato de enviar uma chamada pelo rede não bloqueia o microsserviço que emite a chamada. É capaz de continuar com qualquer outro processamento sem precisar esperar por uma resposta.

A comunicação assíncrona sem bloqueio vem em várias formas, mas vamos examiná-la com mais detalhes os três estilos mais comuns que vejo em arquitetura de microsserviços Eles são:

Comunicação por meio de dados comuns

O microsserviço upstream altera alguns dados comuns, quais ou mais microsserviços serão usados posteriormente.

Solicitação-resposta

Um microsserviço envia uma solicitação para outro microsserviço solicitando que ele faça alguma coisa. Quando a operação solicitada for concluída, se com sucesso ou não, o microsserviço upstream recebe a resposta. Especificamente, qualquer instância do microsserviço upstream deve ser capaz de lidar com a resposta.

Interação orientada por eventos

Um microsserviço transmite um evento, que pode ser considerado factual declaração sobre algo que aconteceu. Outros microsserviços podem ouvir os eventos nos quais eles estão interessados e reaja de acordo.

Vantagens

Com comunicação assíncrona sem bloqueio, o microsserviço torna o a chamada inicial e o microsserviço (ou microsserviços) que recebe a chamada são desacoplado temporariamente. Os microsserviços que recebem a chamada não precisam estar acessível ao mesmo tempo em que a chamada é feita. Isso significa que evitamos o preocupações de desacoplamento temporal que discutimos no Capítulo 2 (ver "A Breve nota sobre acoplamento temporal").

Esse estilo de comunicação também é benéfico se a funcionalidade for acionado por uma chamada levará muito tempo para ser processado. Vamos voltar ao nosso exemplo da MusicCorp e, especificamente, o processo de envio de um pacote. Na Figura 4-5, o Processador de Pedidos recebeu o pagamento e tem decidido que é hora de despachar o pacote, então ele envia uma chamada para o Microsserviço de armazém. O processo de encontrar os CDs, retirá-los do prateleira, empacotá-los e retirá-los pode levar muitas horas, e potencialmente até dias, dependendo de como o despacho real é processado.

funciona. Faz sentido, portanto, que o Processador de Pedidos emita um chamada assíncrona sem bloqueio para o Depósito e tenha o Depósito ligue mais tarde para informar o Processador de Pedidos sobre seu progresso. Este é um tipo de comunicação assíncrona de solicitação-resposta.

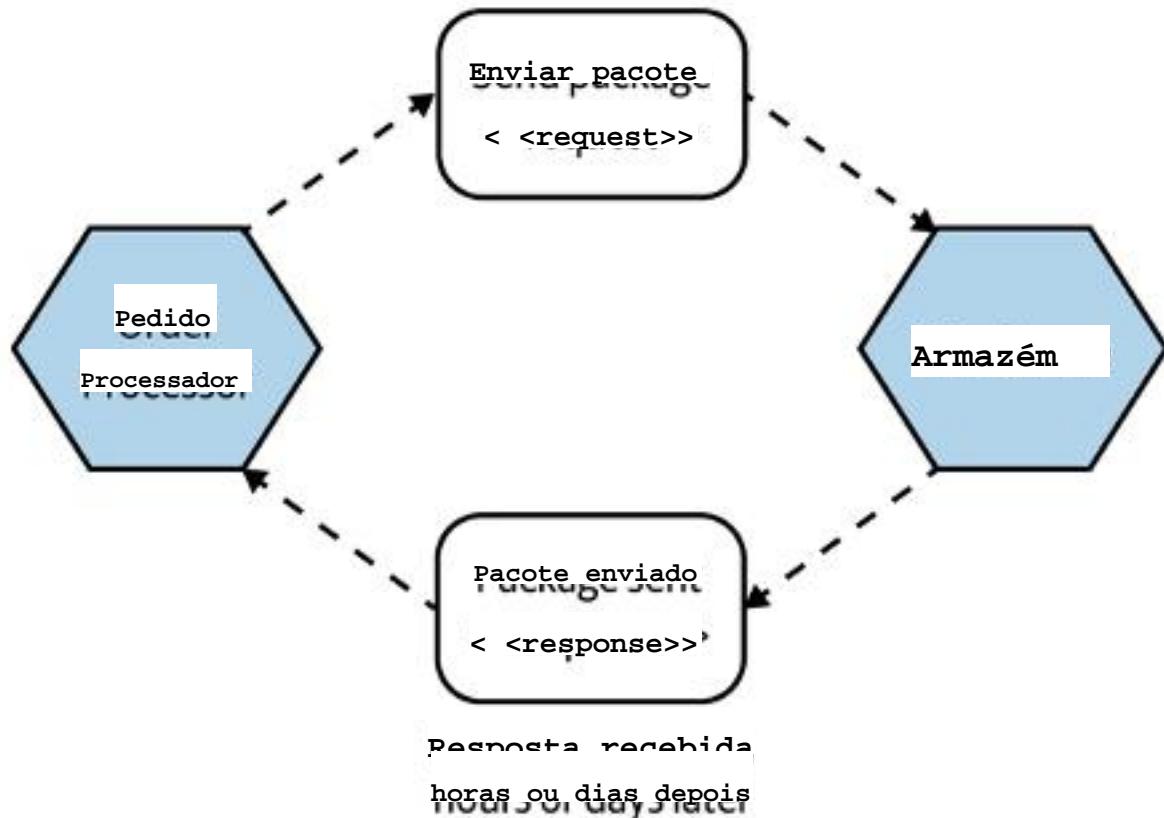


Figura 4-5. O Processador de Pedidos inicia o processo de empacotar e enviar um pedido, que é feito de forma assíncrona.

Se tentássemos fazer algo semelhante com chamadas de bloqueio síncronas, então teríamos que reestruturar as interações entre o Processador de Pedidos e Armazém - não seria viável para o Order Processor abrir um conexão, envie uma solicitação, bloqueie qualquer operação adicional ao chamar o thread, e aguarde uma resposta por horas ou dias.

Desvantagens

As principais desvantagens da comunicação assíncrona sem bloqueio, relativas para bloquear a comunicação síncrona, estão o nível de complexidade e a variedade de escolha. Como já descrevemos, existem diferentes estilos de

comunicação assíncrona para escolher - qual é a certa para você?

Quando começamos a investigar como são esses diferentes estilos de comunicação

implementada, há uma lista potencialmente desconcertante de tecnologias que poderíamos
veja.

Se a comunicação assíncrona não estiver mapeada para seus modelos mentais de
computação, adotando um estilo de comunicação assíncrono será
desafiador no início. E, como exploraremos mais detalhadamente,
os vários estilos de comunicação assíncrona, existem muitos diferentes,
maneiras interessantes pelas quais você pode se meter em muitos problemas.

ASYNC/AWAIT E QUANDO O ASSÍNCRONO AINDA ESTÁ BLOQUEANDO

Como em muitas áreas da computação, podemos usar o mesmo termo em diferentes contextos devem ter significados muito diferentes. Um estilo de programação que parece ser especialmente popular o uso de construções como `async/await` para trabalhar com uma fonte de dados potencialmente assíncrona, mas em um bloqueio, estilo síncrono.

No exemplo 4-1, vemos um exemplo muito simples em JavaScript disso em ação. As taxas de câmbio flutuam com frequência ao longo do dia, e nós os recebemos por meio de um agente de mensagens. Nós definimos uma promessa. Genericamente, uma promessa é algo que, em algum momento, se resolverá em um estado ponto no futuro. No nosso caso, nosso EUR/GBP acabará por resolver sendo a próxima taxa de câmbio de euro para GBP.

Exemplo 4-1. Um exemplo de como trabalhar com um potencial chamada assíncrona de forma síncrona e bloqueada

```
funcão assíncrona f () {  
    deixe EURtoGBP. nova promessa (resolver, rejeitar) => {  
        //código para obter a taxa de câmbio mais recente entre EUR e GBP  
        ..  
    };  
  
    var lateRate = aguardar EUR para GBP; ①  
    processo (LateRate); ②  
}  
  
① Espere até que a última taxa de câmbio de EUR para GBP seja obtida.  
② Não correrá até que a promessa seja cumprida.
```

Quando referenciamos `eur a GBP` usando `await`, bloqueamos até o estado do `LateRate` é cumprido - o processo não é alcançado até que resolvemos o estado do euro em relação ao GBP. ③

Mesmo que nossas taxas de câmbio estejam sendo recebidas de forma assíncrona moda, o uso de `await` neste contexto significa que estamos bloqueando até o

o estado de `latesRate` foi resolvido. Portanto, mesmo que a tecnologia subjacente estamos usando para obter a taxa que pode ser considerada de natureza assíncrona (por exemplo, aguardando a tarifa), do ponto de vista do nosso código, isso é inherentemente uma interação síncrona e bloqueadora.

Onde usá-lo

Em última análise, ao considerar se a comunicação assíncrona está correta para você, você também deve considerar qual tipo de assíncrono comunicação que você deseja escolher, pois cada tipo tem suas próprias desvantagens. Em geral, porém, existem alguns casos de uso específicos que me aceitariam buscando alguma forma de comunicação assíncrona Longa duração os processos são candidatos óbvios, conforme exploramos na Figura 4-5. Além disso, situações em que você tem cadeias de chamadas longas que não podem ser facilmente reestruturadas poderia ser um bom candidato. Vamos nos aprofundar nisso quando analisarmos três das formas mais comuns de solicitação de comunicação assíncrona- chamadas de resposta, comunicação orientada por eventos e comunicação por meio de dados comuns.

Padrão: Comunicação por meio de dados comuns

Um estilo de comunicação que abrange uma infinidade de implementações é comunicação por meio de dados comuns. Esse padrão é usado quando um microserviço coloca os dados em um local definido e outro microserviço (ou potencialmente vários microserviços) então faz uso dos dados. Pode ser como simples como um microserviço colocando um arquivo em um local e, em algum momento mais tarde, outro microserviço pega esse arquivo e faz algo com ele. Esse estilo de integração é fundamentalmente assíncrono por natureza.

Um exemplo desse estilo pode ser visto na Figura 4-6, onde o Novo Produto O importador cria um arquivo que será então lido pelo inventário posterior e microserviços de catálogo.

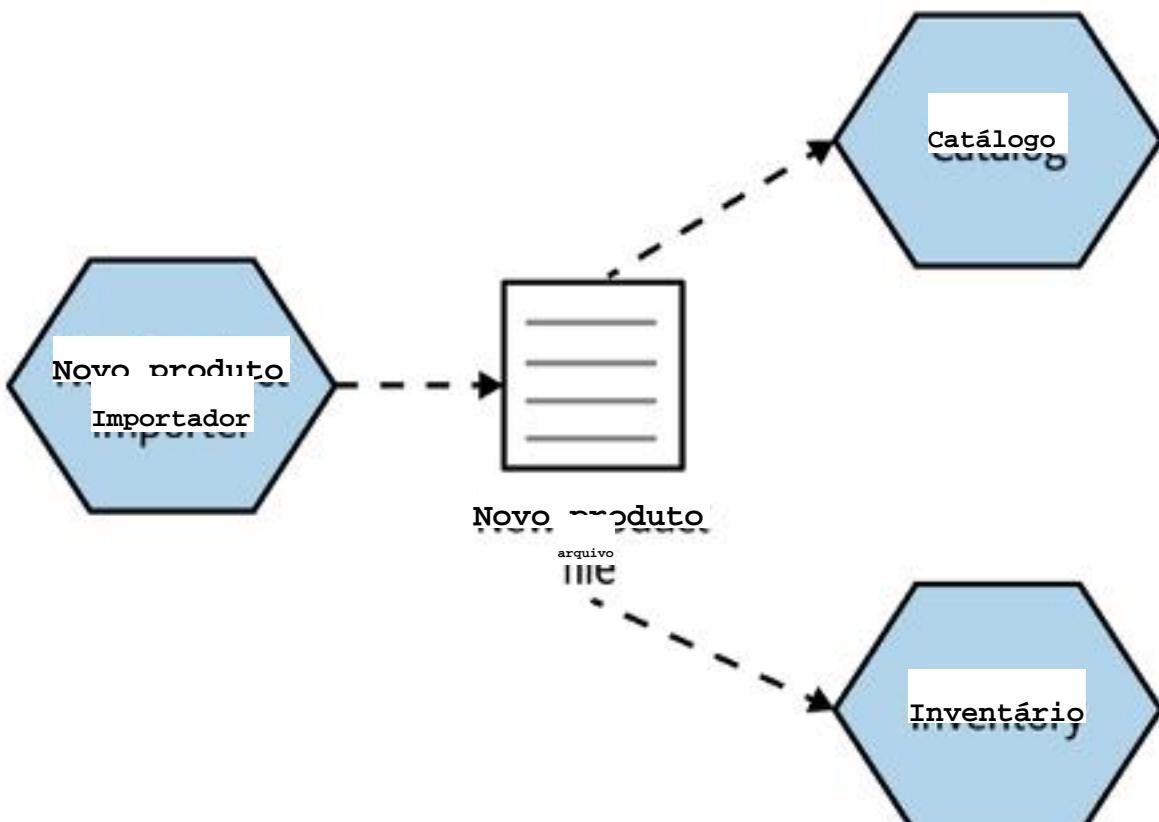


Figura 4-6. Um microsserviço grava um arquivo que outros microsserviços usam.

Esse padrão é, de certa forma, o interprocesso geral mais comum. Padrão de comunicação que você verá, mas às vezes deixamos de vê-lo como um padrão de comunicação em geral, acho que em grande parte porque a comunicação entre os processos geralmente é tão indireta que é difícil de detectar.

Implantação

Para implementar esse padrão, você precisa de algum tipo de armazenamento persistente para os dados. Em muitos casos, um sistema de arquivos pode ser suficiente. Eu construí muitos sistemas que apenas escaneie periodicamente um sistema de arquivos, observe a presença de um novo arquivo e reaja a ele em conformidade. Você poderia usar algum tipo de armazenamento robusto de memória distribuída como bem. É claro. É importante notar que qualquer microsserviço downstream que seja a agir com base nesses dados precisará de seu próprio mecanismo para identificar esses novos os dados estão disponíveis - a pesquisa é uma solução frequente para esse problema.

Dois exemplos comuns desse padrão são o data lake e os dados armazém. Em ambos os casos, essas soluções geralmente são projetadas para ajudar

processam grandes volumes de dados, mas é provável que eles existam em extremos opostos do espectro em relação ao acoplamento. Com um data lake, as fontes carregam dados brutos no qualquer formato que considerem adequado, e os consumidores posteriores desses dados brutos são esperava-se que soubesse como processar as informações. Com um data warehouse, o próprio armazém é um armazenamento de dados estruturado. Microsserviços enviando dados para o data warehouse precisa conhecer a estrutura do data warehouse - se a estrutura muda de uma forma incompatível com versões anteriores, então esses produtores irão precisam ser atualizados.

Com o data warehouse e o data lake, a suposição é que o fluxo de informações está em uma única direção. Um microsserviço publica dados para o armazenamento de dados comum, e os consumidores posteriores leem esses dados e realizar ações apropriadas. Esse fluxo unidirecional pode facilitar razão sobre o fluxo de informações. Uma implementação mais problemática seria o uso de um banco de dados compartilhado no qual vários microsserviços ler e gravar no mesmo armazenamento de dados, um exemplo do qual discutimos em O capítulo 2, quando exploramos o acoplamento comum. A figura 4-7 mostra ambos

Processador de pedidos e armazém atualizando o mesmo registro.

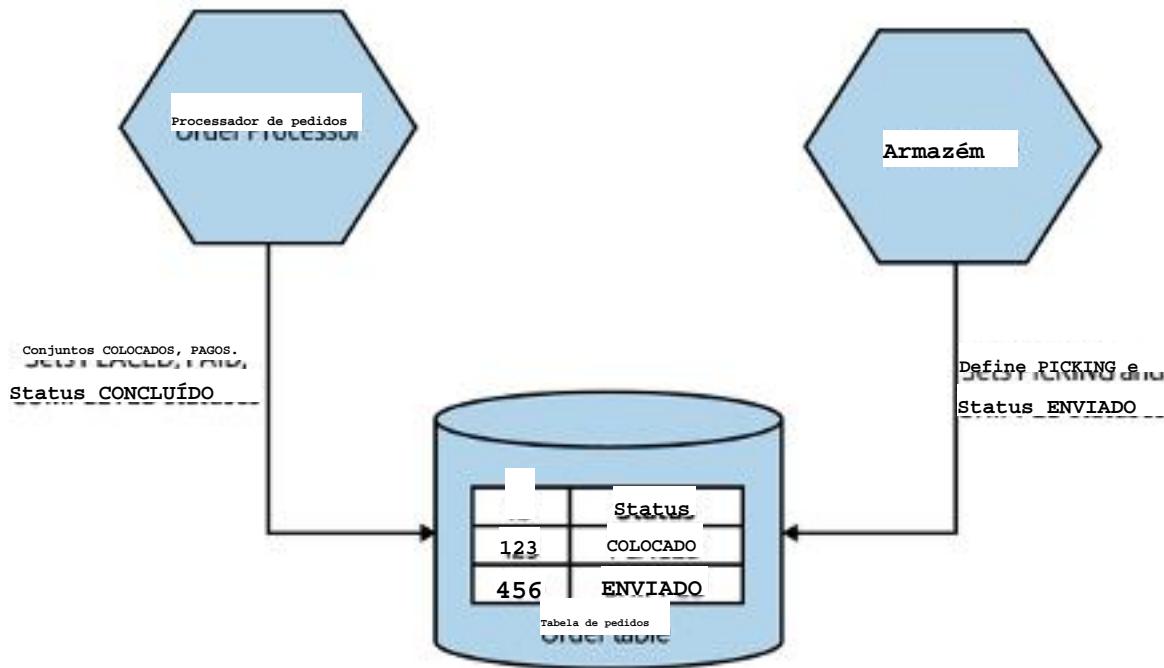


Figura 4-7 Um exemplo de acoplamento comum em que tanto o processador de pedidos quanto o armazém estão atualizando o mesmo registro de pedido

Vantagens

Esse padrão pode ser implementado de forma muito simples, usando o comumente entendido tecnologia. Se você puder ler ou gravar em um arquivo ou ler e gravar em um banco de dados, você pode usar esse padrão. O uso de substâncias prevalentes e bem compreendidas a tecnologia também permite a interoperabilidade entre diferentes tipos de sistemas, incluindo aplicativos de mainframe mais antigos ou personalizáveis prontos para uso produtos de software (COTS). Os volumes de dados também são menos preocupantes aqui, se se você estiver enviando muitos dados de uma só vez, esse padrão pode funcionar bem.

Desvantagens

Os microsserviços que consomem downstream normalmente estarão cientes de que existem novos dados para processar por meio de algum tipo de mecanismo de pesquisa, ou talvez por meio de um trabalho cronometrado acionado periodicamente. Isso significa que esse mecanismo é pouco provável que seja útil em situações de baixa latência. É claro que você pode combinar esse padrão com algum outro tipo de chamada informando um downstream... microsserviço em que novos dados estão disponíveis. Por exemplo, eu poderia escrever um arquivo para um sistema de arquivos compartilhado e, em seguida, envie uma chamada para o microsserviço interessado informando que há novos dados que ele pode querer. Isso pode fechar a lacuna entre os dados publicados e os dados processados. Em geral, porém, se você está usando esse padrão para volumes muito grandes de dados, é menos provável que a baixa latência está no topo da sua lista de requisitos. Se você estiver interessado em enviando volumes maiores de dados e fazendo com que eles sejam mais processados em "reais" time", então usar algum tipo de tecnologia de streaming como Kafka seria um melhor ajuste.

Outra grande desvantagem, e algo que deveria ser bastante óbvio se você lembre-se de nossa exploração do acoplamento comum na Figura 4-7, é que o armazenamento comum de dados se torna uma fonte potencial de acoplamento. Se esses dados armazenar mudanças de estrutura de alguma forma, pode interromper a comunicação entre microsserviços.

A robustez da comunicação também se resumirá à robustez do armazenamento de dados subjacente. Isso não é uma desvantagem estritamente falando, mas é algo que você deve conhecer. Se você estiver soltando um arquivo em um sistema de arquivos,

talvez queira ter certeza de que o sistema de arquivos em si não falhará ... maneiras interessantes.

Onde usá-lo

Onde esse padrão realmente brilha é ao permitir a interoperabilidade entre processos que podem ter restrições sobre qual tecnologia eles podem usar.

Ter um sistema existente conversando com a interface GRPC do seu microserviço ou assinar seu tópico de Kafka pode muito bem ser mais conveniente do ponto de vista de visão do microserviço, mas não do ponto de vista do consumidor.

Sistemas mais antigos podem ter limitações sobre a tecnologia que eles podem suportar e pode ter altos custos de mudança. Por outro lado, até mesmo o antigo mainframe os sistemas devem ser capazes de ler dados de um arquivo. É claro que isso faz tudo depender do uso de tecnologia de armazenamento de dados que seja amplamente suportada - eu também poderia implementar esse padrão usando algo como um cache Redis. Mas pode o seu antigo sistema de mainframe falar com o Redis?

Outro ponto ideal para esse padrão é o compartilhamento de grandes volumes de dados.

Se você precisar enviar um arquivo de vários gigabytes para um sistema de arquivos ou carregar alguns milhões linhas em um banco de dados, então esse padrão é o caminho a seguir.

Padrão: Comunicação entre solicitação e resposta

Com a solicitação-resposta, um microserviço envia uma solicitação para um downstream serviço solicitando que ele faça algo e espera receber uma resposta com o resultado da solicitação. Essa interação pode ser realizada por meio de um síncrono bloqueando a chamada, ou pode ser implementada em uma chamada assíncrona sem bloqueio moda. Um exemplo simples dessa interação é mostrado na Figura 4-8, onde o microserviço Chart, que reúne os CDs mais vendidos para diferentes gêneros, envia uma solicitação ao serviço de inventário solicitando o estoque atual níveis para alguns CDs.

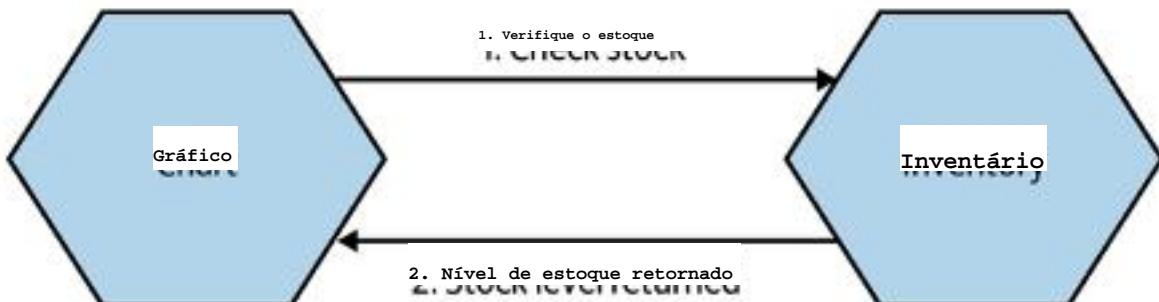


Figura 4-8. O microsserviço Chart envia uma solicitação ao Inventário solicitando os níveis de estoque

Recuperar dados de outros microsserviços como esse é um caso de uso comum para um chamada de solicitação-resposta. Às vezes, porém, você só precisa ter certeza algo é feito. Na Figura 4-9, o microsserviço Warehouse recebe um solicitação do Processador de Pedidos solicitando que ele reserve estoque. A Ordem de processador só precisa saber que o estoque foi reservado com sucesso antes que possa continuar com o pagamento. Se o estoque não puder ser reservado talvez porque um item não esteja mais disponível, o pagamento pode ser cancelado. Usando chamadas de solicitação-resposta em situações como essa, em que chamadas precisar ser preenchido em uma determinada ordem é comum.

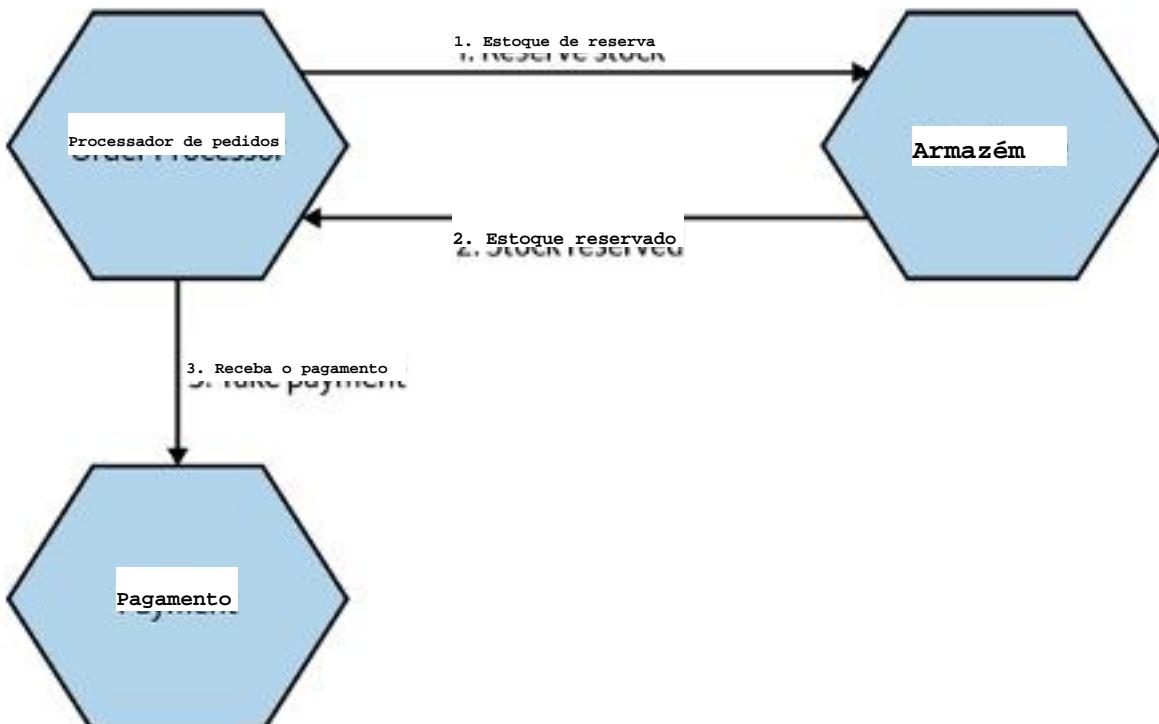


Figura 4-9. O processador de pedidos precisa garantir que o estoque possa ser reservado antes que o pagamento seja feito.

COMANDOS VERSUS SOLICITAÇÕES

Já ouvi algumas pessoas falarem sobre o envio de comandos, em vez de solicitações, especificamente no contexto de solicitação-resposta assíncrona comunicação. A intenção por trás do termo comando é indiscutivelmente a igual ao da solicitação, ou seja, um microsserviço upstream está solicitando um microsserviço downstream para fazer alguma coisa.

Pessoalmente falando, porém, eu prefiro muito mais o termo solicitação. Um comando implica uma diretiva que deve ser obedecida e pode levar a uma situação em que as pessoas acham que o comando deve ser cumprido. Um pedido implica algo que pode ser rejeitado. É certo que um microsserviço examina cada solicitação por seus méritos e, com base em sua própria lógica interna, decide se a solicitação deve ser atendida. Se a solicitação foi enviado viola a lógica interna, o microsserviço deve rejeitá-la. Apesar é uma diferença sutil, não acho que o termo comando transmita o mesmo significado.

Vou continuar usando solicitação em vez de comando, mas seja qual for o termo que você decidir para usar, lembre-se de que um microsserviço pode rejeitar o solicitação/comando, se apropriado.

Implementação: síncrona versus assíncrona

Chamadas de solicitação-resposta como essa podem ser implementadas em um bloqueio estilo síncrono ou assíncrono sem bloqueio. Com uma chamada síncrona, o que você normalmente vê é uma conexão de rede sendo aberta com o microsserviço downstream, com a solicitação sendo enviada por meio dessa conexão. A conexão é mantida aberta enquanto o microsserviço upstream aguarda o microsserviço downstream para responder. Nesse caso, o envio do microsserviço a resposta realmente não precisa saber nada sobre o microsserviço que enviou a solicitação - é apenas enviar coisas de volta por uma conexão de entrada. Se essa conexão morre, talvez porque a corrente ascendente ou a jusante a instância de microsserviço morre, então podemos ter um problema.

Com uma solicitação-resposta assíncrona, as coisas são menos simples. Vamos revisar o processo associado à reserva de estoque. Na Figura 4-10, a solicitação de reserva de estoque é enviada como uma mensagem sobre algum tipo de mensagem corretor (exploraremos os corretores de mensagens mais adiante neste capítulo). Em vez do mensageiro indo diretamente para o microsserviço de inventário do Pedido Processador, em vez disso, ele fica em uma fila. O inventário consome mensagens dessa fila quando possível. Ele lê a solicitação, executa o trabalho associado de reservar o estoque e, em seguida, precisa enviar a resposta voltar para uma fila que o Order Processor está lendo do The Inventory. O microsserviço precisa saber para onde encaminhar a resposta. Em nosso exemplo, envia essa resposta de volta por outra fila que, por sua vez, é consumida por Processador de pedidos.

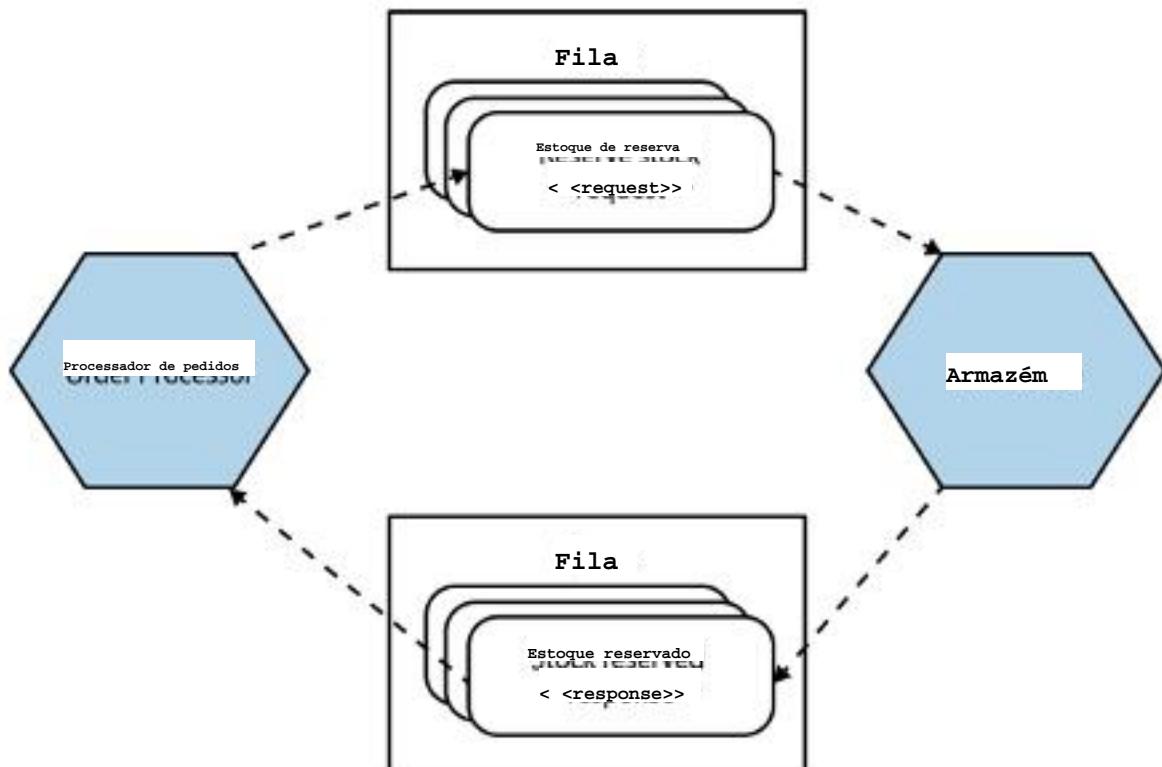


Figura 4-10 Usando filas para enviar solicitações de reserva de estoque

Então, com uma interação assíncrona sem bloqueio, o microsserviço que recebe a solicitação ou precisa saber implicitamente para onde rotear o resposta ou então saber para onde a resposta deve ir. Ao usar um fila, temos o benefício adicional de que várias solicitações podem ser armazenadas em buffer.

na fila esperando para ser tratada. Isso pode ajudar em situações em que o...
as solicitações não podem ser tratadas com rapidez suficiente. O microserviço pode consumir o...
próxima solicitação quando estiver pronta, em vez de ficar sobrecarregada por muitas...
chamadas. Obviamente, muito depende da fila que absorve essas solicitações.

Quando um microserviço recebe uma resposta dessa forma, talvez ele precise se relacionar
a resposta à solicitação original. Isso pode ser um desafio, pois muitas vezes
pode ter passado e, dependendo da natureza do protocolo que está sendo usado, o
a resposta pode não voltar para a mesma instância do microserviço que
enviou a solicitação. Em nosso exemplo de reserva de estoque como parte da colocação de um
pedido, precisaríamos saber como associar a resposta de "estoque reservado"
com um determinado pedido para que possamos continuar processando esse pedido específico. Um
uma maneira fácil de lidar com isso seria armazenar qualquer estado associado ao
solicitação original em um banco de dados, de forma que, quando a resposta chegar, o
a instância receptora pode recarregar qualquer estado associado e agir de acordo.

Uma última observação: todas as formas de interação solicitação-resposta provavelmente serão
exigem alguma forma de tratamento de tempo limite para evitar problemas no sistema.
bloqueado esperando por algo que talvez nunca aconteça. Como é esse intervalo
a funcionalidade implementada pode variar de acordo com a implementação
tecnologia, mas será necessária. Analisaremos os intervalos com mais detalhes em
Capítulo 12.

CHAMADAS PARALELAS VERSUS SEQUENCIAIS

Ao trabalhar com interações de solicitação-resposta, você frequentemente encontrará uma situação em que você precisará fazer várias chamadas antes de poder continuar com algum processamento.

Considere uma situação em que a MusicCorp precisa verificar o preço de um determinado item de três armazeneiros diferentes, o que fazemos emitindo API chamadas. Queremos recuperar os preços de todos os três armazeneiros antes de decidindo de qual delas queremos encomendar novas ações. Podemos decidir para fazer as três chamadas em sequência, esperando que cada uma termine antes de prosseguir com a próxima. Em tal situação, estariamos esperando por a soma das latências de cada uma das chamadas. Se a API chamar cada provedor demorou um segundo para retornar, estariamos esperando três segundos antes de podermos decidir de quem devemos fazer o pedido.

Uma opção melhor seria executar essas três solicitações em paralelo; então o a latência geral da operação seria baseada na chamada de API mais lenta, em vez da soma das latências de cada chamada de API.

Extensões e mecanismos reativos como `async/await` podem ser muito úteis em ajudar a executar chamadas em paralelo, e isso pode resultar em significativas melhorias na latência de algumas operações.

Onde usá-lo

As chamadas de solicitação-resposta fazem todo o sentido para qualquer situação em que o resultado de uma solicitação é necessário antes que um processamento adicional possa ocorrer. Eles também se encaixam muito bem em situações em que um microserviço quer saber se uma chamada não funcionou para que pudesse realizar algum tipo de ação compensatória, como um tente novamente. Se algum deles estiver alinhado com sua situação, a solicitação-resposta é sensata abordagem; a única questão restante é decidir sobre uma relação síncrona versus implementação assíncrona, com as mesmas vantagens que discutimos anteriormente.

Padrão: Comunicação orientada por eventos

A comunicação orientada por eventos parece bastante estranha em comparação com a solicitação-resposta chamadas. Em vez de um microsserviço pedindo que outro microsserviço faça algo, um microsserviço emite eventos que podem ou não ser recebidos por outros microsserviços. É uma interação inherentemente assíncrona, como o evento os ouvintes serão executados em seu próprio segmento de execução.

Um evento é uma declaração sobre algo que ocorreu, quase sempre algo que aconteceu dentro do mundo do microsserviço que é emitindo o evento. O microsserviço que emite o evento não tem conhecimento de a intenção de outros microsserviços de usar o evento e, na verdade, pode nem mesmo esteja ciente de que existem outros microsserviços. Ele emite o evento quando necessário, e esse é o fim de suas responsabilidades.

Na Figura 4-11, vemos o Warehouse emitindo eventos relacionados ao processo de empacotar um pedido. Esses eventos são recebidos por dois microsserviços, Notificações e inventário, e eles reagem de acordo. O microsserviço de notificações envia um e-mail para atualizar nosso cliente sobre mudanças no status do pedido, enquanto o microsserviço de inventário pode atualizar o estoque níveis à medida que os itens são embalados no pedido do cliente.

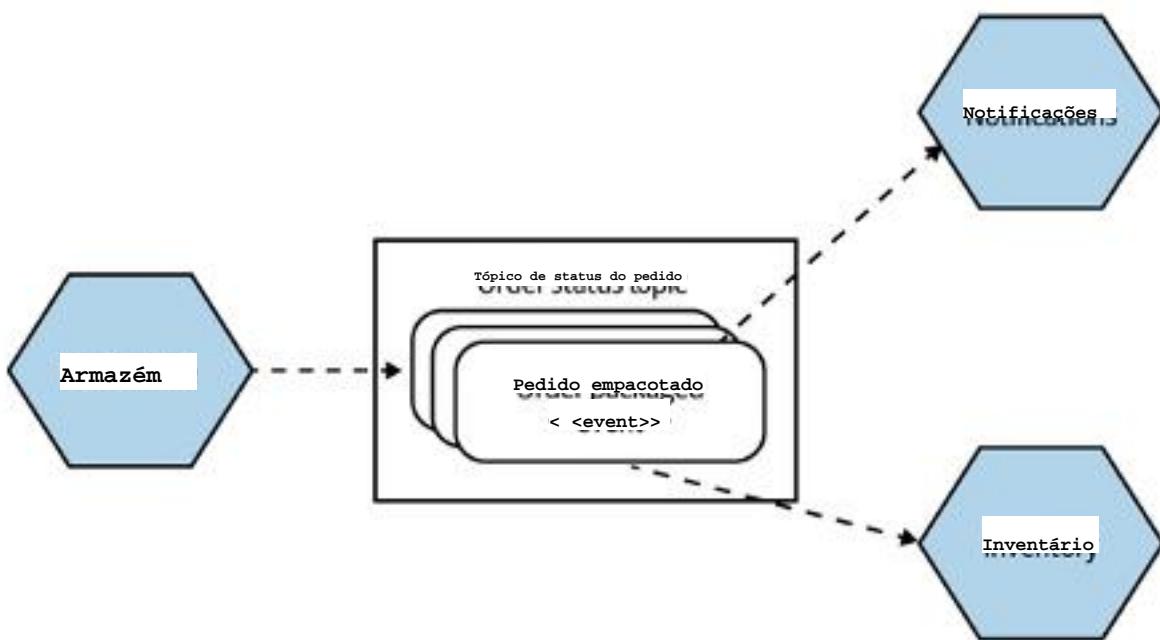


Figura 4-11. O Warehouse emite eventos que alguns microsserviços downstream assinam

The Warehouse está apenas transmitindo eventos, assumindo que as partes interessadas reagirão de acordo. Não sabe quem são os destinatários dos eventos, tornando as interações orientadas por eventos muito mais fracamente acopladas em geral. Quando você compara isso a uma chamada de solicitação-resposta, pode demorar um pouco até entenda a inversão da responsabilidade. Com solicitação-resposta, em vez disso, podemos esperar que o Warehouse informe o microserviço Notifications para enviar e-mails quando apropriado. Nesse modelo, Warehouse precisaria saiba quais eventos exigem a notificação do cliente. Com um orientado por eventos interação, em vez disso, estamos empurrando essa responsabilidade para o Microserviço de notificações.

A intenção por trás de um evento pode ser considerada o oposto de uma solicitação. O emissor do evento está deixando que os destinatários decidam o que fazer. Com solicitação-resposta, o microserviço que envia a solicitação sabe o que deve ser feito e está dizendo ao outro microserviço o que ele acha que precisa acontecer próximo. Obviamente, isso significa que, na solicitação-resposta, o solicitante deve ter conhecimento do que o destinatário posterior pode fazer, o que implica uma maior grau de acoplamento de domínio. Com a colaboração orientada por eventos, o evento o emissor não precisa saber o que qualquer microserviço downstream é capaz de fazer existem e, na verdade, podem nem saber que existem - como resultado, o acoplamento é muito importante reduzido

A distribuição de responsabilidade que vemos em nossas interações orientadas por eventos pode refletir a distribuição de responsabilidade que vemos nas organizações que estão tentando para criar equipes mais autônomas. Em vez de assumir toda a responsabilidade centralmente, queremos colocar isso nas próprias equipes para permitir que elas operar de uma forma mais autônoma - um conceito que revisitaremos em Capítulo 15. Aqui, estamos transferindo a responsabilidade do Warehouse para Notificações e pagamentos - isso pode nos ajudar a reduzir a complexidade do microserviços como o Warehouse e levam a uma distribuição mais uniforme de "inteligência" em nosso sistema. Exploraremos essa ideia com mais detalhes quando compare coreografia e orquestração no Capítulo 6.

EVENTS AND MESSAGES EVENTOS E MENSAGENS

Ocasionalmente, vi os termos mensagens e eventos se confundirem. Um evento é um fato - uma afirmação de que algo aconteceu, junto com algumas informações sobre exatamente o que aconteceu. Uma mensagem é algo que enviamos por meio de um mecanismo de comunicação assíncrona, como um agente de mensagens.

Com a colaboração orientada por eventos, queremos transmitir esse evento e um uma forma típica de implementar esse mecanismo de transmissão seria colocar o evento em uma mensagem. A mensagem é o meio; o evento é o carga útil.

Da mesma forma, talvez queiramos enviar uma solicitação como carga útil de uma mensagem - nesse caso, estariámos implementando uma forma de assíncrono solicitação-resposta.

Implantação

Há dois aspectos principais que precisamos considerar aqui: uma forma de microserviços para emitir eventos e uma forma de nossos consumidores descobrirem esses eventos eventos aconteceram.

Tradicionalmente, corretores de mensagens como o RabbitMQ tentam lidar com os dois problemas.

Os produtores usam uma API para publicar um evento na corretora. O corretor administra assinaturas, permitindo que os consumidores sejam informados quando um evento chegar.

Esses corretores podem até mesmo lidar com o estado dos consumidores, por exemplo, por ajudando a acompanhar quais mensagens eles viram antes. Esses sistemas normalmente são projetados para serem escaláveis e resilientes, mas isso não acontece grátis. Ele pode adicionar complexidade ao processo de desenvolvimento, porque é outro sistema que você pode precisar executar para desenvolver e testar seus serviços. Adicional

máquinas e experiência também podem ser necessários para manter essa infraestrutura em funcionamento e correndo. Mas, uma vez que é, pode ser uma forma incrivelmente eficaz de implementar arquiteturas fracamente acopladas e orientadas a eventos. Em geral, sou fã.

Porém, tenha cuidado com o mundo do middleware, do qual a mensagem corretor é apenas uma pequena parte. As filas por si só são perfeitas.

coisas sensatas e úteis. No entanto, os fornecedores tendem a querer empacotar muitos software com eles, o que pode fazer com que cada vez mais inteligência seja promovida no middleware, conforme evidenciado por coisas como o barramento de serviço corporativo. Certifique-se de saber o que está comprando: mantenha seu middleware idiota e mantenha a inteligência nos endpoints.

Outra abordagem é tentar usar o HTTP como forma de propagar eventos. Atom é uma especificação compatível com REST que define semântica (entre outras coisas) para publicar feeds de recursos. Existem muitas bibliotecas de clientes que nos permitem criar e consumir esses feeds. Então, nosso serviço ao cliente poderia basta publicar um evento nesse feed sempre que nosso atendimento ao cliente mudar. Nossos consumidores simplesmente pesquisam o feed em busca de mudanças. Por um lado, o fato de que podemos reutilizar a especificação Atom existente e qualquer outra associada bibliotecas são úteis e sabemos que o HTTP lida muito bem com a escala. No entanto, esse uso de HTTP não é bom em baixa latência (onde algumas mensagens corretores (excel), e ainda precisamos lidar com o fato de que os consumidores precisam acompanhe quais mensagens eles viram e gerencie suas próprias pesquisas cronograma).

Tenho visto pessoas passarem anos implementando cada vez mais comportamentos que você tira da caixa com um corretor de mensagens apropriado para criar o Atom funcionam para alguns casos de uso. Por exemplo, o padrão de consumo concorrente descreve um método pelo qual você abre várias instâncias de trabalho para compita por mensagens, o que funciona bem para aumentar o número de trabalhadores para lidar com uma lista de empregos independentes (voltaremos a isso no próximo capítulo). No entanto, queremos evitar o caso em que dois ou mais os trabalhadores veem a mesma mensagem, pois acabaremos fazendo mais a mesma tarefa do que precisamos. Com um agente de mensagens, uma fila padrão resolverá isso. Com o Atom, agora precisamos gerenciar nosso próprio estado compartilhado entre todos os trabalhadores para tentar reduzir as chances de reproduzir o esforço.

Se você já tem um corretor de mensagens bom e resiliente disponível, considere usá-lo para lidar com a publicação e a assinatura de eventos. Se você não já tenho um, dê uma olhada no Atom, mas esteja ciente da falácia do custo irrecuperável. Se você se vê querendo cada vez mais apoio do que uma mensagem. O corretor fornece que, em um determinado momento, você pode querer mudar sua abordagem.

Em termos do que realmente enviamos por meio desses protocolos assíncronos, as mesmas considerações se aplicam à comunicação síncrona. Se você estiver atualmente satisfeito com a codificação de solicitações e respostas usando JSON, fique com isso.

O que há em um evento?

Na Figura 4-12, vemos um evento sendo transmitido pelo Cliente microserviço, informando às partes interessadas que um novo cliente se registrou com o sistema. Dois dos microserviços downstream, Loyalty e Notifications, preocupam-se com esse evento. O microserviço Loyalty reage a recebendo o evento configurando uma conta para o novo cliente para que ele pode começar a ganhar pontos, enquanto o microserviço Notifications envia um e-mail para o cliente recém-registrado, dando as boas-vindas ao maravilhoso delícias da MusicCorp.

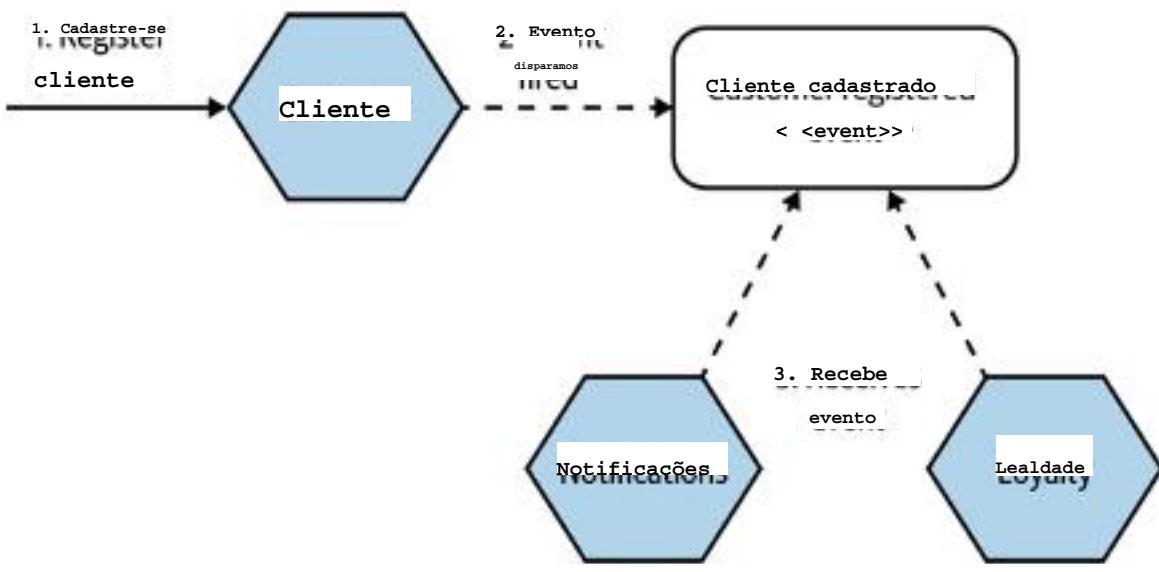


Figura 4-12. Notificações e microserviços Loyalty recebem um evento quando um novo cliente está registrado

Com uma solicitação, estamos solicitando que um microserviço faça algo e fornecendo as informações necessárias para que a operação solicitada seja realizada. Com um evento, estamos transmitindo um fato no qual outras partes possam estar interessadas, mas como o microserviço que emite um evento não pode e não deve saber quem

recebe o evento, como sabemos quais informações outras partes podem necessidade do evento? O que, exatamente, deveria estar dentro do evento?

Apenas um ID

Uma opção é que o evento contenha apenas um identificador para o novo cliente registrado, conforme mostrado na Figura 4-13. O microserviço Loyalty precisa apenas desse identificador para criar a conta de fidelidade correspondente, então tem tudo as informações de que precisa. No entanto, enquanto o microserviço Notifications sabe que precisa enviar um e-mail de boas-vindas quando esse tipo de evento é recebido, ele precisará de informações adicionais para fazer seu trabalho - pelo menos um e-mail endereço, e provavelmente o nome do cliente também para fornecer o e-mail que toque pessoal. Como essas informações não estão no caso de as Notificações o microserviço recebe, ele não tem escolha a não ser buscar essas informações do Microserviço para clientes, algo que vemos na Figura 4-13.

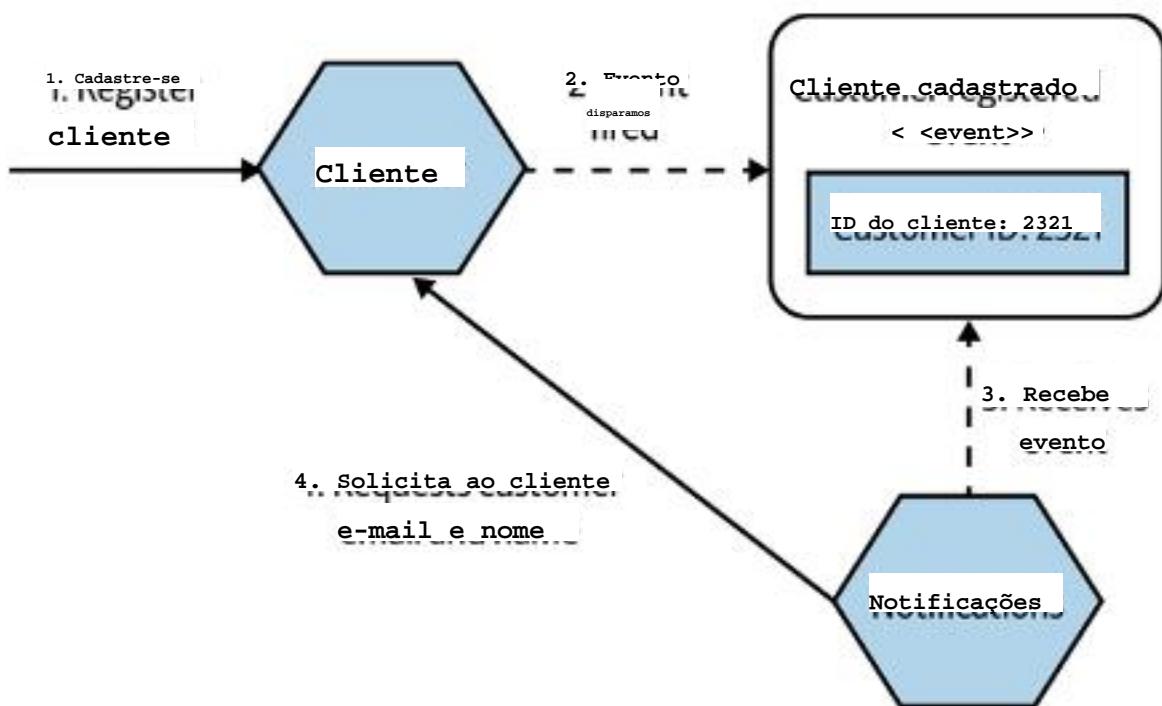


Figura 4-13. O microserviço Notificações precisa solicitar mais detalhes do Microserviços de clientes (que estão) incluídos no evento.

Há algumas desvantagens com essa abordagem. Em primeiro lugar, as notificações o microservice agora precisa saber sobre o microservice do cliente, acrescentando

acoplamento de domínio adicional. Durante o acoplamento de domínios, conforme discutimos em O capítulo 2, está na extremidade mais solta do espectro de acoplamento, ainda gostaríamos de evite isso sempre que possível. Se o evento for que o microsserviço Notificações recebeu continha todas as informações necessárias, então esse retorno de chamada não ser exigido. O retorno de chamada do microsserviço receptor também pode levar a a outra grande desvantagem, a saber, que em uma situação com um grande número de recebendo microsservicos, o microsserviço que emite o evento pode receber um Como resultado, uma enxurrada de solicitações. Imagine se cinco microsserviços diferentes fossem todos recebeu o mesmo evento de criação de clientes e tudo o que precisou solicitar informações adicionais - todos eles teriam que enviar imediatamente uma solicitação ao Microsserviço ao cliente para obter o que eles precisavam. Como o número de os microsservicos interessados em um determinado evento aumentam, o impacto desses as chamadas podem se tornar significativas.

Eventos totalmente detalhados

A alternativa, que eu prefiro, é colocar tudo em um evento que você ficaria feliz, caso contrário, compartilharia por meio de uma API. Se você deixasse o o microsserviço de notificações solicita o endereço de e-mail e o nome de um determinado cliente, por que não colocar essas informações no evento em primeiro lugar? Em Figura 4-14, vemos essa abordagem: as notificações agora são mais próprias suficiente e é capaz de fazer seu trabalho sem precisar se comunicar com o Microsserviço ao cliente Na verdade, talvez nunca seja necessário conhecer o cliente o microsserviço existe.

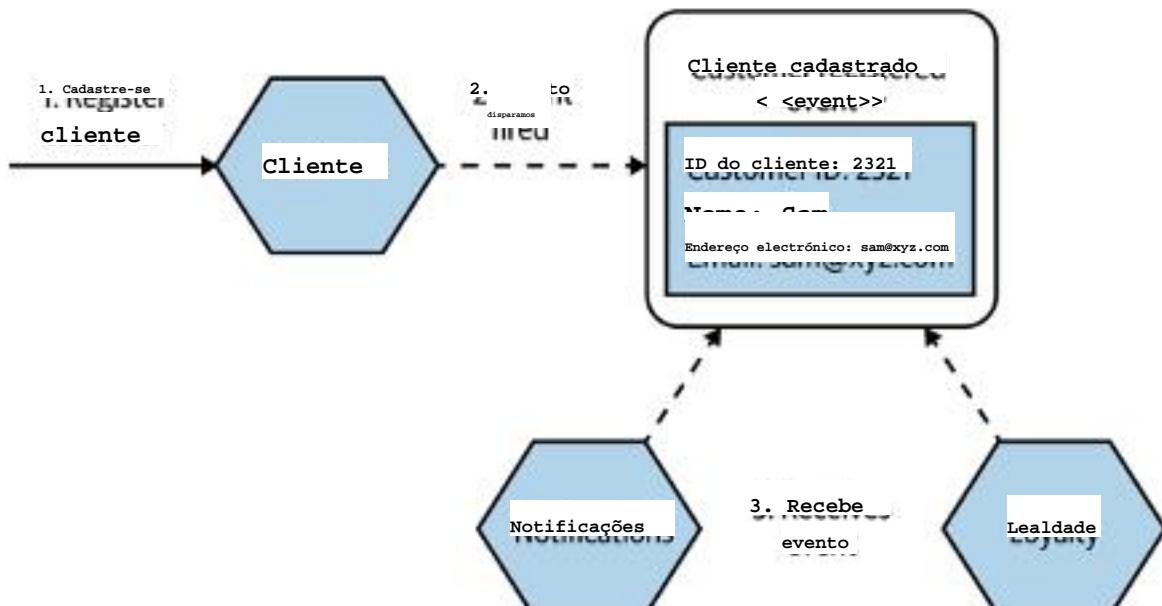


Figura 4-14. Um evento com mais informações pode permitir que o recebimento de microserviços atue sem exigindo mais chamadas para a fonte do evento

Além do fato de que eventos com mais informações podem permitir uma maior flexibilidade de acoplamento, eventos com mais informações também podem funcionar como históricos de registro do que aconteceu com uma determinada entidade. Isso pode ajudá-lo como parte de implementar um sistema de auditoria, ou talvez até mesmo fornecer a capacidade de reconstituir uma entidade em determinados pontos no tempo, o que significa que esses eventos poderiam ser usado como parte de um fornecimento de eventos, um conceito que exploraremos brevemente em um momento.

Embora essa abordagem seja definitivamente minha preferência, ela tem desvantagens. Em primeiro lugar, se os dados associados a um evento forem grandes, podemos ter dúvidas sobre o tamanho do evento. Corretores de mensagens modernos (supondo que você esteja usando um para implementar seu mecanismo de transmissão de eventos) seja bastante generoso limites para o tamanho da mensagem; o tamanho máximo padrão para uma mensagem no Kafka é 1 MB, e a versão mais recente do RabbitMQ tem um limite superior teórico de 512 MB para uma única mensagem (abaixo do limite anterior de 2 GB!), até embora se possa esperar que haja alguns problemas interessantes de desempenho com mensagens grandes como essa. Mas até mesmo o 1 MB que nos é oferecido como o máximo o tamanho de uma mensagem no Kafka nos dá muito espaço para enviar muitos dados. Em última análise, se você está se aventurando em um espaço no qual está começando a se preocupar sobre o tamanho de seus eventos, então eu recomendaria uma abordagem híbrida na qual

algumas informações estão no evento, mas outros dados (maiores) podem ser consultados se obrigatório.

Na Figura 4-14, a Lealdade não precisa saber o endereço de e-mail ou o nome do cliente, mas ainda assim o recebe por meio do evento. Isso pode levar às preocupações se estamos tentando limitar o escopo de quais microserviços podem veja que tipo de dados - por exemplo, talvez eu queira limitar quais microserviços podem ver informações de identificação pessoal (ou PII). pagamento, detalhes do cartão ou dados confidenciais semelhantes. Uma forma de resolver isso poderia ser enviar dois tipos diferentes de eventos - um que contém PII e pode ser visto por alguns microservicos e outro que exclui PII e pode ser transmitido mais amplamente. Isso aumenta a complexidade em termos de gerenciamento da visibilidade de diferentes eventos e garantindo que ambos os eventos realmente sejam acionados. O que acontece quando um microservico envia o primeiro tipo de evento, mas morre antes do segundo evento pode ser enviado?

Outra consideração é que, uma vez que colocamos dados em um evento, eles se tornam parte do nosso contrato com o mundo exterior. Temos que estar cientes de que, se removermos um campo de um evento, poderemos desvincular terceiros. Informações ocultar ainda é um conceito importante na colaboração baseada em eventos - quanto mais dados que colocamos em um evento, mais suposições as partes externas terão sobre o evento. Minha regra geral é que não há problema em colocar informações em um evento se eu ficasse feliz em compartilhar os mesmos dados em uma API de solicitação-resposta.

Onde usá-lo

A colaboração orientada por eventos prospera em situações em que as informações desejam para ser transmitido e em situações em que você está feliz em inverter a intenção. Afastando-se de um modelo de dizer a outras coisas o que fazer e, em vez disso, deixar que os microserviços downstream resolvam isso sozinhos tem uma ótima oferta de atracão.

Em uma situação em que você está se concentrando mais no acoplamento solto do que em outras. Por outro lado, a colaboração baseada em eventos terá um apelo óbvio.

A nota cautelar é que muitas vezes existem novas fontes de complexidade que venha à tona com esse estilo de colaboração, especialmente se você já teve

exposição limitada a ele. Se você não tiver certeza sobre essa forma de comunicação, lembre-se de que nossa arquitetura de microserviços pode (e provavelmente conterá) um mistura de diferentes estilos de interação. Você não precisa apostar tudo no evento... colaboração impulsionada; talvez comece com apenas um evento e comece a partir daí.

Pessoalmente, estou gravitando em direção à colaboração impulsionada por eventos quase como padrão. Meu cérebro parece ter se reconectado de tal forma que esses tipos de comunicação parecem óbvios para mim. Isso não é totalmente útil, pois pode ser complicado tentar explicar por que esse é o caso, além de diga que parece certo. Mas esse é apenas meu próprio preconceito embutido - eu naturalmente gosto o que eu sei, com base em minhas próprias experiências. Há uma forte possibilidade de que minha atração por essa forma de interação é motivada quase inteiramente pela minha experiências ruins anteriores com sistemas excessivamente acoplados. Eu poderia ser apenas o general lutando a última batalha repetidamente sem considerar isso talvez desta vez seja realmente diferente

O que vou dizer, deixando meus próprios preconceitos de lado, é que vejo muito mais equipes substituindo interações de solicitação-resposta por interações orientadas por eventos do que o inverso.

Prossiga com cuidado

Algumas dessas coisas assíncronas parecem divertidas, certo? Arquiteturas orientadas por eventos parecem levar a sistemas significativamente mais desacoplados e escaláveis. E eles lata. Mas esses estilos de comunicação levam a um aumento na complexidade. Essa não é apenas a complexidade necessária para gerenciar a publicação e a assinatura. As mensagens, como acabamos de discutir, mas também à complexidade dos outros problemas podemos enfrentar. Por exemplo, ao considerar uma solicitação assíncrona de longa duração-resposta, temos que pensar no que fazer quando a resposta voltar. Ele volta para o mesmo nó que iniciou a solicitação? Em caso afirmativo, o que acontece se esse nó estiver inativo? Caso contrário, preciso armazenar informações em algum lugar para que eu possa reagir de acordo? A assíncrona de curta duração pode ser mais fácil de gerenciar se você tem as APIs certas, mas mesmo assim, é uma forma diferente de pensando para programadores que estão acostumados com a síncrona entre processos chamadas por mensagem.

É hora de um conto de advertência. Em 2006, eu estava trabalhando na construção de um sistema de preços para um banco. Analisávamos os eventos do mercado e nos exercitávamos quais itens em um portfólio precisavam ser reavaliados. Uma vez que determinamos o lista de coisas para resolver, colocamos todas elas em uma fila de mensagens. Nós estamos usando uma rede para criar um grupo de trabalhadores de preços, o que nos permite aumentar e reduzir a faixa de preços, mediante solicitação. Esses trabalhadores usaram o padrão de consumidores concorrentes, cada um devorando mensagens o mais rápido possível até que não houvesse mais nada para processar.

O sistema estava funcionando e estávamos nos sentindo um pouco presunçosos. Um dia, no entanto, logo após lançarmos um lançamento, encontramos um problema terrível: nossos trabalhadores continuaram morrendo. E morrendo. E morrendo.

Eventualmente, rastreamos o problema. Um bug se infiltrou por meio do qual um determinado tipo de solicitação de preços faria com que um trabalhador falhasse. Nós estávamos usando uma fila transacionada: quando o trabalhador morreu, seu bloqueio na solicitação expirou e a solicitação de preços foi colocada de volta na fila, somente para outro trabalhador pegar o e morrer. Este foi um exemplo clássico do que Martin Fowler chama de falha catastrófica.

Além do bug em si, falhamos em especificar um limite máximo de novas tentativas para o trabalho na fila. Então, corrigimos o bug e configuramos uma nova tentativa máxima. Mas também percebemos que precisávamos de uma maneira de visualizá-las e potencialmente reproduzi-las. mensagens ruins. Acabamos tendo que implementar um hospital de mensagens (ou morto). fila de cartas), onde as mensagens eram enviadas se falhassem. Também criamos uma interface de usuário para visualizar essas mensagens e tente novamente, se necessário. Esses tipos de problemas não são imediatamente óbvios se você estiver familiarizado apenas com o ponto síncrono-comunicação a ponto

A complexidade associada a arquiteturas orientadas por eventos e assíncronas a programação em geral me leva a acreditar que você deve ser cauteloso em com que entusiasmo você começa a adotar essas ideias. Garanta que você tenha um bom monitoramento em vigor e considere fortemente o uso de IDs de correlação, que permitem que você rastreie solicitações além dos limites do processo, conforme abordaremos detalhadamente em Capítulo 10.

Também recomendo fortemente que você confira os Padrões de Integração Corporativa por Gregor Hohpe e Bobby Woolf,⁴ que contém muito mais detalhes sobre os diferentes padrões de mensagens que você talvez queira considerar neste espaço.

No entanto, também temos que ser honestos sobre os estilos de integração que podemos usar. considere "mais simples" - os problemas associados a saber se as coisas funcionou ou não não, não se limita às formas assíncronas de integração. Com um chamada síncrona e bloqueadora, se você tiver um tempo limite, isso aconteceu porque a solicitação foi perdida e a parte posterior não a recebeu? Ou fez a solicitação de conclusão, mas a resposta foi perdida? O que você faz nisso? situação? Se você tentar novamente, mas a solicitação original for atendida, o que acontecerá? (Bem, é aqui que entra a idempotência, um tópico que abordamos no Capítulo 12.)

Indiscutivelmente, com relação ao tratamento de falhas, chamadas de bloqueio síncrono podem nos causa tantas dores de cabeça quando se trata de malhar, se as coisas acontecerem aconteceu (ou não). Só que essas dores de cabeça podem ser mais familiares para nós!

Resumo

Neste capítulo, detalhei alguns dos principais estilos de microserviço comunicação e discutiu as várias vantagens e desvantagens. Nem sempre há uma única opção certa, mas espero ter detalhado informações suficientes sobre chamadas síncronas e assíncronas e orientadas por eventos e resposta à solicitação estilos de comunicação para ajudá-lo a fazer a ligação correta para seu dado contexto. Meus próprios preconceitos em relação à colaboração assíncrona e orientada por eventos são uma função não apenas das minhas experiências, mas também da minha aversão ao acoplamento geral. Mas esse estilo de comunicação vem com uma complexidade significativa. isso não pode ser ignorado, e cada situação é única.

Neste capítulo, mencionei brevemente algumas tecnologias específicas que podem ser usado para implementar esses estilos de interação. Agora estamos prontos para começar a segunda parte da implementação deste livro. No próximo capítulo, exploraremos implementando a comunicação de microserviços com mais profundidade.

⁴ História verídica.

2 Maarten van Steen e Andrew S. Tanenbaum, *Sistemas Distribuídos*, 3^a ed. (Vale de Scotts, CA: CreateSpace Independent Publishing Platform, 2017).

3 Observe que isso é muito simplificado - omitti completamente o código de tratamento de erros, por exemplo. Se você quer saber mais sobre `async/await`, especificamente em JavaScript, o JavaScript moderno. O tutorial é um ótimo lugar para começar.

4 Gregor Hohpe e Bobby Woolf, *Padrões de integração empresarial* (Boston: Addison-Wesley, 2003).

Parte II. Implantação

Capítulo 5. Implementar Comunicação por microsserviços

Conforme discutimos no capítulo anterior, sua escolha de tecnologia deve seja motivado em grande parte pelo estilo de comunicação que você deseja. Decidindo entre o bloqueio de chamadas assíncronas síncronas ou não bloqueadoras, solicitação-resposta ou colaboração orientada por eventos, ajudará você a reduzir o que Caso contrário, poderia ser uma lista muito longa de tecnologia. Neste capítulo, vamos para analisar algumas das tecnologias comumente usadas para microsserviços comunicação

Procurando a tecnologia ideal

Há uma variedade desconcertante de opções de como um microsserviço pode se comunicar outro. Mas qual é o One-soap certo? XML-RPC? DESCANSAR? qRPC? E novas opções estão sempre surgindo. Então, antes de discutirmos algo específico tecnologia, vamos pensar sobre o que queremos da tecnologia que escolhemos.

Facilite a compatibilidade com versões anteriores

Ao fazer alterações em nossos microsserviços, precisamos ter certeza de que não quebrar a compatibilidade com qualquer microsserviço consumidor. Como tal, queremos garantir que qualquer tecnologia que escolhemos facilite a reconstrução retroativa mudanças compatíveis. Operações simples, como adicionar novos campos, não deveriam quebrar clientes. Idealmente, também queremos a capacidade de validar que as mudanças que fiz com que sejam compatíveis com versões anteriores e tenham uma maneira de obter esse feedback antes de implantarmos nosso microsserviço na produção.

Torne sua interface explícita

É importante que a interface que um microsserviço expõe seja externa o mundo é explícito. Isso significa que está claro para um consumidor de um microsserviço sobre quais funcionalidades esse microsserviço expõe. Mas isso também significa que é deixe claro para um desenvolvedor que trabalha no microsserviço qual funcionalidade precisa permanecer intacto para partes externas - queremos evitar uma situação em que um a mudança para um microsserviço causa uma quebra acidental na compatibilidade.

Esquemas explícitos podem ajudar muito a garantir que a interface seja as exposições de microsserviços são explícitas. Algumas das tecnologias que podemos analisar requer o uso de um esquema; para outras tecnologias, o uso de um esquema é opcional. De qualquer forma, eu encoro fortemente o uso de um esquema explícito, como bem como haver documentação de apoio suficiente para esclarecer o que funcionalidade que um consumidor pode esperar que um microsserviço forneça.

Mantenha suas APIs independentes da tecnologia

Se você está no setor de TI há mais de 15 minutos, não precisa Eu quero dizer que trabalhamos em um espaço que está mudando rapidamente. Aquele certeza é mudança. Novas ferramentas, frameworks e linguagens estão surgindo, todos o tempo, implementando novas ideias que podem nos ajudar a trabalhar mais rápido e mais efetivamente. Neste momento, você pode ser uma loja da NET. Mas que tal um ano depois agora, ou daqui a cinco anos? E se você quiser experimentar com um pilha de tecnologia alternativa que pode torná-lo mais produtivo?

Sou um grande fã de manter minhas opções abertas, e é por isso que sou tão fã de microsserviços. É também por isso que acho muito importante garantir que você mantenha as APIs usadas para comunicação entre a tecnologia de microsserviços agnóstico. Isso significa evitar a tecnologia de integração que determina o que pilhas de tecnologia que podemos usar para implementar nossos microsserviços.

Simplifique seu serviço para os consumidores

Queremos facilitar o uso do nosso microsserviço pelos consumidores. Ter um um microsserviço bem elaborado não conta muito se o custo de uso como consumidor está muito alto! Então, vamos pensar sobre o que torna isso mais fácil consumidores para usar nosso novo serviço maravilhoso. Idealmente, gostaríamos de permitir que nosso

liberdade total aos clientes em sua escolha de tecnologia; por outro lado, fornecendo um a biblioteca cliente pode facilitar a adoção. Muitas vezes, no entanto, essas bibliotecas são incompatível com outras coisas que queremos alcançar. Por exemplo, podemos usar bibliotecas de clientes para facilitar as coisas para os consumidores, mas isso pode acontecer no custo do aumento do acoplamento.

Ocultar detalhes de implementação interna

Não queremos que nossos consumidores fiquem vinculados à nossa implementação interna, pois isso leva a um maior acoplamento; isso, por sua vez, significa que, se quisermos mudar algo dentro do nosso microserviço, podemos superar nossos consumidores ao exigir para que eles também mudem. Isso aumenta o custo da mudança – exatamente o que somos tentando evitar. Isso também significa que temos menos probabilidade de querer fazer uma mudança para medo de ter que atualizar nossos consumidores, o que pode levar a um aumento de dívida técnica dentro do serviço. Então, qualquer tecnologia que nos leve a expor detalhes de representação interna devem ser evitados.

Opções de tecnologia

Há toda uma série de tecnologias que poderíamos analisar, mas em vez de olhar amplamente, em uma longa lista de opções, destacarei algumas das mais populares e escolhas interessantes. Aqui estão as opções que veremos:

Chamadas de procedimentos remotos

Estruturas que permitem que chamadas de métodos locais sejam invocadas em um remoto processo. As opções comuns incluem SOAP e gRPC.

DESCANSAR

Um estilo arquitetônico em que você expõe recursos (cliente, pedido, etc.) que podem ser acessados usando um conjunto comum de verbos (GET, POST). O REST é um pouco mais do que isso, mas abordaremos isso em breve.

GraphQL

Um protocolo relativamente novo que permite que os consumidores definam de forma personalizada consultas que podem buscar informações de vários downstream microserviços, filtrando os resultados para retornar somente o necessário.

Corretores de mensagens

Middleware que permite a comunicação assíncrona por meio de filas ou tópicos.

Chamadas de procedimento remoto

Chamada de procedimento remoto (RPC) refere-se à técnica de fazer uma chamada local e executá-lo em um serviço remoto em algum lugar. Existem vários diferentes implementações de RPC em uso. A maior parte da tecnologia neste espaço requer um esquema explícito, como SOAP ou gRPC. No contexto do RPC, o esquema geralmente é chamado de linguagem de definição de interface (IDL), com SOAP, referindo-se ao seu formato de esquema como uma linguagem de definição de serviço web (WSDL). O uso de um esquema separado facilita a geração de clientes e stubs de servidor para diferentes pilhas de tecnologia - então, por exemplo, eu poderia ter um Servidor Java expondo uma interface SOAP e um cliente .NET gerado a partir da mesma definição WSDL da interface. Outra tecnologia, como Java RMI, exige um acoplamento mais estreito entre o cliente e o servidor, exigindo que ambos usam a mesma tecnologia subjacente, mas evitam a necessidade de uma definição de serviço, pois a definição do serviço é fornecida implicitamente pelo Java definicionais de tipo. Todas essas tecnologias, no entanto, têm o mesmo núcleo característica: eles fazem com que uma chamada remota pareça uma chamada local.

Normalmente, usar uma tecnologia RPC significa que você está comprando uma serialização protocolo. A estrutura RPC define como os dados são serializados e desserializado. Por exemplo, o gRPC usa o formato de serialização de buffer de protocolo para esse propósito. Algumas implementações estão vinculadas a uma rede específica protocolo (como SOAP, que faz uso nominal de HTTP), enquanto outros pode permitir que você use diferentes tipos de protocolos de rede, que podem fornecer recursos adicionais. Por exemplo, o TCP oferece garantias sobre entrega, enquanto o UDP não, mas tem uma sobrecarga muito menor. Isso pode permitir que você use diferentes tecnologias de rede para diferentes casos de uso.

As estruturas de RPC que têm um esquema explícito facilitam a geração.
código do cliente. Isso pode evitar a necessidade de bibliotecas de clientes, pois qualquer cliente pode simplesmente gerar seu próprio código com base nessa especificação de serviço. Para o lado do cliente a geração de código para funcionar, porém, o cliente precisa de alguma forma de obter o esquema fora da banda - em outras palavras, o consumidor precisa ter acesso ao esquema antes de planejar fazer chamadas. O Avro RPC é um outlier interessante aqui, pois tem a opção de enviar o esquema completo junto com a carga, permitindo clientes para interpretar dinamicamente o esquema.

A facilidade de geração do código do lado do cliente é um dos principais pontos de venda do RPC. O fato de eu poder simplesmente fazer uma chamada de método normal e teoricamente ignorar o resto é uma grande bênção.

Desafios

Como vimos, o RPC oferece algumas grandes vantagens, mas não deixa de ter desvantagens - e algumas implementações de RPC podem ser mais problemáticas do que outros. Muitas dessas questões podem ser tratadas, mas elas merecem mais exploração.

Acoplamento tecnológico

Alguns mecanismos de RPC, como o Java RMI, estão fortemente vinculados a um determinado plataforma, que pode limitar qual tecnologia pode ser usada no cliente e servidor. Thrift e gRPC têm uma quantidade impressionante de suporte para alternativas linguagens, o que pode reduzir um pouco essa desvantagem, mas esteja ciente de que Às vezes, a tecnologia RPC vem com restrições de interoperabilidade.

De certa forma, esse acoplamento tecnológico pode ser uma forma de exposição interna detalhes da implementação técnica. Por exemplo, o uso de laços RMI não apenas o cliente para a JVM, mas também para o servidor.

Para ser justo, há várias implementações de RPC que não têm isso. Restriction-gRPC, SOAP e Thrift são exemplos que permitem interoperabilidade entre diferentes pilhas de tecnologia.

Chamadas locais não são como chamadas remotas

A ideia central da RPC é esconder a complexidade de uma chamada remota. No entanto, isso pode fazer com que você se esconda demais. O drive em algumas formas de RPC a ser feito chamadas de métodos remotos parecem chamadas de métodos locais, escondendo o fato de que essas duas as coisas são muito diferentes. Posso fazer um grande número de chamadas locais em andamento sem se preocupar muito com o desempenho. Com o RPC, porém, o o custo de empacotamento e desempacotamento de cargas úteis pode ser significativo, não para mencione o tempo gasto para enviar coisas pela rede. Isso significa que você precisa pensar de forma diferente sobre o design de API para interfaces remotas versus locais interfaces. Basta pegar uma API local e tentar torná-la um limite de serviço sem pensar mais é provável que você tenha problemas. Em alguns dos piores exemplos, os desenvolvedores podem estar usando chamadas remotas sem saber, se a abstração é excessivamente opaca.

Você precisa pensar na própria rede. Famosamente, a primeira das falácias da computação distribuída é "A rede é confiável". As redes não são confiável. Eles podem e falharão, mesmo se você for seu cliente e o servidor falar com eles está bem. Eles podem falhar rápido, podem falhar lentamente e podem até mesmo malformar seus pacotes. Você deve presumir que suas redes estão infestadas com entidades malévolas prontas para liberar sua ira por capricho. Portanto, você pode esperar encontrar tipos de modos de falha que você talvez nunca tenha tido para lidar com um software mais simples e monolítico. Uma falha pode ser causada por o servidor remoto retornando um erro ou por você ter feito uma chamada incorreta. Você pode diga a diferença e, em caso afirmativo, você pode fazer alguma coisa a respeito? E o que você faz fazer quando o servidor remoto começa a responder lentamente? Nós vamos cobrir isso tópico quando falamos sobre resiliência no Capítulo 12.

Fragilidade

Algumas das implementações mais populares da RPC podem levar a algumas situações desagradáveis formas de fragilidade, sendo o Java RMI um exemplo muito bom. Vamos considerar um interface Java muito simples para a qual decidimos criar uma API remota nosso serviço ao cliente. O exemplo 5-1 declara os métodos que vamos usar exponha remotamente. O Java RMI então gera os stubs de cliente e servidor para nosso método.

Exemplo 5-1. Definindo um endpoint de serviço usando Java RMI

```
importar java.rmi.Remote;
importar java.rmi.RemoteException;

interface pública CustomerRemote estende Remote {
    public Customer FindCustomer (String id) lança RemoteException;

    cliente público CreateCustomer
        String (nome, sobrenome da string, endereço de e-mail da sequência)
        lança RemoteException;
}
```

Nessa interface, CreateCustomer usa o nome, o sobrenome e o e-mail endereço. O que acontece se decidirmos permitir que o objeto do Cliente também seja criado com apenas um endereço de e-mail? Poderíamos adicionar um novo método neste momento muito facilmente, assim:

```
public Customer CreateCustomer (String EmailAddress) lança
Exceção remota:
```

O problema é que agora também precisamos regenerar os stubs do cliente. Clientes que desejam consumir o novo método precisam dos novos stubs e, dependendo da natureza das mudanças na especificação, consumidores que não precisam do novo método também pode precisar ter seus stubs atualizados. Isso é gerenciável, claro, mas só até certo ponto. A realidade é que mudanças como essa são justas. comum. Os endpoints de RPC geralmente acabam tendo um grande número de métodos para diferentes formas de criar ou interagir com objetos. Isso se deve em parte a o fato de ainda estarmos pensando nessas chamadas remotas como chamadas locais.

No entanto, há outro tipo de fragilidade. Vamos dar uma olhada no que é nosso o objeto do cliente se parece com:

```
classe pública Customer implementa Serializable {
    string privada firstName;
    sobrenome de string privado;
    endereço de e-mail de string privado;
    página de string privada;
}
```

E se descobrir que, embora exponhamos o campo de idade em nosso Cliente objetos, nenhum de nossos consumidores jamais os usa? Decidimos que queremos remover esse campo. Mas se a implementação do servidor remover a idade de sua definição de desse tipo, e não fazemos o mesmo com todos os consumidores, mesmo que eles nunca usaram o campo, o código associado à desserialização do objeto do cliente no lado do consumidor será quebrado. Para implementar essa mudança, precisaríamos fazer alterações no código do cliente para dar suporte à nova definição e implante esses clientes atualizados ao mesmo tempo em que lançamos o novo versão do servidor. Esse é um desafio fundamental com qualquer mecanismo de RPC que promove o uso da geração de stub binário: você não consegue separar o cliente e implantações de servidores. Se você usa essa tecnologia, as versões lockstep podem ser no seu futuro.

Problemas semelhantes ocorrem se quisermos reestruturar o objeto do Cliente, até se não removermos campos - por exemplo, se quiséssemos encapsular Nome e sobrenome em um novo tipo de nomenclatura para facilitar gerenciar. Poderíamos, é claro, corrigir isso passando os tipos de dicionário como os parâmetros de nossas chamadas, mas nesse ponto, perdemos muitos dos benefícios do os stubs gerados porque ainda teremos que combinar e extrair manualmente os campos que queremos.

Na prática, objetos usados como parte da serialização binária em toda a rede podem sejam considerados tipos "somente para expansão". Essa fragilidade resulta nos tipos sendo exposto sobre o fio e se tornando uma massa de campos, alguns dos quais não são mais usados, mas não podem ser removidos com segurança

Onde usá-lo

Apesar de suas deficiências, eu realmente gosto bastante do RPC, e dos mais modernos implementações, como gRPC, são excelentes, enquanto outras as implementações têm problemas significativos que fariam com que eu lhes desse um amplo berço. O Java RMI, por exemplo, tem vários problemas relacionados fragilidade e opções limitadas de tecnologia, e o SOAP é bastante pesado do ponto de vista do desenvolvedor, especialmente quando comparado com os mais modernos escolhas.

Esteja ciente de algumas das possíveis armadilhas associadas à RPC se você estiver
vou escolher esse modelo. Não abstraia suas chamadas remotas a ponto de
a rede está completamente oculta e garanta que você possa evoluir o servidor.
interface sem precisar insistir em atualizações contínuas para os clientes. Descobrindo
o equilíbrio certo para seu código de cliente é importante, por exemplo. Certifique-se
seus clientes não estão alheios ao fato de que uma chamada de rede será
feito. As bibliotecas de cliente são frequentemente usadas no contexto da RPC e, se não,
estruturados corretamente, eles podem ser problemáticos. Falaremos mais sobre eles em breve.

Se eu estivesse procurando opções nesse espaço, o gRPC estaria no topo da minha lista.
Construído para aproveitar as vantagens do HTTP/2, ele tem um desempenho impressionante
características e boa facilidade geral de uso. Eu também aprecio o ecossistema
sobre gRPC, incluindo ferramentas como Protocolock, que é algo que faremos
discutiremos mais adiante neste capítulo, quando discutirmos esquemas.

O gRPC se encaixa bem em um modelo síncrono de solicitação-resposta, mas também pode funcionar em
em conjunto com extensões reativas. Está no topo da minha lista sempre que estou dentro
situações em que tenho muito controle sobre o cliente e o servidor
extremidades do espectro. Se você está tendo que apoiar uma grande variedade de outros
aplicativos que talvez precisem se comunicar com seus microsserviços, a necessidade de
compilar código do lado do cliente em um esquema do lado do servidor pode ser problemático. Em
Nesse caso, alguma forma de REST sobre API HTTP provavelmente seria mais adequada.

DESCANSAR

A Transferência de Estado Representacional (REST) é um estilo arquitetônico inspirado em
a web. Há muitos princípios e restrições por trás do estilo REST,
mas vamos nos concentrar naqueles que realmente nos ajudam quando enfrentamos
desafios de integração em um mundo de microsserviços e quando estamos procurando
uma alternativa ao RPC para nossas interfaces de serviço.

O mais importante quando se pensa em REST é o conceito de recursos. Você
pode pensar em um recurso como algo que o próprio serviço conhece, como um
Cliente O servidor cria diferentes representações desse Cliente em
pedido. A forma como um recurso é mostrado externamente é completamente dissociada da
como ele é armazenado internamente. Um cliente pode solicitar uma representação JSON de um

Cliente, por exemplo, mesmo que esteja armazenado em um formato completamente diferente.

Uma vez que um cliente tenha uma representação desse cliente, ele pode então fazer solicita a alteração, e o servidor pode ou não cumpri-las.

Existem muitos estilos diferentes de REST, e eu os abordo apenas brevemente aqui. Eu recomendo fortemente que você dê uma olhada na maturidade de Richardson Modelo, onde os diferentes estilos de REST são comparados.

O REST em si não fala realmente sobre protocolos subjacentes, embora seja a maioria comumente usado em HTTP. Eu vi implementações de REST usando muito protocolos diferentes antes, embora isso possa exigir muito trabalho. Alguns dos recursos que o HTTP nos fornece como parte da especificação, como verbos, facilita a implementação de REST via HTTP, enquanto que com outros protocolos você terá que lidar com esses recursos sozinho.

REST e HTTP

O próprio HTTP define alguns recursos úteis que funcionam muito bem com o Estilo REST. Por exemplo, os verbos HTTP (por exemplo, GET, POST e PUT) já tem significados bem compreendidos na especificação HTTP sobre como eles devem trabalhar com recursos. O estilo arquitetônico REST realmente diz que nós sabemos que esses verbos devem se comportar da mesma maneira em todos os recursos, e o

A especificação HTTP define vários verbos que podemos usar. OBTER recupera um recurso de forma idempotente, por exemplo, e o POST cria um novo recurso. Isso significa que podemos evitar muitos tipos diferentes de CreateCustomer ou Editar os métodos do cliente. Em vez disso, podemos simplesmente POSTAR um cliente representação para solicitar que o servidor crie um novo recurso, e então nós pode iniciar uma solicitação GET para recuperar uma representação de um recurso.

Conceptualmente, há um endpoint na forma de um recurso de cliente em esses casos, e as operações que podemos realizar neles, são incorporadas ao Protocolo HTTP.

O HTTP também traz um grande ecossistema de ferramentas e tecnologias de suporte. Nós comece a usar proxies de cache HTTP como Varnish e平衡adores de carga como mod_proxy, e muitas ferramentas de monitoramento já têm muito suporte para HTTP pronto para uso. Esses blocos de construção nos permitem lidar com grandes

volumes de tráfego HTTP e roteie-os de forma inteligente e bastante transparente... caminho. Também podemos usar todos os controles de segurança disponíveis com HTTP para proteja suas comunicações Da autenticação básica aos certificados do cliente, o HTTP o ecossistema nos fornece muitas ferramentas para facilitar o processo de segurança e exploraremos mais esse tópico no Capítulo 11. Dito isso, para obter esses benefícios, você precisa usar bem o HTTP. Use-o mal, e pode ser tão inseguro quanto difícil... para escalar como qualquer outra tecnologia existente. Use-o corretamente, porém, e você terá um muita ajuda.

Observe que o HTTP também pode ser usado para implementar o RPC. O SOAP, por exemplo, recebe roteado por HTTP, mas infelizmente usa muito pouco da especificação. Os verbos são ignorados, assim como coisas simples, como códigos de erro HTTP. Por outro lado manualmente, o gRPC foi projetado para aproveitar as capacidades do HTTP/2, como a capacidade de enviar vários fluxos de solicitação-resposta por meio de uma conexão única. Mas é claro que, ao usar o gRPC, você não está fazendo REST só porque você está usando HTTP!

A hipermídia como motor do estado do aplicativo

Outro princípio introduzido no REST que pode nos ajudar a evitar o acoplamento entre cliente e servidor está o conceito de hipermídia como motor de estado do aplicativo (geralmente abreviado como HATEOAS) e, cara, ele precisava de um abreviatura). Este é um texto bastante denso e um conceito bastante interessante, então vamos detalhar um pouco.

Hipermídia é um conceito em que um conteúdo contém links para vários outros conteúdos em vários formatos (por exemplo, texto, imagens, sons). Isso deve ser bem familiar para você, pois é o que acontece em uma web comum página: você segue os links, que são uma forma de controles hipermídia, para ver conteúdo relacionado. A ideia por trás do HATEOAS é que os clientes devem atuar interações com o servidor (potencialmente levando a transições de estado) por meio desses links para outros recursos. Um cliente não precisa saber exatamente onde os clientes vivem no servidor sabendo qual URI acessar; em vez disso, o cliente procura e navega pelos links para encontrar o que precisa.

Esse é um conceito um pouco estranho, então vamos primeiro dar um passo atrás e considerar como as pessoas interagem com uma página da web, que já estabelecemos ser rica.

com controles hipermídia.

Pense no site de compras da Amazon.com. A localização do carrinho de compras tem mudou com o tempo. O gráfico mudou. O link foi alterado. Mas como humanos, somos inteligentes o suficiente para ainda ver um carrinho de compras, saber o que é e interaja com ele. Entendemos o que significa um carrinho de compras, mesmo que a forma exata e o controle subjacente usados para representá-la tenham alterado. Sabemos que, se quisermos ver o carrinho, esse é o controle que temos quero interagir com. É assim que as páginas da web podem mudar de forma incremental tempo. Desde que esses contratos implícitos entre o cliente e o site ainda foi atendido, as mudanças não precisam ser mudanças significativas.

Com os controles hipermídia, estamos tentando alcançar o mesmo nível de "inteligência" para nossos consumidores eletrônicos. Vamos dar uma olhada em um controle hipermídia que talvez tenhamos para a MusicCorp. Acessamos um recurso representando uma entrada de catálogo para um determinado álbum no Exemplo 5-2. Junto com informações sobre o álbum, vemos vários controles de hipermídia.

Exemplo 5-2. Controles hipermídia usados em uma lista de álbuns

```
<album>
  <name>Dê sangue</name>
  <link rel="/artist" href="/artist/theBrakes" />①
  <description>
    Incrível, curto, brutal, engraçado e barulhento. Preciso comprar!
  </description>
  <link rel="/compra instantânea" href="/compra instantânea/1234" />②
</al>
  ① Esse controle de hipermídia nos mostra onde encontrar informações sobre o artista.
  ② E se quisermos comprar o álbum, agora sabemos para onde ir.
```

Neste documento, temos dois controles de hipermídia. O cliente lendo isso um documento precisa saber que um controle com uma relação de artista está onde está preciso navegar para obter informações sobre o artista, e que instantpurchase faz parte do protocolo usado para comprar o álbum. O cliente precisa entender a semântica da API da mesma forma que um

O ser humano precisa entender que, em um site de compras, o carrinho está onde os itens a serem comprados serão.

Como cliente, não preciso saber qual esquema de URI acessar para comprar o álbum; eu só preciso acessar o recurso, encontrar o controle de compra e navegar até isso. O controle de compra pode mudar a localização, o URI pode mudar ou o site poderia até mesmo me enviar para outro serviço e, como cliente, eu não o faria cuidado. Isso nos dá uma grande quantidade de dissociação entre o cliente e o servidor.

Estamos muito abstraídos dos detalhes subjacentes aqui. Nós poderíamos mudar completamente a implementação de como o controle é apresentado, desde que pois o cliente ainda pode encontrar um controle que corresponda à sua compreensão do protocolo, da mesma forma que um controle de carrinho de compras pode deixar de ser um link simples para um controle JavaScript mais complexo. Também somos livres para adicionar novos controles para o documento, talvez representando novas transições de estado que podemos executar no recurso em questão. Nós acabaríamos quebrando nossos consumidores somente se mudarmos fundamentalmente a semântica de um dos controles para que ele se comportasse de maneira muito diferente, ou se removêssemos um controle completamente.

A teoria é que, usando esses controles para desacoplar o cliente e o servidor, obtemos benefícios significativos ao longo do tempo que, esperançosamente, compensam o aumento do tempo necessário para colocar esses protocolos em funcionamento. Infelizmente, embora todas essas ideias parecem sensatas em teoria, descobri que essa forma de REST é raramente praticada, por razões que ainda não entendi totalmente. Isso faz HATEOAS, em particular, um conceito muito mais difícil de promover para aqueles já comprometido com o uso do REST. Fundamentalmente, muitas das ideias em O REST é baseado na criação de sistemas hipermídia distribuídos, e isso não é o que a maioria das pessoas acaba construindo.

Desafios

Em termos de facilidade de consumo, historicamente, você não seria capaz de gerar código do lado do cliente para seu protocolo de aplicativo REST via HTTP como você pode com implementações de RPC. Isso geralmente leva a pessoas criando APIs REST que fornecem bibliotecas de cliente para os consumidores usarem. Essas bibliotecas de cliente fornecem uma vinculação à API para criar um cliente

integração mais fácil. O problema é que as bibliotecas de clientes podem causar alguns desafios com relação ao acoplamento entre o cliente e o servidor, algo que discutiremos em "DRY e os perigos da reutilização de código em um Mundo dos microserviços".

Nos últimos anos, esse problema foi um pouco aliviado. O OpenAPI A especificação que surgiu do projeto Swagger agora fornece a capacidade de definir informações suficientes em um endpoint REST para permitir o geração de código do lado do cliente em uma variedade de linguagens. Em minha experiência, eu não vi muitas equipes realmente fazendo uso dessa funcionalidade, mesmo, que já estavam usando o Swagger para documentação. Suspeito que isso pode ser devido às dificuldades de adaptar seu uso às APIs atuais. Eu faço também se preocupam com uma especificação que foi usada anteriormente apenas para documentação que está sendo usada agora para definir um contrato mais explícito. Isso pode levam a uma especificação muito mais complexa - comparando um esquema OpenAPI com um esquema de buffer de protocolo, por exemplo, é um contraste bastante gritante. Apesar minhas reservas, porém, é bom que essa opção exista agora.

O desempenho também pode ser um problema. As cargas REST sobre HTTP podem, na verdade seja mais compacto que o SOAP porque o REST suporta formatos alternativos, como JSON ou mesmo binário, mas ainda não será nem de longe tão enxuto quanto um binário protocolo como o Thrift poderia ser. A sobrecarga de HTTP para cada solicitação também pode seja uma preocupação com os requisitos de baixa latência. Todos os protocolos HTTP convencionais em uso atual exigem o uso do Transmission Control Protocol (TCP) sob o capô, que tem ineficiências em comparação com outras redes protocolos e algumas implementações de RPC permitem que você use alternativas protocolos de rede para TCP, como o User Datagram Protocol (UDP).

As limitações impostas ao HTTP devido à necessidade de usar TCP estão sendo abordado. O HTTP/3, que está atualmente em processo de finalização, é pretendemos passar a usar o protocolo QUIC mais recente. O QUIC fornece o mesmos tipos de recursos do TCP (como garantias aprimoradas em relação ao UDP) mas com algumas melhorias significativas que comprovadamente oferecem resultados melhorias na latência e reduções na largura de banda. É provável que seja vários anos antes de o HTTP/3 ter um impacto generalizado na Internet pública,

mas parece razoável supor que as organizações possam se beneficiar antes do isso dentro de suas próprias redes.

Com relação especificamente ao HATEOAS, você pode encontrar outros problemas de desempenho. Como os clientes precisam navegar por vários controles para encontrar o terminais corretos para uma determinada operação, isso pode levar a protocolos muito falantes- várias viagens de ida e volta podem ser necessárias para cada operação. Em última análise, isso é um compensação. Se você decidir adotar um estilo REST no estilo HateOAS, eu sugeriria você começar fazendo com que seus clientes naveguem primeiro por esses controles e depois otimizem mais tarde, se necessário. Lembre-se de que o uso de HTTP nos fornece uma grande quantidade de ajuda pronta para uso, que discutimos anteriormente. Os males da otimização prematura já foi bem documentada antes, então eu não preciso expanda-os aqui. Observe também que muitas dessas abordagens foram desenvolvida para criar sistemas de hipertexto distribuídos, e nem todos se encaixam!

Às vezes, você só quer o bom e velho RPC.

Apesar dessas desvantagens, REST sobre HTTP é uma escolha padrão sensata para interações de serviço a serviço. Se você quiser saber mais, eu recomendo REST na prática: arquitetura de hipermídia e sistemas (O'Reilly), de Jim Webber, Savas Parastatidis e Ian Robinson, que aborda o tópico de REST sobre HTTP em profundidade.

Onde usá-lo

Devido ao seu amplo uso no setor, uma API baseada em REST sobre HTTP é uma escolha óbvia para uma interface síncrona de solicitação-resposta, se você estiver procurando permitir o acesso da maior variedade possível de clientes. É seria um erro pensar em uma API REST apenas como sendo "boa e suficiente" para a escolha da maioria das coisas, mas há algo nisso. É amplamente compreendido estilo de interface com o qual a maioria das pessoas está familiarizada e que garante interoperabilidade a partir de uma grande variedade de tecnologias.

Em grande parte devido aos recursos do HTTP e à extensão em que o REST baseia-se nesses recursos (em vez de ocultá-los), APIs baseadas em REST se sobressai em situações nas quais você deseja um armazenamento em cache eficaz e em grande escala de solicitações. É por esse motivo que eles são a escolha óbvia para expor APIs para partes externas ou interfaces de clientes. Eles podem muito bem sofrer, no entanto,

quando comparado a protocolos de comunicação mais eficientes, e embora você pode construir protocolos de interação assíncrona sobre os baseados em REST APIs, isso não é realmente uma boa opção em comparação com as alternativas em geral comunicação de microsserviço para microsserviço.

Apesar de apreciar intelectualmente os objetivos por trás do HATEOAS, eu não vi muitas evidências de que o trabalho adicional para implementar esse estilo de REST oferece benefícios valiosos a longo prazo, nem me lembro dos últimos anos conversando com qualquer equipe implementando uma arquitetura de microsserviços que possa fale sobre o valor de usar o HATEOAS. Minhas próprias experiências são obviamente apenas um conjunto de pontos de dados, e não duvido que, para algumas pessoas, ODEIE O OAS pode ter funcionado bem. Mas esse conceito não parece ter se popularizado tanto quanto eu pensei que seria. Pode ser que os conceitos por trás do HATEOAS são muito estranhos para entendermos, ou pode ser a falta de ferramentas ou padrões em este espaço, ou talvez o modelo simplesmente não funcione para os tipos de sistemas acabamos construindo. Também é possível, é claro, que os conceitos Por trás do HATEOAS realmente não combinam bem com a forma como criamos microsserviços. Portanto, para uso no perímetro, ele funciona muito bem e é síncrono comunicação baseada em solicitações e respostas entre microsserviços, é ótima.

GraphQL

Nos últimos anos, o GraphQL ganhou mais popularidade, devido em grande parte ao fato de se destacar em uma área específica. Ou seja, torna possível que um dispositivo do lado do cliente para definir consultas que podem evitar a necessidade de fazer várias solicitações para recuperar as mesmas informações. Isso pode oferecer uma quantidade significativa melhorias em termos de desempenho de dispositivos restritos do lado do cliente e também pode evitar a necessidade de implementar agregação personalizada do lado do servidor.

Para dar um exemplo simples, imagine um dispositivo móvel que queira exibir uma página que mostra uma visão geral dos pedidos mais recentes de um cliente. A página precisa contém algumas informações sobre o cliente, juntamente com informações sobre os cinco pedidos mais recentes do cliente. A tela precisa de apenas alguns campos do registro do cliente e somente a data, o valor e o status de envio de cada um pedido. O dispositivo móvel pode emitir chamadas para dois microsserviços downstream

para recuperar as informações necessárias, mas isso envolveria a criação de várias chamadas, incluindo a retirada de informações que não são realmente necessárias. Especialmente com dispositivos móveis, isso pode ser um desperdício - consumo mais o plano de dados do dispositivo móvel é maior do que o necessário, e isso pode levar mais tempo.

O GraphQL permite que o dispositivo móvel emita uma única consulta que pode recuar todas as informações necessárias. Para que isso funcione, você precisa de um microsserviço que expõe um endpoint GraphQL ao dispositivo cliente. Esse endpoint do GraphQL é a entrada para todas as consultas do cliente e expõe um esquema para os dispositivos do cliente usar. Esse esquema expõe os tipos disponíveis para o cliente e um bom construtor de consultas gráficas também está disponível para criar essas consultas mais fácil. Ao reduzir o número de chamadas e a quantidade de dados recuperados pelo no dispositivo cliente, você pode lidar perfeitamente com alguns dos desafios que ocorrem ao criar interfaces de usuário com arquiteturas de microsserviços.

Desafios

No início, um desafio era a falta de suporte de linguagem para o GraphQL especificação, com JavaScript sendo sua única opção inicialmente. Isso tem melhorou muito, com todas as principais tecnologias agora tendo suporte para o especificação. Na verdade, houve melhorias significativas no GraphQL e as várias implementações em geral, tornando o GraphQL muito perspectiva menos arriscada do que poderia ter sido há alguns anos. Dito isso, você talvez queira estar ciente de alguns desafios restantes com a tecnologia. Por um lado, o dispositivo cliente pode emitir consultas que mudam dinamicamente, e eu tenho ouvi falar de equipes que tiveram problemas com consultas do GraphQL causando carga significativa no lado do servidor como resultado dessa capacidade. Quando nós comparando o GraphQL com algo como SQL, vemos um problema semelhante. Um instrução SQL cara pode causar problemas significativos para um banco de dados e potencialmente têm um grande impacto no sistema mais amplo. O mesmo problema se aplica com GraphOL A diferença é que, com SOL, pelo menos temos ferramentas como planejadores de consultas para nossos bancos de dados, que podem nos ajudar a diagnosticar consultas problemáticas, enquanto um problema semelhante com o GraphQL pode ser mais difícil de rastrear. A limitação de solicitações pelo lado do servidor é um potencial

solução, mas como a execução da chamada pode ser distribuída por vários microserviços, isso está longe de ser simples.

Em comparação com as APIs HTTP normais baseadas em REST, o armazenamento em cache também é mais complexo. Com APIs baseadas em REST, posso definir um dos muitos cabeçalhos de resposta como ajudar dispositivos do lado do cliente ou caches intermediários, como entrega de conteúdo redes (CDNs), armazene respostas em cache para que não precisem ser solicitadas novamente. Isso não é possível da mesma forma com o GraphQL. O conselho que eu vi em esse problema parece girar em torno de apenas associar um ID a cada devolução recurso (e lembre-se, uma consulta GraphQL pode conter vários recursos) e, em seguida, fazer com que o dispositivo cliente armazene a solicitação em cache com esse ID. Na medida em que Posso dizer que isso torna o uso de CDNs ou o armazenamento em cache de proxies reversos incrivelmente difícil sem trabalho adicional.

Embora eu tenha visto algumas soluções específicas de implementação para esse problema (como os encontrados na implementação do JavaScript Apollo), a sensação de cache como se tivesse sido ignorado consciente ou inconscientemente como parte da inicial desenvolvimento do GraphQL. Se as consultas que você está emitindo forem altamente específicas em natureza para um usuário específico, então essa falta de cache no nível da solicitação pode não ser um obstáculo, é claro, pois é provável que sua taxa de acerto de cache seja baixa. Eu faço Gostaria de saber, porém, se essa limitação significa que você ainda vai acabar com um híbrido solução para dispositivos clientes, com algumas solicitações (mais genéricas) sendo processadas APIs HTTP normais baseadas em REST e outras solicitações que passam pelo GraphQL.

Outro problema é que, embora o GraphQL teoricamente possa lidar com gravações, ele não parece se encaixar tão bem quanto para leituras. Isso leva a situações em que as equipes estão usando o GraphQL para leitura, mas o REST para gravações.

A última questão é algo que pode ser totalmente subjetivo, mas ainda acho que é vale a pena levantar. O GraphQL faz com que pareça que você está apenas trabalhando com dados, o que pode reforçar a ideia de que os microserviços com os quais você está falando são apenas invólucros em bancos de dados. Na verdade, já vi várias pessoas compararem GraphQL para OData, uma tecnologia projetada como uma API genérica para acessando dados de bancos de dados. Como já discutimos detalhadamente, a ideia de tratar microserviços da mesma forma que invólucros em bancos de dados pode ser muito problemático. Os microserviços expõem a funcionalidade em interfaces de rede.

Algumas dessas funcionalidades podem exigir ou resultar na exposição de dados, mas eles ainda devem ter sua própria lógica e comportamento internos. Só porque você está usando o GraphQL, não pense tão pouco em seus microsserviços mais do que uma API em um banco de dados - é essencial que sua API do GraphQL não seja acoplado aos armazenamentos de dados subjacentes de seus microsserviços.

Onde usá-lo

O ponto ideal do GraphQL é para uso no perímetro do sistema, expondo funcionalidade para clientes externos. Esses clientes geralmente são GUIs, e é uma adequação óbvia para dispositivos móveis, dadas suas restrições em termos de limitação capacidade de transmitir dados ao usuário final e a natureza das redes móveis. Mas o GraphQL também foi usado para APIs externas, sendo o GitHub um dos primeiros a adotar do GraphQL. Se você tem uma API externa que geralmente exige clientes externos para fazer várias chamadas para obter as informações de que precisam, o GraphQL pode ajudam a tornar a API muito mais eficiente e amigável.

Fundamentalmente, o GraphQL é um mecanismo de agregação e filtragem de chamadas, portanto, em o contexto de uma arquitetura de microsserviços que seria usado para agregar chamadas sobre vários microsserviços downstream. Como tal, não é algo que substituiria a comunicação geral de microsserviço para microsserviço.

Uma alternativa ao uso do GraphQL seria considerar uma alternativa padrão como o backend para o frontend (BFF) - veremos isso e compare-o com o GraphQL e outras técnicas de agregação no Capítulo 14.

Corretores de mensagens

Os corretores de mensagens são intermediários, geralmente chamados de middleware, que se sentam entre processos para gerenciar a comunicação entre eles. Eles são uma escolha popular para ajudar a implementar a comunicação assíncrona entre microsserviços, pois oferecem uma variedade de recursos poderosos.

Como discutimos anteriormente, uma mensagem é um conceito genérico que define a coisa que um agente de mensagens envia. Uma mensagem pode conter uma solicitação, uma resposta, ou um evento. Em vez de um microsserviço se comunicando diretamente com

outro microsserviço, o microsserviço, em vez disso, envia uma mensagem para um corretor, com informações sobre como a mensagem deve ser enviada.

Tópicos e filas

Os corretores tendem a fornecer filas ou tópicos, ou ambos. Normalmente, as filas são ponto a ponto. Um remetente coloca uma mensagem em uma fila e um consumidor lê dessa fila. Com um sistema baseado em tópicos, vários consumidores podem assinar um tópico e cada consumidor inscrito receberá uma cópia da essa mensagem.

Um consumidor pode representar um ou mais microsserviços, normalmente modelados como um grupo de consumidores. Isso seria útil quando você tem várias instâncias de um microsserviço, e você deseja que qualquer um deles possa receber uma mensagem. Na Figura 5-1, vemos um exemplo em que o Processador de Pedidos tem três instâncias implantadas, todas como parte do mesmo grupo de consumidores. Quando uma mensagem é colocada na fila, apenas um membro do grupo de consumidores o fará receber essa mensagem; isso significa que a fila funciona como uma distribuição de carga mecanismo. Este é um exemplo do padrão de consumidores concorrentes que abordado brevemente no Capítulo 4.

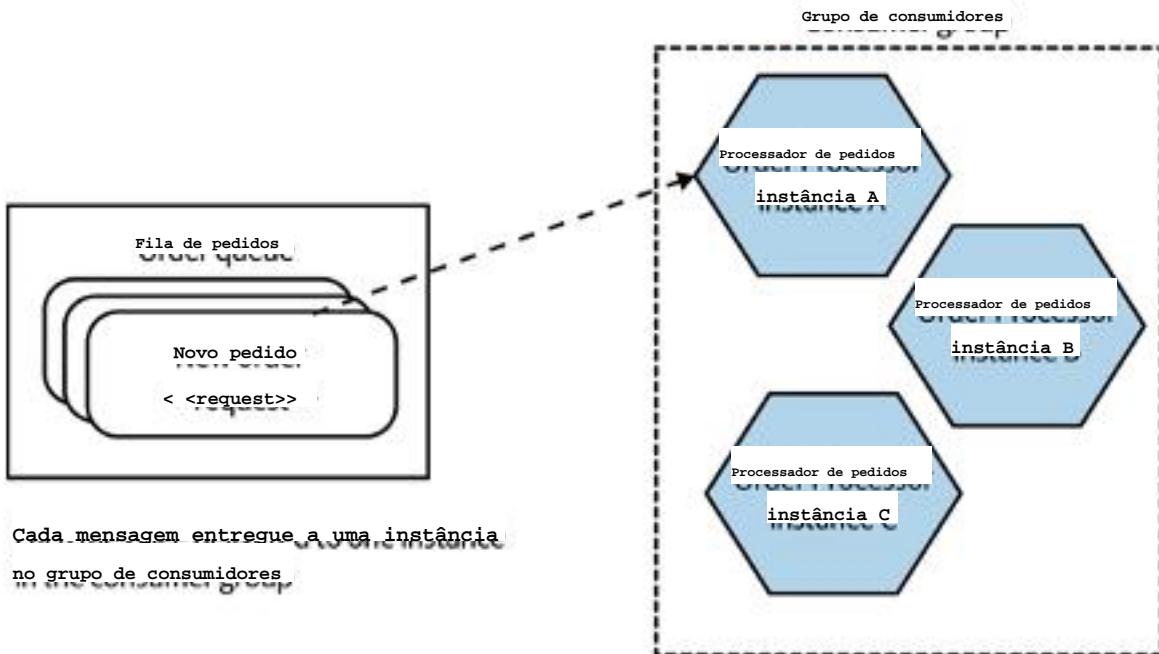


Figura 5-1. Uma fila permite um grupo de consumidores

Com os tópicos, você pode ter vários grupos de consumidores. Na Figura 5-2, um evento representar um pedido que está sendo pago é colocado no tópico Status do pedido. UMA uma cópia desse evento é recebida pelo microsserviço Warehouse e pelo Microsserviço de notificações, que estão em grupos de consumidores separados. Somente uma instância de cada grupo de consumidores verá esse evento.

À primeira vista, uma fila parece um tópico com um único grupo de consumidores. Uma grande parte da distinção entre os dois é que quando uma mensagem é enviada em uma fila, é possível saber para onde a mensagem está sendo enviada. Com um tópico, essa informação está oculta do remetente da mensagem - o remetente não sabe quem (se é que alguém) acabará recebendo a mensagem.

Os tópicos são uma boa opção para a colaboração baseada em eventos, enquanto as filas seriam ser mais apropriado para a comunicação de solicitação/resposta. Isso deve ser considerado como orientação geral em vez de uma regra estrita, no entanto.

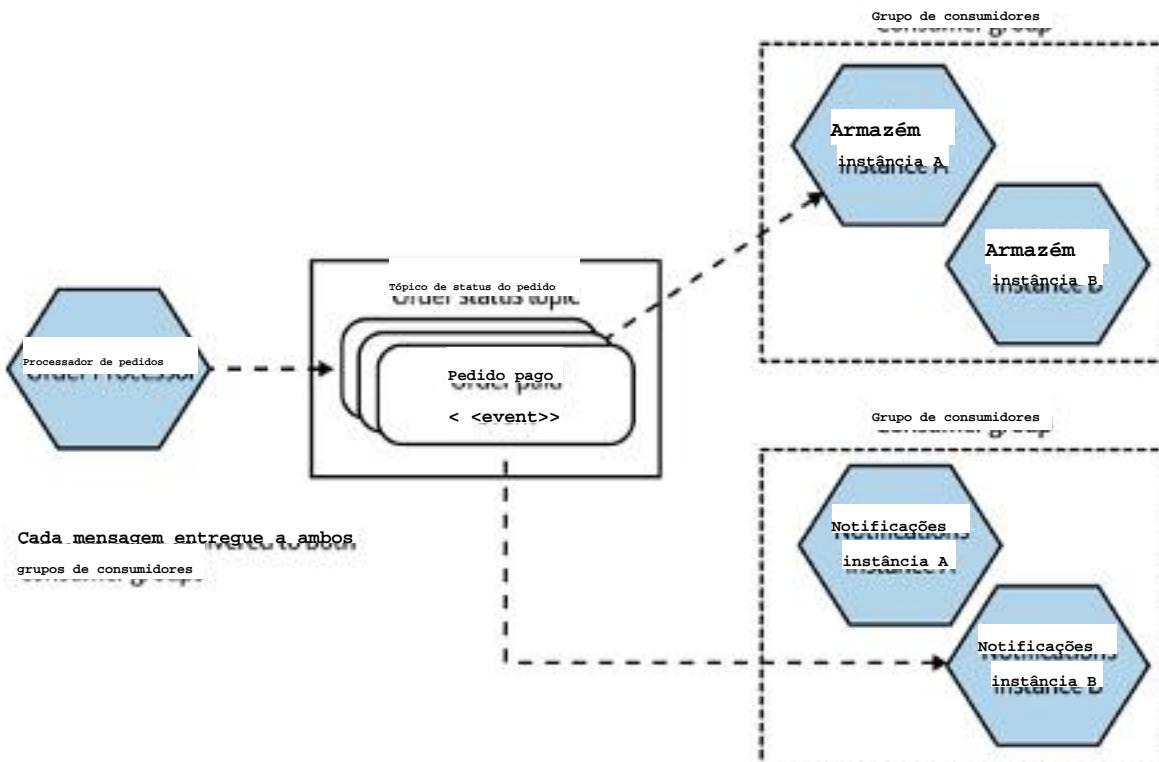


Figura 5-2. Os tópicos permitem que vários assinantes recebam as mesmas mensagens, o que é útil para transmissão de eventos

Entrega garantida

Então, por que usar um corretor? Fundamentalmente, eles fornecem alguns recursos que podem ser muito úteis para comunicação assíncrona. As propriedades que eles fornecem variam, mas a característica mais interessante é a entrega garantida, algo que todos os corretores amplamente usados apoiam de alguma forma. Entrega garantida descreve um compromisso do corretor de garantir que a mensagem seja entregue.

Do ponto de vista do microserviço que envia a mensagem, isso pode ser muito útil. Não é um problema se o destino posterior não estiver disponível - o corretor guardará a mensagem até que ela possa ser entregue. Isso pode reduzir o número de coisas com as quais um microserviço upstream precisa se preocupar. Compare isso com uma chamada direta síncrona, por exemplo, uma solicitação HTTP: se o destino downstream não está acessível, o microserviço upstream precisaria descobrir o que fazer com a solicitação; deve tentar ligar novamente ou dar para cima?

Para que a entrega garantida funcione, um corretor precisará garantir que qualquer mensagem ainda não entregues serão mantidas de forma durável até que elas podem ser entregues. Para cumprir essa promessa, um corretor normalmente administrará como algum tipo de sistema baseado em cluster, garantindo a perda de um único dispositivo não faz com que uma mensagem seja perdida. Normalmente, há muita coisa envolvida na administração correta de uma corretora, em parte devido aos desafios de gerenciamento de software baseado em cluster. Muitas vezes, a promessa de entrega garantida pode ser prejudicada se o corretor não estiver configurado corretamente. Como exemplo, RabbitMQ exige que instâncias em um cluster se comuniquem com latência relativamente baixa entre redes: caso contrário, as instâncias podem começar a ficar confusas sobre o estado das mensagens que estão sendo tratadas, resultando em perda de dados. Não estou destacando essa limitação específica como uma forma de dizer que o RabbitMQ é de alguma forma ruim - todos os corretores têm restrições quanto à forma como precisam ser administrados para entregar a promessa de entrega garantida. Se você planeja administrar seu próprio corretor, faça certifique-se de ler a documentação com atenção.

Também é importante notar que o que qualquer corretor quer dizer com garantido a entrega pode variar. Novamente, ler a documentação é um ótimo começo.

Um dos grandes atrativos de um corretor é a propriedade da entrega garantida. Mas para que isso funcione, você precisa confiar não apenas nas pessoas que criaram o corretor mas também a forma como a corretora tem operado. Se você construiu um sistema que é com base na suposição de que a entrega é garantida, e isso acaba não seja o caso devido a um problema com o corretor subjacente, isso pode causar problemas significativos problemas. A esperança, é claro, é que você esteja transferindo esse trabalho para o software criado por pessoas que podem fazer esse trabalho melhor do que você. Em última análise, você tem que decidir o quanto você quer confiar no corretor que você está usando.

Outras características

Além da entrega garantida, os corretores podem fornecer outras características que você pode achar útil.

A maioria dos corretores pode garantir a ordem em que as mensagens serão entregues, mas isso não é universal e, mesmo assim, o escopo dessa garantia pode ser limitado. Com o Kafka, por exemplo, o pedido é garantido somente dentro de um único particionamento. Se você não tiver certeza de que as mensagens serão recebidas em ordem, o consumidor pode precisar compensar, talvez adiando o processamento de mensagens que são recebidas fora de ordem até que as mensagens faltantes sejam recebido.

Alguns corretores oferecem transações por escrito - por exemplo, o Kafka permite que você para escrever sobre vários tópicos em uma única transação. Alguns corretores também podem fornecer transacionalidade de leitura, algo que eu aproveitei ao usar vários corretores por meio das APIs do Java Message Service (JMS). Isso pode ser útil se você quiser garantir que a mensagem possa ser processada pelo consumidor antes de removê-lo do corretor.

Outra característica um tanto controversa prometida por alguns corretores é a de exatamente uma vez na entrega. Uma das maneiras mais fáceis de fornecer entrega garantida está permitindo que a mensagem seja reenviada. Isso pode fazer com que um consumidor veja a mesma mensagem mais de uma vez (mesmo que seja uma situação rara). A maioria dos corretores farão o possível para reduzir a chance disso, ou ocultar esse fato do consumidor, mas alguns corretores vão além ao garantir exatamente uma entrega única. Este é um tópico complexo, pois falei com alguns especialistas que afirmam que garantir exatamente uma entrega em todos os casos é impossível, enquanto outros

especialistas dizem que você basicamente pode fazer isso com algumas soluções alternativas simples. Ou dessa forma, se o corretor de sua escolha alegar implementar isso, pague de verdade atenção cuidadosa à forma como ele é implementado. Melhor ainda, construa seus consumidores de tal forma que estejam preparados para o fato de poderem receber um envie uma mensagem mais de uma vez e pode lidar com essa situação. Um exemplo muito simples seria que cada mensagem tivesse um ID, que um consumidor pudesse verificar hora em que uma mensagem é recebida. Se uma mensagem com esse ID já tiver sido enviada processada, a mensagem mais recente pode ser ignorada.

Escolhas

Existe uma variedade de corretores de mensagens. Exemplos populares incluem RabbitMQ, ActiveMQ e Kafka (que exploraremos mais em breve). O público principal os fornecedores de nuvem também fornecem uma variedade de produtos que desempenham esse papel, desde versões gerenciadas desses corretores que você pode instalar por conta própria infraestrutura para implementações personalizadas que são específicas para um determinado plataforma. A AWS, por exemplo, tem Simple Queue Service (SQS), Simple Notification Service (SNS) e Kinesis, todos oferecendo diferentes sabores de corretores totalmente gerenciados. O SQS foi, de fato, o segundo produto de todos os tempos lançado pela AWS, tendo sido lançado em 2006.

Kafka

Vale a pena destacar a Kafka como corretora específica, em grande parte devido à sua recente popularidade. Parte dessa popularidade se deve ao uso de Kafka em ajudar a se movimentar volumes de dados disponíveis como parte da implementação de pipelines de processamento de fluxos. Isso pode ajudar a passar do processamento orientado por lotes para um processamento mais em tempo real processamento.

Há algumas características de Kafka que merecem destaque. Em primeiro lugar, foi projetado para uma escala muito grande - foi construído no LinkedIn para substituir vários clusters de mensagens existentes com uma única plataforma. Kafka foi construído para permitir vários consumidores e produtores - falei com um especialista em um grande empresa de tecnologia que tinha mais de cinqüenta mil produtores e consumidores trabalhando no mesmo cluster. Para ser justo, muito poucas organizações

têm problemas nesse nível de escala, mas para algumas organizações, a capacidade de escalar Kafka facilmente (relativamente falando) pode ser muito útil.

Outra característica bastante singular de Kafka é a permanência da mensagem. Com um normal corretor de mensagens, uma vez que o último consumidor tenha recebido uma mensagem, o corretor não precisa mais guardar essa mensagem. Com Kafka, as mensagens podem ser armazenado por um período configurável. Isso significa que as mensagens podem ser armazenadas para sempre. Isso pode permitir que os consumidores reingiram mensagens que já tinham. processado ou permita que consumidores recém-implantados processem mensagens que foram enviados anteriormente.

Finalmente, o Kafka está lançando suporte integrado para processamento de streams. Em vez de usar o Kafka para enviar mensagens para um processamento de stream dedicado Com uma ferramenta como o Apache Flink, algumas tarefas podem ser feitas dentro do próprio Kafka. Usando o KSQL, você pode definir instruções semelhantes a SQL que podem processar uma ou mais tópicos em tempo real. Isso pode lhe dar algo semelhante a um dinamicamente atualizando a visualização materializada do banco de dados, com a fonte de dados sendo Kafka tópicos em vez de um banco de dados. Esses recursos abrem alguns muito possibilidades interessantes de como os dados são gerenciados em sistemas distribuídos. Se você gostaria de explorar essas ideias com mais detalhes, posso recomendar Designing Sistemas orientados por eventos (O'Reilly), de Ben Stopford. (Eu tenho que recomendar o livro de Ben, enquanto eu escrevia o prefácio dele!) Para um mergulho mais profundo em Kafka em geral, eu sugeriria Kafka: The Definitive Guide (O'Reilly), de Neha Narkhede, Gwen Shapira e Todd Palino.

Formatos de serialização

Algumas das opções de tecnologia que analisamos especificamente, algumas das Implementações de RPC - faça escolhas para você em relação à forma como os dados são serializado e desserializado. Com o gRPC, por exemplo, todos os dados enviados serão convertido em formato de buffer de protocolo. Muitas das opções de tecnologia, no entanto, nos dê muita liberdade em termos de como convertemos dados em rede chamadas. Escolha a Kafka como sua corretora preferida e você poderá enviar mensagens em um variedade de formatos. Então, qual formato você deve escolher?

Formatos textuais

O uso de formatos textuais padrão oferece aos clientes muita flexibilidade na forma como eles consomem recursos. As APIs REST geralmente usam um formato textual para o órgãos de solicitação e resposta, mesmo que teoricamente você possa enviar com prazer dados binários via HTTP. Na verdade, é assim que o gRPC funciona usando HTTP embaixo, mas enviando buffers de protocolo binário.

O JSON usurpou o XML como o formato de serialização de texto preferido. Você pode apontar uma série de razões pelas quais isso ocorreu, mas o principal motivo é que um dos principais consumidores de APIs geralmente é um navegador, onde o JSON é um ótimo ajuste. O JSON se tornou popular em parte como resultado da reação contra XML, e os proponentes citam sua relativa compacidade e simplicidade quando comparado ao XML como outro fator vencedor. A realidade é que raramente um grande diferencial entre o tamanho de uma carga JSON e a de uma Carga XML, especialmente porque essas cargas geralmente são compactadas. É também vale ressaltar que parte da simplicidade do JSON tem um custo — é nossa pressa em adotar protocolos mais simples, os esquemas saíram da janela (mais sobre isso mais tarde).

O Avro é um formato de serialização interessante. Ele usa o JSON como base, estrutura e a usa para definir um formato baseado em esquema. Avro encontrou muitos, popularidade como formato para cargas de mensagens, em parte devido à sua capacidade de enviar o esquema como parte da carga útil, o que pode fazer com que o suporte seja múltiplo. formatos de mensagens diferentes são muito mais fáceis.

Pessoalmente, porém, ainda sou fã de XML. Parte do suporte da ferramenta é melhor. Por exemplo, se eu quiser extrair apenas certas partes da carga (a técnica que discutiremos mais em "Como lidar com a mudança entre Microservices"), posso usar o XPATH, que é um padrão bem compreendido com muito suporte de ferramentas, ou até mesmo seletores de CSS, que muitos acham até mais fácil. Com o JSON, eu tenho o JSONPath, mas isso não é tão amplamente suportado. EU Acho estranho que as pessoas escolham o JSON porque é bom e leve, mas então tente inserir conceitos nele, como controles de hipermídia que já existem em XML. Eu aceito, porém, que provavelmente estou em minoria aqui e que o JSON é o formato preferido de muitas pessoas!

Formatos binários

Embora os formatos textuais tenham benefícios, como facilitar a leitura por humanos, eles fornecem muita interoperabilidade com diferentes ferramentas e tecnologias, o mundo dos protocolos de serialização binária está onde você quer a ser se você começar a se preocupar com o tamanho da carga útil ou com as eficiências de escrever e ler as cargas úteis. Os buffers de protocolo existem há um embora e sejam frequentemente usados fora do escopo do gRPC - eles provavelmente representam o formato de serialização binária mais popular para microserviços-comunicação baseada.

Esse espaço é grande, no entanto, e vários outros formatos foram desenvolvido com uma variedade de requisitos em mente. Codificação binária simples, Cap'n Proto e FlatBuffers vêm à mente. Embora os benchmarks sejam abundantes para cada um desses formatos, destacando seus benefícios relevantes em comparação com buffers de protocolo, JSON ou outros formatos, os benchmarks sofrem de um problema fundamental, pois eles podem não representar necessariamente como você é vou usá-los. Se você deseja extrair os últimos bytes do seu formato de serialização, ou para reduzir microssegundos do tempo gasto para ler ou escreva essas cargas, eu sugiro fortemente que você faça sua própria comparação desses vários formatos. Na minha experiência, a grande maioria dos sistemas raramente precisam se preocupar com essas otimizações, pois muitas vezes elas podem alcançar o melhoriias que eles estão procurando enviando menos dados ou não fazendo o ligue para qualquer coisa. No entanto, se você estiver criando um distribuído de latência ultrabaixa sistema, certifique-se de estar preparado para mergulhar de cabeça no mundo binário formatos de serialização.

Esquemas

Uma discussão que surge repetidamente é se devemos usar esquemas para definir o que nossos endpoints expõem e o que eles aceitam. Os esquemas podem vêm em vários tipos diferentes, e escolher um formato de serialização normalmente definem qual tecnologia de esquema você pode usar. Se você está trabalhando com XML bruto, você usaria XML Schema Definition (XSD); se você estiver trabalhando com JSON bruto, você usaria o esquema JSON. Algumas das opções de tecnologia

que mencionamos (especificamente, um subconjunto considerável das opções de RPC) exigem o uso de esquemas explícitos, então, se você escolher essas tecnologias, você teria que faça uso de esquemas. O SOAP funciona por meio do uso de um WSDL, enquanto o gRPC requer o uso de uma especificação de buffer de protocolo. Outras opções de tecnologia exploramos como tornar o uso de esquemas opcional, e é aqui que as coisas fique mais interessante.

Como já discuti, sou a favor de ter esquemas explícitos para endpoints de microserviços, por dois motivos principais. Em primeiro lugar, eles percorrem um longo caminho para ser uma representação explícita do que é um endpoint de microserviço expõe e o que ela pode aceitar. Isso facilita a vida dos desenvolvedores trabalhando no microserviço e para seus consumidores. Os esquemas não podem substituir a necessidade de uma boa documentação, mas certamente podem ajudar a reduzir a quantidade de documentação necessária.

A outra razão pela qual eu gosto de esquemas explícitos, porém, é como eles ajudam em termos de detectando quebras accidentais de terminais de microserviços. Vamos explorar como para lidar com as mudanças entre microserviços em um momento, mas primeiro vale a pena explorando os diferentes tipos de quebras e o papel que os esquemas podem desempenhar.

Quebras estruturais versus semânticas do contrato

De um modo geral, podemos dividir as quebras de contrato em duas categorias — quebras estruturais e quebras semânticas. Uma quebra estrutural é uma situação em que a estrutura do endpoint muda de tal forma que um o consumidor agora é incompatível — isso pode representar campos ou métodos sendo removidos ou novos campos obrigatórios sendo adicionados. Uma quebra semântica se refere a uma situação em que a estrutura do endpoint de microserviços continua sendo a o mesmo, mas o comportamento muda de forma a prejudicar os consumidores expectativas.

Vamos dar um exemplo simples. Você tem um Hard altamente complexo. Microserviço de cálculos que expõe um método de cálculo em seu ponto final. Esse método de cálculo usa dois números inteiros, ambos são campos obrigatórios. Se você alterou os cálculos rígidos de forma que o o método de cálculo agora usa apenas um número inteiro, então os consumidores quebrariam

-eles estariam enviando solicitações com dois números inteiros que os cálculos difíceis o microsserviço rejeitaria. Este é um exemplo de uma mudança estrutural, e em geral, essas mudanças podem ser mais fáceis de detectar.

Uma mudança semântica é mais problemática. É aqui que a estrutura do endpoint não muda, mas o comportamento do endpoint sim. Voltando ao nosso método de cálculo, imagine que na primeira versão, os dois forneciam números inteiros são somados e os resultados retornados. Até agora, tudo bem. Agora nós altere os cálculos rígidos para que o método de cálculo multiplique os números inteiros juntos e retorna o resultado. A semântica do cálculo o método mudou de uma forma que poderia quebrar as expectativas dos consumidores.

Você deve usar esquemas?

Usando esquemas e comparando diferentes versões de esquemas, podemos capturar quebras estruturais. Detectar quebras semânticas requer o uso de testes. Se você não tiver esquemas, ou se tiver esquemas, mas decidir não comparar mudanças no esquema em termos de compatibilidade e, em seguida, o ônus de capturar a estrutura as quebras antes de chegar à produção também recaem nos testes. Indiscutivelmente, a situação é um pouco análoga à digitação estática versus dinâmica em linguagens de programação. Com uma linguagem digitada estaticamente, os tipos são fixos em tempo de compilação, se seu código fizer algo com uma instância de um tipo que não é permitido (como chamar um método que não existe), então o compilador pode detectar esse erro. Isso pode fazer com que você concentre os esforços de teste em outros tipos de problemas. Com uma linguagem digitada dinamicamente, porém, alguns de seus testes precisará detectar erros que um compilador detecta quando digitados estaticamente idiomas.

Agora, estou bastante relaxado sobre linguagens estáticas versus linguagens digitadas dinamicamente, e Eu me descobri muito produtivo (relativamente falando) em ambos. Certamente, linguagens digitadas dinamicamente oferecem alguns benefícios significativos isso, para muitas pessoas, justifica desistir da segurança em tempo de compilação. Pessoalmente falando, porém, se trouxermos a discussão de volta ao microsserviço interações, não descobri que exista uma compensação igualmente equilibrada quando

se trata de esquema versus comunicação "sem esquema". Simplificando, acho que ter um esquema explícito mais do que compensa qualquer benefício percebido de ter comunicação sem esquema.

Realmente, a questão não é se você tem um esquema ou não - é se esse esquema é explícito ou não. Se você estiver consumindo dados de um API sem esquema, você ainda tem expectativas sobre quais dados devem estar lá e como esses dados devem ser estruturados. Seu código que lidará com os dados serão escritos com um conjunto de suposições em mente sobre como esses dados são estruturados. Nesse caso, eu diria que você tem um esquema, mas é apenas totalmente implícito em vez de explícito. Muito do meu desejo por um esquema explícito é motivado pelo fato de que eu acho importante ser o mais explícito possível sobre o que um microsserviço expõe (ou não).

O principal argumento para endpoints sem esquema parece ser que os esquemas precisam trabalhar mais e não ofereça valor suficiente. Isso, na minha humilde opinião, é em parte uma falha de imaginação e em parte uma falha de boas ferramentas para ajudar esquemas têm mais valor quando se trata de usá-los para capturar estruturas quebras.

Em última análise, muito do que os esquemas fornecem é uma representação explícita da parte do contrato de estrutura entre um cliente e um servidor. Eles ajudam a fazer coisas explícito e pode ajudar muito na comunicação entre equipes e no trabalho como uma rede de segurança. Em situações em que o custo da mudança é reduzido - para exemplo, quando o cliente e o servidor são de propriedade da mesma equipe - eu sou mais relaxado por não ter esquemas.

Lidando com mudanças entre microsserviços

Provavelmente a pergunta mais comum que recebo sobre microsserviços, depois de "Como deveriam ser grandes?" é "Como você lida com o controle de versões?" Quando esta pergunta é perguntado, raramente é uma pergunta sobre que tipo de esquema de numeração você deve usar e saber mais sobre como você lida com as mudanças nos contratos entre microsserviços.

A forma como você lida com a mudança realmente se divide em dois tópicos. Em um momento, veremos o que acontece se você precisar fazer uma alteração significativa. Mas antes disso, vamos ver o que você pode fazer para evitar fazer uma alteração significativa em primeiro lugar.

Evitando mudanças significativas

Se você quiser evitar fazer alterações significativas, existem algumas ideias-chave vale a pena explorar, muitos dos quais já abordamos no início do capítulo. Se você puder colocar essas ideias em prática, achará muito mais fácil permitem que os microsserviços sejam alterados independentemente uns dos outros.

Mudanças de expansão

Adicione coisas novas a uma interface de microsserviço; não remova coisas antigas.

Leitor tolerante

Ao consumir uma interface de microsserviço, seja flexível no que você esperar.

Tecnologia certa

Escolha uma tecnologia que facilite a compatibilidade com versões anteriores mudanças na interface.

Interface explícita

Seja explícito sobre o que um microsserviço expõe. Isso torna as coisas mais fáceis para o cliente e mais fácil para os mantenedores do microsserviço entendam o que pode ser mudado livremente.

Detecte alterações accidentais de interrupção antecipada

Tenha mecanismos para detectar alterações na interface que serão interrompidas consumidores em produção antes que essas mudanças sejam implantadas.

Essas ideias se reforçam mutuamente, e muitas se baseiam nesse conceito-chave de informações ocultas que discutimos com frequência. Vamos analisar cada ideia em virar.

Mudanças de expansão

Provavelmente, o lugar mais fácil para começar é adicionando apenas coisas novas a um contrato de microsserviço e não removendo mais nada. Considere o exemplo de adicionar um novo campo a uma carga útil, supondo que o cliente esteja de alguma forma tolerante a tais mudanças, isso não deve ter um impacto material. Adicionando um novo campo DateOfBirth em um registro de cliente deve ser adequado, por exemplo.

Leitor tolerante

A forma como o consumidor de um microsserviço é implementado pode trazer muita coisa a suportar sobre como facilitar as alterações compatíveis com versões anteriores. Especificamente, queremos evite a vinculação excessiva do código do cliente à interface de um microsserviço. Vamos considerar um microsserviço de e-mail cujo trabalho é enviar e-mails para nossos clientes de tempos em tempos. É solicitado o envio de um e-mail de "pedido enviado" para um cliente com o ID 1234; ele dispara e recupera o cliente com esse ID e recebe algo parecido com a resposta mostrada no Exemplo 5-3.

Exemplo 5-3. Exemplo de resposta do serviço ao cliente

```
<customer>
  <firstname>Sam</firstname>
  <lastname>Newman</lastname>
  <email>sam@macdierain.com</email>
  <Número de telefone> 555-1234-5678</Número de telefone>
</customer>
```

Agora, para enviar o e-mail, o microsserviço de e-mail precisa apenas do primeiro nome, campos de sobrenome e e-mail. Não precisamos saber o número do telefone. Queremos simplesmente retirar os campos que nos interessam e ignorar o resto. Alguma tecnologia de encadernação, especialmente aquela usada por linguagens fortemente tipadas, pode tentar vincular todos os campos, quer o consumidor os queira ou não. O que acontece se percebermos que ninguém está usando o número de telefone e nós decide removê-lo? Isso pode fazer com que os consumidores quebrem desnecessariamente.

Da mesma forma, e se quiséssemos reestruturar nosso objeto de suporte ao cliente? mais detalhes, talvez adicionando alguma estrutura adicional, como no Exemplo 5-4? O os dados que nosso serviço de e-mail deseja ainda estão lá, com o mesmo nome, mas se nosso o código faz suposições muito explícitas sobre onde o primeiro nome e os campos de sobrenome serão armazenados e, em seguida, poderão ser interrompidos novamente. Neste caso, nós poderia, em vez disso, usar o XPath para extrair os campos que nos interessam, o que nos permite seja ambivalente sobre onde estão os campos, desde que possamos encontrá-los. Isso padrão de implementar um leitor capaz de ignorar mudanças que não nos importam sobre - é o que Martin Fowler chama de leitor tolerante.

Exemplo 5-4. Um recurso reestruturado do cliente: os dados ainda estão lá, mas nossos consumidores podem encontrá-lo?

```
<customer>
  <naming>
    <firstname>Sam</firstname>
    <lastname>Newman</lastname>
    <nickname>Cérebro de pega</nickname>
    <fullname>Sam "Magpiebrain" Newman</fullname>
  </naming>
  <email>sam@magpiebrain.com</email>
</customer>
```

O exemplo de um cliente tentando ser o mais flexível possível ao consumir um serviço demonstra a lei de Postel (também conhecida como robustez), princípio, que afirma: "Seja conservador no que você faz, seja liberal no que você aceita dos outros." O contexto original para essa peça de sabedoria era a interação de dispositivos em redes, onde você deve esperar todos os tipos de coisas estranhas que aconteceram. No contexto de interações baseadas em microserviços, nos leva a tentar estruturar nosso código de cliente para tolerar mudanças em cargas úteis.

Tecnologia certa

Como já exploramos, algumas tecnologias podem ser mais frágeis quando vêm nos permitir mudar as interfaces - eu já destaquei a minha frustrações pessoais com o Java RMI. Por outro lado, alguma integração as implementações se esforçam para facilitar ao máximo

mudanças a serem feitas sem prejudicar os clientes. No final simples do espectro, buffers de protocolo, o formato de serialização usado como parte do gRPC, têm o conceito de número de campo. Cada entrada em um buffer de protocolo deve definir um número de campo, que o código do cliente espera encontrar. Se novos campos forem acrescentados, o cliente não se importa. O Avro permite que o esquema seja enviado junto com a carga útil, permitindo que os clientes potencialmente interpretem muito uma carga útil como um tipo dinâmico.

Na extremidade mais extrema do espectro, o conceito REST do HATEOAS é em grande parte, trata-se de permitir que os clientes usem endpoints REST, mesmo quando eles mudam usando os links hipermídia discutidos anteriormente.

Isso exige que você acredite em toda a mentalidade da HATEOAS, é claro.

Interface explícita

Sou um grande fã de um microsserviço que expõe um esquema explícito que denota o que seus terminais fazem. Ter um esquema explícito deixa isso claro para os consumidores o que eles podem esperar, mas também deixa tudo muito mais claro para um desenvolvedor trabalhando em um microsserviço sobre quais coisas devem permanecer intocadas garanta que você não prejudique os consumidores. Dito de outra forma, um esquema explícito funciona um longo caminho para tornar os limites da ocultação de informações mais explícitos - o que está exposto no esquema, por definição, não está oculto.

Ter um esquema explícito para RPC está estabelecido há muito tempo e, na verdade, é um requisito para muitas implementações de RPC, o REST, por outro lado, tem normalmente vejo o conceito de um esquema como opcional, a ponto de eu achar esquemas explícitos para endpoints REST são extremamente raros. Isso é mudando, com coisas como a especificação OpenAPI mencionada acima ganhando tracção e com a especificação JSON Schema também ganhando maturidade.

Os protocolos de mensagens assíncronas têm tido mais dificuldades nesse espaço. Você pode ter um esquema para a carga útil de uma mensagem com bastante facilidade e, de fato, esta é uma área na qual o Avro é usado com frequência. No entanto, ter um explícito a interface precisa ir além disso. Se considerarmos um microsserviço que eventos de incêndios, quais eventos ele expõe? Algumas tentativas de tornar explícito esquemas para endpoints baseados em eventos já estão em andamento. Um é o AsyncAPI,

que conquistou vários usuários renomados, mas o que mais está ganhando a tração parece ser CloudEvents, uma especificação apoiada pelo Fundação de computação nativa em nuvem (CNCF). Produto de grade de eventos do Azure suporta o formato CloudEvents, um sinal de que diferentes fornecedores estão apoiando isso formato, que deve ajudar na interoperabilidade. Isso ainda é relativamente novo espaço, então será interessante ver como as coisas acontecerão nos próximos anos.

CONTROLE DE VERSÃO SEMÂNTICO

Não seria ótimo se, como cliente, você pudesse ver apenas a versão número de um serviço e sabe se você pode se integrar a ele?

O controle de versão semântico é uma especificação que permite exatamente isso. Com controle de versão semântico, cada número de versão está no formato

MAJOR.MINOR.PATCH. Quando o número MAJOR aumenta, significa que alterações incompatíveis com versões anteriores foram feitas. Quando MENOR incrementos, foi adicionada uma nova funcionalidade que deve ser invertida compatível. Finalmente, uma alteração no PATCH afirma que as correções de bugs foram feito com a funcionalidade existente.

Para ver como o controle de versão semântico pode ser útil, vamos dar uma olhada em um uso simples desse. Nossa aplicativo de helpdesk foi desenvolvido para funcionar com a versão 1.2.0 do atendimento ao cliente. Se um novo recurso for adicionado, fazendo com que o Cliente serviço a ser alterado para 1.3.0, nosso aplicativo de helpdesk não deve sofrer alterações no comportamento e não se deve esperar que faça nenhuma alteração. Não pudemos garantia de que poderíamos trabalhar com a versão 1.1.0 do Cliente serviço, no entanto, pois podemos contar com a funcionalidade adicionada na versão 1.2.0 liberar. Também podemos esperar fazer alterações em nosso aplicativo se uma nova versão 2.0.0 do atendimento ao cliente for lançada.

Você pode decidir ter uma versão semântica para o serviço, ou mesmo para um endpoint individual em um serviço, se você os estiver coexistindo, como detalhado na próxima seção.

Esse esquema de controle de versão nos permite empacotar muitas informações e expectativas em apenas três campos. Os esboços completos da especificação em muito termos simples: as expectativas que os clientes podem ter de mudanças nesses números, e pode simplificar o processo de comunicação sobre se as mudanças devem impactar os consumidores. Infelizmente, eu não vi isso abordagem usada o suficiente em sistemas distribuídos para entender sua eficácia nesse contexto - algo que realmente não mudou desde a primeira edição deste livro.

Detecte alterações accidentais de interrupção antecipada

É crucial que captarmos mudanças que prejudicarão os consumidores o mais rápido possível. Possível, porque mesmo se escolhermos a melhor tecnologia possível, um. Uma mudança inocente de um microsserviço pode fazer com que os consumidores falhem. Como já mencionamos que o uso de esquemas pode nos ajudar a entender a estrutura mudanças, supondo que usemos algum tipo de ferramenta para ajudar a comparar o esquema versões. Existe uma grande variedade de ferramentas para fazer isso para diferentes tipos de esquema. Temos Protolock para buffers de protocolo, json-schema-diff- validador para o esquema JSON e openapi-diff para o OpenAPI specification. Mais ferramentas parecem estar surgindo o tempo todo nesse espaço. O que você está procurando, porém, é algo que não apenas informe sobre as diferenças entre dois esquemas, mas passarão ou falharão com base em compatibilidade; isso permitiria que você falhasse em uma compilação de CI se fosse incompatível esquemas são encontrados, garantindo que seu microsserviço não seja implantado.

O Confluent Schema Registry de código aberto oferece suporte ao esquema JSON, Avro, e buffers de protocolo e é capaz de comparar versões recém-carregadas para compatibilidade com versões anteriores. Embora tenha sido construído para ajudar como parte de um ecossistema no qual o Kafka está sendo usado e precisa de Kafka para funcionar, existe nada que impeça você de usá-lo para armazenar e validar esquemas que estão sendo usados para Comunicação não baseada em Kafka

As ferramentas de comparação de esquemas podem nos ajudar a detectar quebras estruturais, mas o que sobre quebras semânticas? Ou se você não estiver fazendo uso de esquemas em o primeiro lugar? Então, estamos analisando os testes. Este é um tópico que exploraremos em mais detalhes em "Testes de contrato e contratos orientados ao consumidor (CDCs)", mas Eu queria destacar os testes de contratos orientados pelo consumidor, o que ajuda explicitamente nesta área, o Pacto é um excelente exemplo de uma ferramenta voltada especificamente para esse problema. Lembre-se de que, se você não tiver esquemas, espere seus testes ter que trabalhar mais para detectar mudanças importantes.

Se você oferece suporte a várias bibliotecas de cliente diferentes, execute testes usando cada biblioteca que você oferece suporte com o serviço mais recente é outra técnica que pode ajudar. Depois de perceber que vai quebrar um consumidor, você tem o escolha entre tentar evitar completamente a ruptura ou então abraçá-la e começar

ter as conversas certas com as pessoas que cuidam do consumidor serviços.

Gerenciando alterações importantes

Então, você foi o mais longe possível para garantir que as mudanças que está fazendo sejam a interface de um microsserviço é compatível com versões anteriores, mas você percebeu que você só precisa fazer uma alteração que constitua uma alteração significativa. O que você pode fazer isso em tal situação? Você tem três opções principais:

Implantação em etapas

Exija que o microsserviço exponha a interface e todos os consumidores do essa interface é alterada ao mesmo tempo.

Coexistam versões de microsserviços incompatíveis

Execute versões antigas e novas do microsserviço lado a lado.

Emule a interface antiga

Faça com que seu microsserviço exponha a nova interface e também emule o interface antiga.

Implantação Lockstep

Obviamente, a implantação contínua contraria a capacidade de implantação independente. Se quisermos implantar uma nova versão do nosso microsserviço com um interrompendo mudanças em sua interface, mas ainda fazemos isso de forma independente, nós precisamos dar tempo aos nossos consumidores para fazer o upgrade para a nova interface. Isso conduz vamos para as próximas duas opções que eu consideraria.

Coexistam versões de microsserviços incompatíveis

Outra solução de controle de versão frequentemente citada é ter versões diferentes do serviço ao vivo de uma só vez e para consumidores mais velhos direcionarem seu tráfego para os mais velhos versão, com consumidores mais novos vendo a nova versão, conforme mostrado em

Figura 5-3. Essa é a abordagem usada com moderação pela Netflix em situações em que o custo de mudar consumidores mais velhos é muito alto, especialmente em casos raros. Em casos em que dispositivos antigos ainda estão vinculados a versões mais antigas da API. Eu sou fã dessa ideia pessoalmente e entendo por que a Netflix a usa raramente. Primeiro, se eu precisar corrigir um bug interno no meu serviço, agora preciso corrigir e implantar dois conjuntos diferentes de serviços. Isso provavelmente significaria que eu tenho que ramificar a base de código do meu serviço, e isso é sempre problemático. Em segundo lugar, isso significa que preciso de inteligência para lidar com o direcionamento dos consumidores para a direita microsserviço. Esse comportamento inevitavelmente acaba ficando no middleware em algum lugar, ou em vários scripts nginx, tornando mais difícil raciocinar sobre o comportamento do sistema. Por fim, considere qualquer estado persistente em nosso serviço pode gerenciar. Os clientes criados por qualquer uma das versões do serviço precisam ser armazenado e tornado visível para todos os serviços, independentemente da versão usada. Crie os dados em primeiro lugar. Isso pode ser uma fonte adicional de complexidade.

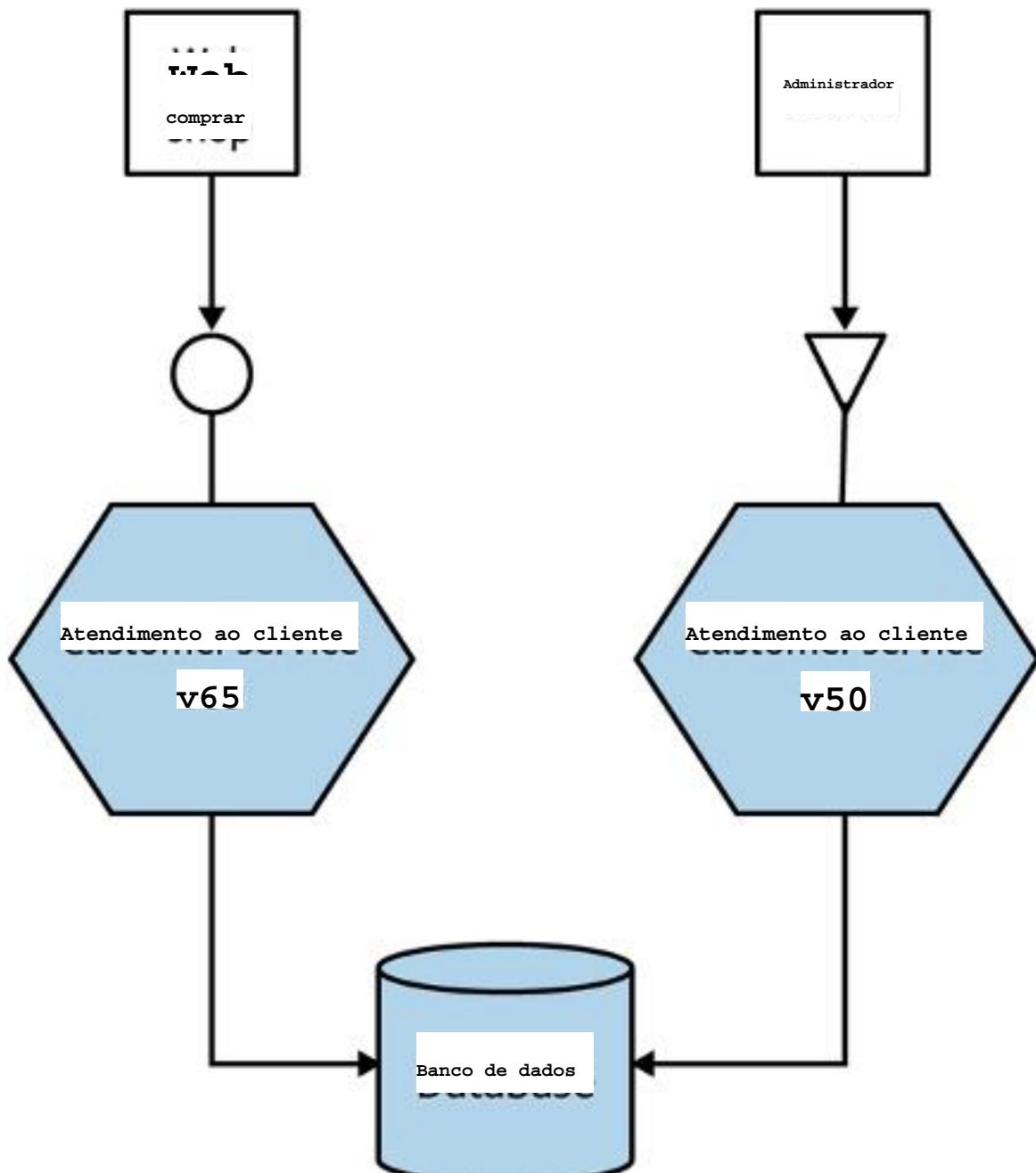


Figura 5-3. Executando várias versões do mesmo serviço para dar suporte a endpoints antigos.

A coexistência de versões simultâneas de serviços por um curto período de tempo pode fazer bom senso, especialmente quando você está fazendo algo como um canário (discutiremos mais sobre esse padrão em "Sobre a entrega progressiva"). Em nessas situações, podemos coexistir com versões por apenas alguns minutos ou talvez horas, e normalmente teremos apenas duas versões diferentes do serviço presente ao mesmo tempo. Quanto mais tempo você demora para conseguir consumidores

atualizado para a versão mais recente e lançada, mas você deve procurar coexistir diferentes endpoints no mesmo microsserviço em vez de coexistir versões totalmente diferentes. Ainda não estou convencido de que esse trabalho valha a pena para um projeto médio.

Emule a interface antiga

Se tivermos feito tudo o que pudemos para evitar a introdução de uma alteração significativa na interface, nosso próximo trabalho é limitar o impacto. O que queremos evitar é forçar consumidores devem se atualizar em sintonia conosco, pois sempre queremos manter a capacidade de liberar microsserviços independentemente uns dos outros. Uma abordagem I ter usado com sucesso para lidar com isso é coexistir tanto o antigo quanto o novo interfaces no mesmo serviço em execução. Então, se quisermos liberar uma pausa mudança, implantamos uma nova versão do serviço que expõe tanto a antiga quanto a novas versões do endpoint.

Isso nos permite lançar o novo microsserviço o mais rápido possível, junto com a nova interface, ao mesmo tempo em que dá tempo para os consumidores se mudarem. Uma vez todos os consumidores não estão mais usando o endpoint antigo, você pode removê-lo junto com qualquer código associado, conforme mostrado na Figura 5-4.

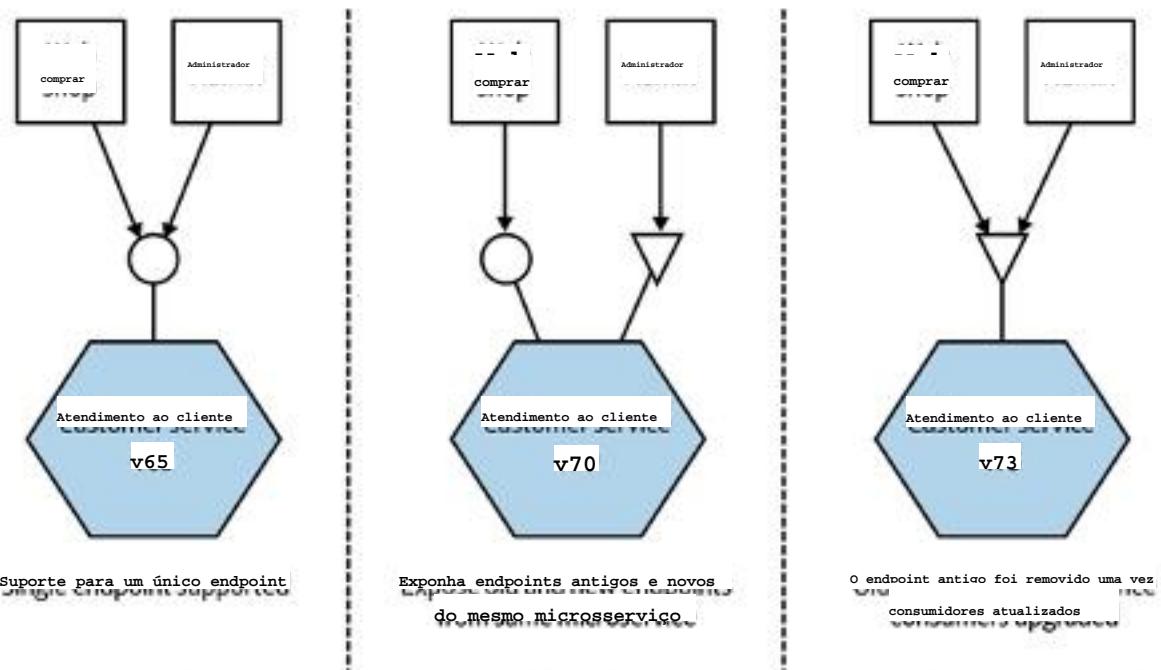


Figura 5-4. Um microsserviço emulando o endpoint antigo e expondo o novo retroativo endpoint incompatível.

Quando usei essa abordagem pela última vez, nos metemos em uma bagunça. com o número de consumidores que tínhamos e o número de mudanças significativas nós fizemos. Isso significava que na verdade estávamos coexistindo três diferentes versões do endpoint. Isso não é algo que eu recomendaria! Mantendo tudo o código e os testes associados necessários para garantir que todos funcionassem foi absolutamente um fardo adicional. Para tornar isso mais gerenciável, nós transformou internamente todas as solicitações do endpoint V1 em uma solicitação V2 e em seguida, solicitações V2 para o endpoint V3. Isso significava que poderíamos delinear claramente qual código seria retirado quando o (s) endpoint (s) antigo (s) morresse (s).

Este é, na verdade, um exemplo do padrão de expansão e contracção, que permite nós introduzimos gradualmente as mudanças significativas. Expandidos as capacidades que oferecemos, apoiando formas antigas e novas de fazer algo. Uma vez o velho os consumidores fazem as coisas da nova maneira, contratamos nossa API, removendo a antiga funcionalidade.

Se você quiser coexistir com endpoints, precisará de uma maneira de os chamadores rotearem seus solicita adequadamente. Para sistemas que usam HTTP, eu vi isso ser feito com os dois números de versão nos cabeçalhos da solicitação e também no próprio URI para exemplo, /v1/customer/ ou /v2/customer/. Estou indeciso sobre qual abordagem

faz mais sentido. Por um lado, gosto que os URIs sejam opacos, para desencorajar os clientes de codificar modelos de URI, mas, por outro lado, isso. A abordagem torna as coisas muito óbvias e pode simplificar o roteamento de solicitações.

Para RPC, as coisas podem ser um pouco mais complicadas. Eu lidei com isso com o protocolo buffers colocando meus métodos em namespaces diferentes - por exemplo, v1. CreateCustomer e v2. CreateCustomer-mas quando você está tentando suportar diferentes versões dos mesmos tipos enviadas pela rede, isso a abordagem pode se tornar muito dolorosa.

Qual abordagem eu prefiro?

Para situações em que a mesma equipe gerencia o microsserviço e tudo consumidores, estou um pouco relaxado com o lançamento de um bloqueio limitado situações. Supondo que seja realmente uma situação única, faça isso quando o impacto é limitado a uma única equipe e pode ser justificável. Eu sou muito cauteloso com isso, porém, pois existe o perigo de que uma atividade única se torne um negócio como normal, e aí vai a capacidade de implantação independente. Use implantações em etapas com muita frequência, e você acabará com um monólito distribuído em pouco tempo.

A coexistência de versões diferentes do mesmo microsserviço pode ser problemática, como discutimos. Eu consideraria fazer isso apenas em situações em que planejou executar as versões de microsserviços lado a lado por apenas um curto período de tempo. A realidade é que, quando você precisa dar aos consumidores tempo para fazer o upgrade, você pode esperar semanas ou mais. Em outras situações em que você possa coexistir versões de microsserviços, talvez como parte de uma implantação azul-esverdeada ou liberação canária, as durações envolvidas são muito mais curtas, compensando o desvantagens dessa abordagem.

Minha preferência geral é usar a emulação de terminais antigos em qualquer lugar possível. Os desafios de implementar a emulação são, na minha opinião, muitos mais fácil de lidar do que com versões coexistentes de microsserviços.

O contrato social

A abordagem que você escolher se deve em grande parte às expectativas. Os consumidores sabem como essas mudanças serão feitas. Mantendo o antigo interface disponível pode ter um custo e, idealmente, você gostaria de desligá-la e remova o código e a infraestrutura associados o mais rápido possível. Sobre o por outro lado, você quer dar aos consumidores o máximo de tempo possível para fazer um mudar. E lembre-se de que, em muitos casos, a incompatibilidade com versões anteriores muda você está fazendo muitas vezes coisas que foram solicitadas pelos consumidores e/ou, na verdade, acabará beneficiando-os. Há um ato de equilíbrio, de claro, entre as necessidades dos mantenedores de microserviços e as do consumidores, e isso precisa ser discutido.

Descobri que, em muitas situações, a forma como essas mudanças serão tratadas foi nunca foi discutido, levando a todos os tipos de desafios. Assim como nos esquemas, ter algum grau de clareza sobre como a incompatibilidade com versões anteriores muda será feito pode simplificar muito as coisas.

Você não precisa necessariamente de muitos papéis e grandes reuniões para chegar acordo sobre como as mudanças serão tratadas, mas supondo que você não vá na rota dos lançamentos por etapas, sugiro que tanto o proprietário quanto o consumidor de um microserviço precisa ter clareza sobre algumas coisas:

- Como você levantará a questão de que a interface precisa mudar?
- Como os consumidores e as equipes de microserviços colaborarão para concorda com a aparência da mudança?
- Quem deve fazer o trabalho para atualizar os consumidores?
- Quando a mudança for acordada, por quanto tempo os consumidores terão que mudar para a nova interface antes de ser removida?

Lembre-se de que um dos segredos de uma arquitetura de microserviços eficaz é adote uma abordagem que prioriza o consumidor. Seus microserviços existem para serem chamados outros consumidores. As necessidades dos consumidores são fundamentais, e se você está fazendo mudanças em um microserviço que causarão consumidores upstream problemas, isso precisa ser levado em consideração.

Em algumas situações, é claro, talvez não seja possível alterar o consumidores. Ouvi da Netflix que eles tiveram problemas (pelo menos historicamente) com decodificadores antigos usando versões mais antigas das APIs da Netflix. Esses set-top boxes caixas não podem ser atualizadas facilmente, então os endpoints antigos precisam permanecer disponível, a menos e até que o número de decodificadores antigos caia para um nível no qual eles podem ter seu suporte desativado. Decisões de parar de envelhecer os consumidores, ao conseguirem acessar seus endpoints, às vezes podem acabar sendo financeiros - quanto dinheiro custa para você apoiar os antigos interface, equilibrada com a quantidade de dinheiro que você ganha com eles consumidores.

Uso de rastreamento

Mesmo que você concorde com um momento em que os consumidores devem parar de usar o antigo interface, você saberia se eles realmente pararam de usá-la? Certificando-se você tem um local de login para cada endpoint que seu microsserviço expõe ajuda, assim como garantir que você tenha algum tipo de identificador de cliente para que possa conversar com a equipe em questão se precisar trabalhar com eles para ajudá-los migrar para longe da sua interface antiga. Isso pode ser algo tão simples quanto pedindo aos consumidores que coloquem seu identificador no cabeçalho do agente do usuário quando fazendo solicitações HTTP, ou você pode exigir que todas as chamadas passem por algum tipo de Gateway de API em que os clientes precisam de chaves para se identificarem.

Medidas extremas

Então, supondo que você saiba que um consumidor ainda está usando uma interface antiga, você querem remover, e eles estão se esforçando para mudar para o novo versão, o que você pode fazer sobre isso? Bem, a primeira coisa a fazer é falar com eles. Talvez você possa ajudá-los a fazer as mudanças acontecerem. Se tudo mais falha e eles ainda não atualizam, mesmo depois de concordarem em fazer isso, existem alguns técnicas extremas que eu vi usadas.

Em uma grande empresa de tecnologia, discutimos como ela lidou com esse problema. Internamente, a empresa teve um período muito generoso de um ano antes de envelhecer as interfaces seriam retiradas. Perguntei como ela sabia se os consumidores ainda estavam

usando as interfaces antigas, e a empresa respondeu que realmente não se incomodou rastreando essas informações; depois de um ano, ele simplesmente desligou a interface antiga. É foi reconhecido internamente que, se isso fez com que um consumidor quebrasse, era o culpa da equipe consumidora do microsserviço: eles tiveram um ano para fazer o mudou e não tinha feito isso. Obviamente, essa abordagem não funcionará para muitos (I disse que era extremo!). Isso também leva a um alto grau de ineficiência. Por não sabendo se a interface antiga era usada, a empresa negou a si mesma a oportunidade de removê-lo antes do fim do ano. Pessoalmente, mesmo se eu fosse para sugerir apenas desligar o endpoint após um certo período de tempo, eu ainda definitivamente quero rastrear quem seria afetado.

Outra medida extrema que vi foi, na verdade, no contexto de depreciação bibliotecas, mas teoricamente também poderia ser usado para endpoints de microsserviços. O exemplo dado foi de uma antiga biblioteca que as pessoas estavam tentando se aposentar. do uso dentro da organização em favor de uma nova e melhor. Apesar de muitos de trabalho para mover o código para usar a nova biblioteca, algumas equipes ainda estavam arrastando seus calcanhares. A solução foi inserir um sono na biblioteca antiga para que que respondeu mais lentamente às chamadas (com registro para mostrar o que foi acontecendo). Com o tempo, a equipe responsável pela depreciação continuou aumentando a duração do sono, até que finalmente as outras equipes receberam a mensagem. Obviamente, você tem que ter certeza absoluta de que esgotou outros esforços razoáveis para fazer com que os consumidores atualizem antes de considerar algo assim!

DRY e os perigos da reutilização de código em um Mundo dos microsserviços

Uma das siglas que nós desenvolvedores ouvimos muito é DRY: não se repita. Embora sua definição às vezes seja simplificada, pois tenta evitar a duplicação, código, DRY, com mais precisão, significa que queremos evitar a duplicação de nosso comportamento e conhecimento do sistema. Este é um conselho muito sensato em geral. Ter muitas linhas de código que fazem a mesma coisa cria sua base de código, maior do que o necessário e, portanto, mais difícil de raciocinar. Quando você quiser mude o comportamento, e esse comportamento é duplicado em muitas partes do seu

sistema, é fácil esquecer em todos os lugares que você precisa fazer uma alteração, o que pode levar a insetos. Portanto, usar o DRY como um mantra em geral faz sentido.

DRY é o que nos leva a criar um código que pode ser reutilizado. Nós puxamos duplicado codifique em abstrações que podemos chamar de vários lugares. Talvez nós

chegue ao ponto de criar uma biblioteca compartilhada que possamos usar em qualquer lugar! Acontece que, no entanto, esse compartilhamento de código em um ambiente de microsserviço é um pouco mais mais envolvido do que isso. Como sempre, temos mais de uma opção a considerar.

Compartilhando código por meio de bibliotecas

Uma coisa que queremos evitar a todo custo é o acoplamento excessivo de um microsserviço e consumidores de forma que qualquer pequena alteração no microsserviço em si possa causar mudanças desnecessárias ao consumidor. Às vezes, no entanto, o uso de o código compartilhado pode criar esse mesmo acoplamento. Por exemplo, em um cliente que tivemos uma biblioteca de objetos de domínio comuns que representavam as entidades principais em uso em nosso sistema. Essa biblioteca foi usada por todos os serviços que tínhamos. Mas quando um foi feita uma alteração em um deles, todos os serviços tiveram que ser atualizados. Nosso sistema comunicados por meio de filas de mensagens, que também precisavam ser drenadas de suas Agora o conteúdo é inválido, e ai de você se você esqueceu...

Se o uso do código compartilhado alguma vez sair do limite do seu serviço, você introduziram uma forma potencial de acoplamento. Usando código comum como registrar bibliotecas é bom, pois são conceitos internos que são invisíveis para o mundo exterior. O site realestate.com.au faz uso de um serviço personalizado modelo para ajudar a iniciar a criação de novos serviços. Em vez de criar esse código compartilhado, a empresa o copia para cada novo serviço para garantir esse acoplamento não vaza.

O ponto realmente importante sobre o compartilhamento de código por meio de bibliotecas é que você não pode atualize todos os usos da biblioteca de uma só vez. Embora vários microsserviços possam todos usam a mesma biblioteca, normalmente o fazem empacotando essa biblioteca na implantação de microsserviços. Para atualizar a versão da biblioteca que está sendo usada, portanto, você precisaria reimplantar o microsserviço. Se você quiser atualizar a mesma biblioteca em todos os lugares exatamente ao mesmo tempo, isso pode levar a um

implantação generalizada de vários microserviços diferentes, todos ao mesmo tempo tempo, com todas as dores de cabeça associadas.

Então, se você estiver usando bibliotecas para reutilização de código além dos limites de microserviços, você tem que aceitar que várias versões diferentes da mesma biblioteca possam estar lá ao mesmo tempo. É claro que você pode atualizar tudo isso para a última versão ao longo do tempo, mas contanto que você esteja de acordo com esse fato, então por tudo significa reutilizar código por meio de bibliotecas. Se você realmente precisa atualizar esse código para todos os usuários exatamente ao mesmo tempo, então você realmente vai querer ver em vez disso, reutilizando código por meio de um microserviço dedicado

Há um caso de uso específico associado à reutilização por meio de bibliotecas, que é No entanto, vale a pena explorar mais.

Bibliotecas de clientes

Falei com mais de uma equipe que insistiu em criar um cliente bibliotecas para seus serviços são uma parte essencial da criação de serviços no primeiro lugar. O argumento é que isso facilita o uso do seu serviço e evita a duplicação do código necessária para consumir o serviço em si.

O problema, claro, é que, se as mesmas pessoas criarem a API do servidor e a API do cliente, existe o perigo de que a lógica exista no servidor começará a vaziar para o cliente. Eu deveria saber: eu fiz isso sozinho. O quanto mais lógica se infiltra na biblioteca do cliente, mais coesão começa a surgir avaria, e você terá que mudar vários clientes para rolar envie correções para o seu servidor. Você também limita as opções de tecnologia, especialmente se exigem que a biblioteca do cliente seja usada.

Um modelo que eu gosto para bibliotecas de clientes é o da Amazon Web Services. (FOI). As chamadas de serviços web SOAP ou REST subjacentes podem ser feitas diretamente, mas todo mundo acaba usando apenas um dos vários softwares existentes kits de desenvolvimento (SDKs), que fornecem abstrações sobre o subjacente API. Esses SDKs, no entanto, são escritos pela comunidade em geral, ou então por pessoas dentro da AWS, exceto aquelas que trabalham na própria API. Este diploma da separação parece funcionar e evita algumas das armadilhas do cliente bibliotecas. Parte da razão pela qual isso funciona tão bem é que o cliente está no comando.

de quando a atualização acontece. Se você seguir o caminho das bibliotecas de clientes você mesmo, certifique-se de que este seja o caso.

A Netflix, em particular, dá ênfase especial à biblioteca do cliente, mas eu me preocupo que as pessoas vejam isso apenas pela lente de evitar a duplicação de código. Em fato, as bibliotecas de clientes usadas pela Netflix são muito, se não mais, sobre garantindo a confiabilidade e a escalabilidade de seus sistemas. O cliente Netflix lidam com descoberta de serviços, modos de falha, registro e outros aspectos que na verdade não têm a ver com a natureza do serviço em si. Sem esses compartilhados clientes, seria difícil garantir que cada peça de cliente/servidor as comunicações se comportaram bem na grande escala em que a Netflix opera. Seu uso na Netflix certamente facilitou a instalação e o funcionamento aumente a produtividade e, ao mesmo tempo, garanta que o sistema se comporte bem. No entanto, de acordo com pelo menos uma pessoa da Netflix, com o tempo isso levou a um grau do acoplamento entre cliente e servidor que tem sido problemático.

Se a abordagem da biblioteca cliente é algo em que você está pensando, pode ser importante separar o código do cliente para lidar com o transporte subjacente protocolo, que pode lidar com coisas como descoberta e falha de serviços, a partir de coisas relacionadas ao próprio serviço de destino. Decida se você é ou não insistirá em que a biblioteca do cliente seja usada, ou se você permitir que pessoas usando diferentes pilhas de tecnologia para fazer chamadas para a API subjacente. E finalmente, certifique-se de que os clientes sejam responsáveis por quando atualizar seu cliente bibliotecas: precisamos garantir que mantenhamos a capacidade de liberar nossos serviços independentemente um do outro!

Descoberta de serviços

Depois de ter mais do que alguns microsserviços disponíveis, sua atenção inevitavelmente se transforma em saber onde tudo está na terra. Talvez você queira para saber o que está sendo executado em um determinado ambiente para que você saiba o que deve esteja monitorando. Talvez seja tão simples quanto saber onde estão suas contas o microsserviço é para que seus consumidores saibam onde encontrá-lo. Ou talvez você apenas quer tornar mais fácil para os desenvolvedores de sua organização saberem quais APIs estão disponíveis para que eles não reinventem a roda. De um modo geral, todos esses

os casos de uso estão sob a bandeira da descoberta de serviços. E como sempre com microsserviços, temos várias opções diferentes à nossa disposição para lidando com isso.

Todas as soluções que analisaremos lidam com as coisas em duas partes. Primeiro, eles forneca algum mecanismo para que uma instância se registre e diga: "Estou aqui!" Em segundo lugar, eles fornecem uma maneira de encontrar o serviço depois de registrado. No entanto, a descoberta de serviços fica mais complicada quando consideramos um ambiente no qual estamos constantemente destruindo e implantando novos instâncias de serviços. Idealmente, gostaríamos de lidar com qualquer solução que escolhermos. com isso.

Vejamos algumas das soluções mais comuns para prestação de serviços e considere nossas opções.

Sistema de nomes de domínio (DNS)

É bom começar de forma simples. O DNS nos permite associar um nome ao endereço IP de uma ou mais máquinas. Poderíamos decidir, por exemplo, que nossas contas o microservice é sempre encontrado em accounts.musiccorp.net. Nós gostaríamos então ter esse ponto de entrada para o endereço IP do host que executa esse microsserviço, ou talvez resolvêssemos isso para um balanceador de carga que está distribuindo carga em várias instâncias. Isso significa que teríamos que lidar com a atualização desses entradas como parte da implantação do nosso serviço.

Ao lidar com instâncias de um serviço em diferentes ambientes, tenho vi que um modelo de domínio baseado em convenções funciona bem. Por exemplo, podemos tenha um modelo definido como <servicename>-<environment>.musiccorp.net fornecendo entradas como accounts-uat.musiccorp.net ou accounts-dev.musiccorp.net.

Uma forma mais avançada de lidar com ambientes diferentes é ter diferentes servidores de nomes de domínio para esses ambientes. Então eu poderia supor que accounts.musiccorp.net é onde eu sempre encontro o microsserviço de contas, mas isso pode ser resolvido para diferentes hosts, dependendo de onde eu faço a pesquisa. Se você já tem seus ambientes em diferentes segmentos de rede e estão confortáveis em gerenciar seus próprios servidores e entradas de DNS, isso poderia

é uma solução bastante bacana, mas dá muito trabalho se você não está recebendo outros benefícios dessa configuração.

O DNS tem uma série de vantagens, a principal delas é que é um bom padrão compreendido e bem usado que quase qualquer pilha de tecnologia usará para isso. Infelizmente, embora existam vários serviços para gerenciar o DNS dentro de uma organização, poucos deles parecem projetados para um ambiente em com os quais estamos lidando com hosts altamente descartáveis, atualizando o DNS entradas um pouco dolorosas. O serviço Route 53 da Amazon faz um trabalho muito bom disso, mas ainda não vi uma opção de hospedagem própria que seja tão boa, embora (como discutiremos em breve) algumas ferramentas dedicadas de descoberta de serviços, como o Consul pode nos ajudar aqui. Além dos problemas na atualização de entradas de DNS, o DNS a especificação em si pode nos causar alguns problemas.

As entradas de DNS para nomes de domínio têm um tempo de vida (TTL). Este é o tempo que um cliente pode considerar a entrada como nova. Quando queremos mudar o host para o qual o nome de domínio se refere, atualizamos essa entrada, mas temos que assumir que os clientes manterão o IP antigo por pelo menos o tempo que o TTL indicar.

As entradas de DNS podem ser armazenadas em cache em vários lugares (até mesmo a JVM armazena em cache)

Entradas de DNS (a menos que você diga para não fazer isso) e quanto mais lugares elas estiverem armazenadas em cache, quanto mais obsoleta a entrada pode ser.

Uma maneira de contornar esse problema é ter a entrada do nome de domínio para seu ponto de serviço para um平衡ador de carga, que por sua vez aponta para as instâncias de seu serviço, conforme mostrado na Figura 5-5. Quando você implanta uma nova instância, você pode retirar o antigo da entrada do balanceador de carga e adicionar o novo. Algumas pessoas usam o DNS round-robin, onde o próprio DNS entra referem-se a um grupo de máquinas. Essa técnica é extremamente problemática, pois o cliente está oculto do host subjacente e, portanto, não pode parar facilmente rotear o tráfego para um dos hosts caso ele fique doente.

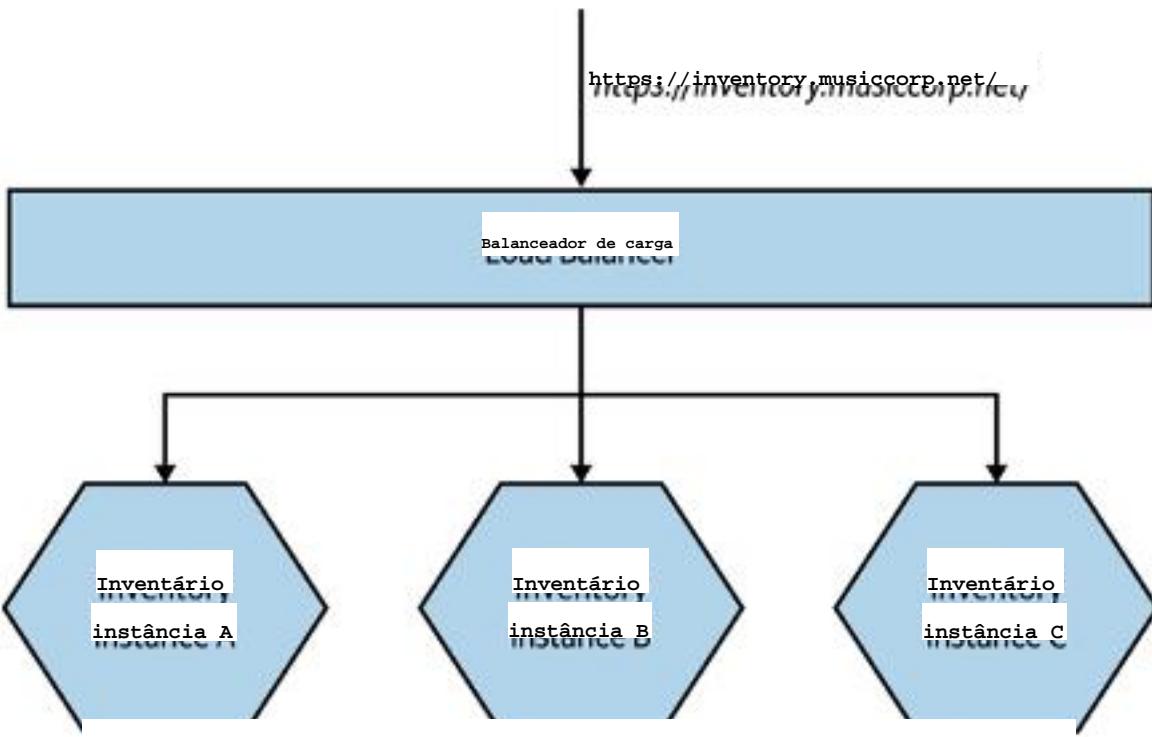


Figura 5-5. Usando o DNS para resolver um balanceador de carga para evitar entradas de DNS obsoletas

Conforme mencionado, o DNS é bem compreendido e amplamente suportado. Mas isso acontece tem uma ou duas desvantagens. Eu sugiro que você investigue se é um bom ajuste para você antes de escolher algo mais complexo. Para uma situação em onde você tem apenas nós únicos, ter o DNS se referindo diretamente aos hosts é provavelmente está bem. Mas para aquelas situações em que você precisa de mais de um instância de um host, faça com que as entradas de DNS sejam resolvidas em平衡adores de carga que possam lidar com a colocação e saída de hosts individuais em serviço, conforme apropriado.

Registros de serviços dinâmicos

As desvantagens do DNS como forma de encontrar nós em um ambiente altamente dinâmico o meio ambiente levou a uma série de sistemas alternativos, a maioria dos quais envolvem o registro do serviço em algum registro central, que por sua vez oferece a capacidade de consultar esses serviços posteriormente. Freqüentemente, esses sistemas fazem mais do que apenas fornecer registro e descoberta de serviços, o que pode ou pode não ser uma coisa boa. Este é um campo lotado, então vamos dar uma olhada em algumas opções para dar uma ideia do que está disponível.

ZooKeeper

O ZooKeeper foi originalmente desenvolvido como parte do projeto Hadoop. É usado para uma variedade quase desconcertante de casos de uso, incluindo configuração gerenciamento, sincronização de dados entre serviços, eleição de líderes, mensagem filas e (útil para nós) como um serviço de nomenclatura.

Como muitos tipos similares de sistemas, o ZooKeeper depende da execução de vários nós em um cluster para fornecer várias garantias. Isso significa que você deve espere executar pelo menos três nós do Zookeeper. A maior parte da inteligência em o ZooKeeper está disponível para garantir que os dados sejam replicados com segurança entre eles nós, e que as coisas permanecem consistentes quando os nós falham.

Em sua essência, o ZooKeeper fornece um namespace hierárquico para armazenar informações. Os clientes podem inserir novos nós nessa hierarquia, alterá-los ou consulte-os. Além disso, eles podem adicionar relógios aos nós para saber quando mudar. Isso significa que podemos armazenar as informações sobre onde nossos serviços estão disponíveis. estão localizados nessa estrutura e, como cliente, são informados quando eles mudam.

O ZooKeeper é frequentemente usado como um repositório de configuração geral, então você também pode armazene a configuração específica do serviço nele, permitindo que você execute tarefas como alterando dinamicamente os níveis de registro ou desativando recursos de um sistema em execução.

Na realidade, existem melhores soluções para o registro dinâmico de serviços, na medida em que que eu evitaria ativamente o ZooKeeper para esse caso de uso hoje em dia.

Cônsul

Como o ZooKeeper, o Consul suporta gerenciamento de configuração e serviço descoberta. Mas vai além do ZooKeeper ao fornecer mais suporte para esses principais casos de uso. Por exemplo, ele expõe uma interface HTTP para serviço descoberta, e uma das características matadoras do Consul é que ele realmente fornece um Servidor DNS pronto para uso; especificamente, ele pode servir registros SRV, que fornece um IP e uma porta para um determinado nome. Isso significa que se parte do seu o sistema já usa DNS e pode suportar registros SRV, você pode simplesmente entrar Consule e comece a usá-lo sem nenhuma alteração em seu sistema existente.

O Consul também incorpora outros recursos que você pode achar úteis, como o capacidade de realizar verificações de saúde nos nós. Assim, o Cônsl poderia muito bem se sobrepor os recursos fornecidos por outras ferramentas de monitoramento dedicadas, embora você

provavelmente usaria o Consul como fonte dessas informações e depois as retiraria em uma configuração de monitoramento mais abrangente.

O Consul usa uma interface HTTP RESTful para tudo, desde o registro de um serviço para consultar o armazenamento de chave/valor ou inserir verificações de integridade. Isso faz a integração com diferentes pilhas de tecnologia é muito simples. Cônslul também tem um conjunto de ferramentas que funcionam bem com ele, melhorando ainda mais sua utilidade. Um exemplo é `consul-template`, que fornece uma maneira de atualizar arquivos de texto com base nas entradas no Cônslul. À primeira vista, isso não parece muito interessante, até considerar o fato de que com `consul-template` agora você pode alterar um valor no Consul-talvez a localização de um microsserviço ou uma configuração valorize e tenha arquivos de configuração em todo o sistema de forma dinâmica. Atualizar. De repente, qualquer programa que lê sua configuração a partir de um arquivo de texto pode tenha seus arquivos de texto atualizados dinamicamente sem precisar saber nada sobre o próprio Consul. Um ótimo caso de uso para isso seria adicionar dinamicamente ou removendo nós para um pool de平衡adores de carga usando um balanceador de carga de software como HAProxy.

Outra ferramenta que se integra bem ao Consul é o Vault, um gerenciamento de segredos ferramenta que revisitaremos em "Segredos". O gerenciamento de segredos pode ser doloroso, mas o A combinação de Consul e Vault certamente pode facilitar a vida.

etcd e Kubernetes

Se você estiver executando em uma plataforma que gerencia cargas de trabalho de contêineres para você, é provável que você já tenha um mecanismo de descoberta de serviços fornecido para você. O Kubernetes não é diferente e vem parcialmente do etcd, uma configuração loja de gerenciamento fornecida com o Kubernetes. etcd tem recursos semelhantes aos os do Consul, e o Kubernetes os usa para gerenciar uma ampla variedade de informações de configuração.

Exploraremos o Kubernetes com mais detalhes em "Kubernetes and Container". Orquestração", mas, em poucas palavras, a forma como a descoberta de serviços funciona Kubernetes é que você implanta um contêiner em um pod e depois um serviço identifica dinamicamente quais pods devem fazer parte de um serviço por padrão correspondência nos metadados associados ao pod. É muito elegante.

mecanismo e pode ser muito poderoso. As solicitações para um serviço serão então recebidas encaminhado para um dos pods que compõem esse serviço.

Os recursos que você obtém imediatamente com o Kubernetes podem muito bem resultar em você só querer se contentar com o que vem com a plataforma principal, evitando o uso de ferramentas dedicadas como o Consul, e para muitos isso torna um muito sentido, especialmente se o ecossistema mais amplo de ferramentas em torno do Consul não for de interesse para você. No entanto, se você estiver executando em um ambiente misto, onde você tem cargas de trabalho em execução no Kubernetes e em outros lugares e, em seguida, tem um ferramenta dedicada de descoberta de serviços que pode ser usada em ambas as plataformas pode ser o caminho a percorrer.

Rolando o seu

Uma abordagem que eu mesmo usei e vi usada em outros lugares é lançar seu sistema próprio. Em um projeto, estávamos fazendo uso intenso da AWS, que oferece a capacidade de adicionar tags às instâncias. Ao iniciar instâncias de serviço, eu aplicaria tags para ajudar a definir o que era a instância e o que ela foi usada para. Isso permitiu que alguns metadados ricos fossem associados a um determinado host -por exemplo:

- serviço = contas
- meio ambiente = produção
- versão = 154

Em seguida, usei as APIs da AWS para consultar todas as instâncias associadas a um determinado Conta da AWS para que eu pudesse encontrar as máquinas que me interessavam. Aqui, a própria AWS é lidar com o armazenamento dos metadados associados a cada instância e fornecendo-nos a capacidade de consultá-lo. Em seguida, criei ferramentas de linha de comando para interagindo com essas instâncias e fornecendo interfaces gráficas para visualização visão geral do status da instância. Tudo isso se torna uma tarefa bastante simples se você pode coletar informações programaticamente sobre as interfaces de serviço.

A última vez que fiz isso, não chegamos ao ponto de fazer com que os serviços usassem a AWS.

APIs para encontrar suas dependências de serviço, mas não há razão para você

não poderia. Obviamente, se você quiser que os serviços upstream sejam alertados quando o a localização de um serviço downstream muda, você está sozinho.

Hoje em dia, esse não é o caminho que eu seguiria. A safra de ferramentas neste espaço está maduro o suficiente para que este seja um caso de não apenas reinventar a roda mas recriando uma roda muito pior...

Não esqueça os humanos!

Os sistemas que analisamos até agora facilitam a execução de uma instância de serviço registre-se e procure outros serviços com os quais ele precisa conversar. Mas, como humanos, nós às vezes também queremos essa informação. Disponibilizando as informações em maneiras que permitem que os humanos o consumam, talvez usando APIs para obter esses detalhes sobre registros humanos (um tópico que veremos em breve) podem ser vitais.

Meshes de serviços e gateways de API

Poucas áreas de tecnologia associadas a microserviços tiveram tanto atenção, entusiasmo e confusão em torno deles, como a de malhas de serviços e API gateways. Ambos têm seu lugar, mas, confusamente, eles também podem se sobrepor em responsabilidades. O gateway da API, em particular, está sujeito ao uso indevido (e indevidamente vendendo), por isso é importante que entendamos como esses tipos de tecnologia pode se encaixar em nossa arquitetura de microserviços. Em vez de tentar entregar uma visão detalhada do que você pode fazer com esses produtos, em vez disso, quero fornecer uma visão geral de onde eles se encaixam, como podem ajudar e algumas armadilhas a serem evitadas.

Em termos típicos de data center, falaríamos sobre o tráfego "leste-oeste" como sendo dentro de um data center, com tráfego "norte-sul" relacionado a interações que entre ou saia do data center vindo do mundo exterior. Do ponto de vista de rede, o que é um data center se tornou um conceito um tanto confuso, então, para nossos propósitos, falaremos mais amplamente sobre um perímetro em rede. Isso pode estar relacionado a um data center inteiro, a um cluster Kubernetes ou talvez apenas um conceito de rede virtual, como um grupo de máquinas em execução na mesma LAN virtual.

De um modo geral, um gateway de API fica no perímetro do seu sistema e lida com o tráfego norte-sul. Suas principais preocupações são gerenciar o acesso de do mundo exterior aos seus microserviços internos. Uma malha de serviços, no por outro lado, lida de forma muito restrita com a comunicação entre microserviços dentro do seu perímetro - o tráfego leste-oeste, como mostra a Figura 5-6.

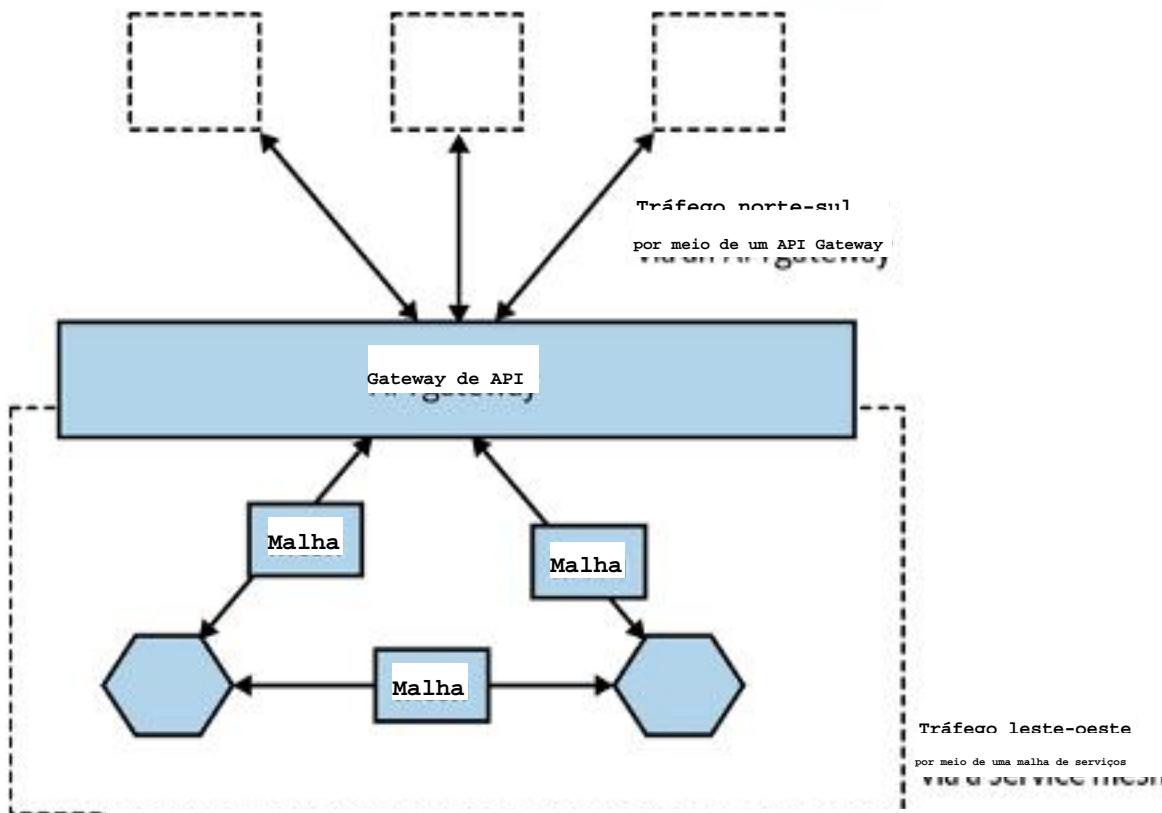


Figura 5-6 Uma visão geral de onde os gateways de API e as malhas de serviços são usados

Malhas de serviços e gateways de API podem potencialmente permitir que microserviços compartilhe código sem exigir a criação de novas bibliotecas de cliente ou novas microserviços. Simplificando, as malhas de serviços e os gateways de API podem funcionam como proxies entre microserviços. Isso pode significar que eles podem ser usados para implementar algum comportamento independente de microserviços que, de outra forma, poderia ter a ser feito em código, como descoberta de serviços ou registro.

Se você estiver usando um gateway de API ou uma malha de serviços para implementar o shared, comportamento comum para seus microserviços, é essencial que esse comportamento seja totalmente genérico - em outras palavras, que o comportamento no proxy não tem valor relação a qualquer comportamento específico de um microserviço individual.

Agora, depois de explicar isso, eu também tenho que explicar que o mundo nem sempre é tão claro quanto isso. Vários gateways de API tentam fornecer capacidades para tráfego leste-oeste também, mas isso é algo que discutiremos em breve. Primeiro, vamos analisar os gateways de API e os tipos de coisas que eles podem fazer.

Gateways de API

Sendo mais focado no tráfego norte-sul, a principal preocupação do gateway de API em um ambiente de microserviços está mapeando solicitações de partes externas para microserviços internos. Essa responsabilidade é semelhante ao que você poderia alcançar com um proxy HTTP simples e, na verdade, os gateways de API geralmente criam mais recursos além dos produtos de proxy HTTP existentes, e eles funcionam amplamente como proxies reversos. Além disso, os gateways de API podem ser usados para implementar mecanismos como chaves de API para terceiros, registro, limitação de taxa e como. Alguns produtos de gateway de API também fornecem portais para desenvolvedores, muitas vezes direcionado a consumidores externos

Parte da confusão em torno do gateway da API tem a ver com o histórico. Um tempo atrás, havia um grande interesse no que foi chamado de "API economia." O setor começou a entender o poder de oferecer APIs até soluções gerenciadas, de produtos SaaS, como Salesforce, a plataformas como AWS, pois ficou claro que uma API oferecia aos clientes muito mais flexibilidade em como seu software foi usado. Isso fez com que várias pessoas começassem a analisar o software que eles já tinham e considerar os benefícios do expondo essa funcionalidade a seus clientes não apenas por meio de uma GUI, mas também por meio de uma API. A esperança era que isso abrisse maiores oportunidades de mercado, e, bem, ganhe mais dinheiro. Em meio a esse interesse, uma safra de gateway de API produtos surgiram para ajudar a tornar possível alcançar esses objetivos. Deles o conjunto de recursos se baseou fortemente no gerenciamento de chaves de API para terceiros, aplicando limites de tarifas e rastreamento de uso para fins de estorno. A realidade é que embora as APIs tenham se mostrado absolutamente uma excelente maneira de oferecer serviços para alguns clientes, o tamanho da economia de API não era tão grande como alguns esperavam, e muitas empresas descobriram que haviam comprado a API produtos de gateway repletos de recursos que eles realmente nunca precisaram.

Na maioria das vezes, tudo o que um gateway de API está realmente sendo usado é para gerenciar acesso aos microserviços de uma organização a partir de seus próprios clientes de GUI (web páginas, aplicativos móveis nativos) via internet pública. Não existe um "terceiro" festa" na mistura aqui. A necessidade de alguma forma de gateway de API para o Kubernetes é essencial, pois o Kubernetes lida de forma nativa apenas com redes dentro do cluster e não faz nada para lidar com a comunicação com e do próprio cluster. Mas, nesse caso de uso, um gateway de API projetado para o acesso externo de terceiros é um grande exagero.

Então, se você quiser um gateway de API, seja bem claro sobre o que você espera dele. Em Na verdade, eu iria um pouco mais longe e diria que você provavelmente deveria evitar ter um Gateway de API que faz demais. Mas vamos falar sobre isso a seguir.

Onde usá-los

Quando você começa a entender que tipo de casos de uso você tem, ele se torna um É um pouco mais fácil ver que tipo de gateway você precisa. Se for apenas um caso de exposição de microserviços em execução no Kubernetes, você pode executar seu próprio reverso proxies - ou melhor ainda, você pode procurar um produto focado como o Ambassador, que foi construído do zero com esse caso de uso em mente. Se você realmente você precisa gerenciar um grande número de usuários terceirizados que acessam sua API, então provavelmente há outros produtos para analisar. É possível, de fato, que você pode acabar com mais de um gateway na mistura para melhor lidar com a separação de preocupações, e eu posso ver que, sendo sensato em muitas situações, apesar das advertências usuais sobre o aumento da complexidade geral do sistema, e o aumento dos saltos de rede ainda se aplica.

De tempos em tempos, estive envolvido no trabalho direto com fornecedores para ajuda na seleção de ferramentas. Posso dizer sem hesitar que tenho experimentou mais vendas incorretas e um comportamento ruim ou cruel na API espaço de gateway do que em qualquer outro - e, como resultado, você não encontrará referências a alguns produtos de fornecedores neste capítulo. Eu atribuí muito disso ao VC-empresas apoiadas que criaram um produto para os tempos de boom da API economia, apenas para descobrir que o mercado não existe, e então eles estão lutando duas frentes: eles estão lutando pelo pequeno número de usuários que realmente precisam o que os gateways mais complexos estão oferecendo, ao mesmo tempo em que perdem negócios para

produtos de gateway de API mais focados, criados para a grande maioria das necessidades mais simples.

O que evitar

Em parte devido ao aparente desespero de alguns fornecedores de gateway de API, todos vários tipos de alegações foram feitas sobre o que esses produtos podem fazer. Isso levou ao uso indevido desses produtos e, por sua vez, a uma desconfiança infeliz de o que é fundamentalmente um conceito bastante simples. Dois exemplos importantes que eu vi de o uso indevido de gateways de API é para agregação de chamadas e reescrita de protocolos, mas Também vi um esforço maior para usar gateways de API para dentro do perímetro (leste-oeste) liga também.

Neste capítulo, já examinamos brevemente a utilidade de um protocolo como o GraphQL para nos ajudar em uma situação na qual precisamos fazer uma série de liga e depois agrupa e filtra os resultados, mas as pessoas geralmente ficam tentadas a resolva esse problema também nas camadas do gateway de API. Tudo começa inocentemente: você combina algumas chamadas e retorna uma única carga. Então você comece fazendo outra chamada downstream como parte do mesmo fluxo agregado. Então você comece a querer adicionar lógica condicional e, em pouco tempo, percebe que você transformou os principais processos de negócios em uma ferramenta de terceiros que não é adequada para a tarefa.

Se você precisar fazer agregação e filtragem de chamadas, veja o potencial do GraphQL ou do padrão BFF, que abordaremos em Capítulo 14. Se a agregação de chamadas que você está realizando for fundamentalmente um processo de negócios, então isso é melhor feito por meio de uma saga explicitamente modelada, que abordaremos no Capítulo 6.

Além do ângulo de agregação, a reescrita do protocolo também é frequentemente usada como algo para o qual os gateways de API devem ser usados. Lembro-me de um fornecedor não identificado promovendo de forma muito agressiva a ideia de que seu produto poderia "mudar qualquer API SOAP em uma API REST." Em primeiro lugar, REST é uma arquitetura completa mentalidade que não pode ser simplesmente implementada em uma camada proxy. Em segundo lugar, a reescrita do protocolo, que é fundamentalmente o que isso está tentando fazer, não deveria deve ser feito em camadas intermediárias, pois está impondo muito comportamento ao lugar errado.

O principal problema com a capacidade de reescrita do protocolo e com o a implementação da agregação de chamadas dentro dos gateways de API é que somos violando a regra de manter os canos mudos e os terminais inteligentes. O A "inteligência" em nosso sistema quer viver em nosso código, onde podemos ter tudo controle sobre eles. O gateway da API neste exemplo é um tubo - nós o queremos como o mais simples possível. Com os microsserviços, estamos buscando um modelo no qual mudanças podem ser feitas e divulgadas mais facilmente por meio de meios independentes capacidade de implantação. Manter a inteligência em nossos microsserviços ajuda nisso. Se nós agora também tem que fazer alterações nas camadas intermediárias, as coisas se tornam mais problemático. Dada a importância dos gateways de API, as alterações neles geralmente ocorrem rigidamente controlado. Parece improvável que equipes individuais recebam gratuitamente As rédeas do autoatendimento mudam esses serviços geralmente gerenciados centralmente. O que faz isso significa? Ingressos. Para implementar uma mudança em seu software, você acaba tendo a equipe do API gateway faz alterações para você. Quanto mais comportamento você vaza em gateways de API (ou em barramentos de serviços corporativos), quanto mais você executa o risco de entregas, maior coordenação e entrega lenta.

O último problema é o uso de um gateway de API como intermediário para todas as relações chamadas de microsserviços. Isso pode ser extremamente problemático. Se inserirmos uma API gateway ou um proxy de rede normal entre dois microsserviços, então nós normalmente adicionaram pelo menos um único salto de rede. Uma chamada do microsserviço A para o microsserviço B vai primeiro de A para o gateway da API e depois do Gateway de API para B. Temos que considerar o impacto da latência do adicional chamada de rede e a sobrecarga de tudo o que o proxy está fazendo. Serviço as malhas, que exploraremos a seguir, estão muito melhor posicionadas para resolver isso problema.

Malhas de serviço

Com uma malha de serviços, funcionalidade comum associada à inter a comunicação de microsserviços é inserida na malha. Isso reduz o funcionalidade que um microsserviço precisa implementar internamente, além de fornecendo consistência em como certas coisas são feitas.

Os recursos comuns implementados pelas malhas de serviços incluem TLS mútuo, IDS de correlação, descoberta de serviços e balanceamento de carga e muito mais. Muitas vezes isso

o tipo de funcionalidade é bastante genérico de um microsserviço para o outro, então acabaríamos usando uma biblioteca compartilhada para lidar com isso. Mas então você tem que lidar com o que acontece se diferentes microsserviços tiverem versões diferentes do as bibliotecas em execução ou o que acontece se você tiver microsserviços escritos em tempos de execução diferentes.

Historicamente, pelo menos, a Netflix exigiria que todas as redes não locais a comunicação tinha que ser feita de JVM para JVM. Isso foi para garantir que o tentado e testou bibliotecas comuns que são uma parte vital do gerenciamento eficaz a comunicação entre microsserviços poderia ser reutilizada. Com o uso de um service mesh, no entanto, temos a possibilidade de reutilizar uma interface comum funcionalidade de microsserviços em microsserviços escritos em diferentes linguagens de programação. As malhas de serviço também podem ser incrivelmente úteis em implementando o comportamento padrão em microsserviços criados por diferentes equipes e o uso de uma malha de serviços, especialmente no Kubernetes, tem tornar-se cada vez mais uma parte presumida de qualquer plataforma que você possa criar para implantação e gerenciamento de microsserviços por autoatendimento.

Facilitar a implementação de comportamentos comuns em microsserviços é uma dos grandes benefícios de uma malha de serviços. Se essa funcionalidade comum fosse implementado exclusivamente por meio de bibliotecas compartilhadas, alterar esse comportamento seria exigem que cada microsserviço obtenha uma nova versão dessas bibliotecas e seja implantado antes que a mudança esteja ativa. Com uma malha de serviços, você tem muito mais flexibilidade na implementação de mudanças em termos de intermicrosserviços comunicação sem exigir uma reconstrução e reimplantação.

Como eles funcionam

Em geral, esperaríamos ter menos tráfego norte-sul do que tráfego leste-oeste com uma arquitetura de microsserviços Uma única chamada norte-sul colocando um a ordem, por exemplo, pode resultar em várias chamadas leste-oeste. Isso significa que ao considerar qualquer tipo de proxy para chamadas dentro do perímetro, temos que ser ciente da sobrecarga que essas chamadas adicionais podem causar, e isso é essencial consideração em termos de como as malhas de serviços são construídas.

As malhas de serviço vêm em diferentes formas e tamanhos, mas o que as une é que sua arquitetura se baseia na tentativa de limitar o impacto causado pelas chamadas para

e do proxy. Isso é obtido principalmente por meio da distribuição do proxy processos a serem executados nas mesmas máquinas físicas do microsserviço instâncias, para garantir que o número de chamadas de rede remotas seja limitado. Em Figura 5-7, vemos isso em ação: o Processador de Pedidos está enviando uma solicitação para o microsserviço de pagamento. Essa chamada é roteada primeiro localmente para um proxy. instância em execução na mesma máquina que o Order Processor, antes continuando com o microsserviço de pagamento por meio de sua instância proxy local. O Order Processor acha que está fazendo uma chamada de rede normal, sem saber que a chamada é roteada localmente na máquina, o que é significativamente mais rápido (e também menos propenso a partícões).

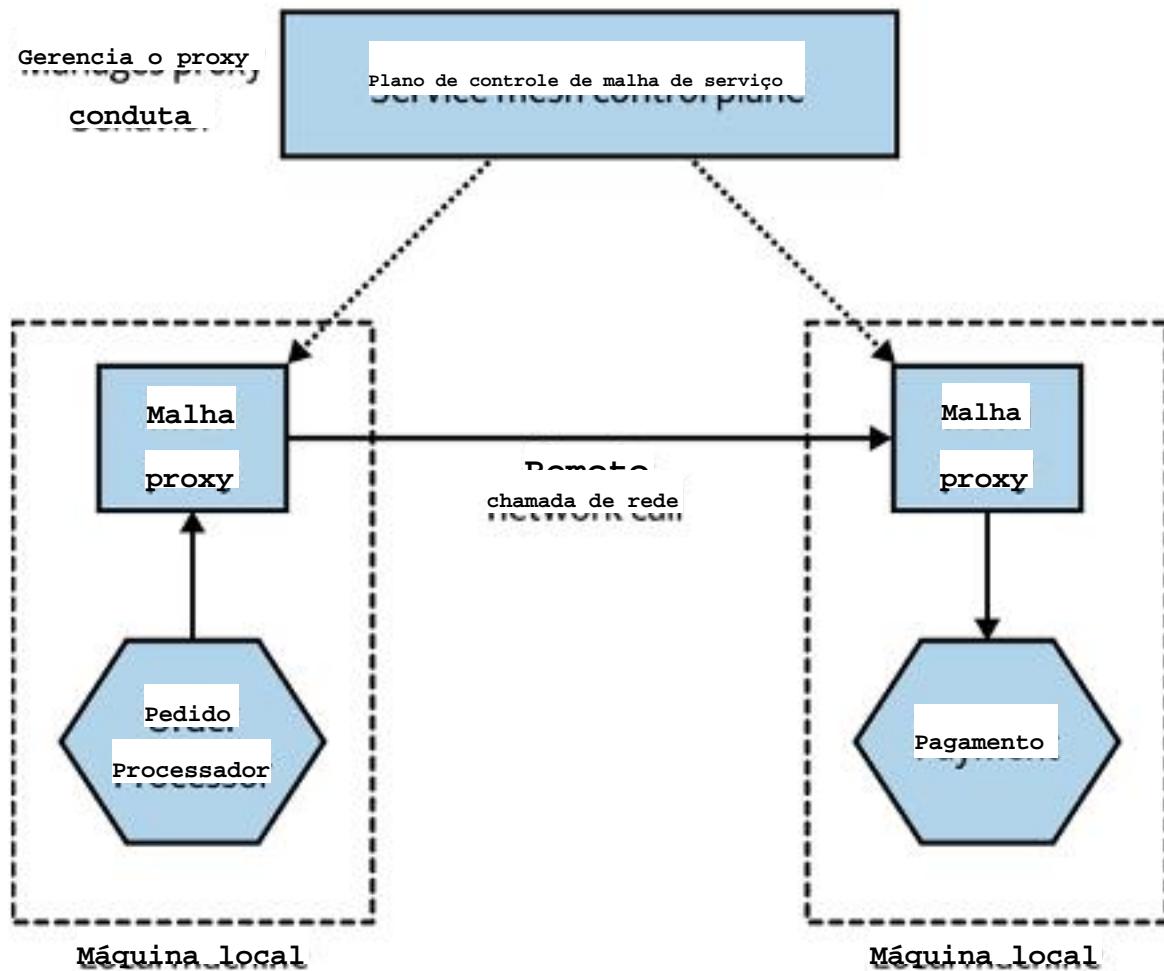


Figura 5-7. Uma malha de serviços é implantada para lidar com toda a comunicação direta entre microsserviços

Um plano de controle ficaria em cima dos proxies de malha locais, atuando como um local em que o comportamento desses proxies pode ser alterado e um local em

onde você pode coletar informações sobre o que os proxies estão fazendo.

Ao implantar no Kubernetes, você implantaria cada microsserviço instância em um pod com seu próprio proxy local. Um único pod é sempre implantado como uma única unidade, para que você sempre saiba que tem um proxy disponível. Além disso, a morte de um único proxy afetaria apenas aquele pod. Esta configuração também permite que você configure cada proxy de forma diferente para finalidades diferentes. Examinaremos esses conceitos com mais detalhes em "Kubernetes and Container".

Orquestração.

Muitas implementações de service mesh usam o proxy Envoy como base de esses processos executados localmente. O Envoy é um proxy C++ leve, frequentemente usado como alicerce para service meshes e outros tipos de proxy. software-it é um componente importante para o Istio e o Ambassador, para exemplo.

Esses proxies, por sua vez, são gerenciados por um plano de controle. Este será um conjunto de software que ajuda você a ver o que está acontecendo e controlar o que está sendo feito. Ao usar uma malha de serviços para implementar o TLS mútuo, por exemplo, o plano de controle seria usado para distribuir certificados de cliente e servidor.

As malhas de serviço não são tubos inteligentes?

Então, toda essa conversa sobre inserir um comportamento comum em uma malha de serviços pode ter campainhas de alarme tocando para alguns de vocês. Essa abordagem não está aberta ao mesmo? tipos de problemas, como barramentos de serviços corporativos ou API excessivamente inchada portais? Não corremos o risco de colocar muita "inteligência" em nosso serviço? malha?

A principal coisa a lembrar aqui é que o comportamento comum que estamos adotando into the mesh não é específico para nenhum microsserviço. Sem negócios a funcionalidade vazou para o exterior. Estamos configurando coisas genéricas como como os tempos limite das solicitações são tratados. Em termos de comportamento comum, isso pode querem ser ajustados por microsserviço, isso normalmente é algo que é bem atendido, sem a necessidade de trabalho a ser feito em uma central plataforma. Por exemplo, com o Istio, posso definir meus requisitos de tempo limite em um base de autoatendimento apenas alterando minha definição de serviço.

Você precisa de um?

Quando o uso de service meshes começou a se tornar popular, logo após o lançamento da primeira edição deste livro, vi muito mérito na ideia, mas também vi muita agitação no espaço. Diferentes modelos de implantação foram sugerido, construído e depois descartado, e o número de empresas que oferecem as soluções nesse espaço aumentaram drasticamente; mas mesmo para aquelas ferramentas que tinham já existido há muito tempo, havia uma aparente falta de estabilidade. Vinculada, que, sem dúvida, fez tanto quanto qualquer um para ser pioneira neste espaço, totalmente reconstruído seu produto do zero na mudança de v1 para v2. Istio, que foi o O service mesh, criado pelo Google, levou anos para chegar a uma versão 1.0 inicial, e mesmo assim, teve mudanças subsequentes significativas em sua arquitetura (em movimento um tanto ironicamente, embora sensatamente, para uma implantação mais monolítica modelo para seu plano de controle).

Durante grande parte dos últimos cinco anos, quando me perguntaram "Devemos obter uma malha de serviços?" meu conselho foi "Se você puder se dar ao luxo de esperar seis meses antes de fazer uma escolha, espere seis meses." Eu estava convencido da ideia, mas preocupado com a estabilidade. E algo como uma malha de serviços não é onde eu pessoalmente quero correr muitos riscos - é tão importante, tão essencial tudo funcionando bem. Você está colocando isso em seu caminho crítico. Está lá em cima com a seleção de um corretor de mensagens ou provedor de nuvem em termos de seriedade. Eu aceitaria.

Desde então, fico feliz em dizer que este espaço amadureceu. A agitação para alguns a extensão diminuiu, mas ainda temos uma pluralidade (saudável) de fornecedores. Isso disse que as malhas de serviço não são para todos. Em primeiro lugar, se você não estiver no Kubernetes, suas opções são limitadas. Em segundo lugar, eles adicionam complexidade. Se você tiver cinco microserviços, não acho que você possa justificar facilmente uma malha de serviços (é discutível se você pode justificar o Kubernetes se tiver apenas cinco microserviços!). Para organizações que têm mais microserviços, especialmente se eles quiserem que a opção desses microserviços sejam escritos de forma diferente linguagens de programação e malhas de serviços valem bem a pena dar uma olhada. Faça o seu dever de casa, mas alternar entre malhas de serviço é doloroso!

Monzo é uma organização que falou abertamente sobre como usar um o service mesh foi essencial para permitir que ele executasse sua arquitetura na escala em que

faz. Seu uso da versão 1 do Linkerd para ajudar a gerenciar RPC entre microsserviços as chamadas se mostraram extremamente benéficas. Curiosamente, Monzo teve que lidar com a dor de uma migração de service mesh para ajudá-la a alcançar a escala necessária quando o A arquitetura mais antiga do Linkerd v1 não atendia mais aos seus requisitos. No final das contas, mudou-se efetivamente para uma malha de serviços interna usando o Envoy proxy.

E quanto aos outros protocolos?

Os gateways de API e as malhas de serviços são usados principalmente para lidar com questões relacionadas a HTTP chamadas. Portanto, REST, SOAP, gRPC e similares podem ser gerenciados por meio desses produtos. As coisas ficam um pouco mais obscuras, porém, quando você comece a olhar comunicação por meio de outros protocolos, como o uso de corredores de mensagens, como Kafka. Normalmente, nesse ponto, a malha de serviços é contornada. A comunicação é feita diretamente com o próprio corretor. Isso significa que você não pode presumir que sua malha de serviços seja capaz de funcionar como intermediária para todos chamadas entre microsserviços.

Serviços de documentação

Ao decompor nossos sistemas em microsserviços mais refinados, esperamos para expor muitas emendas na forma de APIs que as pessoas podem usar para fazer muitas espero que sejam coisas maravilhosas. Se você acertar nossa descoberta, sabemos onde as coisas são. Mas como sabemos o que essas coisas fazem ou como usá-las? Obviamente, uma opção é ter documentação sobre as APIs. Claro, a documentação geralmente pode estar desatualizada. Idealmente, garantiríamos que nosso a documentação está sempre atualizada com a API de microsserviços e torne-a É fácil ver essa documentação quando sabemos onde está um endpoint de serviço.

Esquemas explícitos

Ter esquemas explícitos ajuda muito a tornar mais fácil entendem o que um determinado endpoint expõe, mas por si só eles geralmente são não é suficiente. Como já discutimos, os esquemas ajudam a mostrar a estrutura,

mas eles não ajudam muito a comunicar o comportamento de um endpoint, portanto, uma boa documentação ainda pode ser necessária para ajudar os consumidores entender como usar um endpoint. É importante notar, é claro, que se você decida não usar um esquema explícito, sua documentação acabará funcionando mais trabalho. Você precisará explicar o que o endpoint faz e também documentar a estrutura e os detalhes da interface. Além disso, sem um explícito esquema, detectando se sua documentação está atualizada com o real os endpoints são mais difíceis. Documentação obsoleta é um problema contínuo, mas em pelo menos um esquema explícito oferece mais chances de estar atualizado.

Fui já apresentei o OpenAPI como um formato de esquema, mas também é muito eficaz no fornecimento de documentação e muito código aberto e

Agora existem ferramentas comerciais que podem suportar o consumo da OpenAPI

descritores para ajudar a criar portais úteis para permitir que os desenvolvedores leiam o

documentação Vale a pena notar que os portais de código aberto para visualização

O OpenAPI parece um pouco básico - eu me esforcei para encontrar um que suportasse

funcionalidade de pesquisa, por exemplo. Para quem usa Kubernetes, Ambassador's

o portal do desenvolvedor é especialmente interessante. O embaixador já é um popular

escolha como gateway de API para Kubernetes, e seu Portal do Desenvolvedor tem o

capacidade de descobrir automaticamente os endpoints OpenAPI disponíveis. A ideia de implantar

um novo microserviço e ter sua documentação disponível automaticamente

me atrai muito.

No passado, não tínhamos um bom suporte para documentar com base em eventos

interfaces. Agora, pelo menos, temos opções. O formato AsyncAPI começou como uma adaptação do OpenAPI, e agora também temos o CloudEvents, que é um

Projeto CNCF. Eu não usei nenhum deles com raiva (ou seja, em um ambiente real), mas eu sou

mais atraído pelo CloudEvents simplesmente porque parece ter uma grande variedade de

integração e suporte, devido em grande parte à sua associação com o CNCF.

Historicamente, pelo menos, o CloudEvents parecia ser mais restritivo em termos de

o formato do evento comparado ao AsyncAPI, com apenas o JSON sendo adequado

suportado, até que o suporte ao buffer de protocolo foi reintroduzido recentemente após

sendo removido anteriormente; então isso pode ser uma consideração.

O sistema de autodescrição

Durante a evolução inicial da SOA, padrões como a Descrição Universal, A descoberta e a integração (UDDI) surgiram para nos ajudar a entender o que os serviços estavam funcionando. Essas abordagens eram bastante pesadas, o que levou a técnicas alternativas para tentar entender nossos sistemas. Martin Fowler discutiu o conceito de registro humano, muito mais abordagem leve, na qual os humanos podem registrar informações sobre os serviços na organização em algo tão básico quanto um wiki.

Ter uma ideia do nosso sistema e de como ele está se comportando é importante, especialmente quando estamos em grande escala. Cobrimos uma série de diferentes técnicas que nos ajudarão a obter entendimento diretamente de nosso sistema. Por rastreando a integridade de nossos serviços downstream junto com IDs de correlação para nos ajudar a ver as cadeias de chamadas, podemos obter dados reais sobre como nossos serviços inter-relacionar. Usando sistemas de descoberta de serviços como o Consul, podemos ver onde nossos microserviços estão em execução. Mecanismos como OpenAPI e CloudEvents pode nos ajudar a ver quais recursos estão sendo hospedados em qualquer endpoint, enquanto nossas páginas de verificação de saúde e sistemas de monitoramento nos informam sobre a saúde do sistema geral e dos serviços individuais.

Todas essas informações estão disponíveis de forma programática. Todos esses dados permitem nós para tornar nosso registro humano mais poderoso do que uma simples página wiki que sem dúvida, ficará desatualizado. Em vez disso, devemos usá-lo para aproveitar e exibir todas as informações que nosso sistema emitirá. Ao criar uma personalização painéis, podemos reunir a vasta gama de informações que são disponível para nos ajudar a entender nosso ecossistema

De qualquer forma, comece com algo tão simples quanto uma página da web estática ou wiki que talvez extraia alguns dados do sistema ativo. Mas tente atrair mais e mais informações ao longo do tempo. Tornar essas informações prontamente disponíveis é uma ferramenta fundamental para gerenciar a complexidade emergente que surgirá da execução desses sistemas em grande escala.

Falei com várias empresas que tiveram esses problemas e que acabou criando registros internos simples para ajudar a agrupar metadados serviços. Alguns desses registros simplesmente rastreiam repositórios de código-fonte, procurando arquivos de metadados para criar uma lista de serviços disponíveis. Isso

as informações podem ser mescladas com dados reais provenientes da descoberta de serviços. Sistemas como Consul ou etcd para criar uma imagem mais rica do que está sendo executado e com quem você poderia falar sobre isso.

O Financial Times criou o Biz Ops para ajudar a resolver esse problema. A empresa tem várias centenas de serviços desenvolvidos por equipes em todo o mundo. A ferramenta Biz Ops (Figura 5-8) oferece à empresa um único lugar onde você pode encontrar muitas informações úteis sobre seus microserviços, em além de informações sobre outros serviços de infraestrutura de TI, como redes e servidores de arquivos. Construído sobre um banco de dados gráfico, o Biz Ops tem muitos de flexibilidade sobre quais dados ele coleta e como as informações podem ser modelado.

The screenshot shows the 'PES' (Administrador de operações de negócios) interface. The top navigation bar includes links for 'SOBRE', 'PEQUISAR', 'MINHAS COISAS', 'ADICIONE ALGO', and 'RELATÓRIO (BETA)'. The main content area displays a service named 'Sistema: Página de artigo do FT.com'. The interface is divided into several sections:

- Informações gerais:** Includes fields for 'Nome' (FT.com), 'Descrição' (.serviço que fornece a camada oposta de artigos para FT.com), 'URL principal' (<https://www.ft.com/conten...>), and 'Plataforma' (Production).
- Propriedade e conhecimento:** Shows 'Entregue pela equipe' (Next) and 'Apoiado pela equipe' (Prévio).
- Visão geral técnica:** Displays 'Arquitetura@' with the note: 'next_arti Le_serve as seguintes rotas (*Content/C) > ((0-9)+)> ((0-9)+)> ((0-9)+)> ((0-9)+)> ((0-9)+)'.

On the left side, there is a sidebar with various service management categories: Informações gerais, Conhecimento da propriedade, Visão geral da Techini, Governança de dados, Recursos relacionados, Fallover, Recuperação de desastres, Lançamento, Gerenciamento de chaves, Monitoramento, Solução de problemas, Informações sobre o mar, Análise da operabilidade do serviço, and Diversos.

Figura 5-8. A ferramenta Biz Ops do Financial Times, que reúne informações sobre seus microserviços

A ferramenta Biz Ops vai além da maioria das ferramentas similares que eu já vi, no entanto. A ferramenta calcula o que chama de Pontuação de Operabilidade do Sistema, como mostrado na Figura 5-9. A ideia é que existem certas coisas que atendem e suas equipes devem fazer isso para garantir que os serviços possam ser operados facilmente. Isso pode variar desde garantir que as equipes tenham fornecido as informações corretas no registro para garantir que os serviços tenham verificações de saúde adequadas. O sistema A pontuação de operabilidade, uma vez calculada, permite que as equipes vejam rapidamente se há são coisas que precisam ser consertadas.

Este é um espaço em crescimento. No mundo do código aberto, a ferramenta Backstage do Spotify oferece um mecanismo para criar um catálogo de serviços como o Biz Ops, com um plug-in no modelo para permitir adições sofisticadas, como a capacidade de acionar a criação de um novo microserviço ou a obtenção de informações em tempo real de um Cluster Kubernetes. O catálogo de serviços do Ambassador é mais restrito, focado na visibilidade dos serviços no Kubernetes, o que significa que talvez não têm tanto apelo geral quanto algo como o Biz Ops do FT, mas é. No entanto, é bom ver algumas novas abordagens sobre essa ideia, que são mais gerais.

disponível.

The screenshot shows a web-based application for managing service operability. At the top, a yellow banner displays the message: "Este projeto é atualmente - o alimento e - estoque". Below this, the header includes the acronym PES (Pontuação de operabilidade do sistema) and a "Presente" button. The main navigation bar has tabs for "TODOS OS GRUPOS", "TABELA DA LIGA DE EQUIPES", "TABELA DA LIGA DO SISTEMA", and "SOBRE". The current view is for the group "TODOS OS GRUPOS: ENTERPRISE SERVICES CLOUD NA PLATEFORMA FTPLATFORM2".

Sistema: ftplatform2

Pontuação: 92%

Status da revisão manual: Pronto para revisão adequada

Pontuação atualizada há: 63 minutos

A right-hand sidebar contains a note in Portuguese: "Lembre-se de que o assustador - trabalho em andamento. Alguns possos (particularmente com relação a esta nota de festa) elas não eram sempre fazem o sentido. Ria sua sem críticas. Não dê 100% normal para o novo Pixino M. as coisas que você guarda como corrigir a principal coisa pela qual devem ser feitas".

The main content area lists various operational aspects and their status:

Aspecto do runbook	Status	Falhas	Ações corretivas
dependentes	Erro	Algumas dependentes têm um nível de serviço acima do Silver	
verificações de saúde	Aviso	healthchecked tem Healthcheck sem URL	
mais informações		moreinformation contém HTML, prefira usar markdown mensinformation contém links mensinformation contém referências à documentação não platinada (confluence google sites)	
monitoramento		monitoramento contém HTML, prefira usar markdown monitoramento contém logo	
Próxima linha de bate-papo no público	em	pôs em Solução de problemas de linha de caixa	
Solução de problemas de segunda linha		A segunda linha A planilha de problemas contém links	

Figura 5-9. Um exemplo do Service Operability Score para um microserviço no Financial Times

Resumo

Então, abordamos muitos tópicos neste capítulo - vamos detalhar alguns deles para baixo:

- Para começar, certifique-se de que o problema que você está tentando resolver seja guia sua escolha de tecnologia. Com base no seu contexto e na sua preferência estilo de comunicação, selecione a tecnologia mais apropriada para você, não caia na armadilha de escolher a tecnologia primeiro. O resumo dos estilos de comunicação entre microserviços, primeiro introduzido no Capítulo 4 e mostrado novamente na Figura 5-10, pode ajudar guie sua tomada de decisão, mas apenas seguir esse modelo não é uma substituto para sentar e pensar sobre sua própria situação.

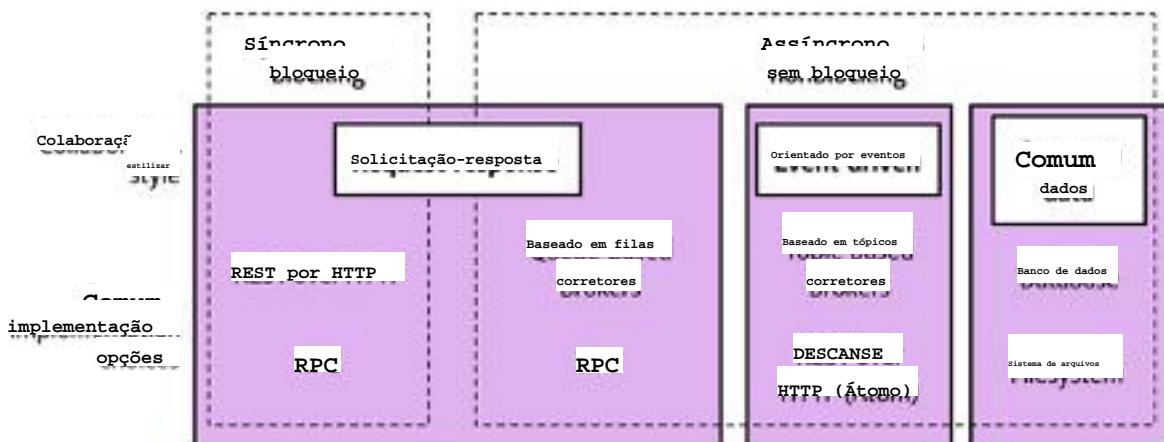


Figura 5-10. Diferentes estilos de comunicação entre microserviços, juntamente com exemplos implementando tecnologias

- Seja qual for a escolha que você fizer, considere o uso de esquemas, em parte para ajude a tornar seus contratos mais explícitos, mas também para ajudar a capturar alterações accidentais de quebra.
- Sempre que possível, esforce-se para fazer mudanças que sejam retrógradas compatível para garantir que a implantação independente continue sendo uma possibilidade.
- Se você precisar fazer alterações incompatíveis com versões anteriores, encontre uma maneira para permitir que os consumidores tenham tempo para fazer o upgrade para evitar implantações paralelas.

- Pense no que você pode fazer para ajudar a revelar informações sobre seu endpoints para humanos - considere o uso de registros humanos e o gostaria de ajudar a entender o caos.

Analisamos como podemos implementar uma chamada entre dois microsserviços, mas o que acontece quando precisamos coordenar operações entre vários microsserviços? Esse será o foco do nosso próximo capítulo.

1 Martin Fowler explora isso com mais detalhes no contexto do armazenamento de dados sem esquema.

2 Observe que, na verdade, existem três ferramentas diferentes neste espaço com o mesmo nome! O openapi-
A ferramenta diff em <https://github.com/Azure/openapi-diff> parece estar mais próxima de uma ferramenta que, na verdade
passa ou falha na compatibilidade.

Capítulo 6. fluxo de trabalho

Nos dois capítulos anteriores, analisamos os aspectos dos microserviços relacionados à forma como um microserviço fala com outro. Mas o que acontece quando nós queremos que vários microserviços colaborem, talvez para implementar um negócio? Como modelamos e implementamos esses tipos de fluxos de trabalho nos sistemas distribuídos podem ser difíceis de acertar.

Neste capítulo, veremos as armadilhas associadas ao uso distribuído de transações para resolver esse problema, e também analisaremos o conceito de saga que podem nos ajudar a modelar nossos fluxos de trabalho de microserviços em muito mais maneira satisfatória.

Transações de banco de dados

Genericamente falando, quando pensamos em uma transação no contexto de computação, pensamos em uma ou mais ações que ocorrerão e queremos tratar como uma única unidade. Ao fazer várias alterações como parte da mesma operação geral, queremos confirmar se todas as alterações têm sido feitas. Também queremos uma maneira de nos limpar se ocorrer um erro, enquanto essas mudanças estão acontecendo. Normalmente, isso resulta em usarmos algo como uma transação de banco de dados.

Com um banco de dados, usamos uma transação para garantir que um ou mais estados das alterações foram feitas com sucesso. Isso pode incluir dados sendo removido, inserido ou alterado. Em um banco de dados relacional, isso pode envolver várias tabelas sendo atualizadas em uma única transação.

Transações ACID

Normalmente, quando falamos sobre transações de banco de dados, estamos falando sobre transações ACID. ACID é um acrônimo que descreve as principais propriedades de transações de banco de dados que conduzem a um sistema em que podemos confiar para garantir a

durabilidade e consistência do nosso armazenamento de dados. ACID significa atomicidade, consistência, isolamento e durabilidade, e aqui está o que essas propriedades oferecem
nós:

Atomicidade

Garante que todas as operações tentadas dentro da transação completo ou tudo falhará. Se alguma das mudanças que estamos tentando fazer falhar alguma razão, então toda a operação é abortada, e é como se não mudanças sempre foram feitas.

Consistência

Quando alterações são feitas em nosso banco de dados, garantimos que ele seja deixado em um formato válido, estado consistente.

Isolamento

Permite que várias transações operem ao mesmo tempo sem interferindo. Isso é conseguido garantindo que qualquer estado provisório mude feitas durante uma transação são invisíveis para outras transações.

Durabilidade

Garante que, uma vez concluída a transação, estejamos confiantes os dados não serão perdidos no caso de alguma falha no sistema.

É importante notar que nem todos os bancos de dados fornecem transações ACID. Todos sistemas de banco de dados relacional que eu já usei fazem, assim como muitos dos mais novos Bancos de dados NoSQL como o Neo4j. O MongoDB por muitos anos suportou o ACID transações somente em alterações feitas em um único documento, o que poderia causar problemas se você quiser fazer uma atualização atômica para mais de um documento.¹

Este não é o livro para uma exploração detalhada desses conceitos; eu tenho certamente simplificado algumas dessas descrições por uma questão de brevidade. Para artigos que costariam de explorar mais esses conceitos, eu recomendo Projetando aplicativos com uso intensivo de dados. 2 Nós nos preocuparemos principalmente

com atomicidade no que segue. Isso não quer dizer que as outras propriedades também não são importantes, mas estão tentando descobrir como lidar com a atomicidade de operações de banco de dados tendem a ser o primeiro problema que encontramos quando começamos a quebrar separar a funcionalidade em microserviços.

Ainda ácido, mas sem atomicidade?

Quero deixar claro que ainda podemos usar transações no estilo ACID ao usar microserviços. Um microserviço é gratuito para usar uma transação ACID para operações em seu próprio banco de dados, por exemplo. É só que o escopo dessas transações são reduzidas à mudança de estado que acontece localmente dentro desse único microserviço. Considere a Figura 6-1. Aqui, estamos acompanhando o processo envolvido na integração de um novo cliente na MusicCorp. Nós temos chegado ao final do processo, o que envolve a alteração do status de cliente 2346 de PENDENTE para VERIFICADO. Como a inscrição é agora completa, também queremos remover a linha correspondente da Tabela de inscrições pendentes. Com um único banco de dados, isso é feito no escopo de uma única transação de banco de dados ACID - qualquer um desses dois estados muda ocorre, ou nenhuma delas ocorre.

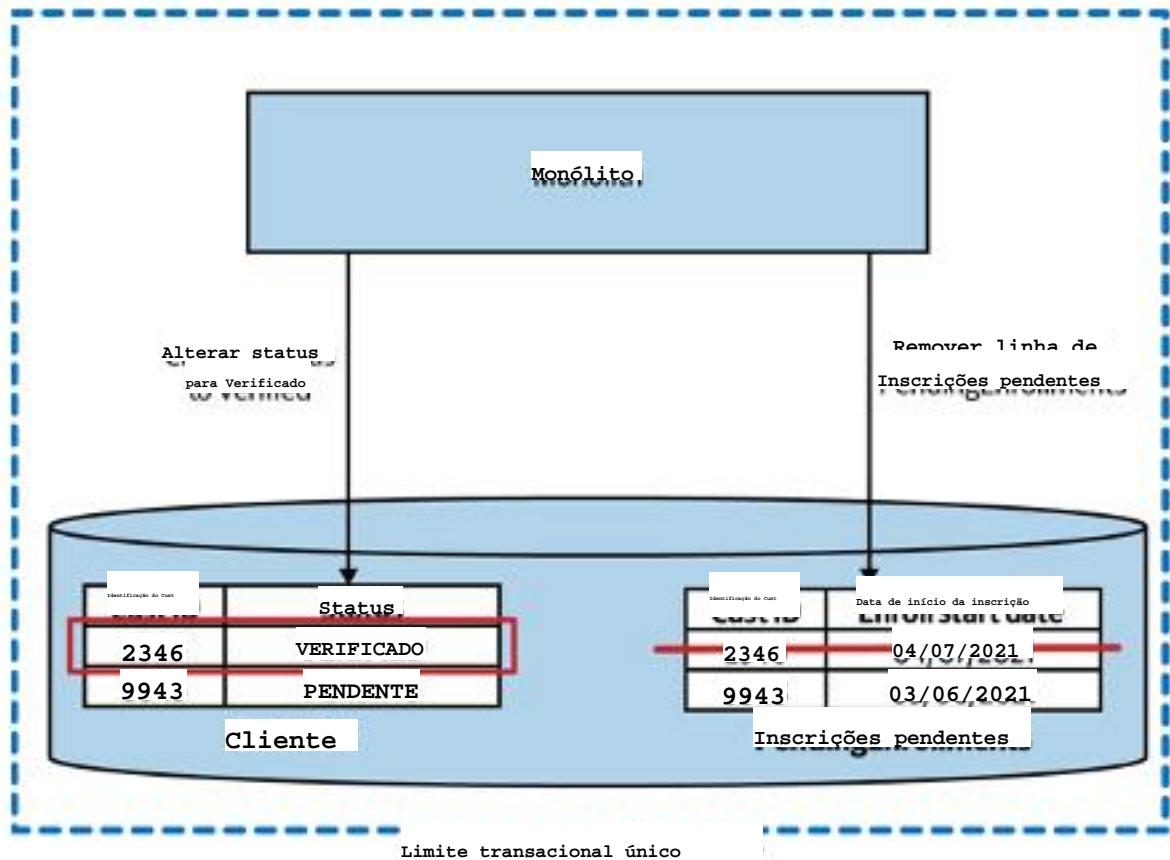


Figura 6-1. Atualizando duas tabelas no escopo de uma única transação ACID

Compare isso com a Figura 6-2, onde estamos fazendo exatamente a mesma alteração, mas cada alteração é feita em um banco de dados diferente. Isso significa que existem duas transações a serem consideradas, cada uma das quais pode funcionar ou falhar independentemente de o outro.

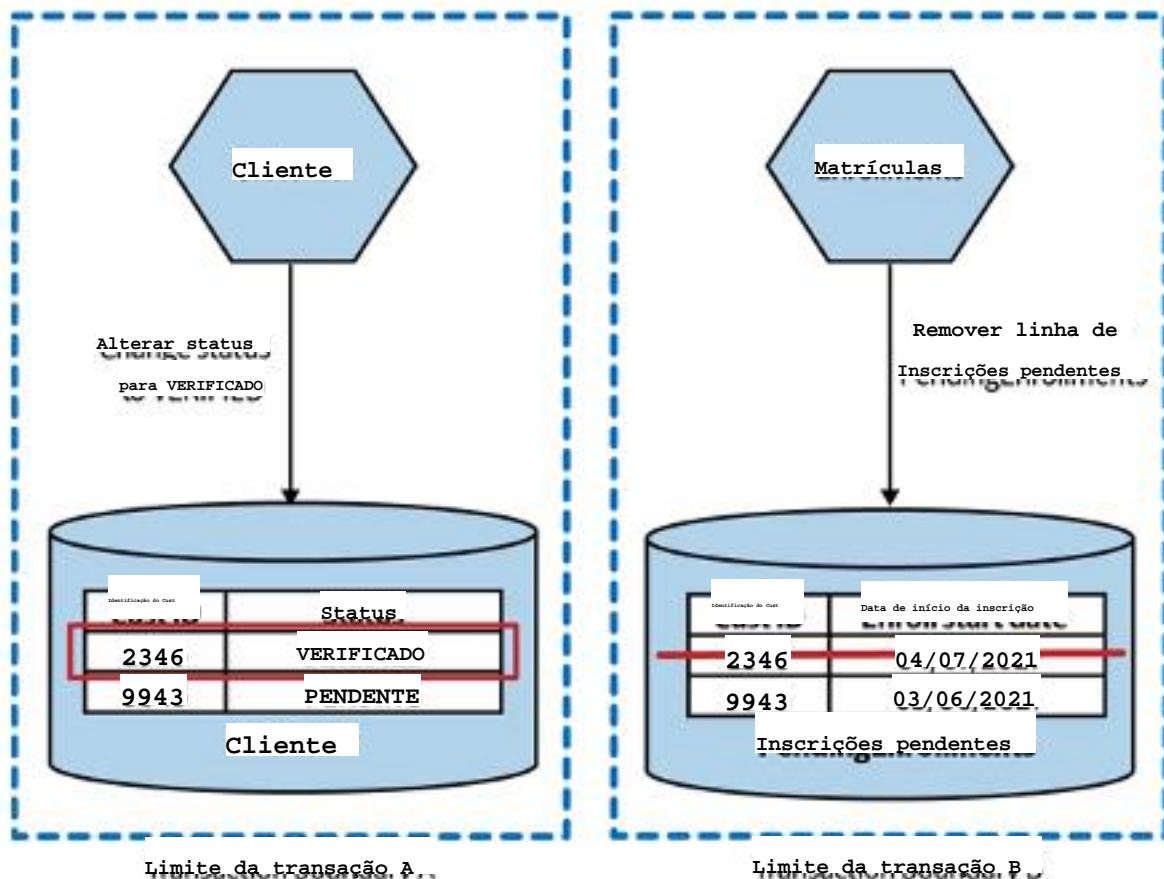


Figura 6-2. As alterações feitas pelos microserviços de Clientes e Inscrições agora são feitas em o escopo de duas transações diferentes.

Poderíamos decidir sequenciar essas duas transações, é claro, removendo uma linha da tabela PendingEnrollments somente se pudéssemos alterar a linha em a tabela de clientes. Mas ainda teríamos que raciocinar sobre o que fazer se o exclusão da tabela PendingEnrollments e, em seguida, falha na lógica de que precisaríamos nos implementar. Ser capaz de reordenar as etapas para melhor. No entanto, lidar com esses casos de uso pode ser uma ideia muito útil (uma em breve) de volta à época em que exploramos sagas). Mas, fundamentalmente, temos que aceitar isso decompondo essa operação em duas transações de banco de dados separadas, perdemos a atomicidade garantida da operação como um todo.

Essa falta de atomicidade pode começar a causar problemas significativos, especialmente se estamos migrando sistemas que anteriormente dependiam dessa propriedade. Normalmente, a primeira opção que as pessoas começam a considerar ainda é usar uma única transação, mas que agora abrange vários processos - uma transação distribuída. Infelizmente, como veremos, as transações distribuídas podem não ser o caminho certo.

para frente. Vamos dar uma olhada em um dos algoritmos mais comuns para implementação de transações distribuídas, o commit em duas fases, como forma de explorar os desafios associados às transações distribuídas como um todo.

Transações distribuídas em duas fases

Compromete-se

O algoritmo de confirmação em duas fases (às vezes reduzido para 2PC) é frequentemente usado na tentativa de nos dar a capacidade de fazer transações mudanças em um sistema distribuído, onde vários processos separados podem precisar a ser atualizado como parte da operação geral. Transações distribuídas e os commits em duas fases, mais especificamente, são frequentemente considerados pelas equipes migrando para arquiteturas de microserviços como forma de resolver desafios rosto. Mas, como veremos, eles podem não resolver seus problemas e podem trazer ainda mais confusão em seu sistema.

O 2PC é dividido em duas fases (daí o nome de confirmação em duas fases): a fase de votação e fase de confirmação. Durante a fase de votação, uma central o coordenador contata todos os trabalhadores que farão parte do transação e solicita confirmação sobre se alguma mudança de estado ou não pode ser feito. Na Figura 6-3, vemos duas solicitações: uma para mudar um cliente status para VERIFICADO e outro para remover uma linha do nosso PendingEnrol ementa uma mesa. Se todos os trabalhadores concordarem que o estado mude, eles se solicitados, podem ocorrer, o algoritmo prossegue para a próxima fase. Se houver trabalhador diz que a mudança não pode ocorrer, talvez porque o solicitado a mudança de estado viola alguma condição local, toda a operação é abortada.

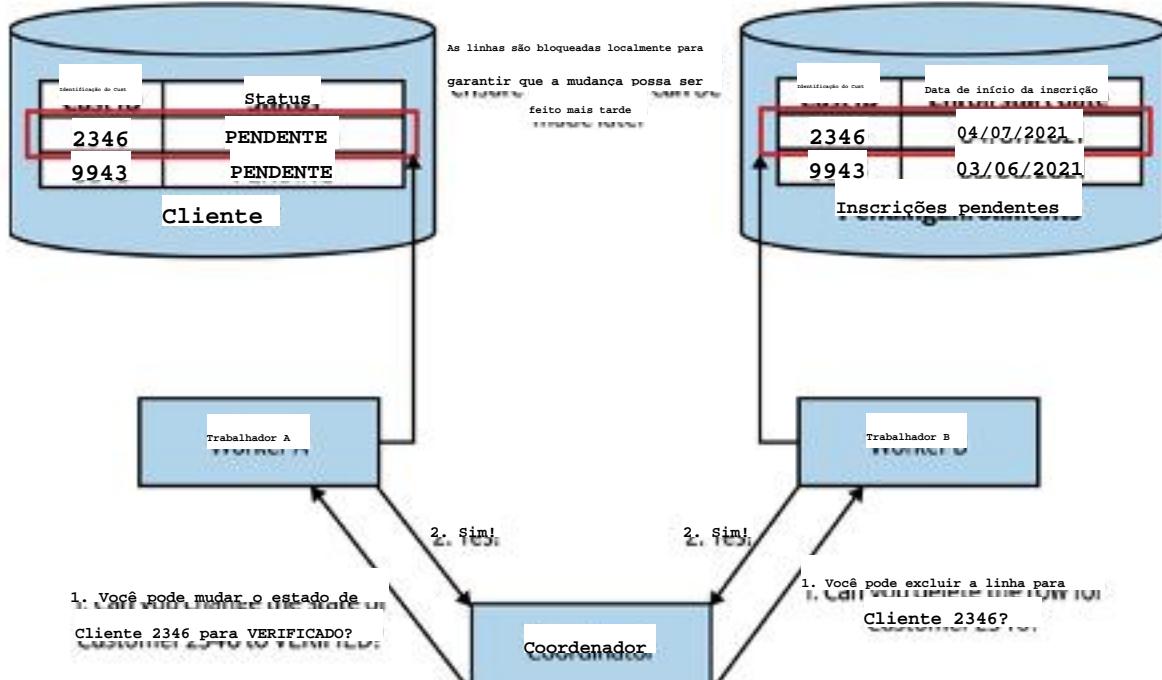


Figura 6-3. Na primeira fase de um comitê de duas fases, os trabalhadores votam para decidir se podem realizar alguma mudança de estado local.

É importante destacar que a mudança não entra em vigor imediatamente depois que um trabalhador indica que pode fazer a alteração. Em vez disso, o trabalhador é garantido que será capaz de fazer essa mudança em algum momento do futuro. Como o trabalhador daria essa garantia? Na Figura 6-3, para exemplo, o trabalhador A disse que será capaz de alterar o estado da linha em a tabela Clientes para atualizar o status desse cliente específico para VERIFICADO. E se uma operação diferente, em algum momento posterior, excluir a linha ou criar alguma outra alteração menor que, no entanto, significa que uma mudança para VERIFICADO mais tarde é inválida? Para garantir que a alteração para VERIFICADO possa ser feita posteriormente, o trabalhador A provavelmente precisará bloquear o registro para garantir que outras alterações não podem acontecer.

Se algum trabalhador não votou a favor do commit, uma mensagem de reversão precisa ser enviado a todas as partes para garantir que elas possam limpar localmente, o que permite os trabalhadores devem liberar quaisquer fechaduras que possam estar segurando. Se todos os trabalhadores concordarem para fazer a alteração, passamos para a fase de confirmação, como na Figura 6-4. Aqui, as alterações são realmente feitas e os bloqueios associados são liberados.

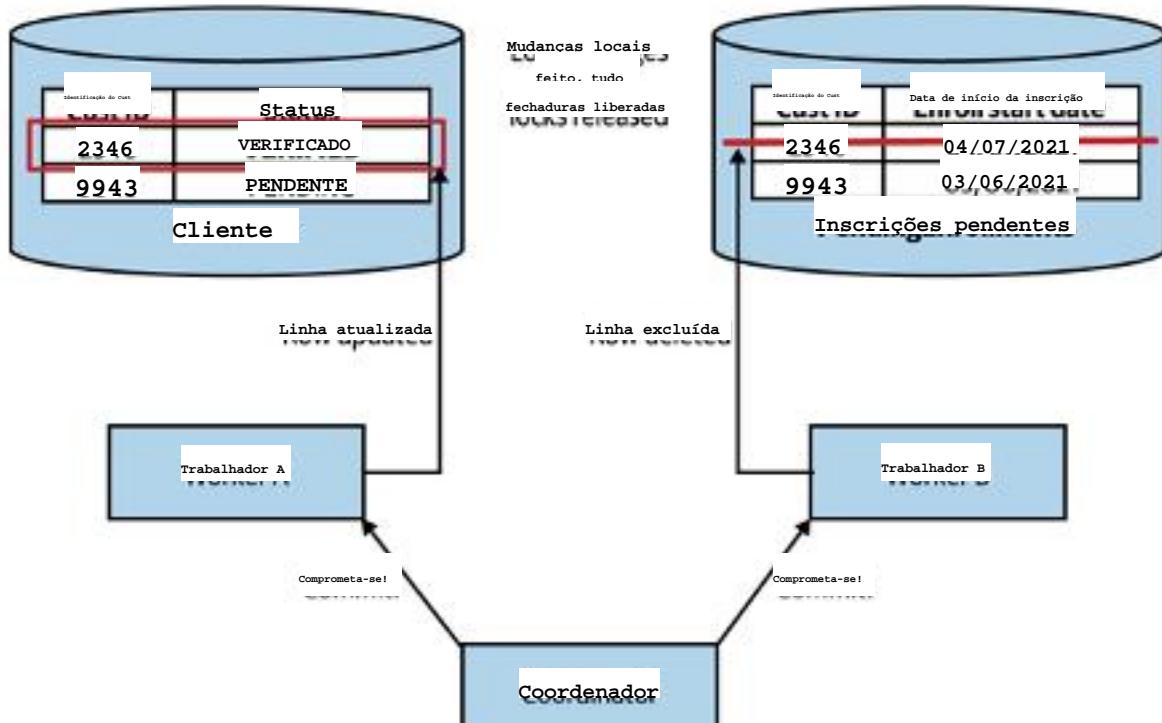


Figura 6-4. Na fase de confirmação de uma confirmação de duas fases, as mudanças são realmente aplicadas

É importante observar que, em tal sistema, não podemos, de forma alguma, garantir que esses commits ocorrerão exatamente ao mesmo tempo. O coordenador precisa enviar a solicitação de confirmação para todos os participantes, e essa mensagem pode chegar e seja processado em horários diferentes. Isso significa que é possível que poderíamos ver a mudança feita no Trabalhador A, mas ainda não no Trabalhador B, se pudéssemos observar diretamente o estado de qualquer um dos trabalhadores. Quanto mais latência houver entre o coordenador e os participantes do compromisso de duas fases, e mais lentamente os trabalhadores processam a resposta, quanto maior essa janela de inconsistência pode ser. Voltando à nossa definição de ACID, isolamento garante que não vejamos estados intermediários durante uma transação. Mas com esta confirmação em duas fases, perdemos essa garantia.

Quando uma confirmação em duas fases funciona, em sua essência, muitas vezes é apenas coordenando fechaduras distribuídas. Os trabalhadores precisam bloquear os recursos locais para garantir que a confirmação possa ocorrer durante a segunda fase. Gerenciando bloqueios e evitar impasses em um sistema de processo único não é divertido. Agora imagine os desafios de coordenar bloqueios entre vários participantes. É não é bonito.

Há uma série de modos de falha associados a confirmações bifásicas que nós não tenho tempo para explorar. Considere o problema de um trabalhador votar em prossiga com a transação, mas não responda quando solicitado a se comprometer. O que devemos fazer então? Alguns desses modos de falha podem ser manipulados automaticamente, mas alguns podem deixar o sistema em tal estado que as coisas precisam a ser corrigido manualmente por um operador.

Quanto mais participantes você tiver e mais latência você tiver no sistema, quanto mais problemas um commit em duas fases tiver. 2PC pode ser uma maneira rápida de injete grandes quantidades de latência em seu sistema, especialmente se o escopo de o bloqueio é grande ou se a duração da transação for grande. É para isso razão pela qual os commits de duas fases são normalmente usados apenas por períodos muito curtos operações. Quanto mais tempo a operação demorar, mais tempo você terá recursos trancado!

Transações distribuídas - basta dizer não

Por todas as razões descritas até agora, sugiro fortemente que você evite o uso de transações distribuídas, como a confirmação em duas fases, para coordenar mudanças em estado em seus microserviços. Então, o que mais você pode fazer?

Bem, a primeira opção poderia ser simplesmente não dividir os dados em primeiro lugar. Se você tem partes do estado que deseja gerenciar de uma forma verdadeiramente atômica e de forma consistente, e você não consegue descobrir como obtê-los de forma sensata características sem uma transação no estilo ACID e, em seguida, deixe esse estado em um banco de dados único e deixe a funcionalidade que gerencia esse estado em um único serviço (ou em seu monólito). Se você está no processo de descobrir onde para dividir seu monólito e quais decomposições podem ser fáceis (ou difíceis), então você pode muito bem decidir que dividir os dados que atualmente são gerenciados em um a transação é muito difícil de lidar no momento. Trabalhe em alguma outra área do sistema e volte a isso mais tarde.

Mas o que acontece se você realmente precisar separar esses dados, mas não quer toda a dor de gerenciar transações distribuídas? Como você pode carregar encerra operações em vários serviços, mas evita o bloqueio? E se a operação

vai levar minutos, dias ou talvez até meses? Em casos como esse, você pode considerar uma abordagem alternativa: sagas.

TRANSAÇÕES DISTRIBUÍDAS DO BANCO DE DADOS

Estou argumentando contra o uso geral de transações distribuídas para coordene a mudança de estado em microserviços. Em tais situações, cada microserviço está gerenciando seu próprio estado durável local (por exemplo, em seu banco de dados). Algoritmos transacionais distribuídos estão sendo usados com sucesso para alguns bancos de dados de grande escala, sendo o Spanner do Google um tal sistema. Nessa situação, a transação distribuída está sendo aplicada de forma transparente do ponto de vista de um aplicativo pelo subalterno banco de dados, e a transação distribuída está apenas sendo usada para coordenar mudanças de estado em um único banco de dados lógico (embora possa ser distribuído em várias máquinas e, potencialmente, em várias centros de dados).

O que o Google conseguiu alcançar com o Spanner é impressionante, mas também é importante notar que o que foi necessário fazer para fazer esse trabalho oferece a você uma ideia dos desafios envolvidos. Digamos que envolve muito data centers caros e relógios atômicos baseados em satélite (na verdade). Para um bom entendimento de como o Spanner faz isso funcionar, eu recomendo a apresentação "Google Cloud Spanner: consistência global em grande escala".

Sagas

Ao contrário de um commit em duas fases, uma saga é, por design, um algoritmo que pode coordenar várias mudanças de estado, mas evita a necessidade de bloqueio de recursos por longos períodos de tempo. Uma saga faz isso modelando as etapas envolvidas como atividades discretas que podem ser executadas de forma independente. Usando sagas vem com o benefício adicional de nos forçar a modelar explicitamente nossos processos de negócios, que podem ter benefícios significativos.

A ideia central, descrita pela primeira vez em "Sagas", de Hector Garcia-Molina e

Kenneth Salem,⁴ aborda a melhor forma de lidar com operações conhecidas como longas

transações vividas (LLTs). Essas transações podem levar muito tempo (minutos, horas ou talvez até dias) e, como parte desse processo, exigem alterações a serem feitas em um banco de dados.

Se você mapeou diretamente um LLT para uma transação normal de banco de dados, uma única a transação do banco de dados abrangeia todo o ciclo de vida do LLT. Isso poderia resultar em várias linhas ou até mesmo tabelas completas sendo bloqueadas por longos períodos de tempo em que o LLT está ocorrendo, causando problemas significativos se outros processos estão tentando ler ou modificar esses recursos bloqueados.

Em vez disso, os autores do artigo sugerem que devemos detalhar esses LLTs em uma sequência de transações, cada uma das quais pode ser tratada de forma independente. A ideia é que a duração de cada uma dessas "subtransações" seja mais curto e modificará somente parte dos dados afetados por todo o LLT. Como resultado, haverá muito menos contêncio no banco de dados subjacente do que o escopo e a duração dos bloqueios são bastante reduzidos.

Embora as sagas tenham sido originalmente concebidas como um mecanismo para ajudar com os LLTs atuando em um único banco de dados, o modelo funciona da mesma forma para coordenando mudanças em vários serviços. Podemos quebrar um único negócio processar em um conjunto de chamadas que serão feitas para serviços colaborativos - isso é o que constitui uma saga.

NOTE NOTA

Antes de prosseguirmos, você precisa entender que uma saga não nos dá atomicidade em termos ACID, como estamos acostumados com uma transação normal de banco de dados. À medida que quebramos o LLT em transações individuais, não temos atomicidade no nível da saga em si. Nós temos atomicidade para cada transação individual dentro da saga geral, pois cada uma das elas pode se relacionar com uma mudança transacional ACID, se necessário. O que uma saga nos dá é informações suficientes para raciocinar sobre em que estado ela se encontra: cabe a nós lidar com as implicações disso.

Vamos dar uma olhada em um fluxo simples de atendimento de pedidos da MusicCorp, descrito na Figura 6-5, que podemos usar para explorar ainda mais as sagas no contexto de um arquitetura de microserviços.

Aqui, o processo de atendimento de pedidos é representado como uma única saga, com cada etapa neste fluxo representando uma operação que pode ser realizada por um serviço diferente. Dentro de cada serviço, qualquer mudança de estado pode ser tratada dentro de uma transação ACID local. Por exemplo, quando verificamos e reservamos estoque usando o serviço de Armazém, internamente, o serviço de Armazém pode criar uma linha em sua tabela de reservas local registrando a reserva; isso a alteração seria tratada dentro de uma transação normal do banco de dados.

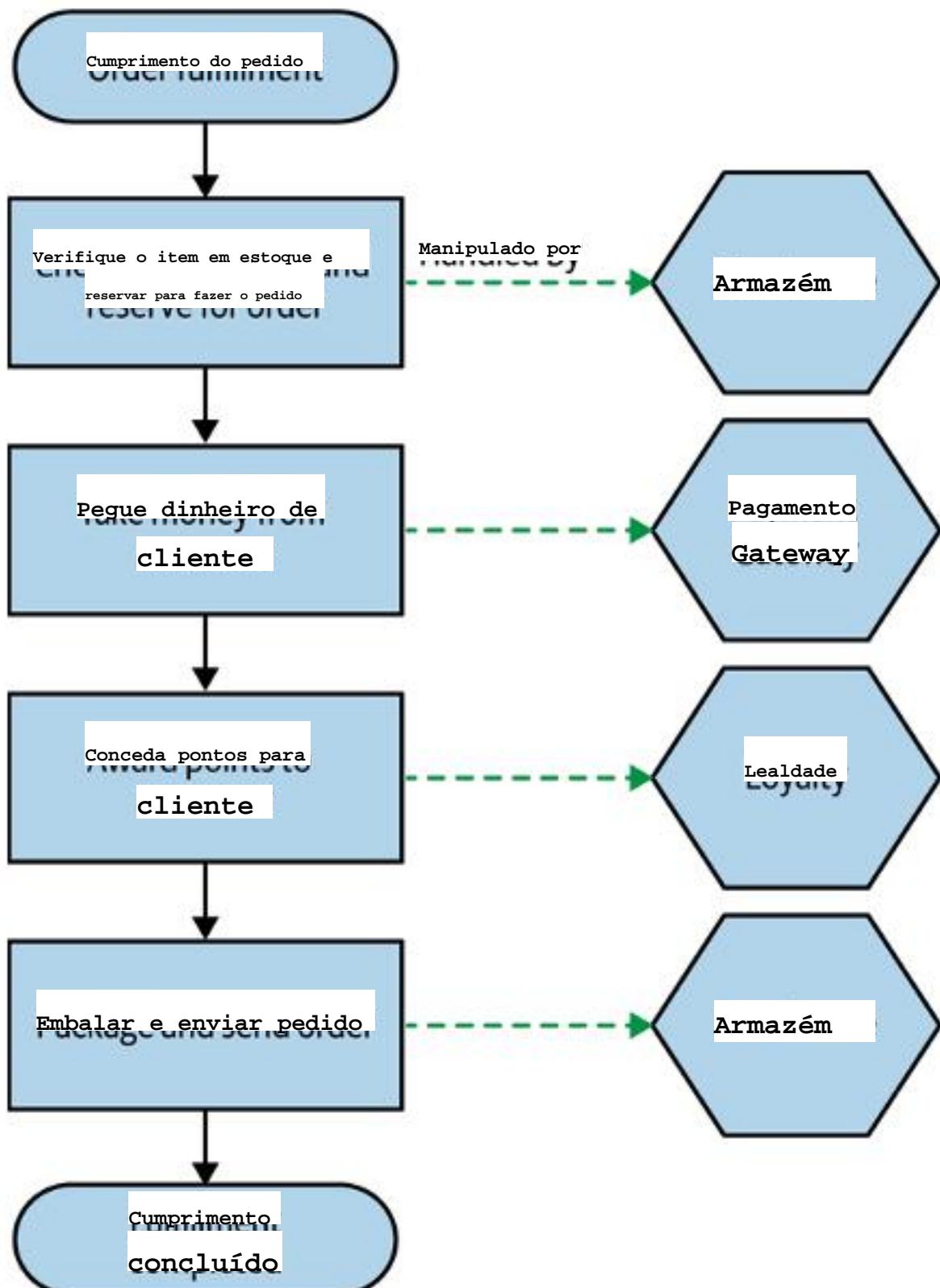


Figura 6-5. Um exemplo de fluxo de atendimento de pedidos, juntamente com os serviços responsáveis pelo transporte fora da operação.

Modos de falha do Saga

Com uma saga sendo dividida em transações individuais, precisamos considerar como lidar com uma falha ou, mais especificamente, como se recuperar em caso de falha acontece. O artigo original da saga descreve dois tipos de recuperação: retroativa recuperação e recuperação futura.

A recuperação reversa envolve a reversão da falha e a limpeza posterior.

- uma reversão. Para que isso funcione, precisamos definir ações compensatórias que nos permitem desfazer transações previamente confirmadas. Recuperação futura nos permite partir do ponto em que a falha ocorreu e manter processamento. Para que isso funcione, precisamos ser capazes de repetir as transações, o que por sua vez, implica que nosso sistema está persistindo informações suficientes para permitir isso tente acontecer novamente.

Dependendo da natureza do processo de negócios que está sendo modelado, você pode espere que qualquer modo de falha acione uma recuperação para trás, uma recuperação para frente recuperação, ou talvez uma mistura dos dois.

É muito importante observar que uma saga nos permite nos recuperar dos negócios falhas, não falhas técnicas. Por exemplo, se tentarmos receber o pagamento de o cliente, mas o cliente não tem fundos suficientes, então isso é um negócio falha que se espera que a saga resolva. Por outro lado, se o

O Payment Gateway atinge o tempo limite ou gera um erro de serviço interno de 500, então essa é uma falha técnica que precisamos tratar separadamente. A saga assume que os componentes subjacentes estão funcionando corretamente - que o sistema subjacente é confiável e, em seguida, coordenamos o trabalho de componentes confiáveis. Exploraremos algumas das maneiras pelas quais podemos fazer nossa componentes técnicos mais confiáveis no Capítulo 12, mas para saber mais sobre isso limitação de sagas, eu recomendo "The Limits of the Saga Pattern" de Uwe Friedrichsen.

Reversões da saga

Com uma transação ACID, se encontrarmos um problema, acionamos uma reversão antes de um a confirmação ocorre. Depois da reversão, é como se nada tivesse acontecido: o A mudança que estávamos tentando fazer não aconteceu. Com nossa saga, porém, nós

têm várias transações envolvidas, e algumas delas podem já ter comprometidos antes de decidirmos reverter toda a operação. Então, como podemos reverter as transações depois que elas já tiverem sido confirmadas?

Vamos voltar ao nosso exemplo de processamento de um pedido, conforme descrito em Figura 6-5. Considere um modo de falha potencial. Chegamos ao ponto de tentar para empacotar o item, apenas para descobrir que o item não pode ser encontrado no armazém, pois mostrado na Figura 6-6. Nosso sistema acha que o item existe, mas simplesmente não está no prateleira!

Agora, vamos supor que decidimos que queremos apenas reverter todo o pedido, em vez de dar ao cliente a opção de colocar o item na parte traseira pedido. O problema é que já recebemos o pagamento e concedemos fidelidade pontos pelo pedido.

Se todas essas etapas tivessem sido executadas em uma única transação de banco de dados, uma simples a reversão limparia tudo. No entanto, cada etapa do atendimento do pedido o processo foi tratado por uma chamada de serviço diferente, cada uma operada em um escopo transacional diferente. Não há uma simples "reversão" para todo o operação.

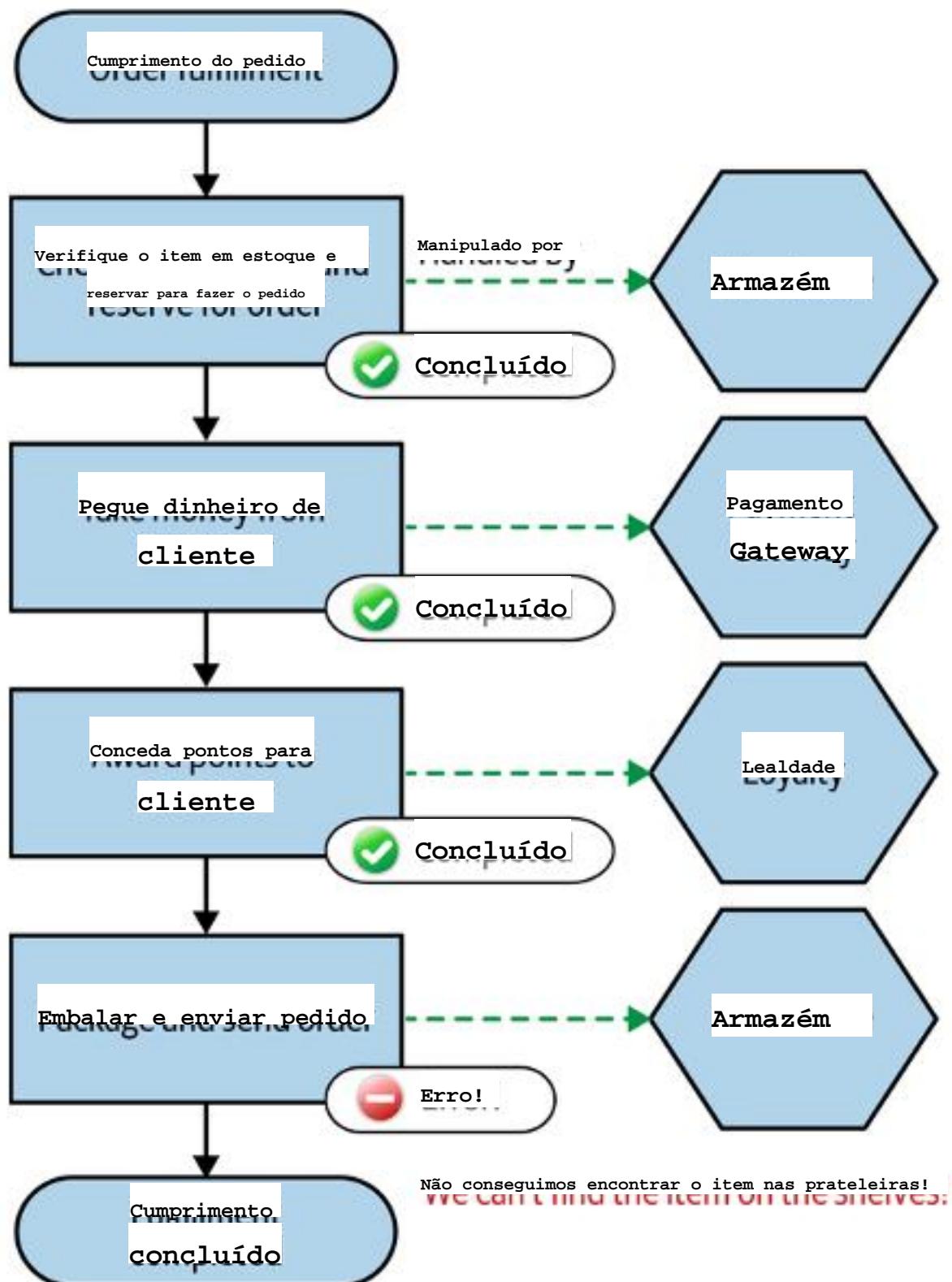


Figura 6-6. Tentamos empacotar nosso item, mas (podemos) encontrá-lo no armazém.

Em vez disso, se você quiser implementar uma reversão, precisará implementar uma transação compensatória. Uma transação compensatória é uma operação que desfaz uma transação previamente confirmada. Para reverter nosso atendimento de pedidos processo, açãoariam os a transação de compensação para cada etapa do nosso saga que já foi cometida, conforme mostrado na Figura 6-7.

Vale ressaltar o fato de que essas transações compensatórias podem não ser se comportam exatamente como os de uma reversão normal do banco de dados. Uma reversão do banco de dados acontece antes da confirmação e, após a reversão, é como se o a transação nunca aconteceu. Nessa situação, é claro, essas transações funcionaram acontecer. Estamos criando uma nova transação que reverte as alterações feitas por a transação original, mas não podemos reverter o tempo e torná-la como se a a transação original não ocorreu.

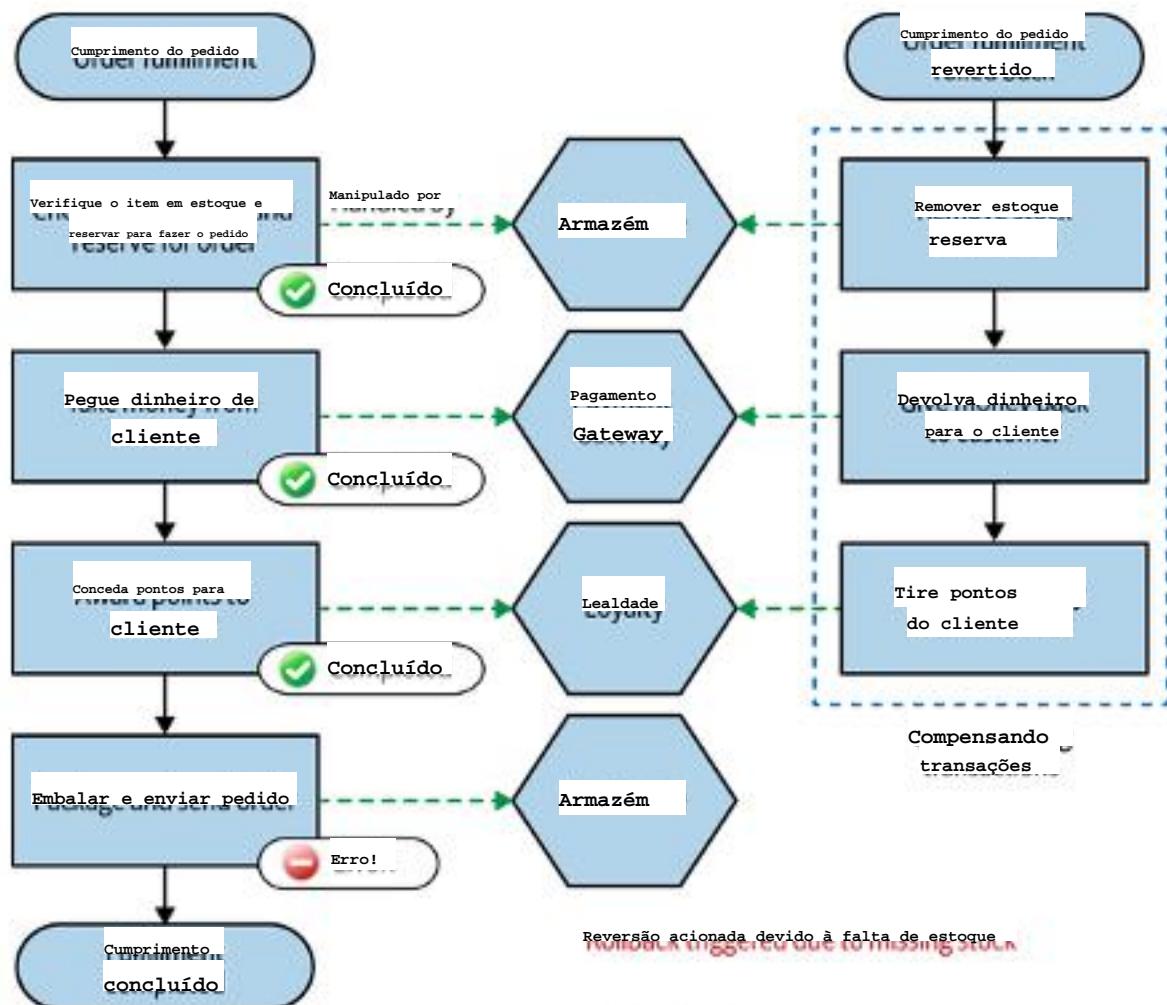


Figura 6-7. Provocando uma reversão de toda a saga

Como nem sempre podemos reverter uma transação de forma limpa, dizemos que essas transações compensatórias são reversões semânticas. Nem sempre podemos limpar preparamos tudo, mas fazemos o suficiente para o contexto da nossa saga. Como exemplo, uma de nossas etapas pode ter envolvido o envio de um e-mail a um cliente para informar para eles seu pedido estava a caminho. Se decidirmos reverter isso, não poderemos cancelar o envio de um e-mail! Em vez disso, nossa transação de compensação pode causar um segundo e-mail a ser enviado ao cliente, informando que havia um problema com o pedido e ele foi cancelado.

É totalmente apropriado que as informações relacionadas à reversão persistam em o sistema. Na verdade, isso pode ser uma informação muito importante. Você pode querer manter um registro no serviço de pedidos para esse pedido abortado, junto com informações sobre o que aconteceu, por uma série de razões.

Reordenando as etapas do fluxo de trabalho para reduzir as reversões

Na Figura 6-7, poderíamos ter criado um pouco nossos prováveis cenários de reversão mais simples ao reordenar as etapas em nosso fluxo de trabalho original. Uma mudança simples seria conceder pontos somente quando o pedido fosse realmente despachado, conforme visto na Figura 6-8.

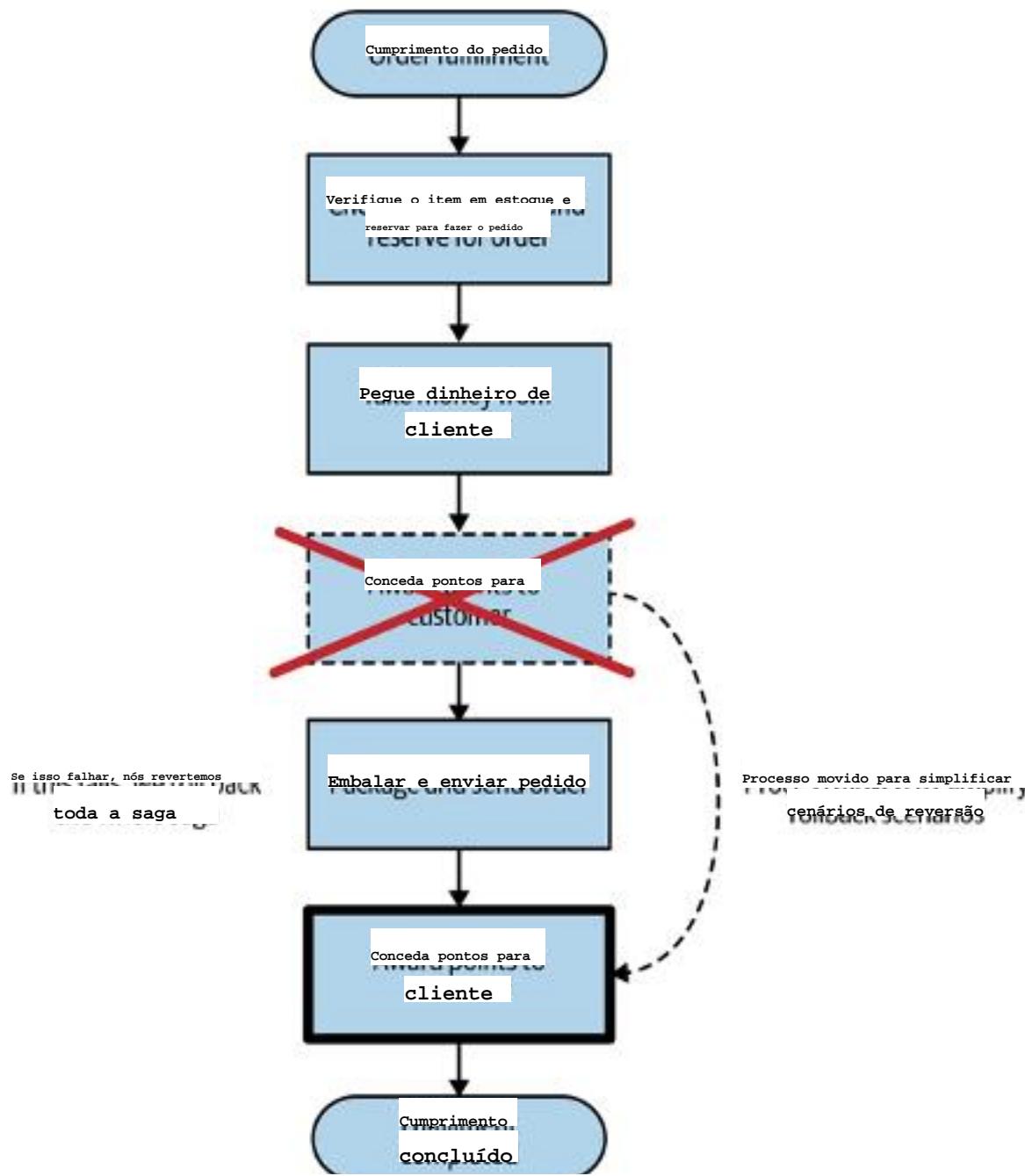


Figura 6-8. Avançar etapas posteriores na saga pode reduzir o que precisa ser revertido no caso de um falha

Dessa forma, evitariamnos ter que nos preocupar com a rolagem desse estágio. de volta se tivéssemos algum problema ao tentar empacotar e enviar o pedido. Às vezes, você pode simplificar suas operações de reversão apenas ajustando como seu fluxo de trabalho é realizado. Ao avançar aqueles passos que são com maior probabilidade de falhar e falhar o processo mais cedo, você evita ter que acionar

~~transações de compensação posteriores, já que essas etapas nem mesmo foram acionadas no primeiro lugar.~~

~~Essas mudanças, se puderem ser acomodadas, podem tornar sua vida muito mais fácil, evitando a necessidade de até mesmo criar transações compensatórias para algumas etapas. Isso pode ser especialmente importante ao implementar uma transação de compensação. é difícil. Talvez você consiga passar uma etapa posterior do processo para uma etapa em que nunca precisa ser revertida.~~

Misturando situações de falha retroativa e falha

~~É totalmente apropriado ter uma combinação de modos de recuperação de falhas. Alguns falhas podem exigir uma reversão (falha para trás); outras podem ser falhas. Para o processamento de pedidos, por exemplo, depois de retirarmos dinheiro do cliente, e o item foi embalado, a única etapa restante é enviar o pacote. Se, por algum motivo, não pudermos enviar o pacote (talvez o (a empresa de entrega que usamos não tem espaço em suas vans para fazer um pedido hoje), parece muito estranho reverter todo o pedido. Em vez disso, provavelmente apenas tente novamente o despacho (talvez colocando-o na fila para o dia seguinte) e, se isso falhar, precisariam de intervenção humana para resolver a situação.~~

Implementando sagas

~~Até agora, analisamos o modelo lógico de como as sagas funcionam, mas precisamos vá um pouco mais fundo para examinar formas de implementar a saga em si. Nós podemos olhar em dois estilos de implementação de saga: sagas orquestradas mais de perto siga o espaço da solução original e confie principalmente na centralização coordenação e rastreamento. Elas podem ser comparadas a sagas coreografadas, que evitam a necessidade de uma coordenação centralizada em favor de uma coordenação mais flexível modelo acoplado, mas pode facilitar o acompanhamento do progresso de uma saga complicado.~~

Sagas orquestradas

~~As sagas orquestradas usam um coordenador central (o que chamaremos de orquestrador (de agora em diante) para definir a ordem de execução e acionar qualquer ação compensatória necessária. Você pode pensar nas sagas orquestradas como uma~~

abordagem de comando e controle: o orquestrador controla o que acontece e quando, e com isso vem um bom grau de visibilidade do que está acontecendo com qualquer saga.

Considerando o processo de atendimento de pedidos na Figura 6-5, vamos ver como essa centralização de coordenação funcionaria como um conjunto de serviços colaborativos, como mostrado na Figura 6-9.

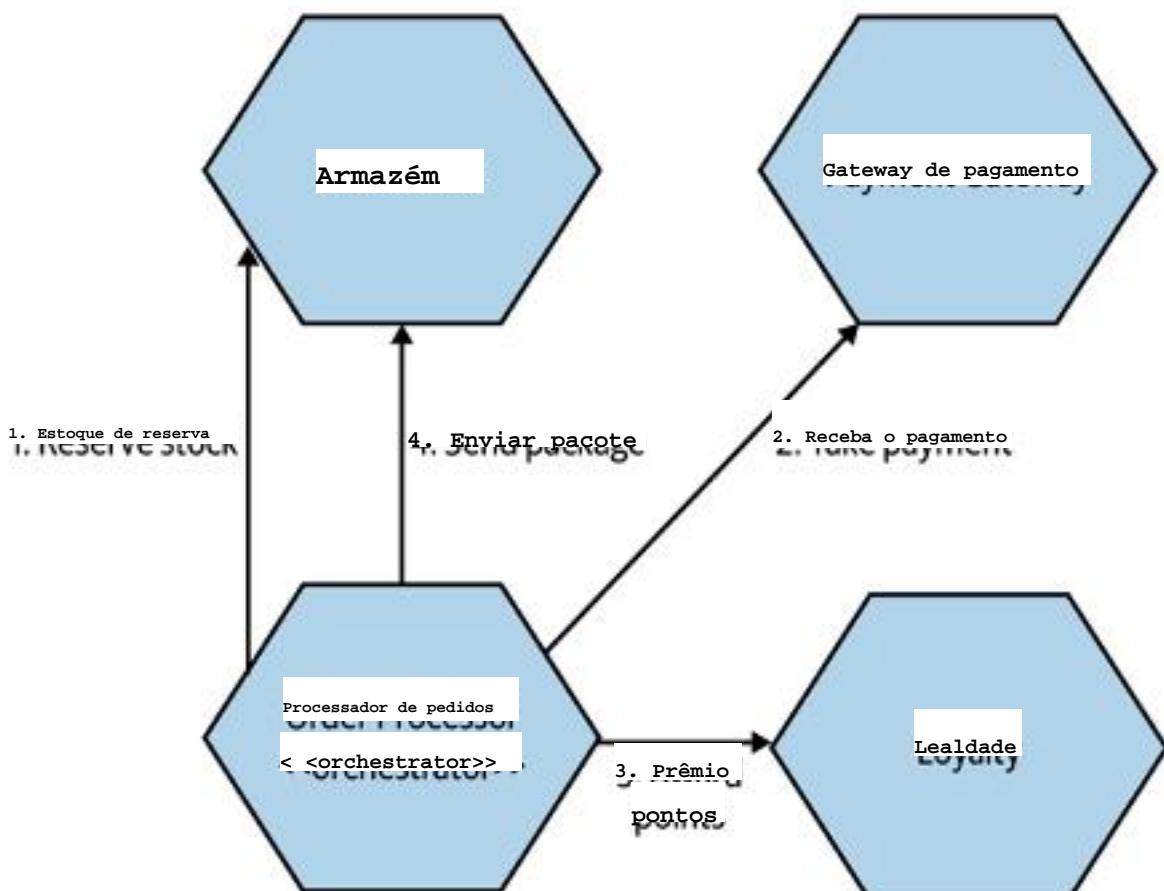


Figura 6-9. Um exemplo de como uma saga orquestrada pode ser usada para implementar nosso pedido de atendimento.

Aqui, nosso processador central de pedidos, desempenhando o papel de orquestrador, coordena nosso processo de atendimento. Ele sabe quais serviços são necessários para realizar a operação e decide quando fazer chamadas para esses serviços.

Se as chamadas falharem, ele poderá decidir o que fazer como resultado. Em geral, orquestradores

as sagas tendem a fazer uso intenso das interações de solicitação-resposta entre

serviços: o Processador de Pedidos envia uma solicitação aos serviços (como um

Payment Gateway), e, espera uma resposta para informar se a solicitação foi bem-sucedido e forneça os resultados da solicitação.

Ter nosso processo de negócios explicitamente modelado dentro do Pedido

O processador é extremamente benéfico. Isso nos permite olhar para um lugar em nosso sistema e entenda como esse processo deve funcionar. Isso pode fazer facilitar a integração de novas pessoas e ajudar a transmitir uma melhor compreensão das partes principais do sistema.

No entanto, há algumas desvantagens a serem consideradas. Primeiro, por sua natureza, este é um abordagem um tanto acoplada. Nosso processador de pedidos precisa saber sobre todos os serviços associados, resultando em um maior grau de acoplamento de domínio.

Embora o acoplamento de domínios não seja inherentemente ruim, ainda gostaríamos de mantê-lo em um mínimo, se possível. Aqui, nosso processador de pedidos precisa conhecer e controle tantas coisas que essa forma de acoplamento é difícil de evitar.

A outra questão, que é mais sutil, é aquela lógica que, de outra forma, deveria ser inserido nos serviços pode começar a ser absorvido pelo orquestrador em vez disso. Se isso começar a acontecer, você poderá descobrir que seus serviços se tornam anêmicos, com pouco comportamento próprio, apenas recebendo ordens de orquestradores como o Order Processor. É importante que você ainda considere os serviços que compõem esses fluxos orquestrados como entidades que têm seus próprios estado e comportamento locais. Eles são responsáveis por seu próprio estado local máquinas.

ADVERTÊNCIA

Se a lógica tiver um lugar onde possa ser centralizada, ela se tornará centralizada!

Uma forma de evitar muita centralização com fluxos orquestrados é garantir que você tenha diferentes serviços desempenhando o papel de orquestrador para fluxos diferentes. Você pode ter um microsserviço de processador de pedidos que lida com a realização de um pedido, um microsserviço de devoluções para lidar com a devolução e processo de reembolso, um microsserviço de recebimento de mercadorias que lida com novos estoques

chegando e sendo colocado nas prateleiras, e assim por diante. Algo como o nosso microserviço de armazém pode ser usado por todos esses orquestradores; como o modelo facilita a manutenção da funcionalidade no Depósito microserviço em si, permitindo que você reutilize a funcionalidade em todos aqueles fluxos.

FERRAMENTAS DE BPM

As ferramentas de modelagem de processos de negócios (BPM) estão disponíveis para muitos anos. Em geral, eles são projetados para permitir que não desenvolvedores definam fluxos de processos de negócios, geralmente usando ferramentas visuais de arrastar e soltar. A ideia é que os desenvolvedores criariam os blocos de construção desses processos, e então os não desenvolvedores conectariam esses blocos de construção em os maiores fluxos do processo. O uso de tais ferramentas parece realmente se alinhar muito bem como uma forma de implementar sagas orquestradas e, de fato, processos a orquestração é praticamente o principal caso de uso de ferramentas de BPM (ou, ao contrário, o uso de ferramentas de BPM faz com que você tenha que adotar orquestração).

Em minha experiência, passei a não gostar muito das ferramentas de BPM. O principal motivo é que o conceito central de que os não desenvolvedores definirão o processo de negócios - na minha experiência quase nunca foi verdadeiro. As ferramentas voltadas para não desenvolvedores acabam sendo usadas por desenvolvedores, e infelizmente, essas ferramentas geralmente funcionam de maneiras que são estranhas à forma como os desenvolvedores gostam de trabalhar. Eles geralmente exigem o uso de GUIs para alterar os fluxos, os fluxos que eles criam podem ser difíceis (ou impossíveis) de configurar corretamente, os fluxos em si podem não ser projetados com testes em mente, e muito mais.

Se seus desenvolvedores vão implementar seus processos de negócios, deixe-os usar ferramentas que eles conheçam e entendam e que sejam adequadas para seus fluxos de trabalho. Em geral, isso significa apenas permitir que eles usem o código para implementar essas coisas! Se você precisar de visibilidade sobre como foi um processo de negócios implementado ou como está operando, então é muito mais fácil projetar uma representação visual de um fluxo de trabalho a partir do código do que usar uma visualização do seu fluxo de trabalho para descrever como seu código deve trabalhar.

Há esforços para criar ferramentas de BPM mais amigáveis ao desenvolvedor. O feedback dos desenvolvedores sobre essas ferramentas parece ser misto, mas as ferramentas funcionaram bem para alguns, e é bom ver pessoas tentando melhorar essas estruturas. Se você sentir a necessidade de explorar essas ferramentas

Além disso, dê uma olhada em Camunda e Zeebe, ambas abertas estruturas de orquestração de origem voltadas para desenvolvedores de microsserviços e que estaria no topo da minha lista se eu decidisse que uma ferramenta de BPM era para eu.

Sagas coreografadas

Uma saga coreografada visa distribuir a responsabilidade pela operação de a saga entre vários serviços colaborativos. Se a orquestração for uma abordagem de comando e controle, as sagas coreografadas representam uma confiança, mas... verifique a arquitetura. Como veremos em nosso exemplo na Figura 6-10, sagas coreografadas geralmente fazem uso intenso de eventos para colaboração entre serviços.

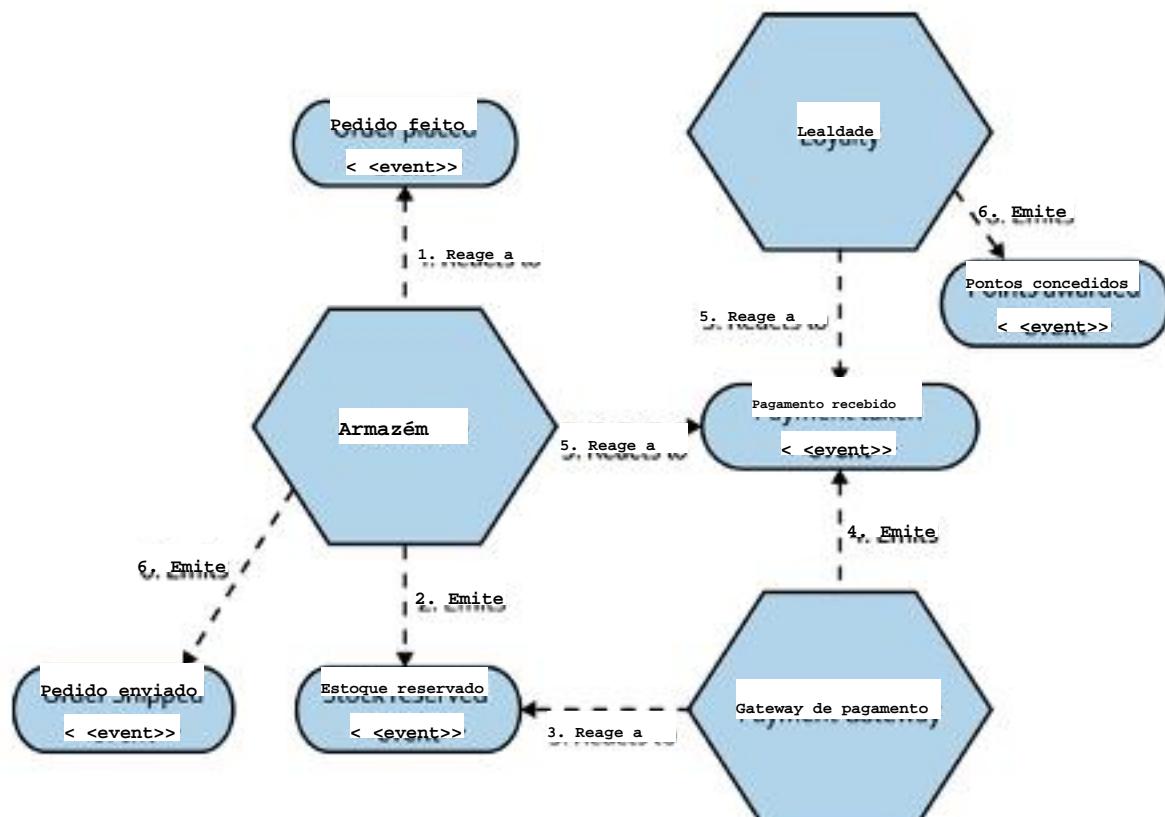


Figura 6-10. Um exemplo de uma saga coreografada para implementar o atendimento de pedidos

Há muita coisa acontecendo aqui, então vale a pena explorar com mais detalhes. Primeiro, esses microsserviços estão reagindo aos eventos recebidos... Conceitualmente, os eventos são transmitidos no sistema e as partes interessadas são

capaz de recebê-los. Lembre-se de que, conforme discutimos no Capítulo 4, você não envia eventos para um microsserviço; basta desligá-los, e os microsserviços que estão interessados nesses eventos podem recebê-los e agir em conformidade. Em nosso exemplo, quando o serviço Warehouse recebe isso primeiro Evento de pedido feito, ele sabe que seu trabalho é reservar o estoque apropriado e acione um evento quando isso for feito. Se o estoque não pudesse ser recebido, o O armazém precisaria levantar um evento apropriado (um insuficiente) Evento de estoque, talvez), o que pode fazer com que o pedido seja abortado.

Também vemos neste exemplo como os eventos podem facilitar o processamento paralelo. Quando o evento Payment Taken é acionado pelo Payment Gateway, ele causa reações nos microsserviços Loyalty e Warehouse. O armazém reage despachando o pacote, enquanto o Loyalty o microsserviço reage concedendo pontos.

Normalmente, você usaria algum tipo de corretor de mensagens para gerenciar o confiável transmissão e entrega de eventos. É possível que vários microsserviços pode reagir ao mesmo evento, e é aí que você usaria um tópico. Festas interessado em um determinado tipo de evento se inscreveria em um tópico específico sem ter que se preocupar com a origem desses eventos, e com o corretor garante a durabilidade do tópico e que os eventos sobre ele sejam bem-sucedidos, entregue aos assinantes. Como exemplo, podemos ter uma recomendação serviço que também escuta eventos Order Placed e os usa para construir um banco de dados de opções musicais de que você pode gostar.

Na arquitetura anterior, nenhum serviço conhece nenhum outro microsserviço. Eles precisam saber apenas o que fazer quando um determinado evento ocorre. recebido - reduzimos drasticamente a quantidade de acoplamento de domínio. Inerentemente, isso cria uma arquitetura muito menos acoplada. Como o a implementação do processo é decomposta e distribuída entre os três microsserviços aqui, também evitamos as preocupações com a centralização da lógica (se você não tem um lugar onde a lógica possa ser centralizada, então não será centralizado!).

O outro lado disso é que pode ser mais difícil descobrir o que está acontecendo.

Com a orquestração, nosso processo foi explicitamente modelado em nosso orquestrador.

Agora, com essa arquitetura apresentada, como você construiria um modelo mental do que o processo deveria ser? Você teria que olhar o comportamento de cada serviço de forma isolada e reconstitua essa imagem em seu próprio, longe de ser simples, mesmo, com um processo de negócios simples como essa.

A falta de uma representação explícita do nosso processo de negócios já é ruim o suficiente, mas também não temos como saber em que estado uma saga se encontra, o que pode nos negar a chance de anexar ações compensatórias quando necessário. Podemos empurrar algumas responsabilidades perante os serviços individuais pela realização de compensações, mas fundamentalmente precisamos de uma maneira de saber em que estado uma saga está para alguns tipos de recuperação. A falta de um lugar central para interrogar o status de uma saga é um grande problema. Conseguimos isso com orquestração, então como resolvemos isso aqui?

Uma das maneiras mais fáceis de fazer isso é projetar uma visão sobre o estado de uma saga consumindo os eventos que estão sendo emitidos. Se gerarmos um ID exclusivo para a saga, o que é conhecido como ID de correlação, podemos colocá-lo em todos os eventos que são emitidos como parte dessa saga. Quando um de nossos serviços reage para um evento, a ID de correlação é extraída e usada para qualquer registro local processos, e também é transmitida a jusante com quaisquer outras chamadas ou eventos que são demitidos. Poderíamos então ter um serviço cujo trabalho é simplesmente aspirar todos esses eventos e apresentar uma visão do estado em que cada pedido se encontra, e talvez realize programaticamente ações para resolver problemas como parte do processo de atendimento se os outros serviços não pudessem fazer isso sozinhos. Eu considero alguma forma de correlação é essencial para sagas coreografadas como essa, mas IDs de correlação também têm muito valor em geral, algo que nós explore mais detalhadamente no Capítulo 10.

Estilos de mistura

Embora possa parecer que as sagas orquestradas e coreografadas sejam visões diametralmente opostas de como as sagas poderiam ser implementadas, você poderia considerar facilmente misturar e combinar modelos. Você pode ter algum negócio

processos em seu sistema que se encaixam mais naturalmente em um modelo ou outro. Você também pode ter uma única saga que tenha uma mistura de estilos. No atendimento do pedido caso de uso, por exemplo, dentro do limite do serviço Warehouse, quando gerenciando a embalagem e o envio de um pedido, podemos usar um orchestrado fluir mesmo que a solicitação original tenha sido feita como parte de uma coreografada maior saga.⁶

Se você decidir misturar estilos, é importante que você ainda tenha uma maneira clara de entender em que estado uma saga se encontra e quais atividades já aconteceram como parte de uma saga. Sem isso, entender os modos de falha se torna complexo, e a recuperação de uma falha é difícil.

RASTREAMENTO DE CHAMADAS

Se você escolheu coreografia ou orquestração, ao implementar um processo de negócios usando vários microserviços, é comum querer ser capaz de rastrear todas as chamadas relacionadas ao processo. Isso às vezes pode seja apenas para ajudá-lo a entender se o processo de negócios está funcionando corretamente, ou pode ser para ajudá-lo a diagnosticar um problema. No capítulo 10 veremos conceitos como IDs de correlação e agregação de registros e como eles podem ajudar nesse sentido.

Devo usar coreografia ou orquestração (ou uma mistura)?

A implementação de sagas coreografadas pode trazer consigo ideias que podem não estão familiarizados com você e sua equipe. Eles normalmente assumem o uso intenso de colaboração baseada em eventos, que não é amplamente compreendida. No entanto, na minha experiência, a complexidade extra associada ao acompanhamento do progresso de um a saga é quase sempre superada pelos benefícios associados a ter uma arquitetura mais fracamente acoplada.

Afastando-se dos meus gostos pessoais, porém, o conselho geral é Dar sobre orquestração versus coreografia é que estou muito relaxado em o uso de sagas orquestradas quando uma equipe possui a implementação do saga inteira. Em tal situação, a arquitetura mais inherentemente acoplada é muito mais fácil de gerenciar dentro dos limites da equipe. Se você tiver várias equipes

envolvido, eu prefiro muito a saga coreografada mais decomposta, pois está
é mais fácil distribuir a responsabilidade pela implementação da saga para as equipes,
com a arquitetura mais fracamente acoplada, permitindo que essas equipes trabalhem
mais isolado.

É importante notar que, como regra geral, você terá mais chances de gravitar
para chamadas baseadas em solicitação-resposta com orquestração, enquanto
a coreografia tende a fazer um uso mais intenso dos eventos. Esta não é uma regra difícil, apenas
uma observação geral. Minha própria inclinação geral para a coreografia é
provavelmente uma função do fato de que eu costumo gravitar em direção a eventos
modelos de interação - se você estiver descobrindo o uso de colaboração baseada em eventos
difícil de entender, a coreografia pode não ser para você.

Sagas versus transações distribuídas

Como espero ter esclarecido até agora, as transações distribuídas vêm com
alguns desafios significativos e, fora de algumas situações muito específicas, eu
tendem a evitá-los. Pat Helland, pioneira em sistemas distribuídos, destila o
desafios fundamentais que surgem com a implementação de transações distribuídas
para os tipos de aplicativos que construímos hoje:⁷

Na maioria dos sistemas de transações distribuídos, a falha de um único nó
faz com que a confirmação da transação pare. Isso, por sua vez, faz com que o aplicativo
para ficar preso. Em tais sistemas, quanto maior ele fica, maior a probabilidade de
o sistema vai ficar inativo. Ao pilotar um avião que precisa de tudo
motores para funcionar, adicionar um motor reduz a disponibilidade do
avião.

Em minha experiência, modelar explicitamente processos de negócios como uma saga evita
muitos dos desafios das transações distribuídas, ao mesmo tempo em que adicionam
benefício de criar processos que, de outra forma, poderiam ser modelados implicitamente
muito mais explícito e óbvio para seus desenvolvedores. Fazendo o núcleo
processos de negócios do seu sistema, um conceito de primeira classe terá uma série de
vantagens.

Resumo

Então, como podemos ver, o caminho para implementar fluxos de trabalho em nosso microsserviço arquitetura se resume a modelar explicitamente o processo de negócios que somos tentando implementar. Isso nos traz de volta à ideia de modelar aspectos de nosso domínio de negócios em nossa arquitetura de microsserviços - modelagem explícita processos de negócios fazem sentido se nossos limites de microsserviços também forem definido principalmente em termos de nosso domínio comercial.

Se você decide gravitar mais em direção à orquestração ou à coreografia, espero que você esteja muito melhor posicionado para saber qual modelo será adapte-se melhor ao seu espaço problemático.

Se você quiser explorar esse espaço com mais detalhes, embora as sagas não sejam explicitamente abordados, Padrões de Integração Corporativa de Gregor Hohpe e

implementando diferentes tipos de fluxo de trabalho.⁸ Eu também posso recomendar vivamente Automação prática de processos de Bernd Ruecker.⁹ O livro de Bernd se concentra muito mais sobre o lado da orquestração das sagas, mas é repleto de coisas úteis informações que o tornam um ponto de acompanhamento natural para este tópico.

Agora temos uma ideia de como nossos microsserviços podem se comunicar e coordenam uns com os outros, mas como podemos construí-los em primeiro lugar? Em no próximo capítulo, veremos como aplicar o controle de origem, contínuo integração e entrega contínua no contexto de um microsserviço arquitetura.

¹ Isso mudou com o suporte para transações ACID de vários documentos sendo liberadas como parte do Mongo 4.0. Eu mesmo não usei esse recurso do Mongo; só sei que ele existe!

² Martin Kleppmann, Projetando aplicativos com uso intensivo de dados (Sebastopol O'Reilly, 2017).

³ Robert Kubis, "Google Cloud Spanner: Global Consistency at Scale", Devoxx, 7 de novembro 2017, Vídeo do YouTube, 33:22. <https://oreil.ly/XHvY5>.

⁴ Hector Garcia-Molina e Kenneth Salem, "Sagas", ACM Sigmod Record 16, nº 3, (1987): 249-59.

⁵ Realmente não podemos, eu tentei!

6 Está fora do escopo deste livro, mas Hector Garcia-Molina e Kenneth Salem continuaram explore como várias sagas podem ser "aninhadas" para implementar processos mais complexos. Para ler mais sobre este tópico, consulte Hector Garcia-Molina et al., "Modelando atividades de longa duração como aninhadas Sagas", Data Engineering 14, nº 1 (março de 1991:14-18).

7 Veja Pat Helland, "A vida além das transações distribuídas: a opinião de um apóstata", acmqueue 14, nº 5 (12 de dezembro de 2016).

8 Gregor Hohpe e Bobby Woolf, Padrões de integração empresarial (Boston: Addison-Wesley, 2003),

9 Bernd Ruecker, Automação prática de processos (Sebastopol: O'Reilly, 2021).

Capítulo 7. Construir

Passamos muito tempo abordando os aspectos de design dos microserviços, mas precisamos começar a nos aprofundar um pouco mais em como seu processo de desenvolvimento pode precisar mudar para acomodar esse novo estilo de arquitetura. Nos próximos capítulos, veremos como implantamos e testamos nossos microserviços, mas antes disso, precisamos ver o que vem primeiro, o que acontece quando um desenvolvedor tem uma mudança pronta para fazer o check-in?

Começaremos essa exploração revisando alguns conceitos fundamentais - integração contínua e entrega contínua. São conceitos importantes não importa que tipo de arquitetura de sistemas você esteja usando, mas os microserviços abrem uma série de perguntas exclusivas. A partir daí, veremos pipelines e em diferentes formas de gerenciar o código-fonte de seus serviços.

Uma breve introdução à integração contínua

A integração contínua (CI) existe há vários anos.

No entanto, vale a pena gastar um pouco de tempo examinando o básico, pois existem algumas opções diferentes a serem consideradas, especialmente quando pensamos sobre o mapeamento entre microserviços, compilações e repositórios de controle de versão.

Com a CI, o objetivo principal é manter todos em sincronia uns com os outros, o que nós conseguimos certificando-se frequentemente de que o código recém-verificado seja corretamente integrado ao código existente. Para fazer isso, um servidor CI detecta que o código tem sido cometido, verifica e realiza algumas verificações, como certificando-se de que o código seja compilado e que os testes sejam aprovados. No mínimo, esperamos que essa integração seja feita diariamente, embora na prática trabalhei, em várias equipes nas quais um desenvolvedor de fato se integrou suas mudanças várias vezes por dia.

Como parte desse processo, geralmente criamos artefatos que são usados para fins futuros de validação, como implantar um serviço em execução para executar testes nele (vamos

explore os testes em profundidade no Capítulo 9). Idealmente, queremos construir esses artefatos apenas uma vez e use-os para todas as implantações dessa versão do código. Isso é para que possamos evitar fazer a mesma coisa repetidamente novamente, e assim podemos confirmar que os artefatos que implantamos são os que testado. Para permitir que esses artefatos sejam reutilizados, nós os colocamos em um repositório de algum tipo, fornecido pela própria ferramenta de CI ou em um sistema separado. Em breve, analisaremos o papel dos artefatos com mais profundidade e analisaremos em profundidade nos testes no Capítulo 9.

A CI tem vários benefícios. Recebemos feedback rápido quanto à qualidade de nossos código, por meio do uso de análises e testes estáticos. A CI também nos permite automatize a criação de nossos artefatos binários. Todo o código necessário para criar o artefato em si é controlado por versão, então podemos recriar o artefato se necessário. Também podemos rastrear desde um artefato implantado até o código e, dependendo das capacidades da própria ferramenta de CI, podemos ver quais foram os testes execute também no código e no artefato. Se adotarmos a infraestrutura como código, podemos também controlam a versão de todo o código necessário para configurar a infraestrutura para nosso microsserviço junto com o código do próprio microsserviço, melhorando transparência em relação às mudanças e facilitando ainda mais a reprodução de compilações. É por esses motivos que a CI tem tido tanto sucesso.

Você está realmente fazendo CI?

A CI é uma prática fundamental que nos permite fazer mudanças de forma rápida e fácil, e sem a qual a jornada para os microsserviços será dolorosa. Eu suspeito de você provavelmente estão usando uma ferramenta de CI em sua própria organização, mas isso pode não ser a mesma coisa que realmente fazer CI. Já vi muitas pessoas confundirem a adoção uma ferramenta de CI que realmente adota a CI. Uma ferramenta de CI, bem usada, ajudará você a fazer CI-mas usando uma ferramenta como Jenkins, CircleCI, Travis ou uma das muitas outras as opções disponíveis não garantem que você esteja realmente fazendo CI corretamente.

Então, como saber se você está realmente praticando CI? Eu gosto muito de Jez As três perguntas de Humble que ele faz às pessoas que testem se elas realmente entendem o que CI é sobre - pode ser interessante fazer a si mesmo as mesmas perguntas:

Você faz check-in na linha principal uma vez por dia?

Você precisa ter certeza de que seu código está integrado. Se você não verificar seu código junto com as mudanças de todos os outros com frequência, você acaba fazendo integração futura mais difícil. Mesmo se você estiver usando filiais de curta duração para gerencie mudanças, integre com a maior frequência possível em uma única linha principal filial - pelo menos uma vez por dia.

Você tem um conjunto de testes para validar suas alterações?

Sem testes, só sabemos que sintaticamente nossa integração funcionou, mas não sabemos se quebramos o comportamento do sistema. CI sem alguma verificação de que nosso código se comporta conforme o esperado não é CI.

Quando a construção é interrompida, é a prioridade #1 da equipe consertá-la?

Uma construção verde passageira significa que nossas mudanças foram integradas com segurança. UMA red build significa que a última alteração possivelmente não foi integrada. Você precisa interrompa todos os outros check-ins que não estejam envolvidos na correção das compilações para obtê-las passando novamente. Se você deixar que mais mudanças se acumulem, o tempo necessário para corrigir o a construção aumentará drasticamente. Eu trabalhei com equipes onde a construção está quebrado há dias, resultando em esforços substanciais para, eventualmente, obter uma construção passageira.

Modelos de ramificação

Poucos tópicos sobre construção e implantação parecem causar tanto problema controvérsia como a do uso de ramificações de código-fonte para o desenvolvimento de recursos.

A ramificação no código-fonte permite que o desenvolvimento seja feito de forma isolada sem interromper o trabalho que está sendo feito por outras pessoas. Na superfície, criando um ramificação do código-fonte para cada recurso que está sendo trabalhado, também conhecida como ramificação de recursos - parece um conceito útil.

O problema é que quando você trabalha em uma ramificação de recursos, você não está regularmente integrando suas mudanças com todos os outros. Fundamentalmente, você é atrasando a integração. E quando você finalmente decidir integrar suas mudanças com todos os outros, você terá uma fusão muito mais complexa.

A abordagem alternativa é fazer com que todos façam o check-in no mesmo "baú" de código-fonte. Para evitar que as mudanças afetem outras pessoas, técnicas como sinalizadores de recursos são usados para "ocultar" trabalhos incompletos. Essa técnica de todos trabalhar no mesmo tronco é chamado de desenvolvimento baseado em troncos.

A discussão em torno desse tópico é sutil, mas minha opinião é que os benefícios da integração frequente e da validação dessa integração são significativo o suficiente para que o desenvolvimento baseado em troncos seja meu estilo preferido de desenvolvimento. Além disso, o trabalho para implementar sinalizadores de recursos é frequente benéfico em termos de entrega progressiva, um conceito que exploraremos em Capítulo 8.

TENHA CUIDADO COM OS GALHOS

Integre com antecedência e integre com frequência Evite o uso de ramificações de longa duração como recurso desenvolvimento. e considere o desenvolvimento baseado em troncos em vez disso, se você realmente precisar usar galhos, mantenha-os curtos!

Além da minha própria experiência anedótica, há um crescente corpo de pesquisas que mostram a eficácia da redução do número de filiais e adotando o desenvolvimento baseado em troncos. O relatório State of DevOps de 2016 da DORA e Puppet realizam pesquisas rigorosas sobre as práticas de entrega de organizações em todo o mundo e estuda quais práticas são comuns usado por equipes de alto desempenho:

Descobrimos que ter galhos ou garfos com vida útil muito curta (menos de um dia) antes de ser incorporado ao tronco e menos de três ativos filiais no total. são aspectos importantes da entrega contínua. e todos contribuem para um melhor desempenho. O mesmo acontece com a fusão do código no tronco ou domine diariamente.

O relatório State of DevOps continuou a explorar esse tópico com mais profundidade nos anos subsequentes, e continuou a encontrar evidências da eficácia de essa abordagem.

Uma abordagem baseada em ramificações ainda é comum no desenvolvimento de código aberto, muitas vezes através da adoção do modelo de desenvolvimento "GitFlow". Vale a pena notar que o desenvolvimento de código aberto não é o mesmo que o desenvolvimento normal do dia-a-dia. O desenvolvimento de código aberto é caracterizado por um grande número de ad hoc contribuições de comitês "não confiáveis" com pouco tempo, cujas mudanças exigem verificação por um número menor de contribuidores "confiáveis". Típico do dia-a-dia o desenvolvimento de código fechado normalmente é feito por uma equipe unida cujo todos os membros têm direitos de compromisso, mesmo que decidam adotar alguma forma de processo de revisão de código. Então, o que pode funcionar para o desenvolvimento de código aberto pode não trabalhar para o seu trabalho diário. Mesmo assim, o relatório State of DevOps de 2019, explorando ainda mais este tópico, encontrei alguns insights interessantes sobre código aberto desenvolvimento e o impacto de filiais "duradouras":

Nossas descobertas de pesquisa se estendem ao desenvolvimento de código aberto em algumas áreas:

- Comprometer código mais cedo é melhor: em projetos de código aberto, muitos observaram que mesclar patches é mais rápido para evitar rebases, ajuda os desenvolvedores a se moverem mais rápido.
- Trabalhar em pequenos lotes é melhor: grandes "bombas de patch" são mais difícil e mais lento de se fundir em um projeto do que um projeto menor, mais conjuntos de patches legíveis, pois os mantenedores precisam de mais tempo para revisar as mudanças.

Se você está trabalhando em uma base de código-fonte fechado ou em uma base aberta projeto de origem, filiais de curta duração; patches pequenos e legíveis; e o teste automático de mudanças torna todos mais produtivos.

Crie pipelines e entrega contínua

Logo no início de fazer CI, meus colegas da Thoughtworks e eu percebemos o valor de, às vezes, ter vários estágios dentro de uma construção. Os testes são muito caso comum em que isso entra em jogo. Talvez eu tenha muitas coisas rápidas, pequenas... testes com escopo e um pequeno número de testes lentos e de grande escopo. Se corrermos tudo os testes juntos, e se estivermos esperando que nossos testes lentos de grande escopo. Para terminar, talvez não consigamos obter um feedback rápido quando nossos testes rápidos falharem. E se

os testes rápidos falham, provavelmente não faz muito sentido executar os testes mais lentos de qualquer forma! Uma solução para esse problema é ter diferentes estágios em nossa construção, criando o que é conhecido como pipeline de construção. Assim, podemos ter um dedicado estágio para todos os testes rápidos, que executamos primeiro, e se todos passarem, então executamos um estágio separado para os testes mais lentos.

Esse conceito de pipeline de construção nos dá uma boa maneira de acompanhar o progresso de nosso software à medida que ele limpa cada estágio, ajudando a nos dar uma visão sobre a qualidade de nosso software. Criamos um artefato implantável, o que acabará sendo implantado na produção e use esse artefato em todo o pipeline. Em nosso contexto, esse artefato se relacionará a um microsserviço que queremos implantar. Em Figura 7-1, vemos isso acontecendo - o mesmo artefato é usado em cada estágio do pipeline, nos dando cada vez mais confiança de que o software funcionará em produção.

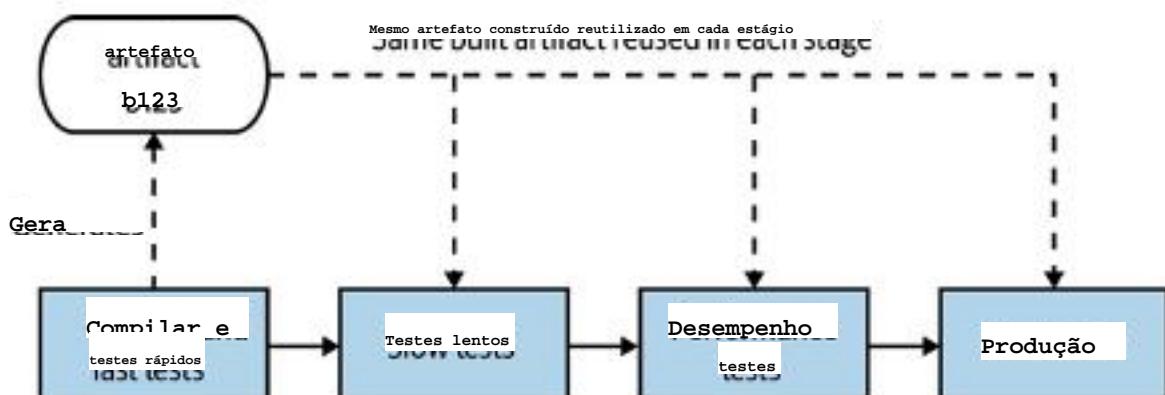


Figura 7-1. Um processo de lançamento simples para nosso serviço de catálogo, modelado como um pipeline de construção

A entrega contínua (CD) se baseia nesse conceito, e muito mais. Conforme descrito no livro homônimo de Jez Humble e Dave Farley, o CD é o abordagem pela qual obtemos feedback constante sobre a prontidão de produção do cada check-in e, além disso, trate cada check-in como um candidato a lançamento.

Para abraçar totalmente esse conceito, precisamos modelar todos os processos envolvidos na obtenção de nosso software desde o check-in até a produção, e precisamos saber onde qualquer versão do software está em termos de liberação liberar. Em CD, fazemos isso modelando cada estágio do nosso software tem que passar, tanto manual quanto automatizado, um exemplo do qual compartilhei

para nosso serviço de catálogo na Figura 7-1. Atualmente, a maioria das ferramentas de CI fornece algum suporte para definir e visualizar o estado dos pipelines de construção, como isso.

Se o novo serviço de catálogo for aprovado, quaisquer verificações sejam realizadas em um estágio no pipeline, ele pode então passar para a próxima etapa. Se não passar de um estágio, nossa ferramenta de CI pode nos informar quais estágios a construção passou e pode chegar visibilidade sobre o que falhou. Se precisarmos fazer algo para consertá-lo, faríamos um altere e faça o check-in, permitindo que a nova versão do nosso microserviço experimente e passe por todas as etapas antes de estar disponível para implantação. Na Figura 7-2, vemos um exemplo disso: build-120 falhou na fase de teste rápido, build-121 falhou nos testes de desempenho, mas o build-122 conseguiu chegar até produção.

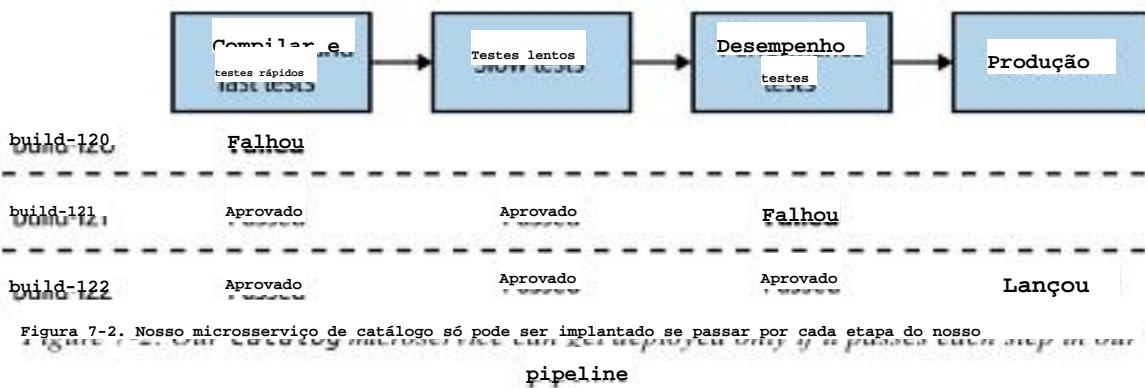


Figura 7-2. Nossa microservice de catálogo só pode ser implantado se passar por cada etapa do nosso pipeline

ENTREGA CONTÍNUA VERSUS ENTREGA CONTÍNUA IMPLANTAÇÃO

Ocasionalmente, vi alguma confusão em torno dos termos contínuos, entrega e implantação contínua. Como já discutimos, entrega contínua é o conceito pelo qual cada check-in é tratado como um candidato a lançamento e por meio do qual podemos avaliar a qualidade de cada versão candidato para decidir se está pronto para ser implantado. Com contínuo implantação, por outro lado, todos os check-ins devem ser validados usando mecanismos automatizados (testes, por exemplo) e qualquer software aprovado essas verificações são implantadas automaticamente, sem humanos intervenção. A implantação contínua pode, portanto, ser considerada uma extensão da entrega contínua. Sem entrega contínua, você não pode fazer implantação contínua. Mas você pode fazer entrega contínua sem fazendo implantação contínua.

A implantação contínua não é adequada para todos – muitas pessoas querem alguma interação humana para decidir se o software deve ser implantado, algo totalmente compatível com a entrega contínua. No entanto, adotar a entrega contínua implica um foco contínuo na otimização seu caminho até a produção, a maior visibilidade facilitando a visualização onde as otimizações devem ser feitas. Freqüentemente, o envolvimento humano no o processo pós-check-in é um gargalo que precisa ser resolvido – veja a mudança do teste de regressão manual ao teste funcional automatizado, para exemplo. Como resultado, à medida que você automatiza cada vez mais sua construção, processo de implantação e liberação, você pode se aproximar e mais perto da implantação contínua.

Ferramentas

Idealmente, você quer uma ferramenta que adote a entrega contínua como uma ferramenta de primeira classe conceito. Já vi muitas pessoas tentarem hackear e estender as ferramentas de CI para criar eles fazem CD, geralmente resultando em sistemas complexos que estão longe de fáceis de usar, pois são ferramentas integradas em CD desde o início. Ferramentas que são totalmente

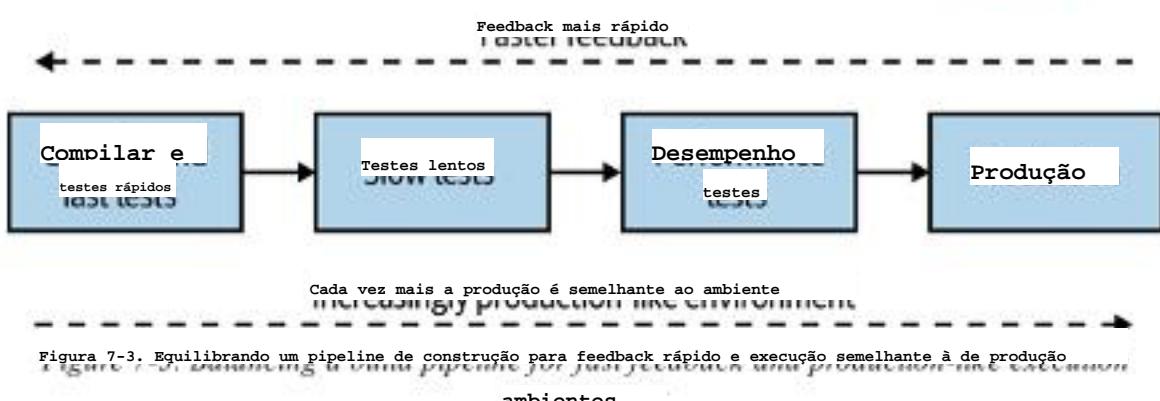
O CD de suporte permite que você defina e visualize esses pipelines, modelando o caminho completo até a produção do seu software. À medida que uma versão do nosso código se move através do pipeline, se ele passar por um desses estágios de verificação automatizada, passa para o próximo estágio.

Alguns estágios podem ser manuais. Por exemplo, se tivermos um usuário manual processo de teste de aceitação (UAT), eu deveria ser capaz de usar uma ferramenta de CD para modelar isso. Eu posso ver a próxima versão disponível pronta para ser implantada em nosso UAT ambiente e, em seguida, implantá-lo e, se ele passar em nossas verificações manuais, marque esse estágio foi bem-sucedido para que ele possa passar para o próximo. Se o subsequente estágio é automatizado e, em seguida, será acionado automaticamente.

Compensações e ambientes

À medida que movemos nosso artefato de microserviço por esse pipeline, nosso microserviço é implantado em diferentes ambientes. Diferentes ambientes servem a propósitos diferentes e podem ter diferentes características.

Estruturar um pipeline e, portanto, descobrir quais ambientes você usará a necessidade é, por si só, um ato de equilíbrio. Logo no início do pipeline, estamos procurando um feedback rápido sobre a prontidão de produção do nosso software. Nós queremos que os desenvolvedores saibam o mais rápido possível se houver algum problema - o quanto mais cedo recebermos feedback sobre um problema que está ocorrendo, mais rápido será corrigi-lo. À medida que nosso software se aproxima da produção, queremos mais certeza de que o software funcionará e, portanto, estaremos implantando cada vez mais ambientes semelhantes à produção - podemos ver essa compensação na Figura 7-3.



Você recebe o feedback mais rápido sobre seu laptop de desenvolvimento, mas isso está longe de semelhante a uma produção. Você pode implantar cada compromisso em um ambiente que seja reprodução fiel de seu ambiente de produção real, mas isso será provavelmente demoram mais e custam mais. Então, encontrar o equilíbrio é fundamental, e continuar analisando a compensação entre feedback rápido e a necessidade de ambientes semelhantes à produção podem ser um processo contínuo extremamente importante atividade.

Os desafios de criar um ambiente semelhante ao de produção também fazem parte do por que mais pessoas estão fazendo formas de testes na produção, incluindo técnicas como testes de fumaça e corridas paralelas. Vamos voltar a isso tópico no Capítulo 8.

Criação de artefatos

À medida que movemos nosso microsserviço para diferentes ambientes, na verdade temos ter algo para implantar. Acontece que existem vários opções para o tipo de artefato de implantação que você pode usar. Em geral, qual o artefato que você criar dependerá muito da tecnologia que você escolheu adote para implantação. Analisaremos isso em profundidade no próximo capítulo, mas eu queria dar algumas dicas muito importantes sobre como criar artefatos deve se encaixar no seu processo de construção de CI/CD.

Para manter as coisas simples, evitaremos exatamente o tipo de artefato que somos criando - basta considerá-lo um único blob implantável no momento. Agora, há duas regras importantes que precisamos considerar. Em primeiro lugar, como mencionei Antes, deveríamos construir um artefato apenas uma vez. Construindo o mesmo Uma e outra vez é uma perda de tempo e é ruim para o planeta, e pode teoricamente, introduzir problemas se a configuração de compilação não for exatamente a o mesmo de cada vez. Em algumas linguagens de programação, um sinalizador de construção diferente pode faça com que o software se comporte de forma bem diferente. Em segundo lugar, o artefato que você verifica deve ser o artefato que você usa! Se você criar um microsserviço, teste-o, digamos "Sim, está funcionando" e, em seguida, construa-o novamente para implantação na produção, como você sabe que o software que você validou é o mesmo software que você implantado?

Juntando essas duas ideias, temos uma abordagem bem simples. Construir seu artefato implantável apenas uma vez e, idealmente, faça-o bem no início o oleoduto. Eu normalmente faria isso depois de compilar o código (se necessário), e executando meus testes rápidos. Depois de criado, esse artefato é armazenado em um repositório apropriado - isso pode ser algo como Artifactory ou Nexus, ou talvez um registro de contêiner. Provavelmente sua escolha de artefato de implantação determina a natureza da loja de artefatos. Esse mesmo artefato pode então ser usado para todas as etapas do pipeline que se seguem, incluindo a implantação no produção. Então, voltando ao nosso pipeline anterior, podemos ver na Figura 7-4 que criemos um artefato para nosso serviço de catálogo durante a primeira etapa do canalize e, em seguida, implante o mesmo artefato 123, mas antigo, como parte do lento testes, testes de desempenho e estágios de produção.

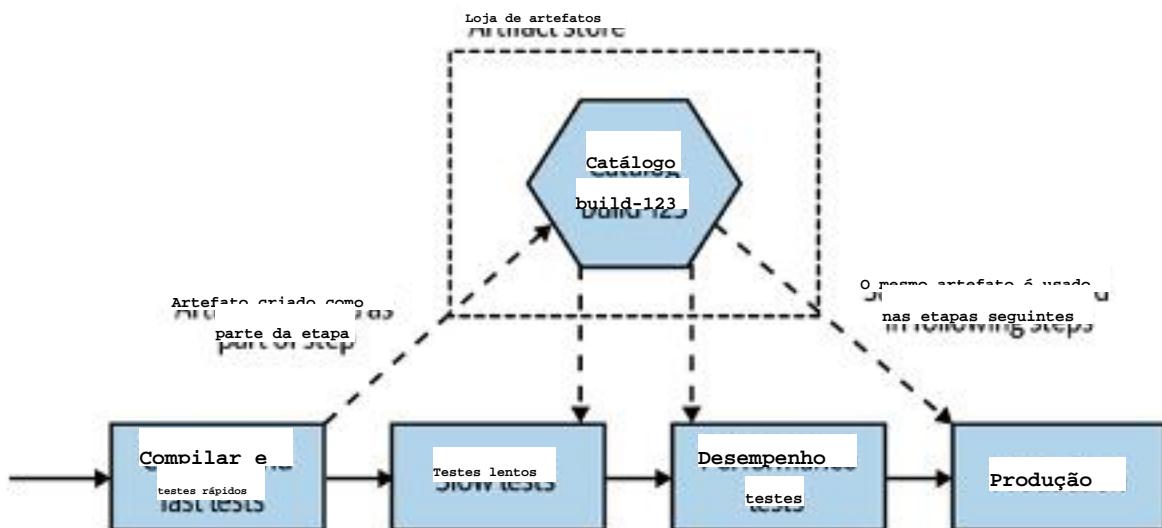


Figura 7-4. O mesmo artefato é implantado em cada ambiente

Se o mesmo artefato for usado em vários ambientes, qualquer aspectos da configuração que variam de ambiente para ambiente precisam seja mantido fora do próprio artefato. Como um exemplo simples, talvez eu queira configure os logs do aplicativo para que tudo no nível DEBUG e superior seja registrado ao executar o estágio de testes lentos, fornecendo mais informações para diagnosticar por que um teste falha. Eu poderia decidir, porém, mudar isso para INFO para reduzir o volume de registros para a implantação de testes de desempenho e produção.

DICAS DE CRIAÇÃO DE ARTEFATOS

Crie um artefato de implantação para seu microserviço uma vez. Reutilize esse artefato em todos os lugares você deseja implantar essa versão do seu microserviço. Mantenha seu artefato de implantação ambiente-agnostic-store a configuração específica do ambiente em outro lugar.

Mapeando código-fonte e compilações para Microsserviços

Já analisamos um tópico que pode estimular o recurso de facções em guerra ramificação versus desenvolvimento baseado em troncos, mas acontece que a controvérsia não acabou neste capítulo. Outro tópico que provavelmente suscitará algumas opiniões bastante diversas são a organização do código para nossos microsserviços. Eu tenho minhas próprias preferências, mas antes de chegarmos a elas, vamos explore as principais opções de como organizamos o código para nossos microsserviços.

Um repositório gigante, uma construção gigante

Se começarmos com a opção mais simples, poderíamos agrupar tudo. Nós temos um único repositório gigante armazenando todo o nosso código, e temos um único construa, como vemos na Figura 7-5. Qualquer check-in neste repositório de código-fonte fará com que nossa compilação seja acionada, onde executaremos todas as etapas de verificação associados a todos os nossos microsserviços e produzem vários artefatos, todos vinculados de volta à mesma construção.

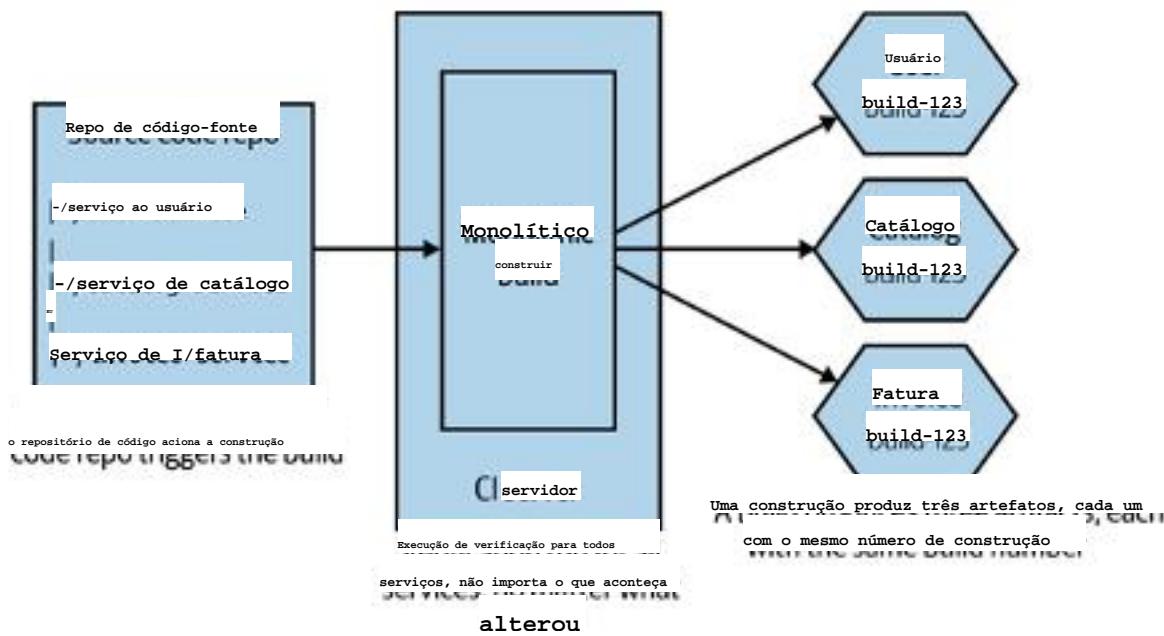


Figura 7-5. Usando um único repositório de código-fonte e uma compilação de CI para todos os microserviços

Em comparação com outras abordagens, isso parece muito mais simples na superfície:

menos repositórios com os quais se preocupar e uma construção conceitualmente mais simples.
Do ponto de vista do desenvolvedor, as coisas também são bem simples. Eu só
verifique o código em. Se eu tiver que trabalhar em vários serviços ao mesmo tempo, eu só preciso
preocupar-se com um compromisso.

Este modelo pode funcionar perfeitamente bem se você acreditar na ideia de lockstep
lançamentos, nos quais você não se importa em implantar vários serviços ao mesmo tempo. Em
geral, esse é absolutamente um padrão a ser evitado, mas logo no início de um projeto,
especialmente se apenas uma equipe estiver trabalhando em tudo, esse modelo pode fazer
sentido por curtos períodos de tempo.

Agora, deixe-me explicar algumas das desvantagens significativas dessa abordagem. Se eu
faça uma alteração de uma linha em um único serviço, por exemplo, alterando o comportamento
no serviço de usuário na Figura 7-5, todos os outros serviços são verificados e
construídos. Isso pode levar mais tempo do que o necessário - estou esperando por coisas que
provavelmente não precisam ser testados. Isso afeta nosso tempo de ciclo, a velocidade em
onde podemos mover uma única mudança do desenvolvimento para a vida. Mais
preocupante, porém, é saber quais artefatos devem ou não ser implantados.

Agora preciso implantar todos os serviços de compilação para inserir minha pequena alteração em
produção? Pode ser difícil dizer; tentar adivinhar quais serviços realmente

mudar apenas lendo as mensagens de confirmação é difícil. Organizações que usam essa abordagem geralmente se resumem a apenas implantar tudo junto, o que nós realmente queremos evitar.

Além disso, se minha alteração de uma linha no serviço de usuário interromper a compilação, não outras alterações podem ser feitas nos outros serviços até que a interrupção seja corrigida. E pense em um cenário em que você tenha várias equipes, todas compartilhando esse gigante construir. Quem está no comando?

Indiscutivelmente, essa abordagem é uma forma de monorepo. Na prática, no entanto, a maioria das implementações de monorepo que eu vi mapeiam várias compilações para diferentes partes do repositório, algo que exploraremos com mais profundidade em breve. Então você poderia ver esse padrão de mapeamento de um repositório para uma única compilação como a pior forma do monorepo para aqueles que desejam criar vários implantáveis de forma independente microsserviços.

Na prática, quase nunca vejo essa abordagem usada, exceto nas primeiras etapas dos projetos. Para ser honesto, qualquer uma das duas abordagens a seguir é significativamente preferível, então vamos nos concentrar neles em vez disso.

Padrão: um repositório por microsserviço (também conhecido como (Multirepositório))

Com o padrão de um repositório por microsserviço (mais comumente chamado como o padrão multirepo (quando comparado ao padrão monorepo), o código para cada microsserviço é armazenado em seu próprio repositório de código-fonte, como vemos na Figura 7-6. Essa abordagem leva a um mapeamento direto, entre alterações no código-fonte e construções de CI.

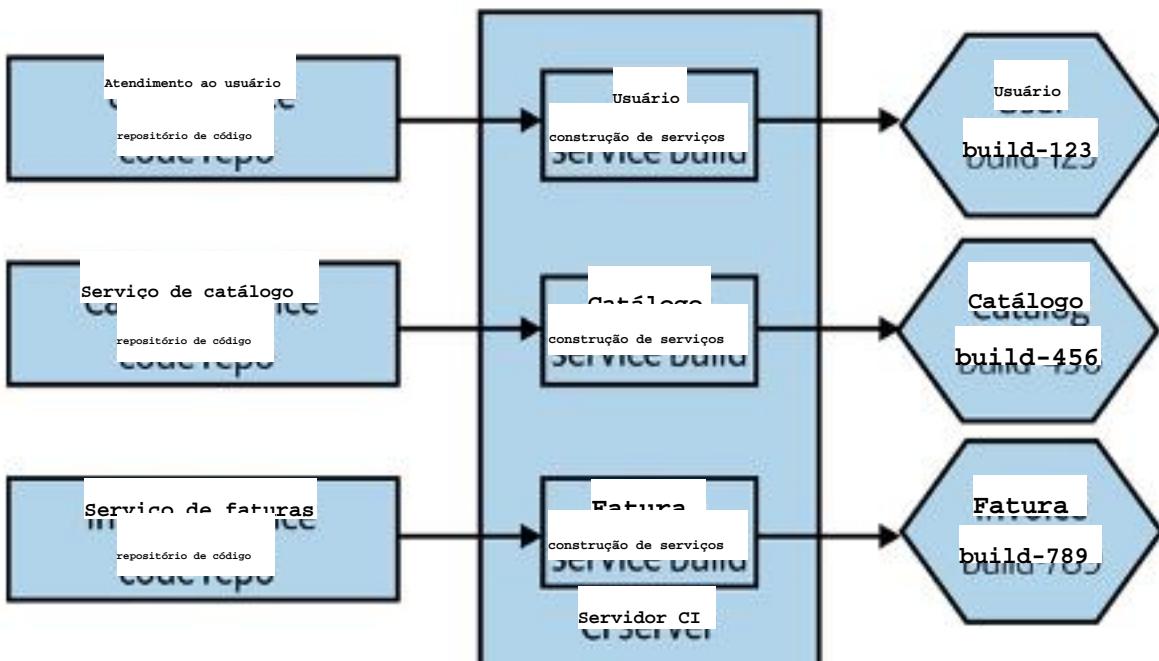


Figura 7-6. O código-fonte de cada microsserviço é armazenado em um repositório de código-fonte separado.

Qualquer alteração no repositório de código-fonte do usuário aciona a compilação correspondente, e se isso passar, terei uma nova versão do meu microsserviço de usuário, disponível para implantação. Ter um repositório separado para cada microsserviço também permite que você altere a propriedade por repositório, algo que faz sentido se você quiser considerar um modelo de propriedade forte para seu microsserviços (mais sobre isso em breve).

A natureza direta desse padrão cria alguns desafios, no entanto. Especificamente, os desenvolvedores podem trabalhar com vários repositórios por vez, o que é especialmente doloroso se eles estiverem tentando fazer alterações em vários repositórios ao mesmo tempo. Além disso, mudanças não pode ser feito de forma atômica em repositórios separados, pelo menos não com o Git.

Reutilizando código em repositórios

Ao usar esse padrão, não há nada que impeça um microsserviço de depender de outro código que é gerenciado em diferentes repositórios. Um simples mecanismo para fazer isso é ter o código que você deseja reutilizar empacotado em uma biblioteca que então se torna uma dependência explícita do downstream.

microserviços. Podemos ver um exemplo disso na Figura 7-7, onde o ...
os serviços de fatura e folha de pagamento usam a biblioteca Connection.

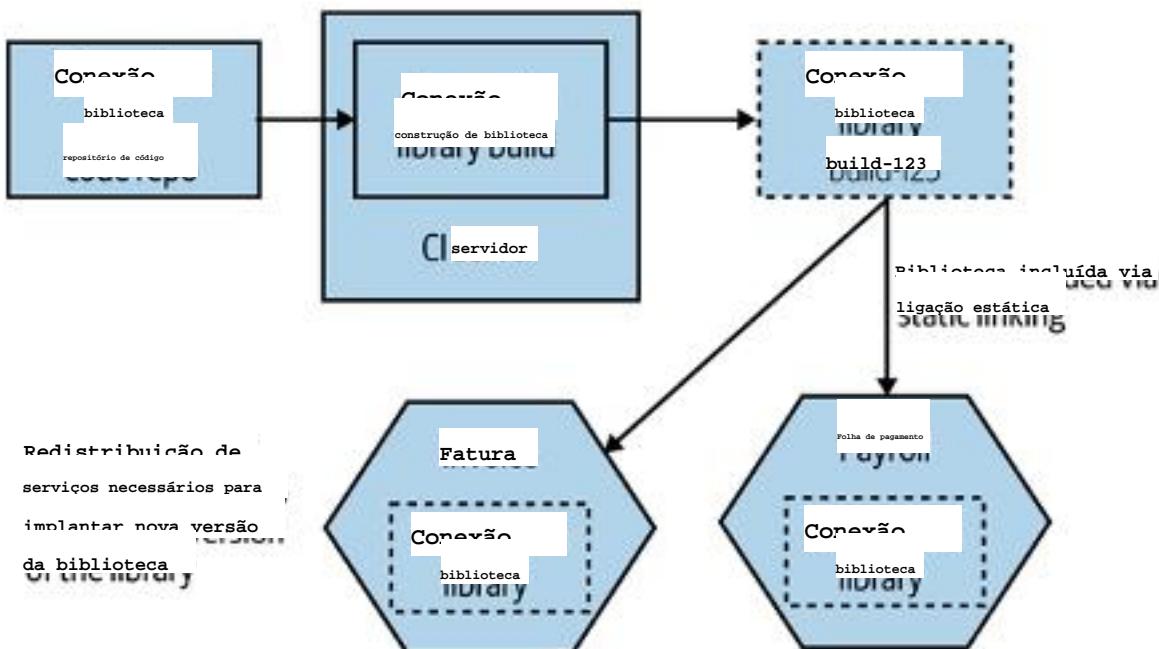


Figura 7-7 Reutilizando código em diferentes repositórios.

Se você quisesse implementar uma alteração na biblioteca do Connection, você teria que ...
faca as alterações no repositório de código-fonte correspondente e aguarde seu ...
construa para ser concluído, oferecendo a você um novo artefato versionado. Para realmente implantar ...
novas versões dos serviços de fatura ou folha de pagamento usando esta nova versão do ...
a biblioteca, você precisaria alterar a versão da biblioteca Connection ...
usar. Isso pode exigir uma alteração manual (se você depender de um item específico). ...
versão), ou pode ser configurado para acontecer dinamicamente, dependendo da ...
natureza das ferramentas de CI que você está usando. Os conceitos por trás disso são descritos ...
mais detalhadamente no livro *Continuous Delivery* de Jez Humble e Dave ...
Farley.⁴

O importante a lembrar, é claro, é que, se você quiser lançar o ...
nova versão da biblioteca Connection, então você também precisa implantar o ...
serviços de fatura e folha de pagamento recém-construídos. Lembre-se, todas as ressalvas que temos ...
explorado em "DRY e os perigos da reutilização de código em um mundo de microserviços" ...
em relação à reutilização e aos microserviços ainda se aplicam - se você optar por reutilizar o código ...
por meio de bibliotecas, então você deve concordar com o fato de que essas mudanças não podem ser ...

lançado de forma atômica, ou então minaremos nosso objetivo de capacidade de implantação independente. Você também precisa estar ciente de que pode ser mais difícil saber se alguns microserviços estão usando uma versão específica de um biblioteca, o que pode ser problemático se você estiver tentando descontinuar o uso de uma versão antiga da biblioteca.

Trabalhando em vários repositórios

Então, além de reutilizar o código por meio de bibliotecas, de que outra forma podemos fazer uma mudança em mais de um repositório? Vamos dar uma olhada em outro exemplo. Na Figura 7-8, quero alterar a API exposta pelo serviço de inventário e também preciso atualizar o serviço de envio para que ele possa fazer uso da nova alteração. Se o código para inventário e envio estava no mesmo repositório, eu poderia confirmar o código uma vez. Em vez disso, terei que dividir as mudanças em duas confirmações: uma para inventário e outra para envio.

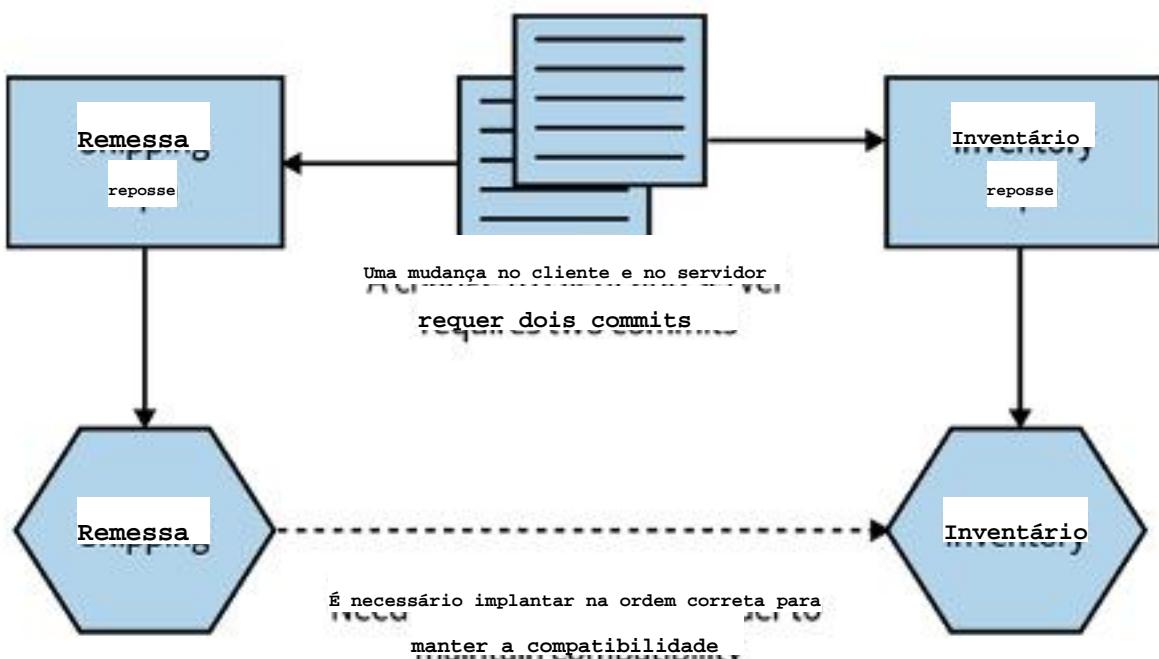


Figura 7-8. Mudanças nos limites do repositório exigem vários commits.

A divisão dessas alterações pode causar problemas se um commit falhar, mas o outros trabalhos - talvez eu precise fazer duas alterações para reverter a alteração, pois exemplo, e isso pode ser complicado se outras pessoas fizerem o check-in enquanto isso. A realidade é que, nessa situação específica, eu provavelmente gostaria

Em qualquer caso, prepare um pouco os commits. Eu gostaria de ter certeza de que o compromisso para alterar o serviço de inventário funcionou antes de eu alterar qualquer código de cliente em o serviço de envio - se a nova funcionalidade na API não estiver presente, aí não adianta ter um código de cliente que faça uso dele.

Falei com várias pessoas que consideram a falta de implantação atômica com isso deve ser um problema significativo. Eu certamente posso apreciar a complexidade disso traz, mas acho que na maioria dos casos isso aponta para um problema subjacente maior. Se você está continuamente fazendo alterações em vários microsserviços e, em seguida, em seu os limites do serviço podem não estar no lugar certo, e isso também pode implicar muito acoplamento entre seus serviços. Como já discutimos, somos tentando otimizar nossa arquitetura e nossos limites de microsserviços, para que é mais provável que as mudanças se apliquem dentro de um limite de microsserviços. Mudanças transversais devem ser a exceção, não a norma.

Na verdade, eu diria que a dor de trabalhar com vários repositórios pode ser útil em ajudar a impor limites de microsserviços, pois isso força você a pensar cuidadosamente sobre onde estão esses limites e sobre a natureza das interações entre eles.

GORJETA

Se você está constantemente fazendo alterações em vários microsserviços, é provável que seus os limites do microsserviço estão no lugar errado. Pode valer a pena considerar a fusão microsserviços juntos novamente se você perceber que isso está acontecendo.

Depois, há o incômodo de precisar extrair de vários repositórios e enviar para vários repositórios como parte de seu fluxo de trabalho normal. Na minha experiência, isso pode seja simplificado usando um IDE que suporte vários repositórios (este é algo que todos os IDEs que usei nos últimos cinco anos podem lidar), ou por escrever scripts de wrapper simples para simplificar as coisas ao trabalhar no linha de comando.

Onde usar esse padrão

Usar a abordagem de um repositório por microsserviço também funciona bem para equipes pequenas, assim como acontece com equipes grandes, mas se você estiver fazendo muitas mudas além dos limites do microsserviço, então pode não ser para você, e para o padrão monorepo que discutiremos a seguir pode ser mais adequado, embora faça lotes de mudanças além dos limites de serviço pode ser considerado um sinal de alerta de que algo não está certo, como discutimos anteriormente. Ele também pode criar código reutilize de forma mais complexa do que usar uma abordagem monorepo, pois você precisa depender no código que está sendo empacotado em artefatos de versão.

Padrão: Monorepo

Com uma abordagem monorepo, codifique para vários microsserviços (ou outros tipos) de projetos) é armazenado no mesmo repositório de código-fonte. Eu vi situações em que um monorepo é usado apenas por uma equipe para gerenciar a fonte controle de todos os seus serviços, embora o conceito tenha sido popularizado por algumas empresas de tecnologia muito grandes onde várias equipes e centenas, se não milhares de desenvolvedores podem trabalhar no mesmo repositório de código-fonte.

Ao ter todo o código-fonte no mesmo repositório, você permite o código-fonte alterações de código a serem feitas em vários projetos de forma atômica, e para uma reutilização mais refinada do código de um projeto para o outro. O Google é provavelmente o exemplo mais conhecido de uma empresa usando uma abordagem monorepo, embora esteja longe de ser o único. Embora existam alguns outros benefícios para essa abordagem, como melhor visibilidade do código de outras pessoas, a capacidade para reutilizar o código com facilidade e fazer alterações que afetem vários tipos projetos são frequentemente citados como o principal motivo para a adoção desse padrão.

Se tomarmos o exemplo que acabamos de discutir, onde queremos fazer uma mudança ao inventário para que ele exponha algum novo comportamento e também atualize o Serviço de remessa para fazer uso dessa nova funcionalidade que expusemos, então essas alterações podem ser feitas em um único commit, como vemos na Figura 7-9.

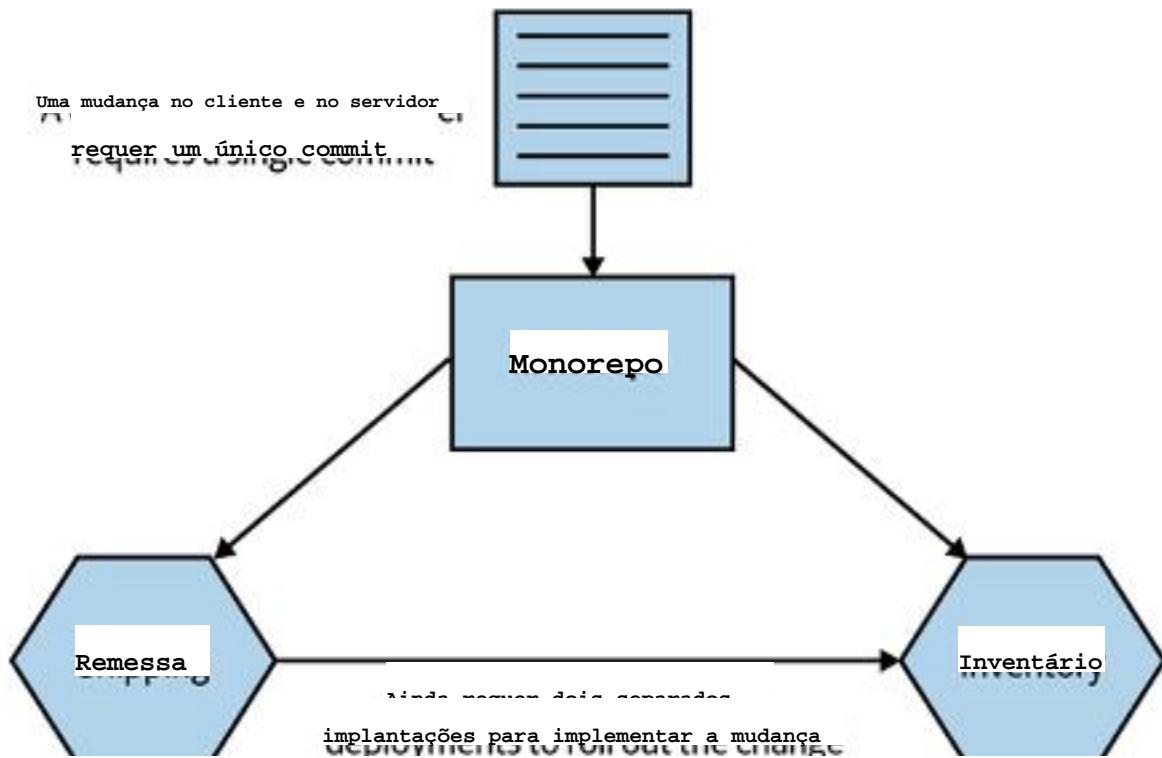


Figura 7-9. Usando um único commit para fazer alterações em dois microserviços usando um monorepo

Obviamente, como no padrão multirepo discutido anteriormente, ainda precisamos para lidar com o lado da implantação disso. Provavelmente precisaríamos ter cuidado considerar a ordem de implantação se quisermos evitar uma implantação em etapas.

CONFIRMAÇÕES ATÔMICAS VERSUS IMPLANTAÇÃO ATÔMICA

Ser capaz de fazer um commit atômico em vários serviços não fornece dados atômicos de lançamento. Se você quiser alterar o código em vários serviços ao mesmo tempo e lançá-lo na produção ao mesmo tempo, isso viola o princípio fundamental da independência de capacidade de implantação. Para saber mais sobre isso, consulte "DRY e os perigos da reutilização de código em um microserviço mundo".

Mapeamento para construir

Com um único repositório de código-fonte por microserviço, mapeando a partir do código-fonte para um processo de construção é simples. Qualquer mudança nessa fonte de repositório de código pode acionar uma compilação de CI correspondente. Com um monorepo, ele recebe um pouco mais complexo.

Um ponto de partida simples é mapear pastas dentro do monorepo para uma construção, como mostrado na Figura 7-10. Uma alteração feita na pasta de serviço do usuário seria acionar a compilação do serviço de usuário, por exemplo. Se você verificou no código que arquivos alterados na pasta user - service e no catalog-service pasta, então a compilação do usuário e a compilação do catálogo seriam acionadas.

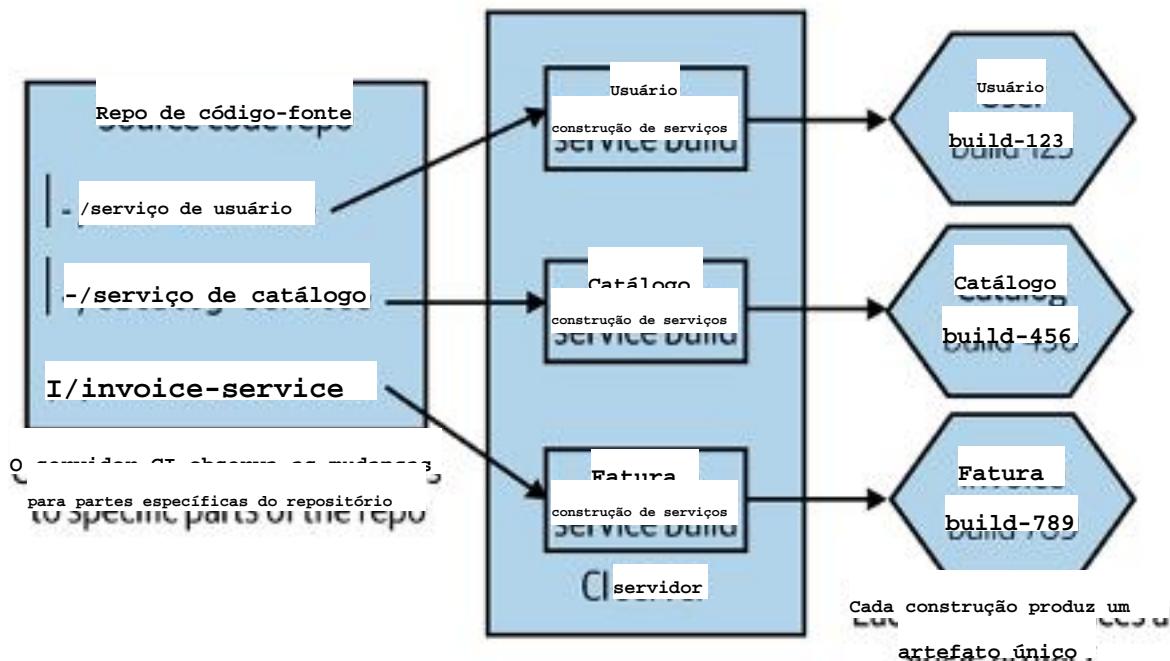


Figura 7-10. Um único repositório de origem com subdiretórios mapeados para compilações independentes

Isso fica mais complexo à medida que você tem estruturas de pastas mais envolvidas. Ligado projetos maiores, você pode acabar com várias pastas diferentes querendo aciona a mesma compilação e com algumas pastas acionando mais de uma compilação. Na extremidade simples do espectro, você pode ter uma pasta "comum" usada por todos os microserviços, uma mudança que faz com que todos os microserviços sejam reconstruído. No lado mais complexo, as equipes acabam precisando adotar mais gráficos ferramentas de construção baseadas, como a ferramenta de código aberto Bazel, para gerenciá-las dependências de forma mais eficaz (o Bazel é uma versão de código aberto do Google própria ferramenta de construção interna). A implementação de um novo sistema de construção pode ser uma empreendimento significativo, então não é algo a ser feito levianamente, mas O monorepo do próprio Google seria impossível sem ferramentas como essa.

Um dos benefícios de uma abordagem monorepo é que podemos praticar melhor reutilização granulada em todos os projetos. Com um modelo multirepo, se eu quiser reutilizar

código de outra pessoa, provavelmente terá que ser empacotado como um código versionado artefato que posso então incluir como parte da minha construção (como um pacote Nuget, um Arquivo JAR ou um NPM). Com nossa unidade de reutilização sendo uma biblioteca, somos potencialmente extraíndo mais código do que realmente queremos. Teoricamente, com um monorepo Eu poderia depender apenas de um único arquivo fonte de outro projeto-embora isso, é claro, faça com que eu tenha um mapeamento de compilação mais complexo.

Definindo propriedade

Com equipes menores e tamanhos pequenos de base de código, os monorepos provavelmente podem funcionar bem com as ferramentas tradicionais de gerenciamento de compilação e código-fonte que você está acostumado. No entanto, à medida que seu monorepo fica maior, você provavelmente precisará para começar a analisar diferentes tipos de ferramentas. Exploraremos os modelos de propriedade em mais detalhes no Capítulo 15, mas, enquanto isso, vale a pena explorar brevemente como isso acontece quando pensamos em controle de origem.

Martin Fowler já escreveu sobre diferentes modelos de propriedade, delineando uma escala móvel de propriedade, da propriedade forte à fraca propriedade e, em seguida, propriedade coletiva. Desde que Martin os capturou termos, as práticas de desenvolvimento mudaram, então talvez valha a pena revisá-lo e redefinindo esses termos.

Com uma forte propriedade, alguns códigos pertencem a um grupo específico de pessoas. Se alguém de fora desse grupo quer fazer uma mudança, eles têm que perguntar ao proprietários para fazer a mudança por eles. A propriedade fraca ainda tem o conceito de proprietários definidos, mas pessoas fora do grupo de propriedade podem fazer mudanças, embora qualquer uma dessas alterações deva ser revisada e aceita por alguém do grupo proprietário. Isso cobriria o envio de uma solicitação de pull para a equipe principal de propriedade para análise, antes que a pull request seja mesclada.

Com a propriedade coletiva, qualquer desenvolvedor pode alterar qualquer parte do código.

Com um pequeno número de desenvolvedores (20 ou menos, como guia geral), você pode se dar ao luxo de praticar a propriedade coletiva, onde qualquer desenvolvedor pode mudar qualquer outro microserviço. Porém, à medida que você tem mais pessoas, é mais provável que querer avançar em direção a um modelo de propriedade forte ou fraco para criar limites de responsabilidade mais definidos. Isso pode causar um desafio para

equipes que usam monorepos se sua ferramenta de controle de origem não oferecer suporte a finos controles de propriedade granulados.

Algumas ferramentas de código-fonte permitem que você especifique a propriedade de um determinado diretórios ou até mesmo caminhos de arquivo específicos dentro de um único repositório. Google implementou inicialmente este sistema em cima do Perforce para seu próprio monorepo antes de desenvolver seu próprio sistema de controle de origem, e também é algo que o GitHub tem suporte desde 2016. Com o GitHub, você cria um CODEONNERS arquivo, que permite mapear proprietários para diretórios ou caminhos de arquivo. Você pode ver alguns exemplos no Exemplo 7-1, extraídos da própria documentação do GitHub, que mostre os tipos de flexibilidade que esses sistemas podem oferecer.

Exemplo 7-1. Exemplos de como especificar a propriedade em diretórios específicos em um arquivo CODEONNERS do GitHub

```
# Neste exemplo, @doctocat possui todos os arquivos na construção/registros
# diretório na raiz do repositório e em qualquer um de seus
# subdiretórios.
/build/logs/ @doctocat

# Neste exemplo, @octocat possui qualquer arquivo em um diretório de aplicativos
# em qualquer lugar do seu repositório.
aplicativos/ @octocat

# Neste exemplo, @doctocat possui qualquer arquivo no `/docs
# diretório na raiz do seu repositório
/docs/ @doctocat

O próprio conceito de propriedade de código do GitHub garante que os proprietários do código sejam a fonte os arquivos são solicitados para revisão sempre que uma solicitação pull é gerada para o arquivos relevantes. Isso pode ser um problema com solicitações de pull maiores, pois você pode acabar precisando da aprovação de vários revisores, mas há muitas boas razões para buscar solicitações de pull menores, em qualquer caso.
```

Ferramentas

O monorepo do próprio Google é enorme e exige quantidades significativas de engenharia para fazer com que funcione em grande escala. Considere coisas como um baseado em gráficos sistema de construção que passou por várias gerações, um objeto distribuído vinculador para acelerar os tempos de construção, plug-ins para IDEs e editores de texto que podem manter dinamicamente os arquivos de dependência sob controle - é uma quantidade enorme de

trabalhar. À medida que o Google crescia, cada vez mais enfrentava limitações no uso do Perforce e acabou tendo que criar sua própria ferramenta proprietária de controle de fonte chamada Piper. Quando trabalhei nessa parte do Google em 2007-2008, havia mais de cem pessoas mantendo várias ferramentas para desenvolvedores, com uma parte significativa desse esforço se dedicou a lidar com as implicações da abordagem monorepo. Isso é algo que você pode justificar se tiver dezenas de milhares de engenheiros, é claro.

Para uma visão geral mais detalhada da lógica por trás do uso de um monorepo, eu recomendo "Por que o Google armazena bilhões de linhas de código em um Repositório único", de Rachel Potvin e Josh Levenberg.⁵ Na verdade, eu sugiro que seja uma leitura obrigatória para quem pensa: "Devemos usar um monorepo, porque o Google faz!" Sua organização provavelmente não é o Google e provavelmente não tem problemas, restrições ou recursos do tipo Google. Dito de outra forma, qualquer monorepo com o qual você acabe provavelmente não será do Google.

A Microsoft teve problemas semelhantes com a escala. Adotou o Git para ajudar a gerenciar o monodiretório principal do código-fonte para Windows. Um trabalho completo. O diretório dessa base de código tem cerca de 270 GB de arquivos de origem.⁶ Baixando tudo isso levaria um tempo, e também não é necessário - os desenvolvedores o farão acabam trabalhando em apenas uma pequena parte do sistema geral. Então, Microsoft teve que criar um sistema de arquivos virtual dedicado, VFS for Git (anteriormente conhecido como GVFS), que garante que apenas os arquivos de origem de que um desenvolvedor precisa sejam realmente baixados.

O VFS for Git é uma conquista impressionante, assim como a própria cadeia de ferramentas do Google, embora justificar esses tipos de investimentos nesse tipo de tecnologia seja muito mais fácil para empresas como essa. Também vale ressaltar que, embora o VFS for Git é de código aberto, ainda não conheci uma equipe fora da Microsoft que o use - e a maior parte da cadeia de ferramentas do Google que suporta seu monorepo é código fechado (o Bazel é uma exceção notável, mas não está claro até que ponto o código aberto Bazel, na verdade, reflete o que é usado dentro do próprio Google).

O artigo de Markus Oberlehner, "Monorepos in the Wild", me apresentou a Lerna, uma ferramenta criada pela equipe por trás do compilador JavaScript Babel.

O Lerna foi projetado para facilitar a produção de vários artefatos versionados do mesmo repositório de código-fonte. Não posso falar diretamente sobre a eficácia Lerna está nessa tarefa (além de várias outras deficiências notáveis, eu Eu não sou um desenvolvedor de JavaScript experiente), mas parece que vem de uma superfície exame para simplificar um pouco essa abordagem.

Como "mono" é mono?

O Google não armazena todo o seu código em um monorepo. Existem alguns projetos especialmente aqueles que estão sendo desenvolvidos ao ar livre, que são realizados em outros lugares. No entanto, pelo menos com base no artigo ACM mencionado anteriormente, 95% do código do Google foi armazenado no monorepo a partir de 2016. Em outras organizações, um monorepo pode ter como escopo apenas um sistema ou um pequeno número de sistemas. Isso significa que uma empresa pode ter um pequeno número de monrepos para diferentes partes da organização.

Também falei com equipes que praticam monrepos por equipe. Enquanto tecnicamente falando, isso provavelmente não corresponde à definição original desse padrão (que normalmente fala em termos de várias equipes compartilhando o mesmo repositório), ainda acho que é mais "monorepo" do que qualquer outra coisa. Neste situação, cada equipe tem seu próprio monorepo que está totalmente sob seu controle. Todos os microserviços pertencentes a essa equipe têm seu código armazenado no da equipe monorepo, conforme mostrado na Figura 7-11.

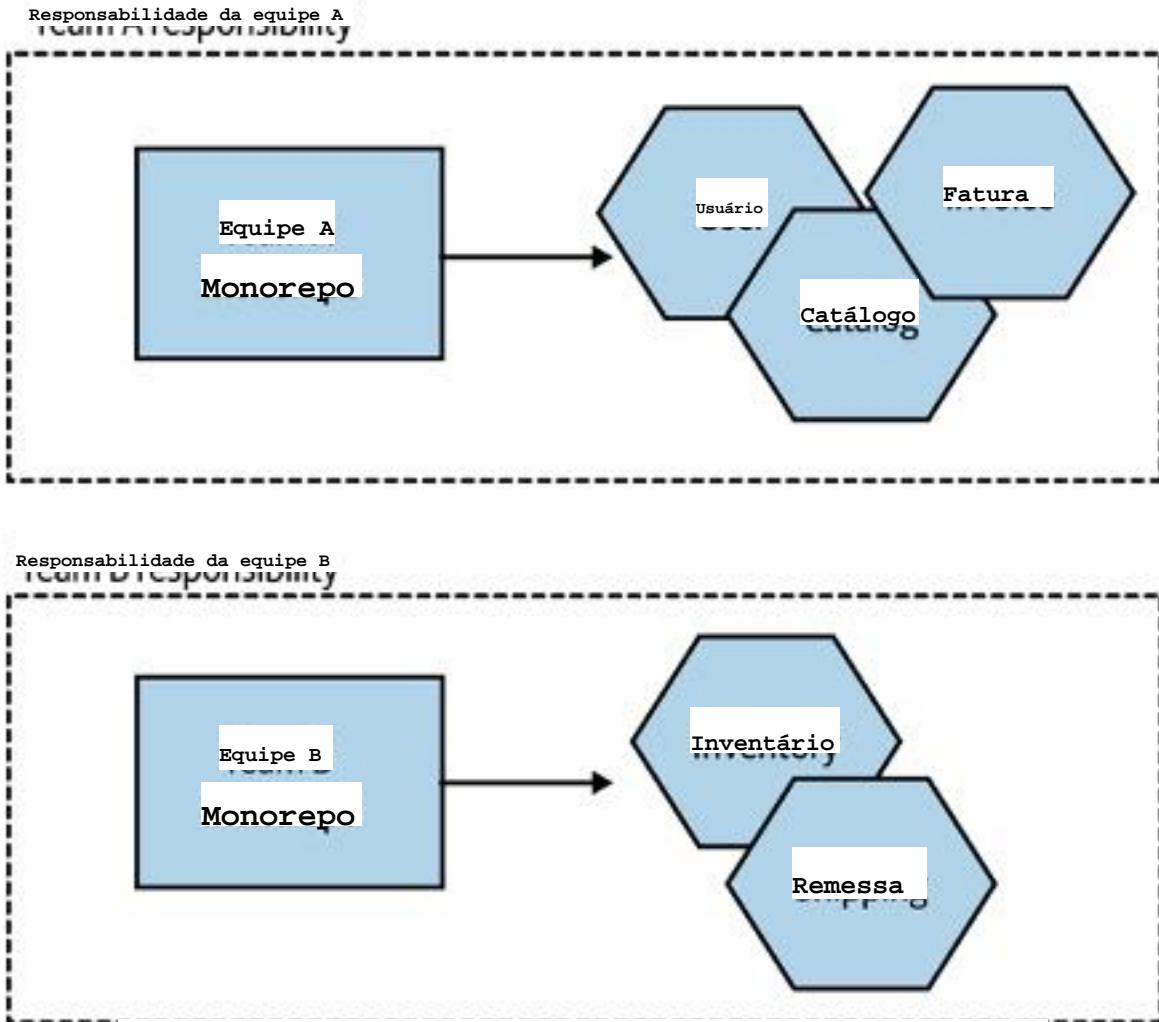


Figura 7-11. Uma variação de padrão em que cada equipe tem seu próprio monorepo.

Para equipes que praticam a propriedade coletiva, esse modelo tem muitos benefícios, sem dúvida, fornecendo a maioria das vantagens de uma abordagem monorepo, enquanto evitando alguns dos desafios que ocorrem em maior escala. Essa metade do caminho casa pode fazer muito sentido em termos de trabalhar dentro de uma casa existente, limites de propriedade organizacional, e isso pode mitigar um pouco as preocupações sobre o uso desse padrão em maior escala.

Onde usar esse padrão

Algumas organizações que trabalham em grande escala encontraram o monorepo abordagem para funcionar muito bem para eles. Já mencionamos o Google e Microsoft, e podemos adicionar Facebook, Twitter e Uber à lista. Essas todas as organizações têm uma coisa em comum: são grandes e focadas em tecnologia.

empresas que são capazes de dedicar recursos significativos para obter o melhor fora desse padrão. Onde vejo que os monorepos funcionam bem é na outra extremidade do espectro, com um número menor de desenvolvedores e equipes. Com 10 a 20 desenvolvedores, é mais fácil gerenciar os limites de propriedade e manter a construção processo simples com a abordagem monorepo. Os pontos problemáticos parecem surgir para organizações intermediárias – aquelas com escala para começar a abordar questões que exigem novas ferramentas ou formas de trabalhar, mas sem a largura de banda extra para invista nessas ideias.

Qual abordagem usaria | ?

Na minha experiência, as principais vantagens de uma abordagem monorepo são mais finas-reutilização granulada e compromissos atómicos não parecem superar os desafios que surgem em grande escala. Para equipes menores, qualquer abordagem é boa, mas como você escala, acho que a abordagem de um repositório por microserviço (multirepositórios) é mais simples. Fundamentalmente, estou preocupado com o incentivo às mudanças entre serviços, as linhas mais confusas de propriedade e a necessidade de novas ferramentas que a monorepos possa trazer.

Um problema que tenho visto repetidamente é que as organizações que começaram pequenas, onde a propriedade coletiva (e, portanto, os monorepos) funcionou bem inicialmente, têm lutou para mudar para modelos diferentes mais tarde, conforme o conceito do monorepo está tão arraigado. À medida que a organização de entrega cresce, a dor do monorepo aumenta, mas também aumenta o custo de migrar para uma alternativa abordagem. Isso é ainda mais desafiador para organizações que cresceram rapidamente, pois muitas vezes é somente após esse rápido crescimento que os problemas torna-se evidente, momento em que o custo da migração para uma abordagem multirrepo parece muito alto. Isso pode levar à falácia do custo irrecuperável: você investiu tanto muito para fazer o monorepo funcionar até agora – só um pouco mais o investimento fará com que funcione tão bem quanto antes, certo? Talvez não, mas é uma alma corajosa que pode reconhecer que está gastando um bom dinheiro atrás ruim e tome a decisão de mudar de rumo.

As preocupações com propriedade e monorepos podem ser aliviadas por meio do uso de controles de propriedade refinados, mas isso tende a exigir ferramentas e/ou um maior nível de diligência. Minha opinião sobre isso pode mudar à medida que

a maturidade das ferramentas em torno de monorepos melhora, mas apesar de muito trabalho sendo feito em relação ao desenvolvimento de código aberto de construção baseada em gráficos ferramentas, ainda estou vendo uma aceitação muito baixa dessas cadeias de ferramentas. Então, é multirepos para mim.

Resumo

Abordamos algumas ideias importantes neste capítulo que devem ajudar você a entender boa opção, independentemente de você acabar usando microserviços ou não. Existem muitos mais aspectos para explorar em torno dessas ideias, desde a entrega contínua até desenvolvimento baseado em troncos, de monorepos a multirepos. Eu lhe dei uma série de recursos e leituras adicionais, mas é hora de passarmos para um assunto que é importante explorar com alguma profundidade de implantação.

1 Alanna Brown, Nicole Forsgren, Jez Humble, Nigel Kersten e Gene Kim, Estado de 2016 Relatório de DevOps. <https://oreil.ly/YqEEh>.

2 Nicole Forsgren, Dustin Smith, Jez Humble, e Jessie Frazelle, aceleraram o estado do DevOps 2019, <https://oreil.ly/mfkll>.

3 Para obter mais detalhes, consulte Jez Humble e David Farley, *Continuous Delivery: Reliable Software Lançamentos por meio de construção, teste e automação de implantação* (Upper Saddle River, NJ: Addison-Wesley, 2010).

4 Consulte "Gerenciando gráficos de dependência" em Entrega contínua, pp. 363-73.

5 Rachel Potvin e Josh Levenberg, "Por que o Google armazena bilhões de linhas de código em uma única Repositório", Comunicações do ACM 59, nº 7 (julho de 2016): 78-87.

6 Consulte o histórico de design do sistema de arquivos virtual Git.

Capítulo 8. Implantação

A implementação de um aplicativo monolítico de processo único é bastante simples. processos. Microserviços, com sua interdependência e riqueza de tecnologia opções, são uma chaleira de peixe completamente diferente. Quando eu escrevi a primeira edição deste livro, este capítulo já tinha muito a dizer sobre a enorme variedade de opções disponíveis para você. Desde então, o Kubernetes veio à tona e As plataformas de função como serviço (FaaS) nos deram ainda mais maneiras de pense em como realmente enviar nosso software.

Embora a tecnologia possa ter mudado na última década, acho que muitos alguns dos princípios fundamentais associados à criação de software não mudaram. Na verdade, acho que é ainda mais importante que os entendamos completamente ideias fundamentais, pois podem nos ajudar a entender como lidar com isso paisagem caótica de novas tecnologias. Com isso em mente, este capítulo será destaque alguns princípios básicos relacionados à implementação que são importantes para entenda, ao mesmo tempo que mostra como as diferentes ferramentas disponíveis para você podem ajudar (ou atrapalhar) em relação à colocação desses princípios em prática.

Para começar, porém, vamos dar uma olhada por trás da cortina e ver o que acontece quando passamos de uma visão lógica de nossa arquitetura de sistemas para uma topologia de implantação física real.

Do lógico ao físico

Até agora, quando discutimos microserviços, falamos sobre eles em um sentido lógico, e não em um sentido físico. Poderíamos falar sobre como nosso microserviço de fatura se comunica com o microserviço de pedidos, conforme mostrado na Figura 8-1, sem realmente examinar a topologia física de como esses serviços são implantados. Uma visão lógica de uma arquitetura normalmente abstrai as preocupações subjacentes de implantação física - essa noção precisa a ser alterado para o escopo deste capítulo.

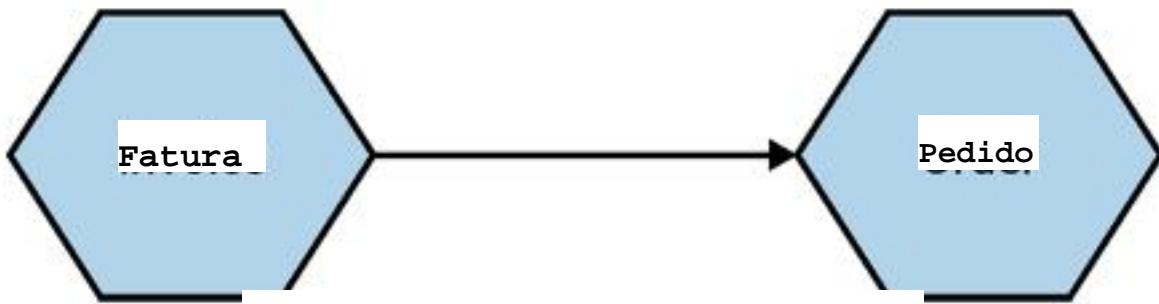


Figura 8-1. Uma visão simples e lógica de dois microserviços

Essa visão lógica de nossos microserviços pode esconder uma riqueza de complexidade quando se trata de realmente executá-los em uma infraestrutura real. Vamos pegar um, veja quais tipos de detalhes podem estar ocultos em um diagrama como esse.

Várias instâncias

Quando pensamos na topologia de implantação dos dois microserviços (em Figura 8-2), não é tão simples quanto uma coisa falar com outra. Para começar, parece bem provável que tenhamos mais de uma instância de cada serviço. Ter várias instâncias de um serviço permite lidar com mais carga e também pode melhorar a robustez do seu sistema, pois você pode mais facilmente tolerar a falha de uma única instância. Então, potencialmente temos um ou mais instâncias de Invoice conversando com uma ou mais instâncias de Order. Exatamente como a comunicação entre essas instâncias será tratada dependerá da natureza do mecanismo de comunicação. mas se assumirmos que neste situação em que estamos usando alguma forma de API baseada em HTTP, um平衡ador de carga seria ser suficiente para lidar com o roteamento de solicitações para diferentes instâncias, como vemos em

Figura 8-2.

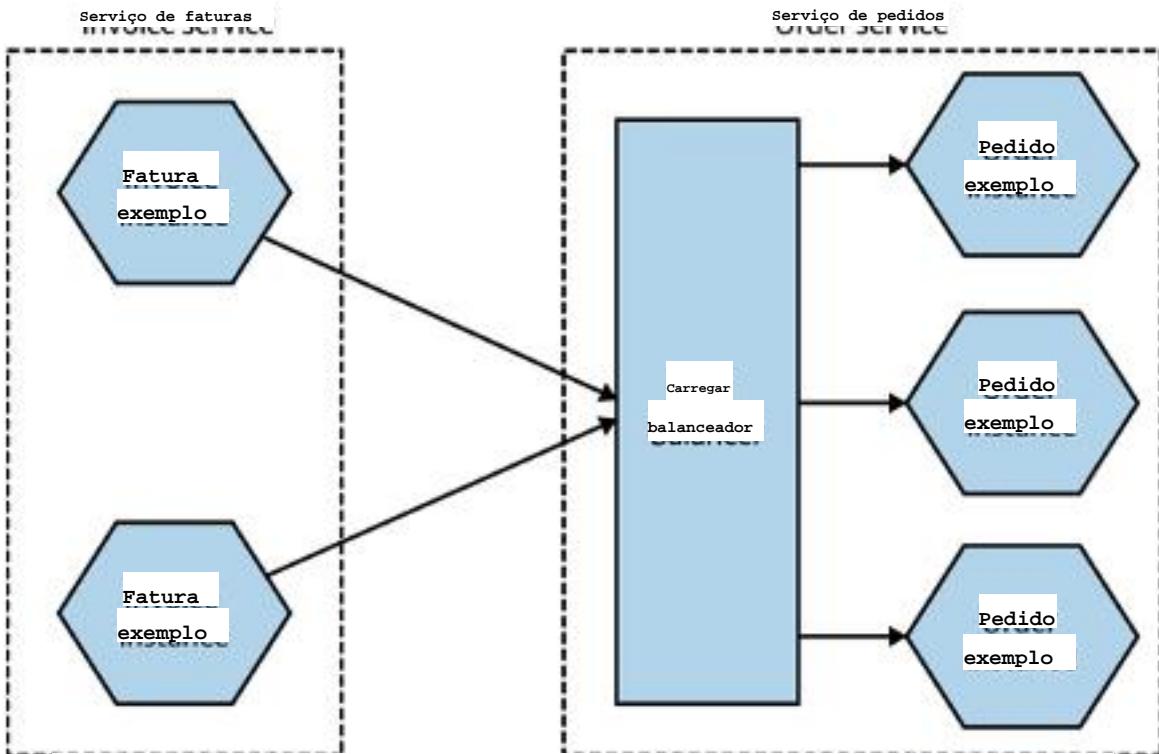


Figura 8-2. Usando um平衡ador de carga para mapear solicitações para instâncias específicas do Pedido microserviço.

O número de instâncias que você desejará dependerá da natureza do seu aplicativo - você precisará avaliar a redundância necessária, a carga esperada níveis e similares para chegar a um número viável. Você também pode precisar para levar em consideração onde essas instâncias serão executadas. Se você tiver vários instâncias de um serviço por motivos de robustez, você provavelmente quer ter certeza que essas instâncias não estão todas no mesmo hardware subjacente. Levado mais longe, isso pode exigir que você tenha instâncias diferentes distribuídas, não apenas entre várias máquinas, mas também em diferentes data centers, para oferecer a você proteção contra a indisponibilidade de um data center inteiro. Isso pode levam a uma topologia de implantação como a da Figura 8-3.

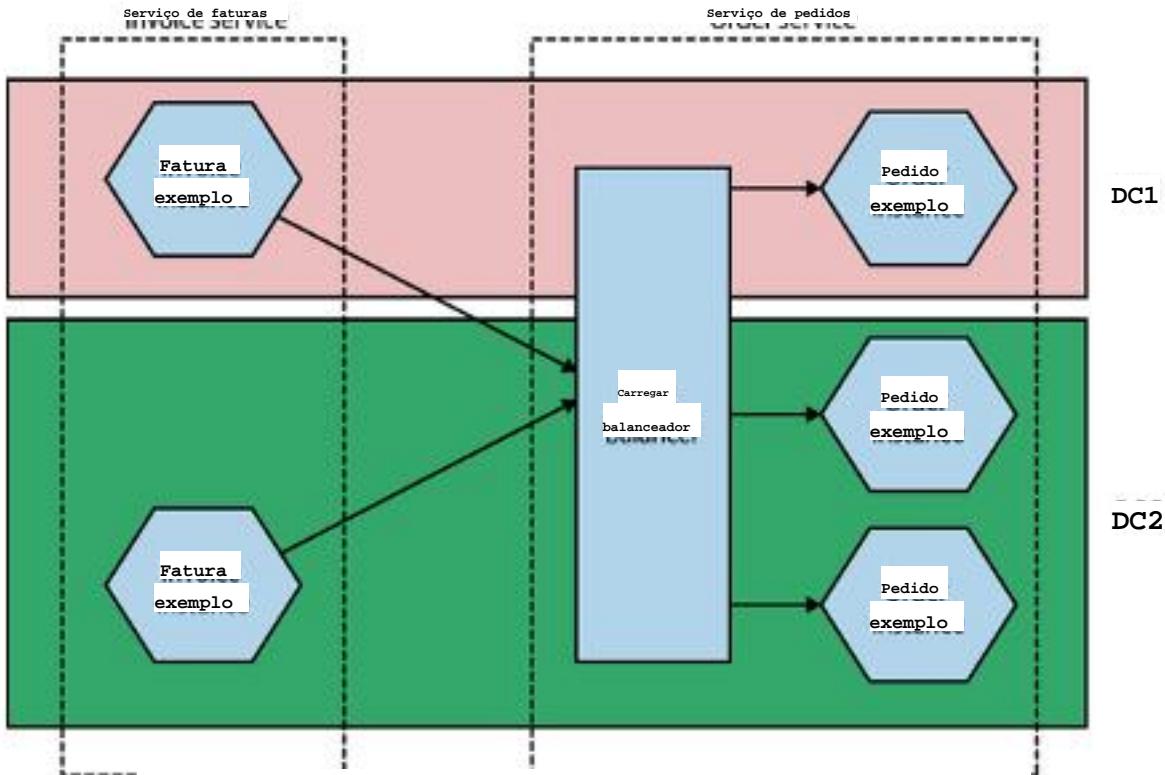


Figura 8-3. Distribuindo instâncias em vários data centers diferentes

Isso pode parecer muito cauteloso - quais são as chances de um dado inteiro?
O centro está indisponível? Bem, eu não posso responder essa pergunta para cada
situação, mas pelo menos ao lidar com os principais provedores de nuvem, isso é
absolutamente algo que você deve levar em conta. Quando se trata de
algo como uma máquina virtual gerenciada, nem AWS nem Azure nem
O Google fornecerá um SLA para uma única máquina, nem um
SLA para uma única zona de disponibilidade (que é o equivalente mais próximo de um dado
centro para esses fornecedores). Na prática, isso significa que qualquer solução você
a implantação deve ser distribuída em várias zonas de disponibilidade.

O banco de dados

Levando isso adiante, há outro componente importante que ignoramos
até este ponto - o banco de dados. Como já discutimos, queremos um
microserviço para ocultar seu gerenciamento de estado interno, portanto, qualquer banco de dados usado por um
microserviço para gerenciar seu estado é considerado oculto dentro do

microsserviço. Isso leva ao mantra frequentemente declarado de “não compartilhe bancos de dados”, o argumento que espero já tenha sido suficientemente apresentado até agora.

Mas como isso funciona quando consideramos o fato de eu ter vários instâncias de microsserviços? Cada instância de microsserviço deve ter sua própria banco de dados? Em uma palavra, não. Na maioria dos casos, se eu acessar qualquer instância do meu Pedido serviço, eu quero ser capaz de obter informações sobre o mesmo pedido. Então nós precisam de algum grau de estado compartilhado entre diferentes instâncias do mesmo serviço lógico. Isso é mostrado na Figura 8-4,

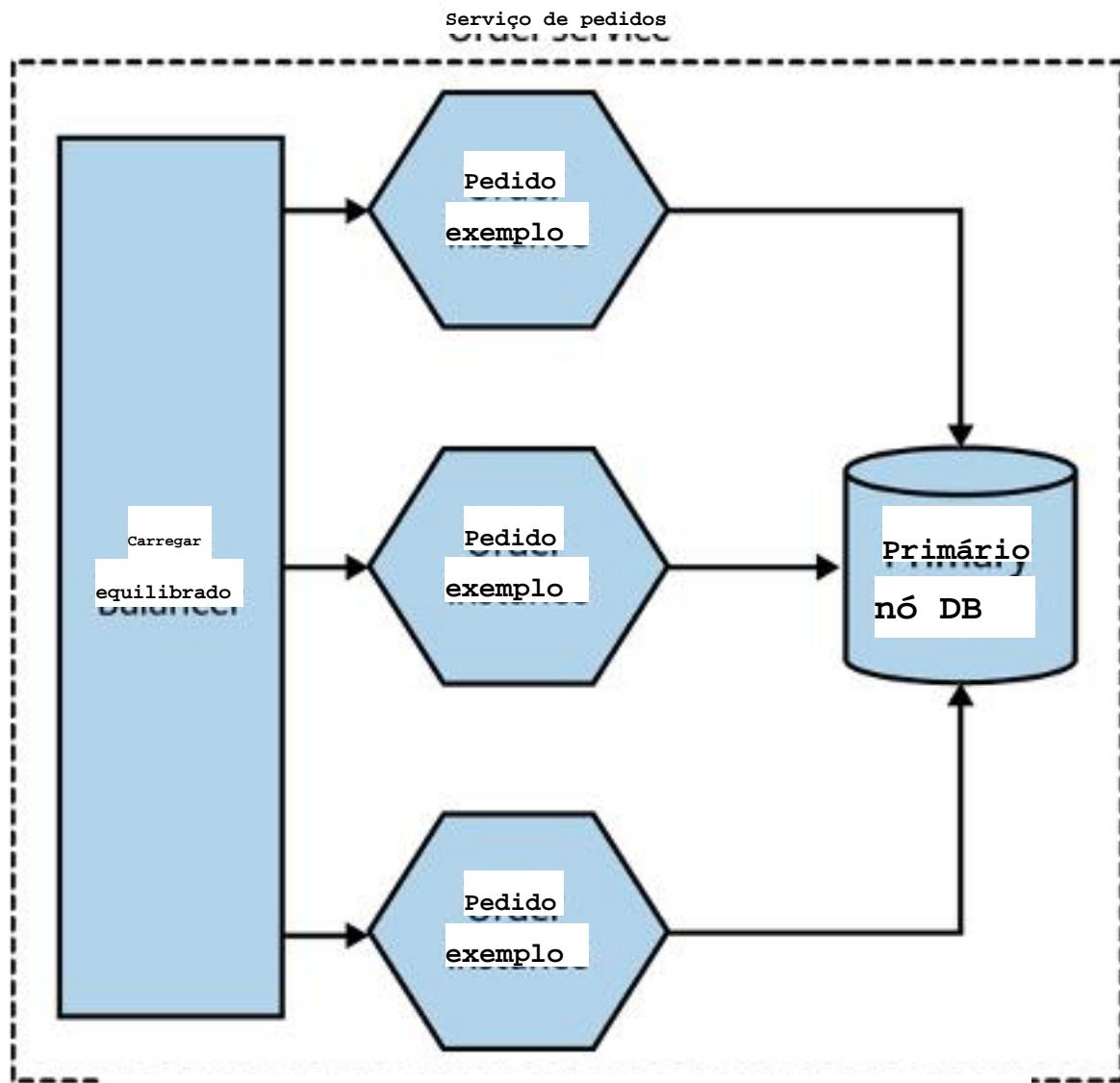


Figura 8-4. Várias instâncias do mesmo microsserviço podem compartilhar um banco de dados.

Mas isso não viola nossa regra de "não compartilhar bancos de dados"? Na verdade, não. Um dos nossas principais preocupações é que, ao compartilhar um banco de dados entre vários diferentes microsserviços, a lógica associada ao acesso e à manipulação desse O estado agora está espalhado por diferentes microsserviços. Mas aqui estão os dados compartilhados por diferentes instâncias do mesmo microsserviço. A lógica para o estado de acesso e manipulação ainda é mantida dentro de uma única lógica microsserviço.

Implantação e escalabilidade do banco de dados

Assim como nossos microsserviços, até agora falamos principalmente sobre um banco de dados em um sentido lógico. Na Figura 8-3, ignoramos qualquer preocupação com a redundância ou necessidades de escalabilidade do banco de dados subjacente.

De um modo geral, uma implantação de banco de dados físico pode ser hospedada em várias máquinas, por uma série de razões. Um exemplo comum seria carga dividida para leituras e gravações entre um nó primário e um ou mais nós que são designados para fins somente de leitura (esses nós normalmente são chamadas de réplicas de leitura). Se estivéssemos implementando essa ideia para o nosso Solicite um serviço, podemos acabar com uma situação como a mostrada em Figura 8-5.

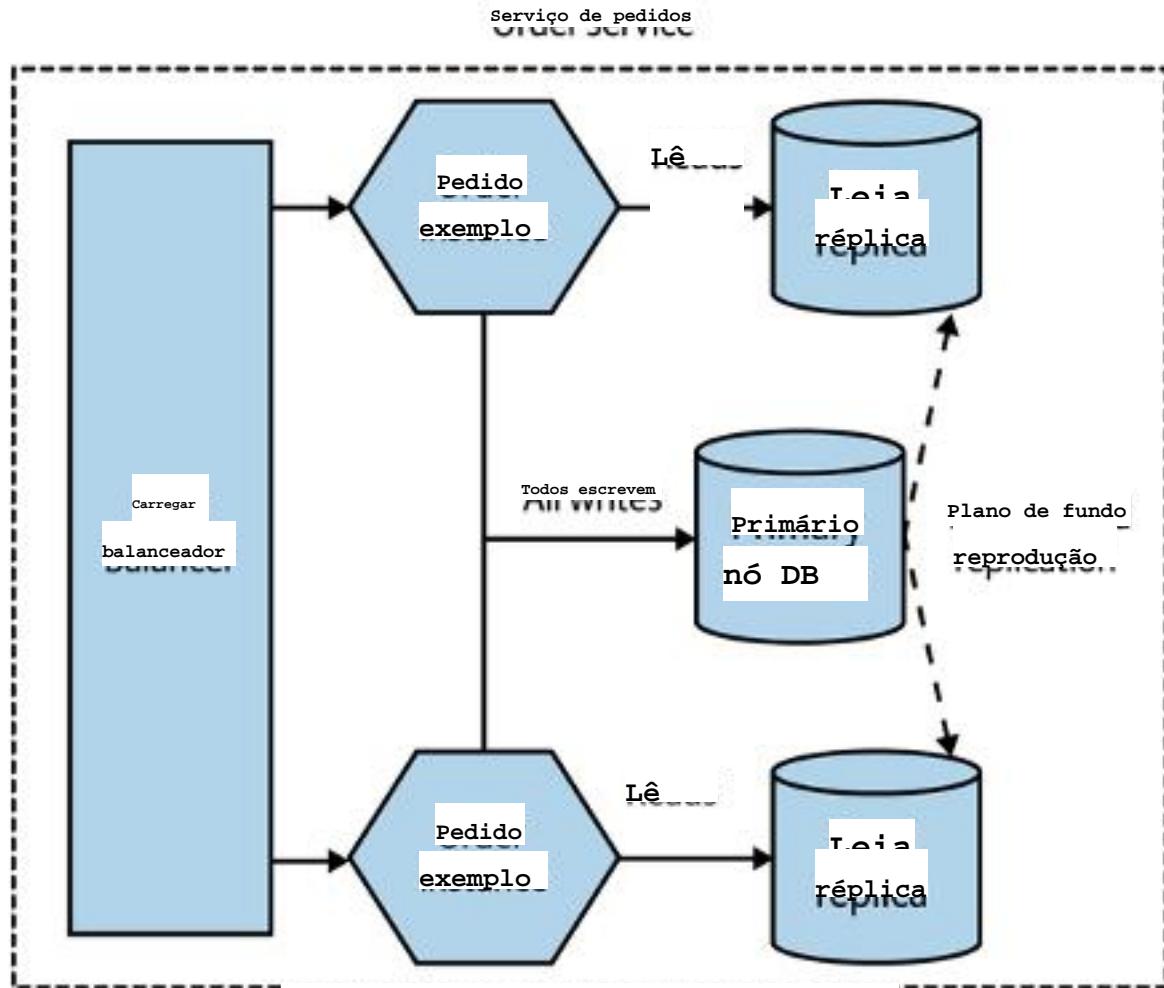


Figura 8-5. Usando réplicas de leitura para distribuir a carga

Todo o tráfego somente de leitura vai para um dos nós de réplica de leitura, e você pode escalar ainda mais o tráfego de leitura adicionando nós de leitura adicionais. Devido à forma como bancos de dados relacionais funcionam, é mais difícil escalar gravações adicionando máquinas adicionais (normalmente são necessários modelos de fragmentação), o que adiciona complexidade adicional), portanto, mover o tráfego somente de leitura para essas réplicas de leitura pode geralmente liberar mais capacidade no nó de gravação para permitir maior escalabilidade.

Soma-se a esse quadro complexo o fato de que a mesma infraestrutura de banco de dados pode suportar vários bancos de dados logicamente isolados. Então, os bancos de dados para motor e hardware, conforme mostrado na Figura 8-6. Isso pode ter um significado benefícios: permite que você reúna hardware para atender a vários microsserviços,

pode reduzir os custos de licenciamento e também pode ajudar a reduzir a solução alternativa gerenciamento do próprio banco de dados.

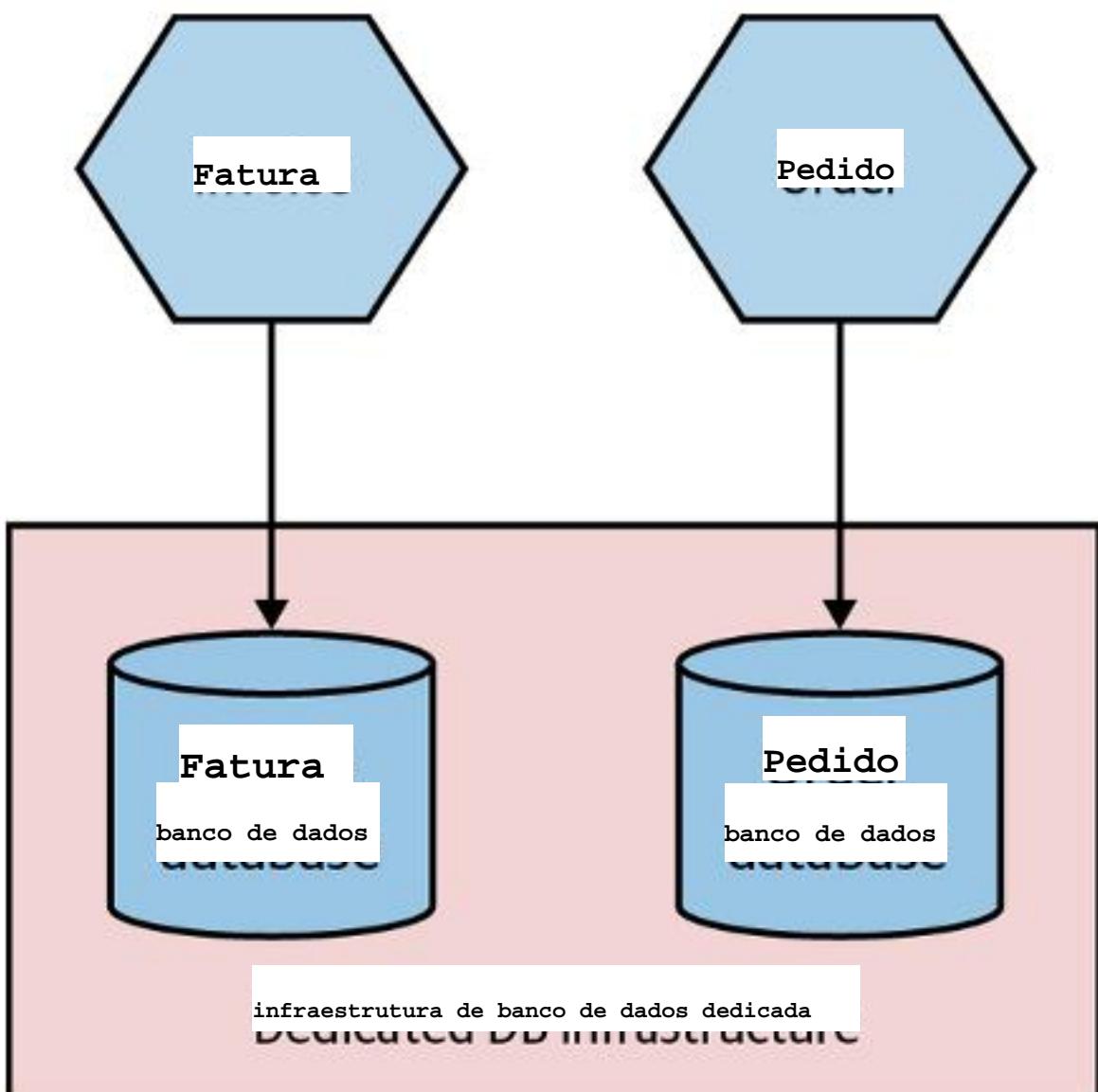


Figura 8-6. A mesma infraestrutura física de banco de dados hospedando dois bancos de dados logicamente isolados

O importante a perceber aqui é que, embora esses dois bancos de dados possam ser executados a partir do mesmo hardware e mecanismo de banco de dados, eles ainda são logicamente bancos de dados isolados. Eles não podem interferir uns com os outros (a menos que você permita isso). A única coisa importante a considerar é que, se esse banco de dados compartilhado falha na infraestrutura, você pode impactar vários microserviços, o que poderia têm um impacto catastrófico.

Em minha experiência, organizações que gerenciam sua própria infraestrutura e operam de uma forma "local", tendem a ter muito mais probabilidade de ter vários bancos de dados diferentes hospedados em uma infraestrutura de banco de dados compartilhada, pelo custo razões que descrevi antes. Provisionar e gerenciar hardware é trabalho (e, historicamente, pelo menos, os bancos de dados têm menos probabilidade de serem executados em sistemas virtualizados na infraestrutura), então você quer menos disso.

Por outro lado, as equipes que operam em provedores de nuvem pública são muito mais propenso a provisionar infraestrutura de banco de dados dedicada em um microsserviço base, conforme mostrado na Figura 8-7. Os custos de provisionar e gerenciar isso a infraestrutura é muito menor. Serviço de banco de dados relacional (RDS) da AWS, por exemplo, pode lidar automaticamente com questões como backups, atualizações e o failover de zonas de multidisponibilidade e produtos similares estão disponíveis no outros provedores de nuvem pública. Isso torna muito mais econômico ter infraestrutura mais isolada para seu microsserviço, fornecendo cada microsserviço o proprietário tem mais controle em vez de depender de um serviço compartilhado.

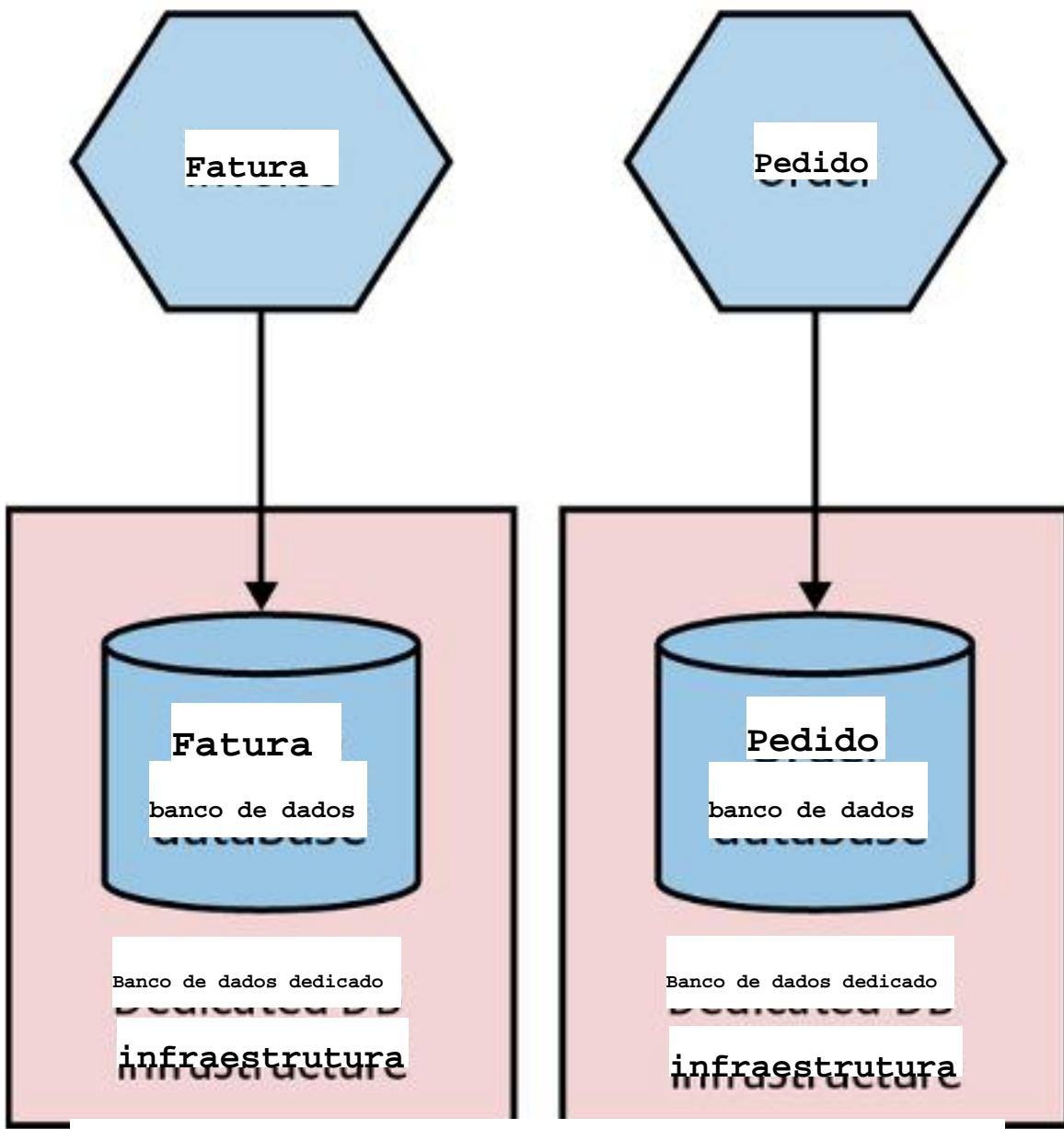


Figura 8-7. Cada microserviço faz uso de sua própria infraestrutura de banco de dados dedicada

Ambientes

Quando você implanta seu software, ele é executado em um ambiente. Cada ambiente normalmente atenderá a diferentes propósitos e ao número exato de ambientes você pode ter variado muito com base em como você desenvolve software e como seu software é implantado para seu usuário final. Alguns ambientes serão têm dados de produção, enquanto outros não. Alguns ambientes podem ter todos

serviços neles; outros podem ter apenas um pequeno número de serviços, com qualquer serviços não presentes substituídos por serviços falsos para fins de teste.

Normalmente, pensamos que nosso software está passando por uma série de ambientes de pré-produção, com cada um servindo a algum propósito para permitir o software a ser desenvolvido e sua prontidão para a produção ser testada. Exploramos isso em "Trade-Offs and Environments". De um desenvolvedor laptop para um servidor de integração contínua, um ambiente de teste integrado e além disso, a natureza e o número exatos de seus ambientes dependerão de um uma série de fatores, mas será impulsionado principalmente pela forma como você escolhe se desenvolver software. Na Figura 8-8, vemos um pipeline para o catálogo da MusicCorp microsserviço. O microsserviço passa por diferentes ambientes antes finalmente entra em um ambiente de produção, onde nossos usuários poderão usar o novo software.

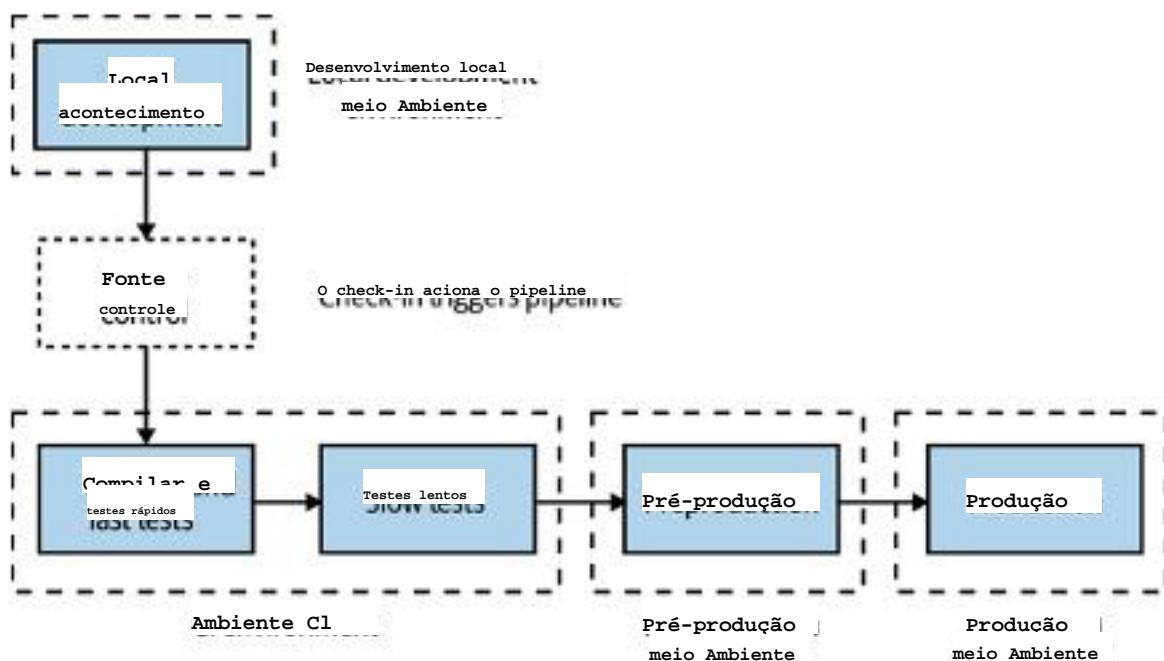


Figura 8-8. Ambientes diferentes usados para diferentes partes da tubulação

O primeiro ambiente em que nosso microsserviço é executado é onde quer que o desenvolvedor esteja. estava trabalhando no código antes de fazer o check-in, provavelmente seu laptop local. Depois de confirmar o código, o processo de CI começa com os testes rápidos. Ambos os estágios de teste rápidos e lentos são implantados em nosso ambiente de CI. Se os testes lentos passem, o microsserviço é implantado no ambiente de pré-produção para

permite a verificação manual (que é totalmente opcional, mas ainda é importante) para muitos). Se essa verificação manual for aprovada, o microsserviço será então implantado na produção.

Idealmente, cada ambiente nesse processo seria uma cópia exata do ambiente de produção. Isso nos daria ainda mais confiança do que nossos o software funcionará quando chegar à produção. No entanto, na realidade, nós muitas vezes não podemos nos dar ao luxo de executar várias cópias de todo o nosso ambiente de produção devido ao quanto caro isso é.

Também queremos ajustar os ambientes no início desse processo para permitir a rapidez feedback. É vital que saibamos o mais cedo possível se nosso o software funciona para que possamos corrigir as coisas rapidamente, se necessário. Quanto mais cedo nós conhecemos um problema com nosso software, quanto mais rápido for corrigi-lo, e o diminua o impacto da quebra. É muito melhor encontrar um problema em nosso local laptop então pegue-o em testes de pré-produção, mas também pegando um problema nos testes de pré-produção pode ser muito melhor para nós do que escolher algo em produção (embora exploremos alguns negócios importantes-contorna isso no Capítulo 9).

Isso significa que os ambientes mais próximos do desenvolvedor serão ajustados fornecem feedback rápido, o que pode comprometer a "aparência de produção" são. Mas à medida que os ambientes se aproximam da produção, queremos que eles sejam cada vez mais, como o ambiente de produção final para garantir que captamos problemas.

Como um exemplo simples disso em ação, vamos revisitar nosso exemplo anterior do Catalogue o serviço e dê uma olhada nos diferentes ambientes. Na Figura 8-9, o laptop do desenvolvedor local tem nosso serviço implantado como uma única instância rodando localmente. O software é rápido de criar, mas é implantado como um único instância em execução em hardware muito diferente do que esperamos em produção. No ambiente de CI, implantamos duas cópias do nosso serviço para teste contra, certificando-se de que nossa lógica de平衡amento de carga esteja funcionando bem. Nós implantamos ambas as instâncias na mesma máquina – isso mantém os custos baixos e transforma as coisas mais rápido e ainda nos dá feedback suficiente nesta fase do processo.

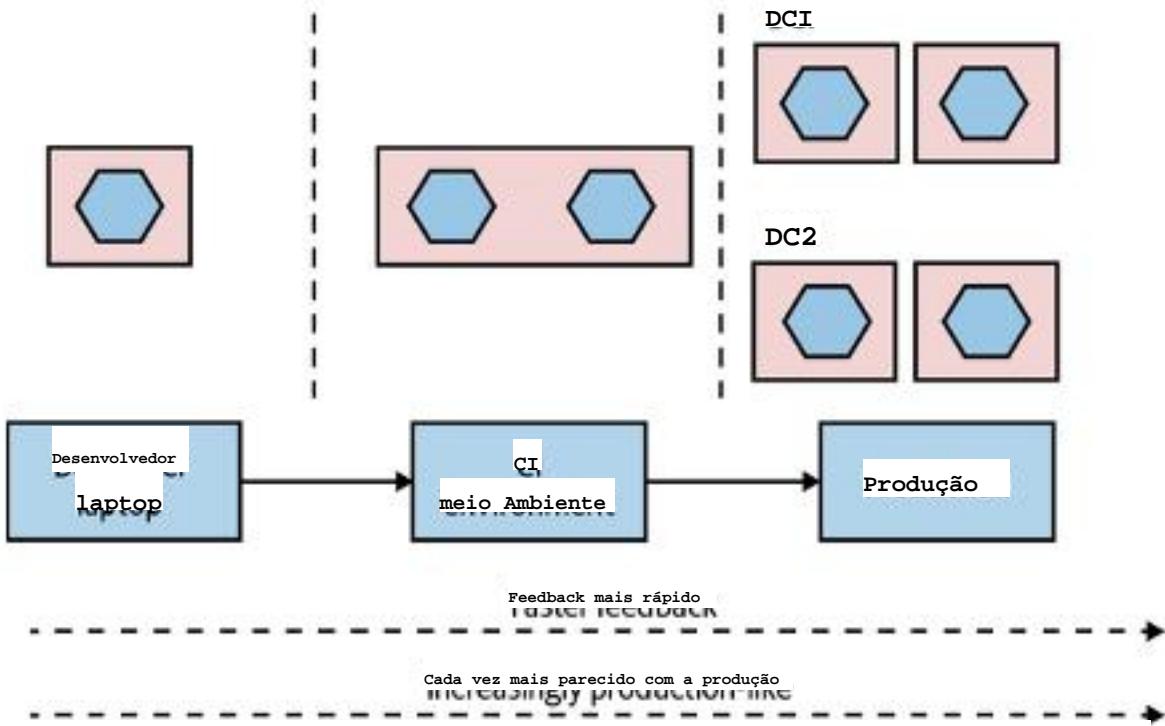


Figura 8-9 Um microsserviço pode variar na forma como é implantado de um ambiente para o outro.

Finalmente, na produção, nosso microsserviço é implantado como quatro instâncias, distribuídos por quatro máquinas, que por sua vez são distribuídas em duas data centers diferentes.

Esse é apenas um exemplo de como você pode usar ambientes; exatamente o que a configuração necessária variará muito dependendo do que você está construindo e como você o implanta. Você pode, por exemplo, ter várias produções... ambientes se você precisar implantar uma cópia do seu software para cada cliente.

O principal, porém, é que a topologia exata do seu microsserviço será mudança de ambiente para ambiente. Portanto, você precisa encontrar maneiras de alterar o número de instâncias de um ambiente para outro, junto com qualquer configuração específica do ambiente. Você também quer criar seu serviço instâncias apenas uma vez, portanto, conclui-se que qualquer ambiente específico as informações precisam ser separadas do artefato de serviço implantado.

Como você pode variar a topologia do seu microsserviço a partir de uma o ambiente para outro dependerá muito do mecanismo que você usa para implantação e também em quanto as topologias variam. Se a única coisa que

mudanças de um ambiente para outro são o número de microsserviços. ... instâncias, isso pode ser tão simples quanto parametrizar esse valor para permitir números diferentes a serem transmitidos como parte da atividade de implantação.

Então, para resumir, um único microsserviço lógico pode ser implantado no vários ambientes. De um ambiente para o outro, o número de as instâncias de cada microsserviço podem variar de acordo com os requisitos de cada um meio ambiente.

Princípios de implantação de microsserviços

Com tantas opções à sua frente sobre como implantar seus microsserviços, eu Acho importante que eu estabeleça alguns princípios fundamentais nessa área. Um sólido a compreensão desses princípios o colocará em boa posição, não importa o escolhas que você acaba fazendo. Examinaremos cada princípio em detalhes em breve, mas só para começar, aqui estão as principais ideias que abordaremos:

Execução isolada

Execute instâncias de microsserviços de forma isolada, de forma que elas tenham seus próprios recursos de computação e sua execução não pode impactar outras instâncias de microsserviços em execução nas proximidades.

Foco na automação

À medida que o número de microsserviços aumenta, a automação se torna cada vez mais importante. Concentre-se na escolha de uma tecnologia que permita alto grau de automação e adote a automação como parte essencial de sua cultura.

Infraestrutura como código

Represente a configuração de sua infraestrutura para facilitar a automação e promover o compartilhamento de informações. Armazene esse código no controle de origem para permitir para que os ambientes sejam recriados.

Implantação sem tempo de inatividade

Aprofunde a implantação independente e garanta a implantação de uma nova versão de um microsserviço pode ser feita sem nenhum tempo de inatividade para os usuários do seu serviço (sejam eles humanos ou outros microsserviços).

Gestão do estado desejado

Use uma plataforma que mantenha seu microsserviço em um estado definido, lançando novas instâncias, se necessário, em caso de interrupções ou tráfego aumenta.

Execução isolada

Você pode se sentir tentado, especialmente no início de sua jornada de microsserviços, a basta colocar todas as suas instâncias de microsserviço em uma única máquina (o que poderia seja uma única máquina física ou uma única VM), conforme mostrado na Figura 8-10. Puramente do ponto de vista do gerenciamento de hosts, esse modelo é mais simples. Em um mundo em que uma equipe gerencia a infraestrutura e outra equipe gerencia o software, a carga de trabalho da equipe de infraestrutura geralmente é uma função do número de hosts que ele precisa gerenciar. Se mais serviços forem agrupados em um host único, a carga de trabalho de gerenciamento do host não aumenta conforme o número de os serviços aumentam.

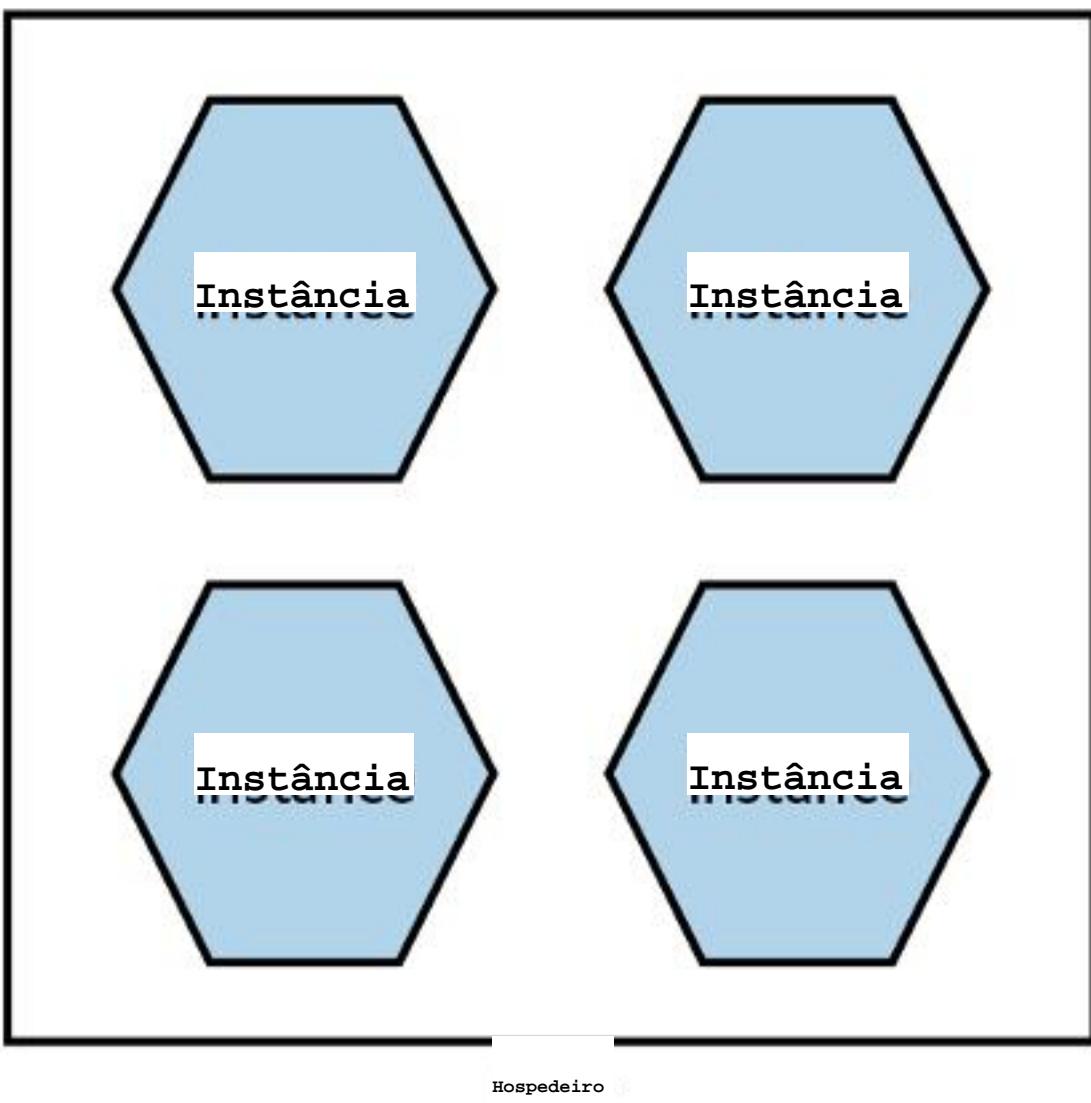


Figura 8-10. Vários microserviços por host.

No entanto, existem alguns desafios com esse modelo. Primeiro, ele pode fazer monitoramento mais difícil. Por exemplo, ao rastrear a CPU, eu preciso rastrear a CPU de um serviço independente dos outros? Ou eu me importo com a CPU do host como um todo? Os efeitos colaterais também podem ser difíceis de evitar. Se um serviço está sob carga significativa, pode acabar reduzindo os recursos disponíveis para outras partes do sistema. Esse foi um problema que Gilt, um site varejista de moda, encontrou. Começando com um monólito Ruby on Rails, Gilt decidiu migrar para microserviços para facilitar a escalabilidade do aplicativo e também para melhor acomodar um número crescente de desenvolvedores. Inicialmente dourado, coexistiam muitos microserviços em uma única caixa, mas uma carga irregular em uma das

os microsserviços teriam um impacto adverso em tudo o mais em execução ...
aquele anfitrião. Isso também tornou a análise de impacto das falhas do host mais complexa-
retirar um único host de serviço pode ter um grande efeito cascata.

A implantação de serviços também pode ser um pouco mais complexa, pois garantir um
a implantação não afeta outra causa dores de cabeca adicionais. Para
exemplo, se cada microsserviço for diferente (e potencialmente contraditório),
dependências que precisam ser instaladas no host compartilhado, como posso fazer
esse trabalho?

Esse modelo também pode inibir a autonomia das equipes. Se os serviços forem diferentes
as equipes são instaladas no mesmo host, que pode configurar o host para seus
serviços? Com toda a probabilidade, isso acaba sendo tratado por um centralizado
equipe, o que significa que é preciso mais coordenação para implantar os serviços.

Fundamentalmente, executar várias instâncias de microsserviços na mesma máquina
(virtual ou físico) acaba minando drasticamente uma das chaves
princípios de microsserviços como uma capacidade de implantação totalmente independente. É
segue, portanto, que realmente queremos executar instâncias de microsserviços em
isolamento, como vemos na Figura 8-11.

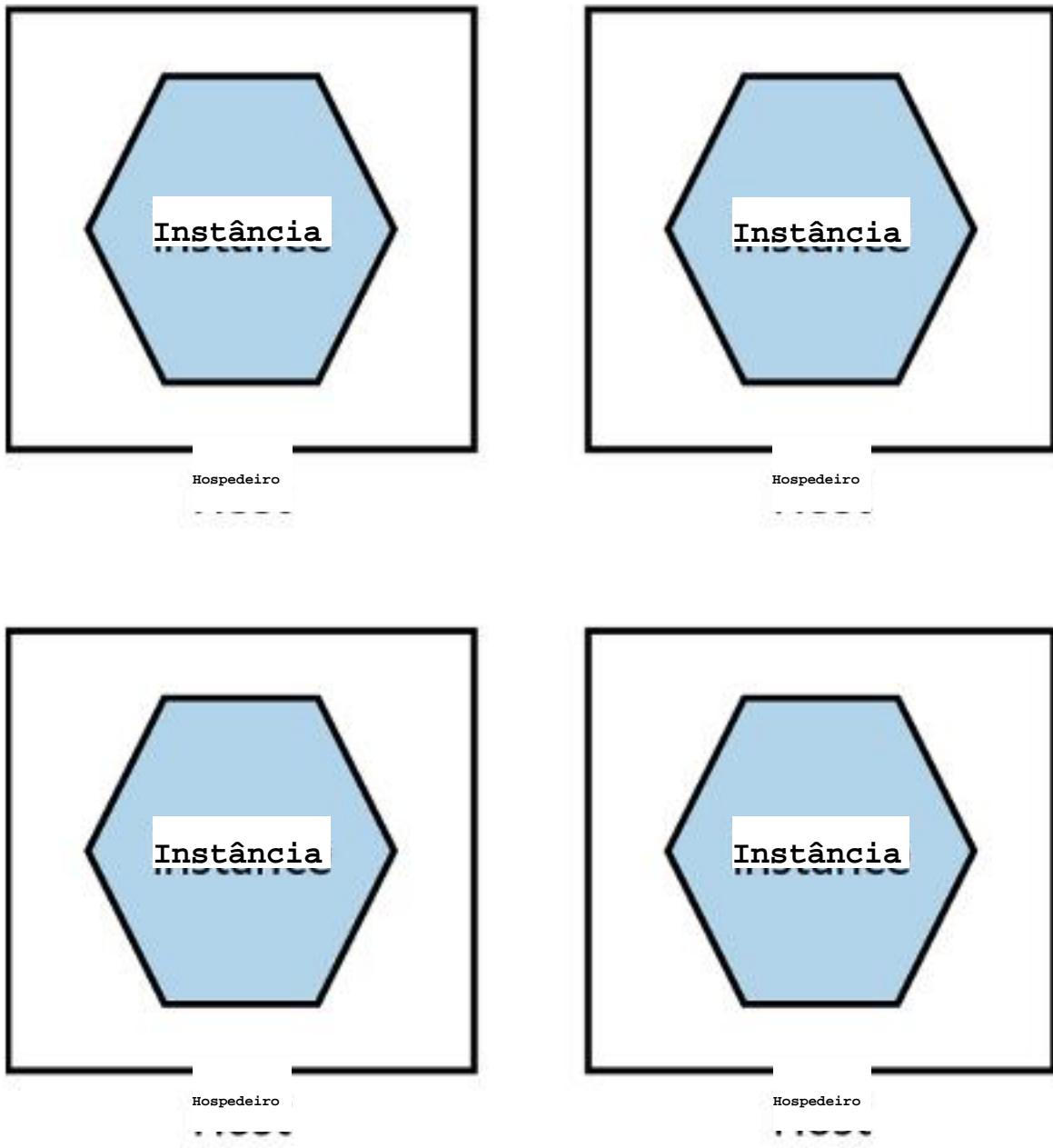


Figura 8-11. Um único microserviço por host

Cada instância de microserviço tem seu próprio ambiente de execução isolado.

pode instalar suas próprias dependências e ter seu próprio conjunto de ring-fenced recursos.

Como diz meu antigo colega Neal Ford, muitas de nossas práticas de trabalho ao redor a implantação e o gerenciamento do host são uma tentativa de otimizar a escassez de recursos. No passado, se quiséssemos que outra máquina alcançasse o isolamento, nosso A única opção era comprar ou alugar outra máquina física. Isso geralmente tinha um

grande prazo de entrega e resultou em um compromisso financeiro de longo prazo. Na minha experiência, não é incomum que os clientes provisionem apenas novos servidores a cada dois ou três anos, e tentando obter máquinas adicionais fora do. Esses cronogramas são difíceis. Mas as plataformas de computação sob demanda têm reduzido drasticamente os custos dos recursos de computação e as melhorias na tecnologia de virtualização significa que há mais flexibilidade, mesmo para uso interno. A infraestrutura hospedada.

Com a conteinerização se juntando ao mix, temos mais opções do que nunca para provisionar um ambiente de execução isolado, conforme mostra a Figura 8-12, de um modo geral, vamos do extremo de ter um físico dedicado máquinas para nossos serviços, o que nos dá o melhor isolamento, mas provavelmente o custo mais alto, para contêineres na outra extremidade, o que nos dá um isolamento mais fraco mas tende a ser mais econômico e muito mais rápido de provisionar. Nós viremos voltando a algumas das especificidades da tecnologia, como a conteinerização mais adiante neste capítulo.

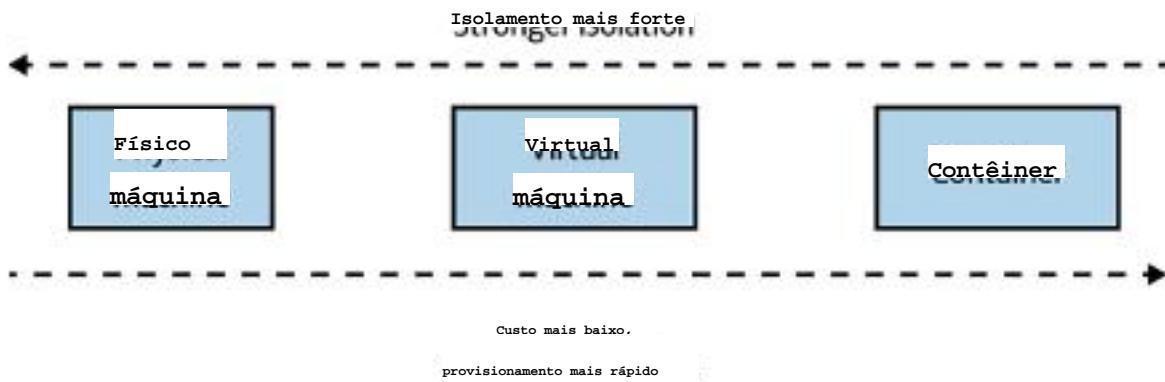


Figura 8-12. Diferentes vantagens em torno dos modelos de isolamento

Se você estivesse implantando seus microserviços em plataformas mais abstratas como o AWS Lambda ou o Heroku, esse isolamento seria fornecido para você. Dependendo da natureza da plataforma em si, você provavelmente poderia esperar que instância de microserviço para acabar sendo executada dentro de um contêiner ou VM dedicada nos bastidores.

Em geral, o isolamento ao redor dos contêineres melhorou o suficiente para fazer eles são uma escolha mais natural para cargas de trabalho de microserviços. A diferença em o isolamento entre contêineres e VMs foi reduzido a tal ponto que, para o

Na grande maioria das cargas de trabalho, os contêineres são "bons o suficiente", o que é grande parte por que eles são uma escolha tão popular e por que eles tendem a ser o meu padrão escolha na maioria das situações.

Foco na automação

À medida que você adiciona mais microserviços, você terá mais partes móveis com as quais lidar -mais processos, mais coisas para configurar, mais instâncias para monitorar.

A mudança para microserviços gera muita complexidade nas operações espaço, e se você estiver gerenciando seus processos operacionais de forma predominantemente forma manual, isso significa que mais serviços exigirão mais e mais pessoas para fazer coisas.

Em vez disso, você precisa de um foco incansável na automação. Selecione ferramentas e tecnologia que permite que as coisas sejam feitas de forma automática, idealmente com o objetivo de trabalhar com infraestrutura como código (que abordaremos em breve).

À medida que o número de microserviços aumenta, a automação se torna cada vez mais importante. Considere seriamente a tecnologia que permite uma alta grau de automação e adote a automação como parte central de sua cultura.

A automação também é a forma como você pode garantir que seus desenvolvedores ainda permaneçam produtivo. Proporcionando aos desenvolvedores a capacidade de fornecer autoatendimento serviços individuais ou grupos de serviços são a chave para construir suas vidas mais fácil.

A tecnologia de seleção que permite a automação começa com as ferramentas usadas para gerenciar anfitriões. Você pode escrever uma linha de código para iniciar uma máquina virtual ou desligar um? Você pode implantar o software que você escreveu automaticamente? Você pode implantar alterações no banco de dados sem intervenção manual? Abraçando um a cultura de automação é fundamental se você quiser manter as complexidades de arquiteturas de microserviços sob controle.

Dois estudos de caso sobre o poder da automação

Provavelmente será útil dar alguns exemplos concretos de que explique o poder de uma boa automação. A empresa australiana

~~realestate.com.au (REA) fornece anúncios de imóveis para varejo e clientes comerciais na Austrália e em outros lugares da região Ásia-Pacífico.~~

~~Ao longo de vários anos, ela estava migrando sua plataforma para uma plataforma distribuída design de microserviços. Quando começou essa jornada, teve que gastar muito hora de preparar as ferramentas dos serviços da maneira certa, facilitando a desenvolvedores para provisionar máquinas, implantar seu código e monitorar seus serviços. Isso causou um carregamento antecipado do trabalho para começar as coisas.~~

~~Nos primeiros três meses deste exercício, a REA conseguiu mover apenas dois novos microserviços em produção, com a equipe de desenvolvimento tomando o máximo responsabilidade por toda a construção, implantação e suporte dos serviços. Em nos próximos três meses, entre 10 a 15 serviços foram lançados em um similar maneira. No final de um período de 18 meses, a REA tinha mais de 70 serviços em produção.~~

~~Esse tipo de padrão também é confirmado pelas experiências de Gilt, que nós mencionado anteriormente. Novamente, a automação, especialmente ferramentas para ajudar os desenvolvedores, impulsionou a explosão no uso de microserviços pela Gilt. Um ano depois de começar sua migração para microserviços, Gilt tinha cerca de 10 microserviços ativos; em 2012, mais de 100; e em 2014, mais de 450 microserviços estavam ativos – ou cerca de três microserviços para cada desenvolvedor em Gilt. Esse tipo de proporção de microserviços para desenvolvedores não são incomuns entre organizações que são madura no uso de microserviços, sendo o Financial Times uma empresa com uma proporção semelhante.~~

Infraestrutura como código (IAC)

~~Levando o conceito de automação adiante, a infraestrutura como código (IAC) é uma conceito pelo qual sua infraestrutura é configurada usando legível por máquina código. Você pode definir sua configuração de serviço em um arquivo chef ou puppet, ou talvez escreva alguns scripts bash para configurar as coisas, mas seja qual for a ferramenta que você usar ao usar, seu sistema pode ser levado a um estado conhecido por meio do uso de código-fonte. Indiscutivelmente, o conceito de IAC pode ser considerado uma forma de implemente automação. Acho, porém, que vale a pena chamá-lo de seu coisa, porque fala sobre como a automação deve ser feita. Infraestrutura como o código trouxe conceitos do desenvolvimento de software para as operações~~

espaço. Ao definir nossa infraestrutura via código, essa configuração pode ser versão controlada, testada e repetida à vontade. Para saber mais sobre esse assunto, eu recomendo Infrastructure as Code, 2ª edição, de Kief Morris.

Teoricamente, você poderia usar qualquer linguagem de programação para aplicar as ideias de infraestrutura como código, mas existem ferramentas especializadas nessa área, como Puppet, Chef, Ansible e outros, todos liderados pelo

CFEngine anterior. Essas ferramentas são declarativas - elas permitem que você defina em forma textual: o que você espera que uma máquina (ou outro conjunto de recursos) pareça como, e quando esses scripts são aplicados, a infraestrutura é trazida para lá estado. Ferramentas mais recentes foram além da configuração de uma máquina.

e passou a analisar como configurar conjuntos inteiros de recursos em nuvem. O Terraform tem tido muito sucesso nesse espaço e estou empolgado em ver o potencial de Pulumi, que tem como objetivo fazer algo semelhante, embora por permitindo que as pessoas usem linguagens de programação normais em vez da

linguagens específicas de domínio que geralmente são usadas por essas ferramentas. AWS

O CloudFormation e o AWS Cloud Development Kit (CDK) são exemplos de ferramentas específicas da plataforma, neste caso suportando apenas a AWS, embora seja vale a pena notar que mesmo se eu estivesse trabalhando apenas com a AWS, eu preferiria a flexibilidade de uma ferramenta multiplataforma como o Terraform.

O controle de versão do seu código de infraestrutura oferece transparência sobre quem fez mudanças, algo que os auditores adoram. Também torna mais fácil reproduzir um ambiente em um determinado momento. Isso é algo que pode seja especialmente útil ao tentar rastrear defeitos. Em um memorável exemplo, um dos meus clientes, como parte de um processo judicial, teve que recriar um todo executando o sistema em um momento específico, alguns anos antes, até o patch níveis dos sistemas operacionais e do conteúdo dos corretores de mensagens. Se a configuração do ambiente foi armazenada no controle de versão, seu trabalho teria sido muito mais fácil - por mais que fosse, eles acabaram gastando mais de três meses tentandometiculosamente reconstruir uma imagem espelhada de uma produção anterior ambiente examinando e-mails e notas de lançamento para tentar se exercitar o que foi feito por quem. O processo judicial, que já estava em andamento longo período de tempo, ainda não estava resolvido quando terminei meu trabalho com o cliente.

Implantação sem tempo de inatividade

Como você provavelmente está cansado de me ouvir dizer, independente

a capacidade de implantação é muito importante. No entanto, também não é uma qualidade absoluta.

Quão independente é algo exatamente? Antes deste capítulo, nós basicamente

analisou a capacidade de implantação independente em termos de evitar a implementação

acoplamento. No início deste capítulo, falamos sobre a importância de fornecer

uma instância de microsserviço com um ambiente de execução isolado, para garantir isso

tem um grau de independência no nível de implantação física. Mas nós podemos

vá mais longe.

Implementar a capacidade de implantação sem tempo de inatividade pode ser um grande avanço

ao permitir que microsserviços sejam desenvolvidos e implantados. Sem zero-

implantação em tempo de inatividade, talvez eu precise coordenar com os consumidores upstream

quando eu libero um software para alertá-los sobre uma possível interrupção.

Sarah Wells, do Financial Times, cita a capacidade de implementar zero-

implantação em tempo de inatividade como sendo o maior benefício individual em termos de

melhorando a velocidade de entrega. Com a confiança de que os lançamentos não seriam

interrompendo seus usuários, o Financial Times conseguiu aumentar drasticamente o

frequência de lançamentos. Além disso, uma versão sem tempo de inatividade pode ser muito

mais facilmente feito durante o horário de trabalho. Além do fato de que fazer

assim melhora a qualidade de vida das pessoas envolvidas com o lançamento

(em comparação com o trabalho noturno e nos fins de semana), uma equipe bem descansada trabalhando

durante o dia tem menos probabilidade de cometer erros e terá o apoio de

muitos de seus colegas quando precisam resolver problemas.

O objetivo aqui é que os consumidores upstream não percebam nada quando você o faz.

um lançamento. Tornar isso possível pode depender muito da natureza do seu

microsserviço. Se você já está usando o suporte de middleware

comunicação assíncrona entre seu microsserviço e seu

consumidores, isso pode ser trivial de implementar - as mensagens enviadas a você serão

entregue quando você estiver de volta. Se você estiver usando o baseado em síncrono

comunicação, no entanto, isso pode ser mais problemático.

Conceitos como atualizações contínuas podem ser úteis aqui, e essa é uma área em que

o uso de uma plataforma como o Kubernetes torna sua vida muito mais fácil. Com um

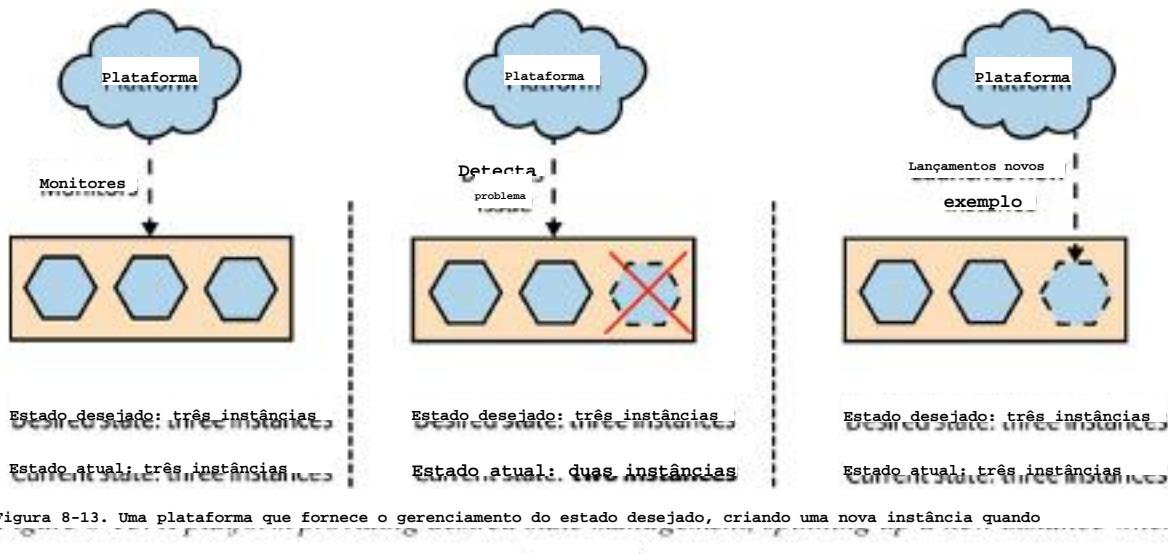
atualização contínua, seu microsserviço não é totalmente desligado antes do novo a versão é implantada, em vez disso, as instâncias do seu microsserviço são lentas reduzido à medida que novas instâncias executando novas versões do seu software são aumentou. É importante notar, porém, que se é a única coisa que você está procurando é algo para ajudar nas implantações sem tempo de inatividade e, em seguida, na implementação O Kubernetes provavelmente é um grande exagero. Algo simples como um azul esverdeado mecanismo de implantação (que exploraremos mais em "Separando Deployment from Release") pode funcionar com a mesma eficiência.

Pode haver desafios adicionais em termos de lidar com problemas como conexões duradouras e afins. Certamente é verdade que se você construir um microsserviço com implantação sem tempo de inatividade em mente, você provavelmente terá um é muito mais fácil fazer isso do que se você usasse uma arquitetura de sistemas existente e tentou modernizar esse conceito posteriormente. Se você é capaz ou não de implemente inicialmente uma implantação sem tempo de inatividade para seus serviços, se você puder Ao chegar lá, você certamente apreciará esse maior nível de independência.

Gestão do estado desejado

O gerenciamento do estado desejado é a capacidade de especificar a infraestrutura requisitos que você tem para sua inscrição e para que esses requisitos sejam mantido sem intervenção manual. Se o sistema em execução mudar de tal forma uma forma de que o estado desejado não seja mais mantido, a plataforma subjacente toma as medidas necessárias para trazer o sistema de volta ao estado desejado.

Como um exemplo simples de como o gerenciamento de estado desejado pode funcionar, você poderia especificar o número de instâncias que seu microsserviço exige, talvez também especificando a quantidade de memória e CPU que essas instâncias precisam. Alguns a plataforma subjacente pega essa configuração e a aplica, trazendo o sistema no estado desejado. Cabe à plataforma, entre outras coisas, identifique quais máquinas têm recursos sobressalentes que podem ser alocados para executar o número solicitado de instâncias. Como mostra a Figura 8-13, se um desses as instâncias morrem, a plataforma reconhece que o estado atual não corresponde ao estado desejado e toma as medidas apropriadas ao lançar um substituto instância.



A beleza do gerenciamento de estado desejado é que a própria plataforma gerencia como o estado desejado é mantido. Isso libera o desenvolvimento e as operações as pessoas também por terem que se preocupar exatamente com a forma como as coisas estão sendo feitas. Eles só precisam se concentrar em obter a definição de estado desejada logo no início lugar. Isso também significa que, no caso de ocorrer um problema, como uma morte da instância, falha do hardware subjacente ou fechamento de um data center inativo, a plataforma pode resolver o problema para você sem intervenção humana sendo exigido.

Embora seja possível criar sua própria cadeia de ferramentas para aplicar o estado desejado gerenciamento, normalmente você usa uma plataforma que já o suporta. Kubernetes é uma dessas ferramentas que abraça essa ideia, e você também pode conseguir algo conceitos de uso semelhantes, como escalonamento automático de grupos em um provedor de nuvem pública como Azure ou AWS. Outra plataforma que pode fornecer esse recurso é Nômada. Ao contrário do Kubernetes, que se concentra na implantação e no gerenciamento cargas de trabalho baseadas em contêineres, o Nomad tem um modelo muito flexível de execução também outros tipos de cargas de trabalho de aplicativos, como aplicativos Java, VMs, trabalhos no Hadoop e muito mais. Pode valer a pena dar uma olhada se você quiser uma plataforma para gerenciar cargas de trabalho mistas que ainda usam conceitos como o desejo gestão estadual.

Essas plataformas estão cientes da disponibilidade subjacente de recursos e são capaz de combinar as solicitações do estado desejado com os recursos disponíveis (ou então

digo que isso não é possível). Como operador, você está distanciado da baixa configuração de nível - você pode dizer algo simples como "Eu quero quatro instâncias espalhadas pelos dois data centers" e confiam em sua plataforma para garantir que isso seja feito para você. Plataformas diferentes oferecem diferentes níveis de controle - você pode ficar muito mais complexo com a definição de estado desejada se você quer.

O uso do gerenciamento de estado desejado pode ocasionalmente causar problemas se você esquece que está fazendo uso dele. Lembro-me de uma situação em que eu estava encerrando um cluster de desenvolvimento na AWS antes de ir para casa. Eu estava encerrando as instâncias de máquinas virtuais gerenciadas (fornecidas pela AWS Produto EC2) para economizar dinheiro - eles não seriam usados da noite para o dia. No entanto, descobri que, assim que matei uma das instâncias, outra a instância apareceu de volta. Demorei um pouco para perceber que eu havia configurado um grupo de escalonamento automático para garantir que houvesse um número mínimo de máquinas. A AWS estava vendo uma instância morrer e criando uma substituta. Me levou 15 minutos jogando whack-a-mole desse jeito antes de eu perceber o que estava acontecendo. O problema era que éramos cobrados pelo EC2 por hora. Mesmo que uma instância foi executada por apenas um minuto, fomos cobrados pela hora inteira. Então meu trabalho no final do dia acabou sendo muito caro. De certa forma, isso foi um sinal de sucesso (pelo menos foi o que eu disse a mim mesmo) - nós configuramos o grupo de escalonamento automático algum tempo antes, e eles tinham acabado de trabalhar direto ao ponto que tínhamos esquecido que eles estavam lá. Era simplesmente uma questão de escrever um script para desativar o grupo de escalonamento automático como parte do desligamento do cluster para corrigir o problema no futuro.

Pré-requisitos

Para aproveitar o gerenciamento de estado desejado, a plataforma precisa de alguma forma para iniciar automaticamente instâncias do seu microsserviço. Então, ter uma implantação automatizada de instâncias de microsserviços é um pré-requisito claro para gestão estadual desejada. Você também pode precisar pensar cuidadosamente sobre como as instâncias demoram muito para serem lançadas. Se você estiver usando o estado desejado gerenciamento para garantir que haja recursos computacionais suficientes para lidar com o usuário carregue, então, se uma instância morrer, você desejará uma instância de substituição o mais rápido quanto possível para preencher a lacuna. Se o provisionamento de uma nova instância levar muito tempo,

talvez seja necessário ter excesso de capacidade para lidar com a carga no caso de um caso morrendo para ter espaço suficiente para respirar uma nova cópia.

Embora você possa criar uma solução de gerenciamento de estado desejada para Você mesmo, não estou convencido de que seja um bom uso do seu tempo. Se você quiser adote esse conceito, acho melhor adotar uma plataforma que o adota como um conceito de primeira classe. Pois isso significa lidar com o que pode representar uma nova plataforma de implantação e todas as ideias associadas e ferramentas, talvez você queira adiar a adoção do gerenciamento de estado desejado até já tenho alguns microserviços em funcionamento. Isso permitirá que você obtenha familiarizado com os conceitos básicos de microserviços antes de ficar sobrecarregado com nova tecnologia. Plataformas como o Kubernetes realmente ajudam quando você tem muitas coisas para gerenciar - se você tiver apenas alguns processos com os quais se preocupar, você poderia esperar até mais tarde para adotar essas ferramentas.

GitOps

O GitOps, um conceito relativamente recente lançado pela Weaveworks, reúne os conceitos de gerenciamento de estado desejado e infraestrutura como código. GitOps foi originalmente concebido no contexto do trabalho com o Kubernetes, e isso é onde as ferramentas relacionadas estão focadas, embora, sem dúvida, descreva um fluxo de trabalho que outros já usaram.

Com o GitOps, o estado desejado para sua infraestrutura é definido em código e armazenado no controle de origem. Quando mudanças são feitas nesse estado desejado, algumas o uso de ferramentas garante que esse estado atualizado desejado seja aplicado à execução sistema. A ideia é oferecer aos desenvolvedores um fluxo de trabalho simplificado com suas aplicações.

Se você já usou ferramentas de configuração de infraestrutura como Chef ou Puppet, este o modelo é familiar para gerenciar a infraestrutura. Ao usar o Chef Server ou Puppet Master, você tinha um sistema centralizado capaz de promover mudanças dinamicamente quando foram feitos. A mudança com o GitOps é que essas ferramentas está usando os recursos do Kubernetes para ajudar a gerenciar aplicativos em vez de apenas infraestrutura.

Ferramentas como o Flux estão facilitando muito a adoção dessas ideias. Vale a pena observando, é claro, que, embora as ferramentas possam facilitar a alteração da maneira como você trabalha, eles não podem forçá-lo a adotar novas abordagens de trabalho.

Dito de outra forma, só porque você tem o Flux (ou outra ferramenta GitOps), não significa que você está adotando as ideias da gestão estadual desejada ou infraestrutura como código.

Se você está no mundo do Kubernetes, adotando uma ferramenta como o Flux e os fluxos de trabalho que ele promove podem muito bem acelerar a introdução de conceitos como gerenciamento de estado e infraestrutura desejados como código. Apenas certifique-se de que você não perca de vista os objetivos dos conceitos subjacentes e fique cego por todos a nova tecnologia neste espaço!

Opções de implantação

Quando se trata das abordagens e ferramentas que podemos usar para nossos cargas de trabalho de microserviços, temos muitas opções. Mas devemos dar uma olhada nessas opções em termos dos princípios que acabei de delineiar. Queremos nosso microserviços para serem executados de forma isolada e, idealmente, implantados em um forma que evita o tempo de inatividade. Queremos que as ferramentas que escolhemos nos permitam adote uma cultura de automação, defina nossa infraestrutura e aplicação configuração no código e, idealmente, também gerencie o estado desejado para nós. Vamos resumir brevemente as várias opções de implantação antes de analisar quão bem eles entregam essas ideias:

Máquina física

Uma instância de microserviço é implantada diretamente em uma máquina física, sem virtualização

Máquina virtual

Uma instância de microserviço é implantada em uma máquina virtual.

Contêiner

Uma instância de microserviço é executada como um contêiner separado em um ambiente virtual ou máquina física. O tempo de execução desse contêiner pode ser gerenciado por um contêiner ferramenta de orquestração como o Kubernetes.

Contêiner de aplicativos

Uma instância de microserviço é executada dentro de um contêiner de aplicativo que gerencia outras instâncias do aplicativo, normalmente no mesmo tempo de execução.

Plataforma como serviço (PaaS)

Uma plataforma mais abstrata é usada para implantar microserviços instâncias, geralmente abstraindo todos os conceitos dos servidores subjacentes usado para executar seus microserviços. Os exemplos incluem Heroku, Google App Engine e AWS Beanstalk

Função como serviço (FaaS)

Uma instância de microserviço é implantada como uma ou mais funções, que são executado e gerenciado por uma plataforma subjacente, como AWS Lambda ou Azure Functions. Provavelmente, o FaaS é um tipo específico de PaaS, mas merece exploração por si só, dada a recente popularidade da ideia e da questões que ele levanta sobre o mapeamento de um microserviço para um implantado artefato.

Máquinas físicas

Uma opção cada vez mais rara, você pode se encontrar implantando microserviços diretamente em máquinas físicas. Por "diretamente", quero dizer que não há camadas de virtualização ou containerização entre você e o subjacente hardware. Isso se tornou cada vez menos comum em alguns diferentes razões. Em primeiro lugar, a implantação diretamente no hardware físico pode levar a uma redução utilização em toda a sua propriedade. Se eu tiver uma única instância de um microserviço rodando em uma máquina física e eu uso apenas metade da CPU, memória ou E/S fornecidos pelo hardware, então os recursos restantes são desperdiçados. Isso o problema levou à virtualização da maior parte da infraestrutura de computação, permitindo que você coexista várias máquinas virtuais no mesmo ambiente físico.

máquina. Ele oferece uma utilização muito maior de sua infraestrutura, que tem alguns benefícios óbvios em termos de relação custo-benefício.

Se você tiver acesso direto ao hardware físico sem a opção de virtualização, a tentação é então empacotar vários microserviços no mesmo máquinas - é claro, isso viola o princípio sobre o qual falamos sobre ter um ambiente de execução isolado para seus serviços. Você poderia usar ferramentas como Puppet ou Chef para configurar a ajuda da máquina implemente a infraestrutura como código. O problema é que, se você estiver trabalhando apenas no nível de uma única máquina física, implementando conceitos conforme desejado gerenciamento de estado, implantação sem tempo de inatividade e assim por diante exigem que trabalhemos em um nível mais alto de abstração, usando algum tipo de camada de gerenciamento na parte superior. Esses tipos de sistemas são mais comumente usados em conjunto com o virtual. máquinas, algo que exploraremos mais adiante em um momento.

Em geral, a implantação direta de microserviços em máquinas físicas é algo que eu quase nunca vejo hoje em dia, e você provavelmente precisará ter algum requisitos (ou restrições) muito específicos em sua situação para justificar isso abordagem sobre a maior flexibilidade que a virtualização ou a containerização pode trazer.

Máquinas virtuais

A virtualização transformou os data centers, permitindo que dividimos máquinas físicas existentes em máquinas virtuais menores. Tradicional virtualização como a VMware ou a usada pelos principais provedores de nuvem, a infraestrutura gerenciada de máquinas virtuais (como o serviço EC2 da AWS) tem gerou enormes benefícios ao aumentar a utilização da infraestrutura de computação, e, ao mesmo tempo, reduz a sobrecarga do gerenciamento do host.

Fundamentalmente, a virtualização permite que você divida uma máquina subjacente em várias máquinas "virtuais" menores que agem exatamente como servidores normais para o software em execução dentro das máquinas virtuais. Você pode atribuir partes de a capacidade subjacente de CPU, memória, E/S e armazenamento de cada virtual máquina, que em nosso contexto permite que você amunte muitas outras isoladas

ambientes de execução para suas instâncias de microserviço em um único máquina física.

Cada máquina virtual contém um sistema operacional completo e um conjunto de recursos que pode ser usado pelo software executado dentro da VM. Isso garante que você tenha um grau muito bom de isolamento entre instâncias quando cada instância é implantado em uma VM separada. Cada instância de microserviço pode totalmente configure o sistema operacional na VM de acordo com suas próprias necessidades locais. Nós ainda No entanto, tenho o problema de que, se o hardware subjacente estiver executando esses virtuais as máquinas falham, podemos perder várias instâncias de microserviços. Existem maneiras para ajudar a resolver esse problema específico, incluindo coisas como o estado desejado gestão, que discutimos anteriormente.

Custo da virtualização

À medida que você empacota mais e mais máquinas virtuais na mesma base hardware, você descobrirá que obtém retornos decrescentes em termos de recursos de computação disponíveis para as próprias VMs. Por que isso?

Pense em nossa máquina física como uma gaveta de meias. Se colocarmos muita madeira divisórias em nossa gaveta, podemos guardar mais meias ou menos? A resposta é menos: as divisórias em si também ocupam espaço! Nossa gaveta pode ser mais fácil para lidar e organizar, e talvez pudéssemos decidir colocar camisetas em um dos espaços agora, em vez de apenas meias, mas mais divisórias significam menos espaço geral.

No mundo da virtualização, temos uma sobrecarga semelhante à nossa meia divisórias de gaveta. Para entender de onde vem essa sobrecarga, vamos dar uma olhada sobre como a maior parte da virtualização é feita. A Figura 8-14 mostra uma comparação de dois tipos de virtualização. À esquerda, vemos as várias camadas envolvidas em o que é chamado de virtualização tipo 2 e, à direita, vemos baseada em contêiner virtualização, que exploraremos mais em breve.

A virtualização tipo 2 é o tipo implementado pela AWS, VMware, vSphere, Xen e KVM. (A virtualização tipo 1 se refere à tecnologia na qual as VMs é executado diretamente no hardware, não em cima de outro sistema operacional.) Em nosso infraestrutura física, temos um sistema operacional hospedeiro. Neste caso, corremos

algo chamado hipervisor, que tem duas funções principais. Primeiro, ele mapeia recursos como CPU e memória do host virtual para o host físico.

Em segundo lugar, ele atua como uma camada de controle, permitindo manipular o virtual máquinas em si.

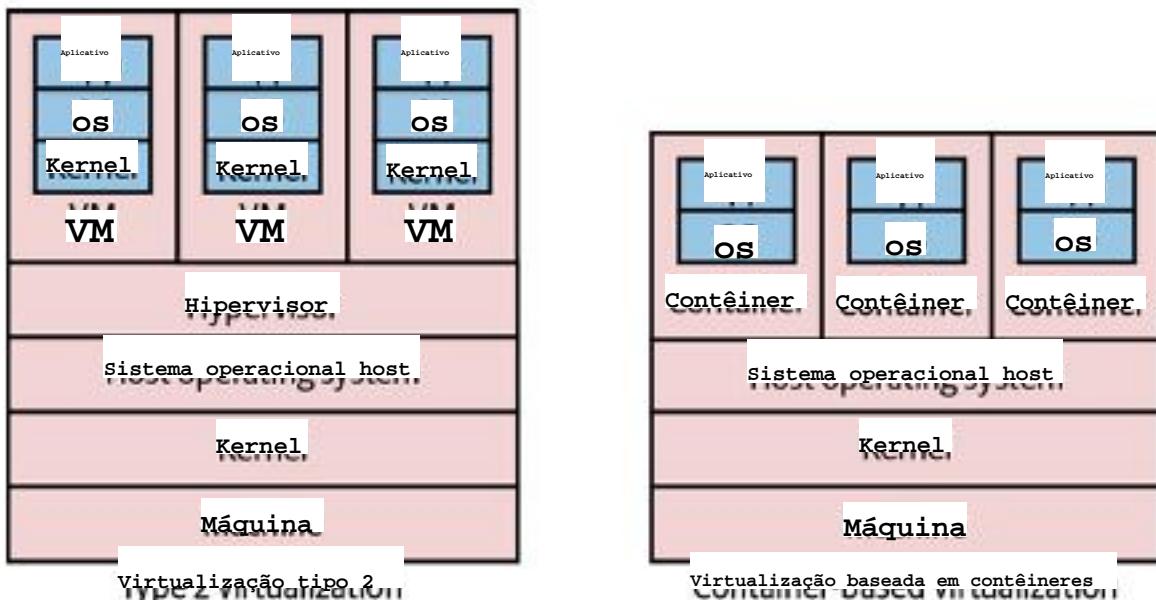


Figura 8-14. Comparação entre virtualização padrão tipo 2 e contêineres leves

Dentro das VMs, temos o que parecem ser hosts completamente diferentes. Eles podem executar seus próprios sistemas operacionais, com seus próprios kernels. Eles podem ser consideradas máquinas quase hermeticamente fechadas, mantidas isoladas do host físico subjacente e as outras máquinas virtuais do hipervisor.

O problema com a virtualização do tipo 2 é que o hipervisor aqui precisa ser configurado reservando recursos para fazer seu trabalho. Isso elimina a CPU, a E/S e a memória que poderia ser usado em outro lugar. Quanto mais hosts o hipervisor gerencia, mais recursos de que precisa. Em um determinado ponto, essa sobrecarga se torna uma restrição em fatiando ainda mais sua infraestrutura física. Na prática, isso significa que muitas vezes há retornos decrescentes em dividir uma caixa física em uma caixa menor e partes menores, à medida que proporcionalmente mais e mais recursos vão para o sobrecarga do hipervisor.

Bom para microsserviços?

Voltando aos nossos princípios, as máquinas virtuais se saem muito bem em termos de isolamento, mas com um custo. Sua facilidade de automação pode variar de acordo com a exatidão tecnologia sendo usada e gerenciada em VMs no Google Cloud, Azure ou AWS, para por exemplo, são todos fáceis de automatizar por meio de APIs bem suportadas e de um ecossistema de ferramentas que se baseiam nessas APIs. Além disso, essas plataformas fornecem conceitos como grupos de escalonamento automático, ajudando a implementar o estado desejado gestão. Implementar uma implantação sem tempo de inatividade exigirá mais funciona, mas se a plataforma de VM que você está usando fornecer uma boa API, a os blocos de construção estão lá. O problema é que muitas pessoas estão fazendo uso de VMs gerenciadas fornecidas por plataformas de virtualização tradicionais, como essas fornecidos pela VMware, que, embora teoricamente possam permitir automação, normalmente não são usadas nesse contexto. Em vez disso, essas plataformas tendem estar sob o controle central de uma equipe de operações dedicada e a capacidade. Como resultado, automatizar diretamente contra eles pode ser restringido.

Embora os contêineres estejam se mostrando mais populares em geral para cargas de trabalho de microserviços, muitas organizações usaram máquinas virtuais para executando sistemas de microserviços em grande escala, com grande efeito. Netflix, um dos garoto-propaganda de microserviços, construiu muitos de seus microserviços no topo das máquinas virtuais gerenciadas da AWS via EC2. Se você precisar do mais rigoroso níveis de isolamento que eles podem trazer, ou você não tem a capacidade de conteinerize seu aplicativo, as VMs podem ser uma ótima opção.

Contentores

Desde a primeira edição deste livro, os contêineres se tornaram dominantes conceito na implantação de software do lado do servidor e, para muitos, é o de fato opção para empacotar e executar arquiteturas de microserviços. O contêiner conceito, popularizado pelo Docker e aliado a um contêiner de suporte uma plataforma de orquestração, como o Kubernetes, tornou-se a preferida de muitas pessoas opção para executar arquiteturas de microserviços em grande escala.

Antes de explicarmos por que isso aconteceu e sobre a relação entre contêineres, Kubernetes e Docker, devemos primeiro explorar o que é um contêiner é exatamente, e veja especificamente como ele difere das máquinas virtuais.

Isolado, de forma diferente

Os contêineres surgiram pela primeira vez em sistemas operacionais no estilo Unix e para muitos anos foram, na verdade, apenas uma perspectiva viável nesses sistemas operacionais, como Linux. Embora os contêineres do Windows sejam muito comuns, foi o Linux sistemas operacionais nos quais os contêineres tiveram o maior impacto até agora.

No Linux, os processos são executados por um determinado usuário e têm certos recursos com base em como as permissões são definidas. Os processos podem gerar outros processos. Por exemplo, se eu inicio um processo em um terminal, esse processo geralmente é considerado filho do processo terminal. O trabalho do kernel Linux é mantendo essa árvore de processos, garantindo que somente usuários permitidos possam acessar os processos. Além disso, o kernel Linux é capaz de atribuir recursos para esses diferentes processos - tudo isso é parte integrante da construção de um sistema operacional multiusuário viável, onde você não quer as atividades de um usuário para matar o resto do sistema.

Os contêineres em execução na mesma máquina usam o mesmo suporte kernel (embora haja exceções a essa regra que exploraremos em breve). Em vez de gerenciar processos diretamente, você pode pensar em um contêiner como uma abstração sobre uma subárvore da árvore geral do processo do sistema, com o kernel fazendo todo o trabalho duro. Esses contêineres podem ter recursos físicos alocados a eles, algo que o kernel manipula para nós. Essa abordagem geral existe de várias formas, como Solaris Zones e OpenVZ, mas foi com o LXC que essa ideia chegou ao mainstream do Linux sistemas operacionais. O conceito de contêineres Linux foi ainda mais avançado quando o Docker forneceu um nível ainda mais alto de abstração em relação aos contêineres, usando inicialmente o LXC sob o capô e depois substituindo-o completamente.

Se observarmos o diagrama de pilha de um host executando um contêiner na Figura 8-14, vemos algumas diferenças ao compará-la com a virtualização tipo 2. Primeiro, não precisamos de um hipervisor. Em segundo lugar, o contêiner não parece ter um kernel - isso porque ele usa o kernel da máquina subjacente.

Na Figura 8-15, vemos isso com mais clareza. Um contêiner pode funcionar sozinho sistema operacional, mas esse sistema operacional faz uso de uma parte do compartilhado kernel - é nesse kernel que a árvore de processos de cada contêiner vive. Isso

significa que nosso sistema operacional hospedeiro pode executar o Ubuntu e nossos contêineres

CentOS, desde que ambos possam ser executados como parte do mesmo kernel subjacente.

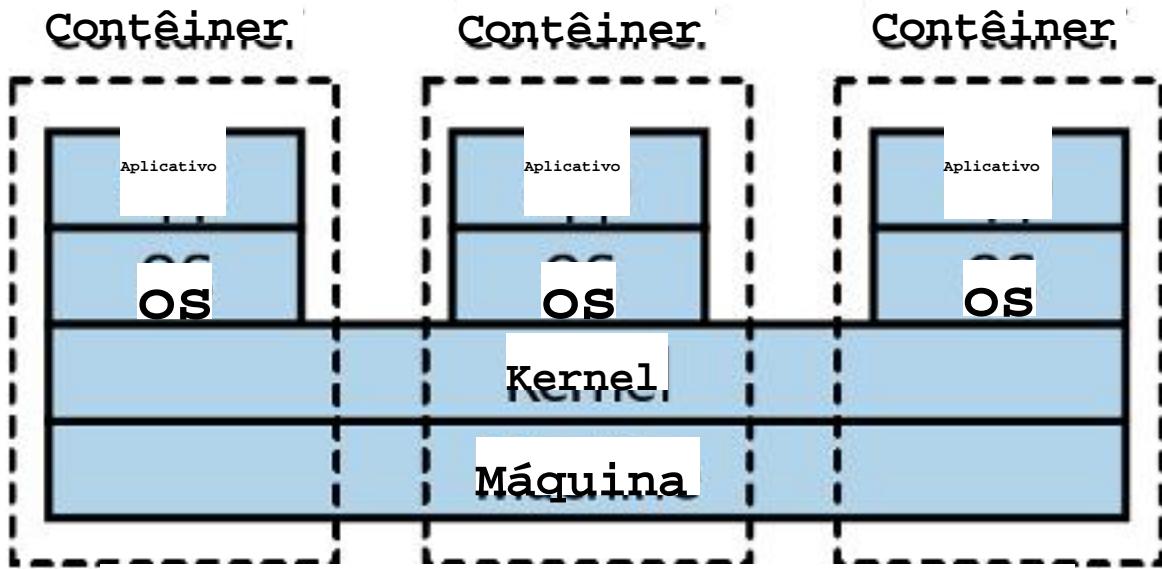


Figura 8-15. Normalmente, contêineres na mesma máquina compartilham o mesmo kernel

Com contêineres, não nos beneficiamos apenas dos recursos economizados por não precisando de um hipervisor; também ganhamos em termos de feedback. Os contêineres Linux são muito mais rápido de provisionar do que máquinas virtuais completas. Não é incomum que uma VM leva muitos minutos para começar, mas com contêineres Linux, a inicialização pode demorar apenas alguns segundos. Você também tem um controle mais refinado sobre o uso dos próprios contêineres em termos de atribuição de recursos a eles, o que torna é muito mais fácil ajustar as configurações para tirar o máximo proveito do subjacente hardware.

Devido à natureza mais leve dos contêineres, podemos ter muito mais eles são executados no mesmo hardware que seria possível com as VMs. Por implantando um serviço por contêiner, como na Figura 8-16, obtemos um grau de isolamento de outros contêineres (embora isso não seja perfeito) e pode fazer isso muito mais econômico do que seria possível se quiséssemos executar cada um serviço em sua própria VM. Voltando à nossa analogia com a gaveta de meias de anteriormente, com contêineres, as divisórias das gavetas de meias são muito mais finas do que elas são para VMs, o que significa que uma proporção maior da gaveta de meias é usada para meias.

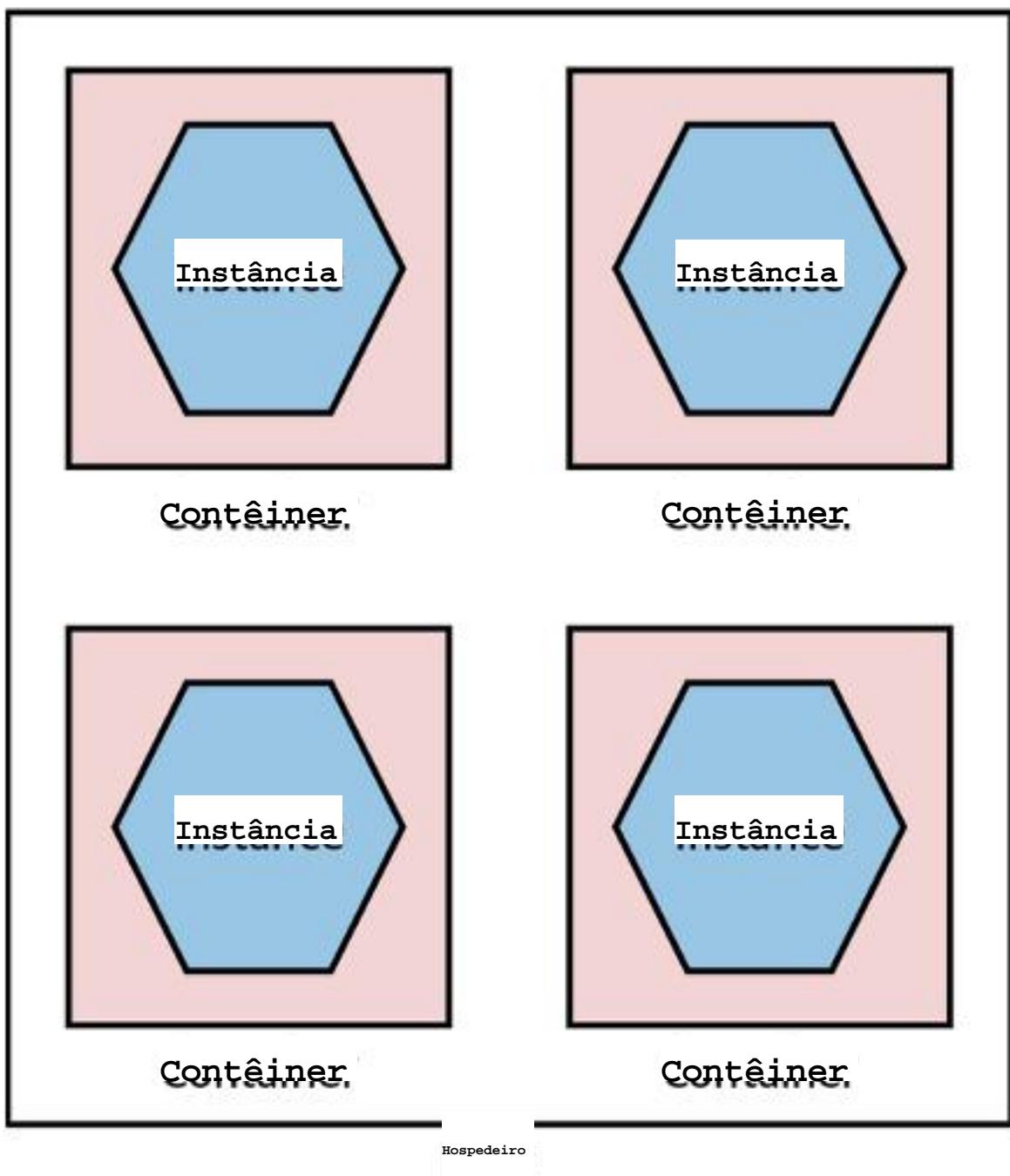


Figura 8-16. Executando serviços em contêineres separados

Os contêineres também podem ser usados bem com a virtualização completa; na verdade, isso é comum. Já vi mais de um projeto provisionar um grande AWS EC2 instancie e execute vários contêineres nela para obter o melhor dos dois mundos: um plataforma de computação efêmera sob demanda na forma de EC2, juntamente com contêineres altamente flexíveis e rápidos rodando sobre ele.

Não é perfeito

No entanto, os contêineres Linux apresentam alguns problemas. Imagine que eu tenho muitos de microserviços executados em seus próprios contêineres em um host. Como funciona o mundo exterior os vê? Você precisa de alguma maneira de direcionar o mundo exterior até os contêineres subjacentes, algo que muitos dos hipervisores fazem para você com virtualização normal. Com tecnologias anteriores, como a LXC, esta era algo que você tinha que cuidar sozinho - esta é uma área em que o Docker assumir contêineres ajudou muito.

Outro ponto a ter em mente é que esses contêineres podem ser considerados isolado do ponto de vista de um recurso - posso alocar conjuntos cercados de CPU, memória e assim por diante para cada contêiner, mas isso não é necessariamente o mesmo grau de isolamento que você obtém de máquinas virtuais ou, aliás, por ter máquinas físicas separadas. No início, havia uma série de maneiras documentadas e conhecidas pelas quais um processo de um contêiner poderia sair e interaja com outros contêineres ou com o hospedeiro subjacente.

Muito trabalho foi feito para resolver esses problemas, e os sistemas de orquestração de contêineres e tempos de execução de contêineres subjacentes fizeram um bom trabalho ao examinar como executar melhor as cargas de trabalho de contêineres, então isso o isolamento é melhorado, mas você precisará pensar devidamente nos tipos de cargas de trabalho que você deseja executar. Minha própria orientação aqui é que, em geral, você deve ver os contêineres como uma ótima maneira de isolar a execução de tarefas confiáveis software. Se você estiver executando um código escrito por outras pessoas e estiver preocupado com uma parte maliciosa tentando contornar o isolamento em nível de contêiner, então você vai querer fazer você mesmo um exame mais profundo sobre o estado da arte atual para lidar com essas situações, algumas das quais abordaremos em breve.

Contêineres Windows

Historicamente, os usuários do Windows olhavam ansiosamente para o uso do Linux contemporâneos, pois os contêineres eram algo negado ao Windows sistema operacional. Nos últimos anos, no entanto, isso mudou, com contêineres agora são um conceito totalmente suportado. O atraso foi, na verdade, quase o sistema operacional Windows subjacente e o kernel que suportam o mesmo tipos de recursos existentes na terra do Linux para fazer os contêineres funcionarem.

Foi com a entrega do Windows Server 2016 que muito disso mudou, e, desde então, os contêineres do Windows continuaram evoluindo.

Um dos obstáculos iniciais na adoção de contêineres Windows tem sido o tamanho do próprio sistema operacional Windows. Lembre-se de que você precisa executar um sistema operacional dentro de cada contêiner, portanto, ao fazer o download uma imagem de contêiner, você também está baixando um sistema operacional. Janelas, porém, é grande - tão grande que tornou os contêineres muito pesados, não apenas em termos de o tamanho das imagens, mas também em termos dos recursos necessários para executá-las.

A Microsoft reagiu a isso criando um sistema operacional reduzido chamado Servidor Windows Nano. A ideia é que o Nano Server tenha um pequeno footprint OS e seja capaz de executar coisas como instâncias de microserviços. Além disso, a Microsoft também oferece suporte a um Windows Server Core maior, que existe para oferecer suporte à execução de aplicativos legados do Windows como contêineres. O problema é que essas coisas ainda são muito grandes quando comparadas ao Linux equivalentes - as primeiras versões do Nano Server ainda estariam bem acima de 1 GB em tamanho, em comparação com sistemas operacionais Linux de pequeno porte, como o Alpine, que ocuparia apenas alguns megabytes.

Embora a Microsoft tenha continuado tentando reduzir o tamanho do Nano Server, este a disparidade de tamanho ainda existe. Na prática, porém, devido à forma como é comum camadas em imagens de contêiner podem ser armazenadas em cache, isso pode não ser uma grande quantidade problema.

De especial interesse no mundo dos contêineres Windows é o fato de que eles suportam diferentes níveis de isolamento. Um contêiner padrão do Windows usa isolamento de processos, assim como seus equivalentes Linux. Com o isolamento do processo, cada contêiner é executado em parte do mesmo kernel subjacente, que gerencia o isolamento entre os contêineres. Com os contêineres do Windows, você também tem a opção de fornecer mais isolamento executando contêineres dentro de seus próprios VM Hyper-V. Isso lhe dá algo mais próximo do nível de isolamento total virtualização, mas o bom é que você pode escolher entre Hyper-V ou isolamento do processo quando você inicia o contêiner - a imagem não precisa mudar.

Ter flexibilidade para executar imagens em diferentes tipos de isolamento pode... tenha seus benefícios. Em algumas situações, seu modelo de ameaça pode exigir que você deseja um isolamento mais forte entre seus processos em execução do que um processo simples isolamento de nível. Por exemplo, você pode estar executando terceiros "não confiáveis" codifique junto com seus próprios processos. Em tal situação, ser capaz de correr essas cargas de trabalho de contêineres como contêineres Hyper-V são muito úteis. Nota, de claro, é provável que o isolamento do Hyper-V tenha um impacto em termos de rotação tempo e um custo de tempo de execução mais próximo ao da virtualização normal.

LINHAS BORRADAS

Há uma tendência crescente de pessoas que procuram soluções que fornecam a isolamento mais forte fornecido pelas VMs, embora tenha a natureza leve de contêineres. Os exemplos incluem os contêineres Hyper-V da Microsoft, que permitem núcleos separados e Firecracker, que é confusamente chamado uma VM baseada em kernel. Firecracker provou ser popular como detalhes de implementação de ofertas de serviços como o AWS Lambda, onde é necessário isolar totalmente as cargas de trabalho de diferentes clientes enquanto ainda tentando reduzir o tempo de rotação e reduzir a operação área ocupada pelas cargas de trabalho.

Docker

Os contêineres estavam em uso limitado antes do surgimento do Docker impulsionar o conceito mainstream. O conjunto de ferramentas do Docker lida com grande parte da solução alternativa contêineres. O Docker gerencia o provisionamento de contêineres, lida com alguns dos problemas de rede para você e até fornece seu próprio conceito de registro que permite que você armazene aplicativos Docker. Antes do Docker, não tínhamos o conceito de uma "imagem" para contêineres - esse aspecto, junto com um aspecto muito melhor conjunto de ferramentas para trabalhar com contêineres, ajudou os contêineres a se tornarem muitos mais fácil de usar.

A abstração da imagem do Docker é útil para nós, pois os detalhes de como nossos os microserviços implementados estão ocultos. Temos as construções para nossos microsserviço: crie uma imagem Docker como um artefato de construção e armazene a imagem em

o registro do Docker e vamos lá. Quando você executa uma instância de um

Imagem do Docker, você tem um conjunto genérico de ferramentas para gerenciar essa instância, não

importam a tecnologia subjacente usada - microsserviços escritos em Go, Python,

NodeJS, ou qualquer outra coisa, pode ser tratada da mesma forma.

O Docker também pode aliviar algumas das desvantagens de executar muitos serviços

localmente para fins de desenvolvimento e teste. Anteriormente, eu poderia ter usado uma ferramenta como

Vagrant que me permite hospedar várias VMs independentes em meu desenvolvimento

máquina. Isso me permitiria ter uma VM semelhante à de produção executando meu

instâncias de serviço localmente. Essa foi uma abordagem bastante pesada, no entanto,

e eu estaria limitado em quantas VMs eu poderia executar. Com o Docker, é fácil, simplesmente

para executar o Docker diretamente na minha máquina de desenvolvedor, provavelmente usando o Docker

Desktop. Agora posso simplesmente criar uma imagem do Docker para minha instância de microsserviço,

ou extraia uma imagem pré-criada e execute-a localmente. Essas imagens do Docker podem

(e deve) ser idêntica à imagem do contêiner na qual eu eventualmente executarei

produção.

Quando o Docker surgiu pela primeira vez, seu escopo estava limitado ao gerenciamento de contêineres em

uma máquina. Isso era de uso limitado - e se você quisesse gerenciar

contêineres em várias máquinas? Isso é algo essencial se

você deseja manter a integridade do sistema, se tiver uma máquina morrendo em você, ou se

você só quer executar contêineres suficientes para lidar com a carga do sistema. Docker

lançou dois produtos próprios totalmente diferentes para resolver esse problema,

confusamente chamado de "Docker Swarm" e "Docker Swarm Mode" - quem disse

nomear coisas foi difícil de novo? Realmente, porém, quando se trata de gerenciar lotes

de contêineres em várias máquinas, o Kubernetes é o rei aqui, mesmo se você

pode usar a cadeia de ferramentas Docker para criar e gerenciar indivíduos

contêineres.

Aptidão para microsserviços

Os contêineres, como conceito, funcionam perfeitamente bem para microsserviços e

O Docker tornou os contêineres significativamente mais viáveis como conceito. Recebemos nosso

isolamento, mas a um custo gerenciável. Também escondemos a tecnologia subjacente,

permitindo-nos misturar diferentes pilhas tecnológicas. Quando se trata de implementar

conceitos como gerenciamento de estado desejado, porém, precisaremos de algo como Kubernetes para lidar com isso para nós.

O Kubernetes é importante o suficiente para justificar uma discussão mais detalhada, então voltaremos a isso mais tarde neste capítulo. Mas, por enquanto, pense nisso como uma forma de gerenciar contêineres em muitas máquinas, o que é suficiente para o momento.

Contentores de aplicativos

Se você estiver familiarizado com a implantação de aplicativos .NET por trás do IIS ou do Java aplicativos em algo como Weblogic ou Tomcat, você ficará bem familiarizado com o modelo no qual vários serviços ou aplicativos distintos ficam dentro de um único contêiner de aplicativo, que, por sua vez, fica em um único host, como vemos na Figura 8-17. A ideia é que o contêiner do aplicativo oferece benefícios em termos de melhor capacidade de gerenciamento, como suporte de agrupamento para lidar com o agrupamento de várias instâncias, ferramentas de monitoramento e afins.

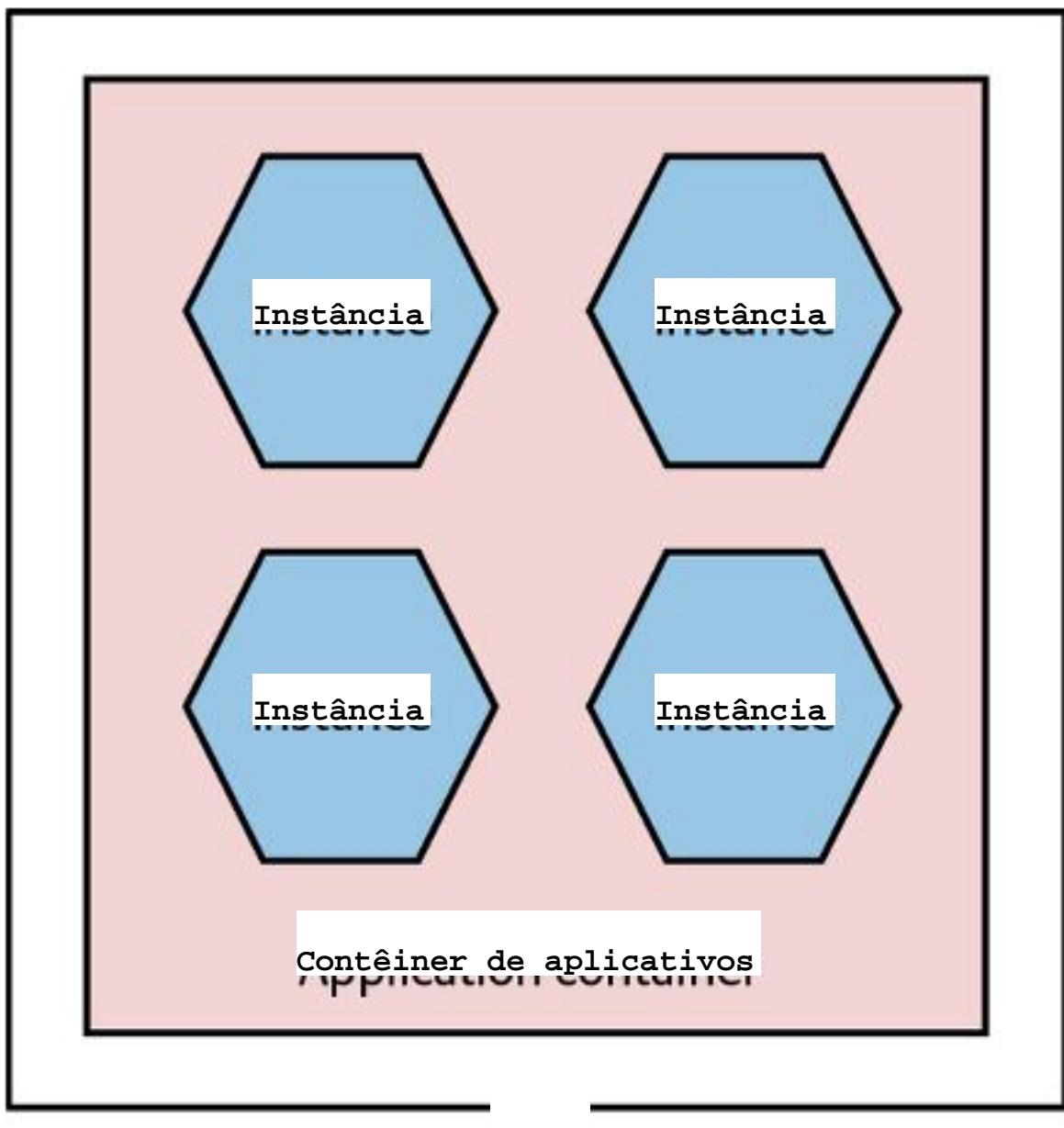


Figura 8-17. Vários microservicos por contêiner de aplicativos

Essa configuração também pode gerar benefícios em termos de redução da sobrecarga de linguagem de tempo de execução. Considere executar cinco serviços Java em um único servlet Java no recipiente. Eu tenho a sobrecarga de apenas uma única JVM. Compare isso com executando cinco JVMs independentes no mesmo host ao usar contêineres. Isso disse, ainda acho que esses contêineres de aplicativos têm desvantagens suficientes para você deve se desafiar para ver se eles são realmente necessários.

A primeira das desvantagens é que elas inevitavelmente restringem a tecnologia escolha. Você precisa comprar uma pilha de tecnologia. Isso pode limitar não apenas o opções de tecnologia para a implementação do serviço em si, mas também do opções que você tem em termos de automação e gerenciamento de seus sistemas. Como discutiremos em breve, uma das maneiras pelas quais podemos lidar com a sobrecarga de gerenciar vários hosts é feito com automação e, portanto, restringindo nossas opções resolver isso pode muito bem ser duplamente prejudicial.

Eu também questionaria parte do valor dos recursos fornecidos por esses contêineres de aplicativos. Muitos deles destacam a capacidade de gerenciar clusters para suporta o estado de sessão compartilhada na memória, algo que absolutamente queremos evite, em qualquer caso, devido aos desafios que isso cria ao escalar nosso serviços. E os recursos de monitoramento que eles fornecem não serão suficientes quando consideramos os tipos de monitoramento conjunto que queremos fazer em um mundo dos microserviços, como veremos no Capítulo 10. Muitos deles também têm tempos de rotação bastante lentos, afetando os ciclos de feedback dos desenvolvedores.

Também existem outros conjuntos de problemas. Tentando fazer um ciclo de vida adequado o gerenciamento de aplicativos em cima de plataformas como a JVM pode ser problemático e mais complexo do que simplesmente reiniciar uma JVM. Analisando o uso de recursos e tópicos também é muito mais complexo, pois você tem vários aplicativos que compartilham o mesmo processo. E lembre-se, mesmo se você conseguir valorizar os contêineres de tecnologia específica, eles não são gratuitos. Além do fato de que muitos deles são comerciais e, portanto, têm uma implicação de custo, eles adicionam uma sobrecarga de recursos por si só.

Em última análise, essa abordagem é novamente uma tentativa de otimizar a escassez de recursos que simplesmente podem não aguentar mais. Quer você decida ou não ter vários serviços por host como modelo de implantação, eu gostaria fortemente sugiro considerar microservices implantáveis independentes como artefatos, com cada instância de microservice em execução como seu próprio processo isolado.

Fundamentalmente, a falta de isolamento que esse modelo fornece é uma das principais razões pelas quais esse modelo é cada vez mais raro para pessoas que adotam arquiteturas de microserviços.

Plataforma como Serviço (PaaS)

Ao usar a Plataforma como Serviço (PaaS), você está trabalhando em um nível superior abstração do que um único host. Algumas dessas plataformas dependem de uma artefato específico de tecnologia, como um arquivo Java WAR ou uma gem Ruby, e provisionando, e executando automaticamente para você. Algumas dessas plataformas tentarão, de forma transparente, lidar com a escalabilidade do sistema para cima e para baixo para você; outros permitirão que você tenha algum controle sobre quantos nós seu serviço pode correr, mas eles cuidam do resto.

Como foi o caso quando escrevi a primeira edição, a maioria das melhores, a maioria soluções PaaS refinadas são hospedadas. A Heroku estabeleceu a referência para oferecer uma interface amigável para desenvolvedores e, sem dúvida, continua sendo o padrão-ouro para PaaS, apesar do crescimento limitado em termos de seu conjunto de recursos nos últimos anos. Plataformas como o Heroku não apenas executam sua instância de aplicativo; elas também fornecem recursos como a execução de instâncias de banco de dados para você algo que pode ser muito doloroso de fazer sozinho.

Quando as soluções PaaS funcionam bem, elas realmente funcionam muito bem. No entanto, quando eles não funcionam bem para você, muitas vezes você não tem muito controle sobre termos de se esconder para consertar as coisas. Isso faz parte da troca de você fazer. Eu diria que, na minha experiência, quanto mais inteligentes as soluções PaaS tentam ser Seja, quanto mais eles erram. Eu usei mais de um PaaS que tenta escala automática com base no uso do aplicativo, mas faz isso mal. Invariavelmente, as heurísticas que impulsionam essa inteligência tendem a ser personalizadas para a média aplicativo em vez de seu caso de uso específico. Quanto mais atípico for seu aplicativo, maior a probabilidade de não funcionar bem com um PaaS.

Como as boas soluções de PaaS cuidam muito de você, elas podem ser excelentes maneira de lidar com o aumento da sobrecarga que obtemos com muito mais partes móveis. Dito isso, ainda não tenho certeza se temos todos os modelos corretos nesse espaço ainda, e as opções limitadas de hospedagem própria significam que essa abordagem pode não funcionar para você. Quando escrevi a primeira edição, eu tinha esperança de que veríamos mais crescimento neste espaço, mas isso não aconteceu da maneira que eu esperado. Em vez disso, acho que o crescimento dos produtos sem servidor foi oferecido principalmente pelos provedores de nuvem pública começou a suprir essa necessidade. Em vez de

oferecendo plataformas de caixa preta para hospedar um aplicativo, eles fornecem soluções gerenciadas prontas para uso para coisas como corretores de mensagens, bancos de dados, armazenamento, e coisas que nos permitem misturar e combinar as peças que gostamos de construir o que precisamos. É nesse contexto que funciona como um serviço, um tipo específico de produto sem servidor, vem ganhando muita força.

É difícil avaliar a adequação das ofertas de PaaS para microserviços, pois eles vêm em várias formas e tamanhos. O Heroku parece bem diferente de Netlify, por exemplo, mas ambos podem funcionar para você como uma plataforma de implantação para seus microserviços, dependendo da natureza do seu aplicativo.

Função como serviço (FaaS)

Nos últimos anos, a única tecnologia que chegou ainda mais perto do Kubernetes em termos de geração de hype (pelo menos no contexto de microserviços) são sem servidor. Sem servidor é, na verdade, um termo genérico para uma série de diferentes tecnologias em que, do ponto de vista da pessoa que as utiliza, os computadores subjacentes não importam. Os detalhes do gerenciamento e da configuração das máquinas são tiradas de você. Nas palavras de Ken Fromm (que, até eu posso dizer que cunhei o termo sem servidor):

A frase "sem servidor" não significa que os servidores não estejam mais envolvidos. É simplesmente significa que os desenvolvedores não precisam mais pensar muito sobre eles. Os recursos de computação são usados como serviços sem a necessidade de gerenciar as capacidades ou limites físicos. Provedores de serviços assumem cada vez mais a responsabilidade de gerenciar servidores e armazenamentos de dados e outros recursos de infraestrutura. Os desenvolvedores poderiam configurar seus próprios soluções de código aberto, mas isso significa que eles precisam gerenciar os servidores e as filas e as cargas.

-Ken Fromm, "Por que o futuro do software e dos aplicativos é
Sem servidor"

A função como serviço, ou FaaS, tornou-se uma parte importante do sistema sem servidor que, para muitos, os dois termos são intercambiáveis. Isso é lamentável, pois ignora a importância de outros produtos sem servidor, como bancos de dados,

filas, soluções de armazenamento e similares. No entanto, fala com o o entusiasmo que o FaaS gerou por ter dominado a discussão.

Foi o produto Lambda da AWS, lançado em 2014, que despertou a empolgação em torno do FaaS. Em um nível, o conceito é deliciosamente simples. Você implanta algum código (uma "função"). Esse código está inativo, até que algo aconteça com acionar esse código. Você está encarregado de decidir o que pode ser esse gatilho. Pode ser um arquivo chegando em um determinado local, um item aparecendo em uma mensagem fila, uma chamada recebida via HTTP ou outra coisa.

Quando sua função é acionada, ela é executada e, quando termina, é desligada. O a plataforma subjacente manipula a rotação dessas funções para cima ou para baixo sob demanda e lidará com execuções simultâneas de suas funções para que você possa ter várias cópias em execução ao mesmo tempo, quando apropriado.

Os benefícios aqui são vários: o código que não está sendo executado não está custando caro. Dinheiro - você paga apenas pelo que usa. Isso pode tornar o FaaS uma ótima opção para situações em que você tem carga baixa ou imprevisível. O subjacente a plataforma controla a rotação das funções para cima e para baixo para você, oferecendo algum grau de alta disponibilidade e robustez implícitas sem que você tenha para fazer qualquer trabalho. Fundamentalmente, o uso de uma plataforma FaaS, como acontece com muitas as outras ofertas sem servidor permitem que você reduza drasticamente a quantidade de sobrecarga operacional com a qual você precisa se preocupar.

Limitações

Nos bastidores, todas as implementações de FaaS que conheço usam algumas tipo de tecnologia de contêineres. Isso está escondido de você - normalmente você não tem que se preocupar em construir um contêiner que será executado, basta fornecer alguma forma empacotada do código. Isso significa, porém, que você não tem um diploma de controle sobre o que exatamente pode ser executado; como resultado, você precisa do FaaS provedor para oferecer suporte ao idioma de sua escolha. O Azure Functions fez o melhor aqui em termos dos principais fornecedores de nuvem, suportando uma ampla variedade de diferentes tempos de execução, enquanto a própria oferta Cloud Functions do Google Cloud suporta muito poucos idiomas em comparação (no momento em que este artigo foi escrito, Google suporta apenas Go, algumas versões do Node e Python). Vale a pena notar que A AWS agora permite que você defina seu próprio tempo de execução personalizado para seu

funcões, teoricamente permitindo que você implemente suporte para linguagens que não são fornecidos fora da caixa, embora isso se torne outra peça de sobrecarga operacional que você precisa manter.

Essa falta de controle sobre o tempo de execução subjacente também se estende à falta de controle sobre os recursos fornecidos para cada invocação de função. Em todo o Google Cloud, Azure e AWS, você só pode controlar a memória dada a cada uma função. Isso, por sua vez, parece implicar que uma certa quantidade de CPU e E/S é dada ao tempo de execução da sua função, mas você não pode controlar esses aspectos diretamente. Isso pode significar que você acaba tendo que dar mais memória a uma função mesmo que não seja necessário, apenas para obter a CPU de que você precisa. Em última análise, se você sentir que você precisa fazer muitos ajustes nos recursos disponíveis para seu funcionamento, eu acho que, pelo menos nesta fase, o FaaS provavelmente não é um ótimo opção para você.

Outra limitação a ser observada é que as invocações de funções podem fornecer limites em termos de quanto tempo eles podem durar. Funções do Google Cloud, para exemplo, atualmente estão limitados a 9 minutos de execução, enquanto o AWS Lambda as funções podem ser executadas por até 15 minutos. As funções do Azure podem ser executadas para sempre se você quer (dependendo do tipo de plano em que você está). Pessoalmente, acho que se você ter funções em execução por longos períodos de tempo, isso provavelmente aponta para o tipo de problema para o qual as funções não são adequadas.

Finalmente, a maioria das invocações de funções é considerada sem estado. Conceitualmente, isso significa que uma função não pode acessar o estado deixado por um anterior invocação de função, a menos que esse estado seja armazenado em outro lugar (por exemplo, em um banco de dados). Isso tornou difícil ter várias funções encadeadas.

- considere uma função orquestrando uma série de chamadas para outra no downstream funções. Uma exceção notável é o Azure Durable Functions, que resolve isso problema de uma forma muito interessante. O Durable Functions suporta a capacidade de suspender o estado de uma determinada função e permitir que ela reinicie onde o invocação deixada de lado - tudo isso é tratado de forma transparente por meio do uso de extensões reativas. Esta é uma solução que eu acho que é significativamente maior. Mais fácil para desenvolvedores do que o Step Functions da AWS, que se une várias funções usando a configuração baseada em JSON.

MONTAGEM NA WEB (WASM)

Wasm é um padrão oficial que foi originalmente definido para fornecer desenvolvedores uma forma de executar programas em sandbox escritos em uma variedade de linguagens de programação em navegadores clientes. Definindo uma embalagem formato e um ambiente de tempo de execução, o objetivo do Wasm é permitir arbitrários código para ser executado de forma segura e eficiente em dispositivos clientes. Isso pode permitir a criação de aplicativos muito mais sofisticados do lado do cliente ao usar navegadores da web normais. Como exemplo concreto, o eBay usou Wasm fornecerá software de scanner de código de barras, cujo núcleo era escrito em C/C++ e anteriormente estava disponível apenas para nativos Aplicativos Android ou iOS, para a web.²

A Interface do Sistema WebAssembly (WASI) foi definida como uma forma de permitir o Wasm sair do navegador e funciona em qualquer lugar com um WASI compatível a implementação pode ser encontrada. Um exemplo disso é a capacidade de executar Wasm em redes de entrega de conteúdo como Fastly ou Cloudflare.

Devido à sua natureza leve e aos fortes conceitos de sandboxing incorporados de acordo com sua especificação principal, o Wasm tem o potencial de desafiar o uso de contêineres como o formato de implantação ideal para aplicativos do lado do servidor. No curto prazo, o que está impedindo isso é provavelmente o lado do servidor plataformas disponíveis para executar o Wasm. Embora você possa teoricamente executar o Wasm no Kubernetes, por exemplo, você acaba incorporando o Wasm dentro contêineres, o que, sem dúvida, acaba sendo um pouco inútil, já que você executando uma implantação mais leve dentro de uma (comparativamente) mais recipiente pesado.

Uma plataforma de implantação do lado do servidor com suporte nativo para WASI seria provavelmente necessário para tirar o máximo proveito do potencial do Wasm. Teoricamente em pelo menos, um programador como o Nomad estaria em melhor posição para apoiar o Wasm, pois suporta um modelo de driver conectável. O tempo dirá...

Desafios

Além das limitações que acabamos de analisar, existem outras desafios que você pode enfrentar ao usar o FaaS.

Em primeiro lugar, é importante abordar uma preocupação que geralmente é levantada com o FaaS e essa é a noção de tempo de rotação. Conceitualmente, as funções não estão sendo executadas em todos, a menos que sejam necessários. Isso significa que eles precisam ser lançados para servir uma solicitação recebida. Agora, para alguns tempos de execução, leva muito tempo para iniciar uma nova versão do tempo de execução, geralmente chamada de tempo de "inicialização a frio". JVM e .NET os tempos de execução sofrem muito com isso, então um tempo de início frio para funções que os usam os tempos de execução geralmente podem ser significativos.

Na realidade, porém, esses tempos de execução raramente iniciam a frio. Pelo menos na AWS, os tempos de execução são mantidos "aquecidos", para que as solicitações recebidas já sejam atendidas por instâncias lançadas e em execução. Isso acontece de tal forma que pode ser difícil avaliar o impacto de uma "partida a frio" hoje em dia devido ao otimizações que estão sendo feitas nos bastidores pelos fornecedores de FaaS. No entanto, se isso for uma preocupação, opte por linguagens cujos tempos de execução tenham um rápido aumento times (Go, Python, Node.js, Ruby, etc). Portanto, car esse problema efetivamente.

Finalmente, o aspecto de escala dinâmica das funções pode, na verdade, acabar sendo um problema. As funções são iniciadas quando acionadas. Todas as plataformas que eu usei ter um limite rígido no número máximo de invocações simultâneas, que é algo que você talvez precise observar com cuidado. Eu falei com mais de uma equipe que teve o problema de aumentar a escala de funções e sobrecarregando outras partes de sua infraestrutura que não tinham a mesma propriedades de escala. Steve Faulkner, da Bustle, compartilhou um exemplo, onde as funções de escalabilidade sobrecarregaram a infraestrutura Redis da Bustle, causando problemas de produção. Se uma parte do seu sistema puder ser dimensionada dinamicamente, mas outras partes do seu sistema não, então você pode descobrir que essa incompatibilidade pode causar dores de cabeça significativas.

Mapeamento para microserviços

Até agora, em nossas discussões sobre as várias opções de implantação, o mapeamento de uma instância de microserviço a um mecanismo de implantação tem sido bonito e simples. Uma única instância de microserviço pode ser implantada em um

máquina virtual, empacotada como um único contêiner, ou até mesmo colocada em um contêiner de aplicativos como Tomcat ou IIS. Com o FaaS, as coisas ficam um pouco mais confuso.

Função por microsserviço

Agora, obviamente, uma única instância de microsserviço pode ser implantada como uma única função, conforme mostrado na Figura 8-18. Este é provavelmente um lugar sensato para começar. Isso mantém o conceito de uma instância de microsserviço como uma unidade de implantação, que é o modelo que mais exploramos até agora.

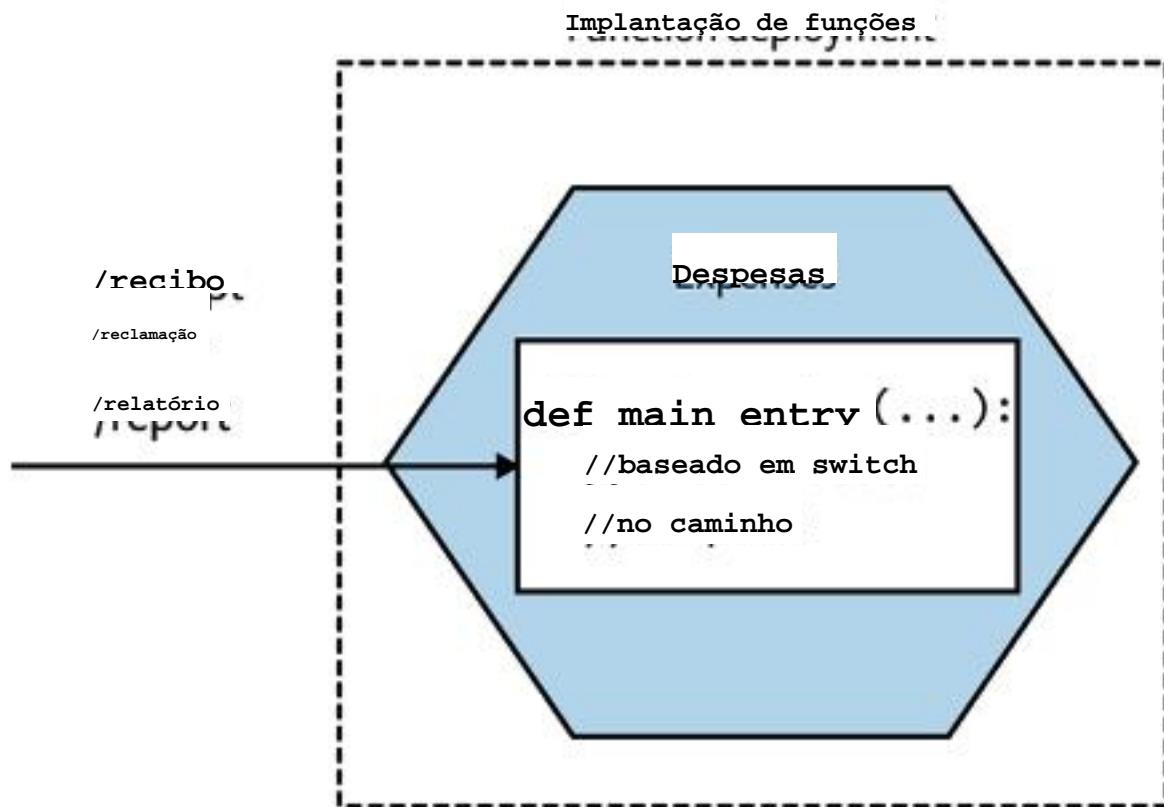


Figura 8-18. Nosso serviço de despesas é implementado como uma única função

Quando invocada, a plataforma FaaS acionará um único ponto de entrada em seu função implantada. Isso significa que se você vai ter uma única função implantação para todo o seu serviço, você precisará ter alguma maneira de despachando desse ponto de entrada para as diferentes partes da funcionalidade em seu microsserviço. Se você estivesse implementando o serviço de despesas como Microsserviço baseado em REST, você pode ter vários recursos expostos, como /recibo, /reclamação ou /relatório. Com este modelo, uma solicitação para qualquer um desses

os recursos entrariam por esse mesmo ponto de entrada, então você precisaria direcione a chamada recebida para a funcionalidade apropriada com base na caminho da solicitação de entrada.

Função por agregado

Então, como dividiríamos uma instância de microsserviço em funções menores? Se você está fazendo uso do design orientado por domínio, talvez já tenha explicitamente modelou seus agregados (uma coleção de objetos que são gerenciados como um único entidade, normalmente se referindo a conceitos do mundo real). Se o seu microsserviço a instância lida com vários agregados, um modelo que faz sentido para mim é separar uma função para cada agregado, conforme mostrado na Figura 8-19. Isso garante que toda a lógica de um único agregado seja independente dentro do função, facilitando a garantia de uma implementação consistente da vida-gerenciamento do ciclo do agregado.

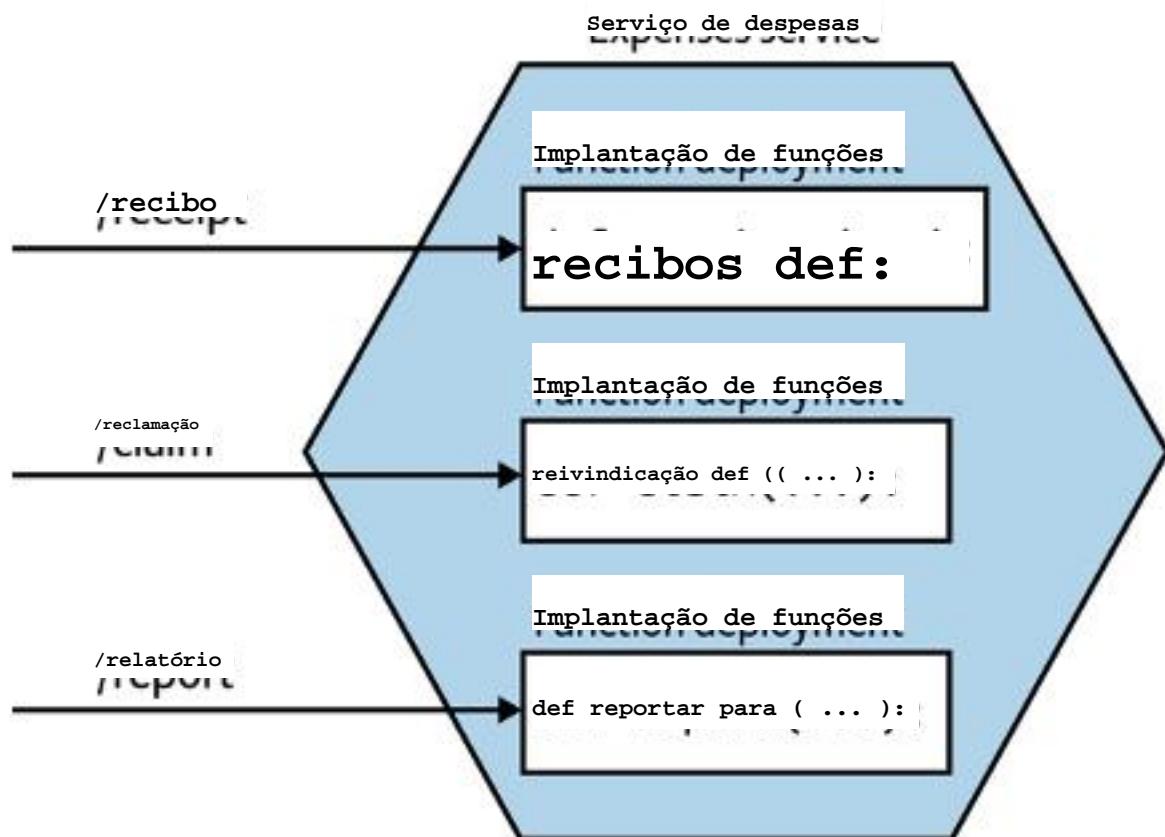


Figura 8-19. Um serviço de despesas sendo implantado como várias funções, cada uma lidando com um agregado diferente

Com esse modelo, nossa instância de microserviço não é mais mapeada para uma única unidade de implantação. Em vez disso, nosso microserviço agora é mais um conceito lógico consistindo em várias funções diferentes que teoricamente podem ser implantadas independentemente um do outro.

Algumas ressalvas aqui. Em primeiro lugar, eu recomendo fortemente que você mantenha uma forma mais grosseira-interface externa granulada. Para os consumidores upstream, eles ainda estão conversando com o Serviço de despesas - eles não sabem que as solicitações são mapeadas para menores - agregados com escopo definido. Isso garante que, caso você mude de ideia e queira para recombinar coisas ou até mesmo reestruturar o modelo agregado, você não impacte os consumidores upstream.

A segunda questão está relacionada aos dados. Esses agregados devem continuar a usar um banco de dados compartilhado? Sobre esse assunto, estou um pouco relaxado. Supondo que a mesma equipe gerencia todas essas funções e, conceitualmente, continua sendo uma um único "serviço", eu ficaria bem se eles ainda usassem o mesmo banco de dados, pois A Figura 8-20 mostra.

Com o tempo, porém, se as necessidades de cada função agregada divergirem, eu seria inclinados a procurar separar seu uso de dados, como pode ser visto na Figura 8-21, especialmente se você começar a ver que o acoplamento na camada de dados prejudica sua capacidade de altere-os facilmente. Nesse estágio, você poderia argumentar que essas funções seriam agora são microserviços por si só, embora, como acabei de explicar, pode haver valor em ainda representá-los como um único microserviço para consumidores upstream.

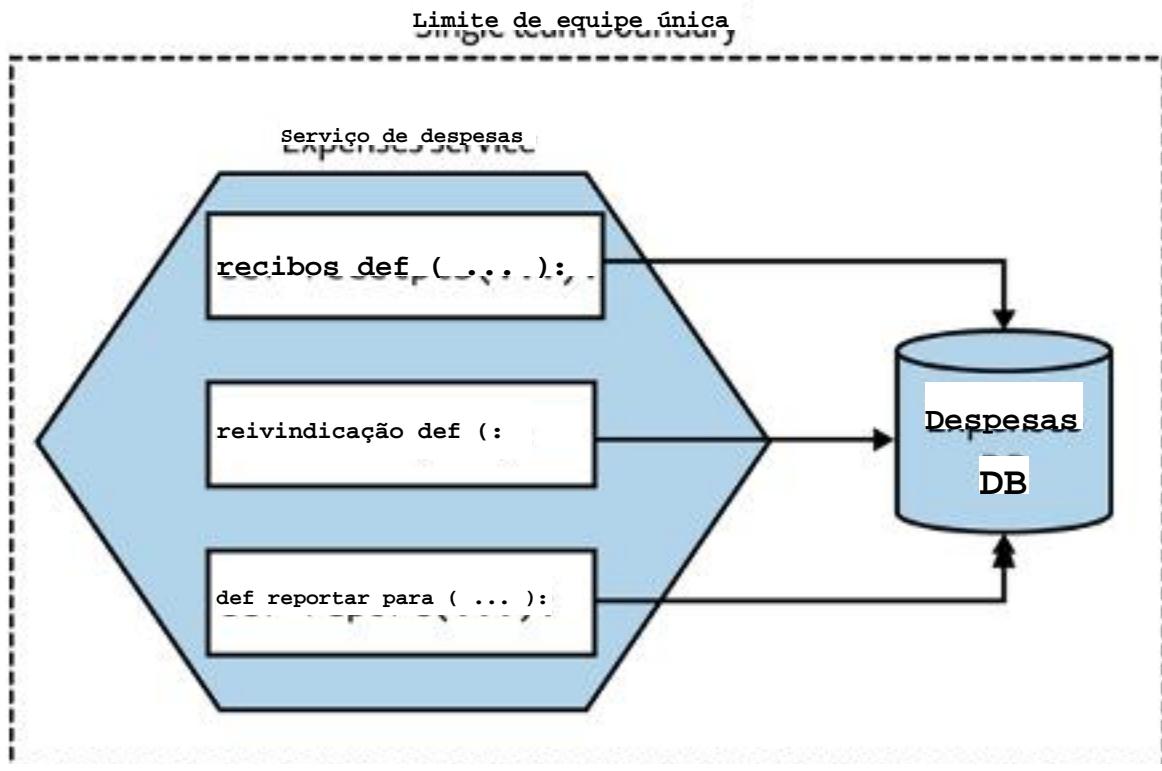


Figura 8-20. Funções diferentes usando o mesmo banco de dados, pois todas fazem parte logicamente do mesmo microserviço e são gerenciados pela mesma equipe

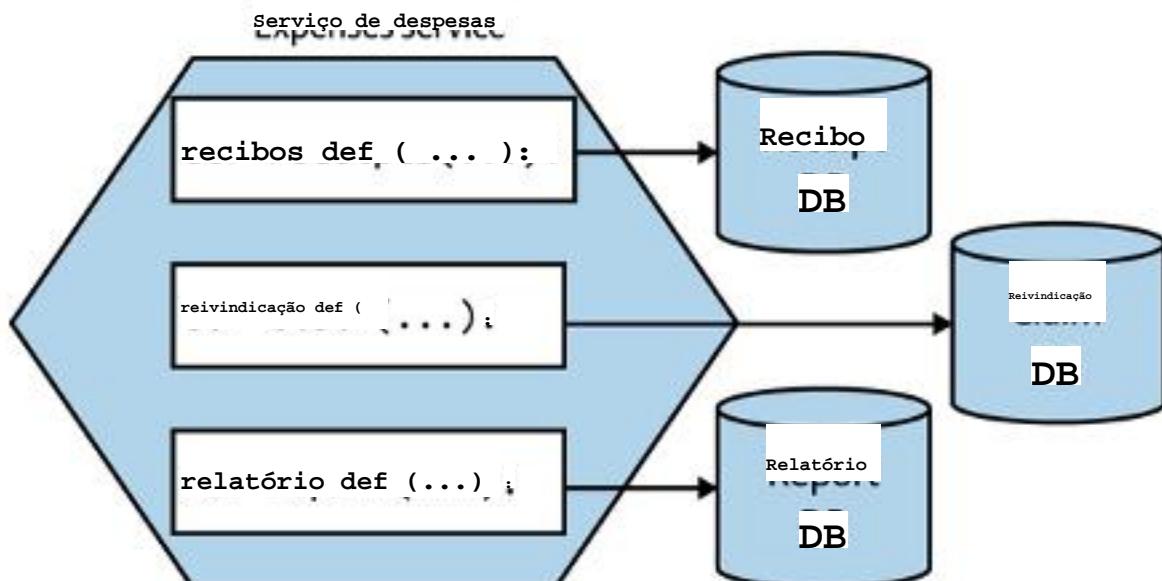


Figura 8-21. Cada função usando seu próprio banco de dados

Esse mapeamento de um único microserviço para vários mais refinados unidades implantáveis distorcem um pouco nossa definição anterior de microserviço. Normalmente, consideramos um microserviço como sendo implantável de forma independente.

unidade-agora, um microsserviço é composto por vários diferentes de forma independente unidades implantáveis. Conceitualmente, neste exemplo, o microsserviço se move no sentido de ser mais um conceito lógico do que físico.

Seja ainda mais refinado

Se você quiser ficar ainda menor, existe a tentação de quebrar seu função por agregado em partes menores. Eu sou muito mais cauteloso aqui. Além da explosão de funções que isso provavelmente criará, também viola um dos princípios fundamentais de um agregado - que queremos tratá-lo como um único unidade para garantir que possamos gerenciar melhor a integridade do próprio agregado.

Eu já tive a ideia de fazer cada transição de estado de um agrava sua própria função, mas eu desisti dessa ideia devido ao problemas associados à inconsistência. Quando você tem diferentes itens implantáveis de forma independente, cada um gerenciando uma parte diferente de um conjunto a transição do estado, garantir que as coisas sejam feitas corretamente, fica realmente muito difícil. É nos coloca no espaço das sagas, que discutimos no Capítulo 6. Quando implementando processos de negócios complexos, conceitos como sagas são importantes, e o trabalho é justificável. Eu me esforço, porém, para ver o valor de adicionar isso complexidade no nível de gerenciamento de um único agregado que poderia facilmente ser gerenciado por uma única função.

O caminho a seguir

Continuo convencido de que o futuro da maioria dos desenvolvedores é usar uma plataforma que esconde grande parte dos detalhes subjacentes deles. Por muitos anos, Heroku foi a coisa mais próxima que eu poderia apontar em termos de algo que encontrou o certo equilíbrio, mas agora temos o FaaS e o ecossistema mais amplo do turnkey ofertas sem servidor que traçam um caminho diferente.

Ainda há problemas a serem resolvidos com o FaaS, mas acho que, embora o a safra atual de ofertas ainda precisa mudar para resolver os problemas com elas, esse é o tipo de plataforma que a maioria dos desenvolvedores acabará usando. Nem todos os aplicativos se encaixarão perfeitamente em um ecossistema de FaaS, dadas as restrições, mas para aqueles que o fazem, as pessoas já estão vendo benefícios significativos. Com mais e mais trabalho investido em ofertas de FaaS apoiadas pelo Kubernetes, pessoas que

são incapazes de fazer uso direto das soluções FaaS fornecidas pelo principal...
os provedores de nuvem poderão cada vez mais aproveitar essa nova maneira
de trabalhar.

Portanto, embora os FAs possam não funcionar para tudo, certamente é algo que eu recomendo
pessoas para explorar. E para meus clientes que estão pensando em migrar para a nuvem-
soluções Kubernetes baseadas em Kubernetes, venho incentivando muitos deles a explorarem o FaaS
primeiro, pois pode dar a eles tudo de que precisam e, ao mesmo tempo, esconder coisas importantes
complexidade e descarregamento de muito trabalho.

Estou vendo mais organizações usando o FaaS como parte de uma
solução, escolhendo o FaaS para casos de uso específicos em que ele se encaixa bem. Um bom
exemplo seria a BBC, que faz uso das funções do Lambda como parte do
sua principal pilha de tecnologia que fornece o site da BBC News. O geral
o sistema usa uma combinação de instâncias Lambda e EC2 - com as instâncias EC2
frequentemente sendo usado em situações nas quais as invocações da função Lambda seriam
ser muito caro. 3

Qual opção de implantação é ideal para você?

Caramba. Então, temos muitas opções, certo? E eu provavelmente não ajudei também
muito me esforçando para compartilhar muitos prós e contras de cada um
abordagem. Se você chegou até aqui, você pode estar um pouco confuso sobre o que
você deveria fazer.

GORJETA

Bem, antes de prosseguir, eu realmente espero que nem seja preciso dizer que se você é
atualmente fazendo trabalhos para você, então continue fazendo isso! Não deixe a moda ditar sua técnica
decisões.

Se você acha que precisa mudar a forma como implanta microsserviços, deixe-me
tente destilar muito do que já discutimos e invente
algumas orientações úteis.

Revisitando nossos princípios de implantação de microservicos, um dos mais aspectos importantes em que nos concentrarmos foram o de garantir o isolamento de nossos microservicos. Mas apenas usar isso como um princípio orientador pode nos guiar rumo ao uso de máquinas físicas dedicadas para cada instância de microservico! Isso, é claro, provavelmente seria muito caro, e como já fizemos discutido, existem algumas ferramentas muito poderosas que não conseguiríamos usar se seguíssemos esse caminho.

As compensações são abundantes aqui. Equilibrando o custo com a facilidade de uso, isolamento, familiaridade, pode se tornar avassaladora. Então, vamos revisar um conjunto de regras I gostaria de chamar as regras básicas de Sam para descobrir onde

Implante coisas:

1. Se não estiver quebrado, não conserte-o.⁴

2. Desista de todo o controle com o qual você se sente feliz e depois doe só um pouco mais. Se você puder transferir todo o seu trabalho para um bom PaaS como o Heroku (ou uma plataforma FaaS), então faça isso e seja feliz. Você realmente precisa mexer em cada configuração?

3. Containerizar seus microserviços não é indolor, mas é realmente bom compromisso em relação ao custo de isolamento e tem alguns fantásticos benefícios para o desenvolvimento local, ao mesmo tempo em que oferece um grau de controle sobre o que acontece. Espere o Kubernetes em seu futuro.

Muitas pessoas estão proclamando "Kubernetes or bust!" que eu sinto que é inútil. Se você está na nuvem pública e seu problema se encaixa no FaaS como um modelo de implantação, faça isso em vez disso e ignore o Kubernetes. Seus desenvolvedores provavelmente acabará sendo muito mais produtivo. Como discutiremos mais em Capítulo 16, não deixe que o medo da prisão o mantenha preso em uma bagunça fabricação própria.

Encontrei um PaaS incrível, como Heroku ou Zeit, e tenho um aplicativo que se encaixa nas restrições da plataforma? Leve todo o trabalho para a plataforma e passe mais tempo trabalhando em seu produto. Tanto o Heroku quanto o Zeit são lindos plataformas fantásticas com incrível usabilidade do ponto de vista do desenvolvedor. Afinal, seus desenvolvedores não merecem ser felizes?

Para o resto de vocês, a conteinerização é o caminho a seguir, o que significa que precisamos para falar sobre o Kubernetes.

PAPEL DE FANTOCHE, CHEF E OUTRAS FERRAMENTAS?

Este capítulo mudou significativamente desde a primeira edição. Isso é devido em parte para a evolução da indústria como um todo, mas também para novas tecnologias que se tornou cada vez mais útil. O surgimento de novas tecnologias também levou a uma diminuição do papel de outras tecnologias - e assim vemos ferramentas como Puppet, Chef, Ansible e Salt desempenham um papel muito menor em implantando arquiteturas de microserviços do que fizemos em 2014.

A principal razão para isso é fundamentalmente a ascensão do contêiner. O poder de ferramentas como Puppet e Chef é que elas oferecem uma maneira de trazer uma máquina para um estado desejado, com esse estado desejado definido em algum código formulário. Você pode definir quais tempos de execução você precisa, onde arquivos de configuração precisam ser, etc., de uma forma que possa ser executada de forma determinística repetidamente na mesma máquina, garantindo que ela sempre possa ser levada ao mesmo estado.

A forma como a maioria das pessoas cria um contêiner é definindo um Dockerfile. Isso permite que você defina os mesmos requisitos que você faria com Puppet ou Chef, com algumas diferenças. Um recipiente é explodido quando reimplantado, para que cada criação de contêiner seja feita do zero (eu sou simplificando um pouco aqui). Isso significa que grande parte da complexidade inerente ao Puppet e ao Chef para lidar com essas ferramentas sendo executadas repetidamente nas mesmas máquinas não é necessário.

Puppet, Chef e ferramentas similares ainda são incrivelmente úteis, mas seu papel agora foi empurrado para fora do contêiner e mais para baixo na pilha. As pessoas usam ferramentas como essas para gerenciar aplicativos legados e infraestrutura ou para criar os clusters que armazenam cargas de trabalho, agora continue correndo. Mas é ainda menos provável que os desenvolvedores entrem em contato com essas ferramentas do que eram no passado.

O conceito de infraestrutura como código ainda é de vital importância. É só que o tipo de ferramentas que os desenvolvedores provavelmente usarão mudou. Para aqueles trabalhar com a nuvem, por exemplo, coisas como o Terraform podem ser muito útil para provisionar a infraestrutura em nuvem. Recentemente, eu me tornei um grande fã do Pulumi, que evita o uso de linguagens específicas de domínio

(DSLs) a favor do uso de linguagens de programação normais para ajudar os desenvolvedores gerenciam sua infraestrutura em nuvem. Eu vejo grandes coisas pela frente Pulumi, à medida que as equipes de entrega assumem cada vez mais a propriedade do mundo operacional, e eu suspeito que Puppet, Chef e similares, enquanto eles continuarão a desempenhar um papel útil nas operações, provavelmente mudarão cada vez mais longe das atividades de desenvolvimento do dia-a-dia.

Kubernetes e orquestração de contêineres

À medida que os contêineres começaram a ganhar força, muitas pessoas começaram a analisar soluções sobre como gerenciar contêineres em várias máquinas. Docker tive duas tentativas de fazer isso (com Docker Swarm e Docker Swarm Mode, respectivamente); empresas como Rancher e CoreOS criaram suas próprias leva; e plataformas de uso mais geral, como Mesos, foram usadas para executar contêineres junto com outros tipos de cargas de trabalho. Em última análise, porém, apesar de um muito esforço nesses produtos, o Kubernetes fez nos últimos dois anos venham a dominar esse espaço.

Antes de falarmos com o próprio Kubernetes, devemos discutir por que há uma necessidade para uma ferramenta como essa em primeiro lugar.

O caso da orquestração de contêineres

Em termos gerais, o Kubernetes pode ser descrito de várias maneiras como um contêiner, plataforma de orquestração ou, para usar um termo que caiu em desuso, um agendador de contêineres. Então, quais são essas plataformas e por que podemos querer eles?

Os contêineres são criados isolando um conjunto de recursos em um subjacente máquina. Ferramentas como o Docker nos permitem definir a aparência de um contêiner curtir e criar uma instância desse contêiner em uma máquina. Mas a maioria das soluções exigem que nosso software seja definido em várias máquinas, talvez para lidar com uma carga suficiente ou para garantir que o sistema tenha redundância para tolerar a falha de um único nó. As plataformas de orquestração de contêineres gerenciam como e onde as cargas de trabalho de contêineres são executadas. O termo "agendamento" começa com

fazem mais sentido nesse contexto. O operador diz: "Eu quero que essa coisa funcione" e o orquestrador descobre como agendar a procura de emprego disponível recursos, realocando-os, se necessário, e tratando dos detalhes do operador.

As várias plataformas de orquestração de contêineres também lidam com o estado desejado gerenciamento para nós, garantindo que o estado esperado de um conjunto de contêineres (instâncias de microserviços, no nosso caso) é mantida. Eles também nos permitem especifique como queremos que essas cargas de trabalho sejam distribuídas, o que nos permite otimize para utilização de recursos, latência entre processos ou robustez razões.

Sem essa ferramenta, você terá que gerenciar a distribuição de seus contêineres, algo que posso dizer por experiência própria envelhece muito rápido.

Escrever scripts para gerenciar instâncias de contêiner de inicialização e de rede não é diversão.

Em termos gerais, todas as plataformas de orquestração de contêineres, incluindo Kubernetes, forneça esses recursos de alguma forma. Se você olhar programadores de uso geral, como Mesos ou Nomad, soluções gerenciadas como o ECS da AWS, o Docker Swarm Mode e assim por diante, você verá um conjunto de recursos semelhante. Mas, por motivos que exploraremos em breve, o Kubernetes ganhou esse espaço. Também tem um ou dois conceitos interessantes que valem a pena explorar brevemente.

Uma visão simplificada dos conceitos do Kubernetes

Existem muitos outros conceitos no Kubernetes, então você vai me perdoar por não examinando todos eles (isso definitivamente justificaria um livro em si). O que eu vou tentar fazer aqui um esboço das principais ideias com as quais você precisará se envolver quando primeiro comece a trabalhar com a ferramenta. Vamos examinar primeiro o conceito de cluster, conforme mostrado na Figura 8-22.

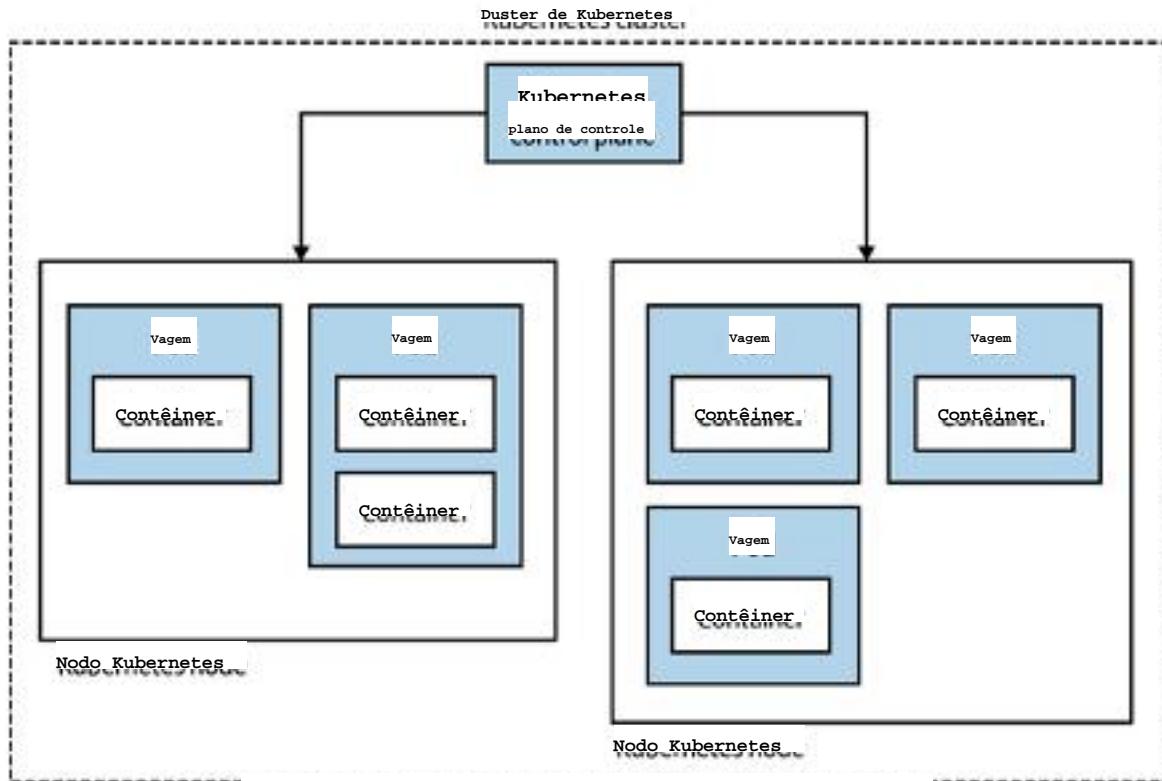


Figura 8-22. Uma visão geral simples da topologia do Kubernetes

Fundamentalmente, um cluster Kubernetes consiste em duas coisas. Primeiro, há um conjunto de máquinas nas quais as cargas de trabalho serão executadas, chamadas de nós. Em segundo lugar, há um conjunto de software de controle que gerencia esses nós e é referenciado para como plano de controle. Esses nós podem estar executando máquinas físicas ou máquinas virtuais sob o capô. Em vez de agendar um contêiner, Em vez disso, o Kubernetes agenda algo que chama de pod. Um pod consiste em um ou mais contêineres que serão implantados juntos.

Normalmente, você terá apenas um contêiner em uma cápsula - por exemplo, um instância do seu microsserviço. Há algumas ocasiões (raras, na minha experiência) em que ter vários contêineres implantados juntos pode fazer. No entanto, sinto. Um bom exemplo disso é o uso de proxies secundários, como Enviado, geralmente como parte de uma malha de serviços - um tópico que discutimos em "Serviço Malhas e gateways de API".

O próximo conceito que é útil conhecer é chamado de serviço. No no contexto do Kubernetes, você pode pensar em um serviço como um endpoint de roteamento estável - basicamente, uma forma de mapear dos pods que você está executando para um estabulo

interface de rede que está disponível no cluster. Alças do Kubernetes roteamento dentro do cluster, como vemos na Figura 8-23.

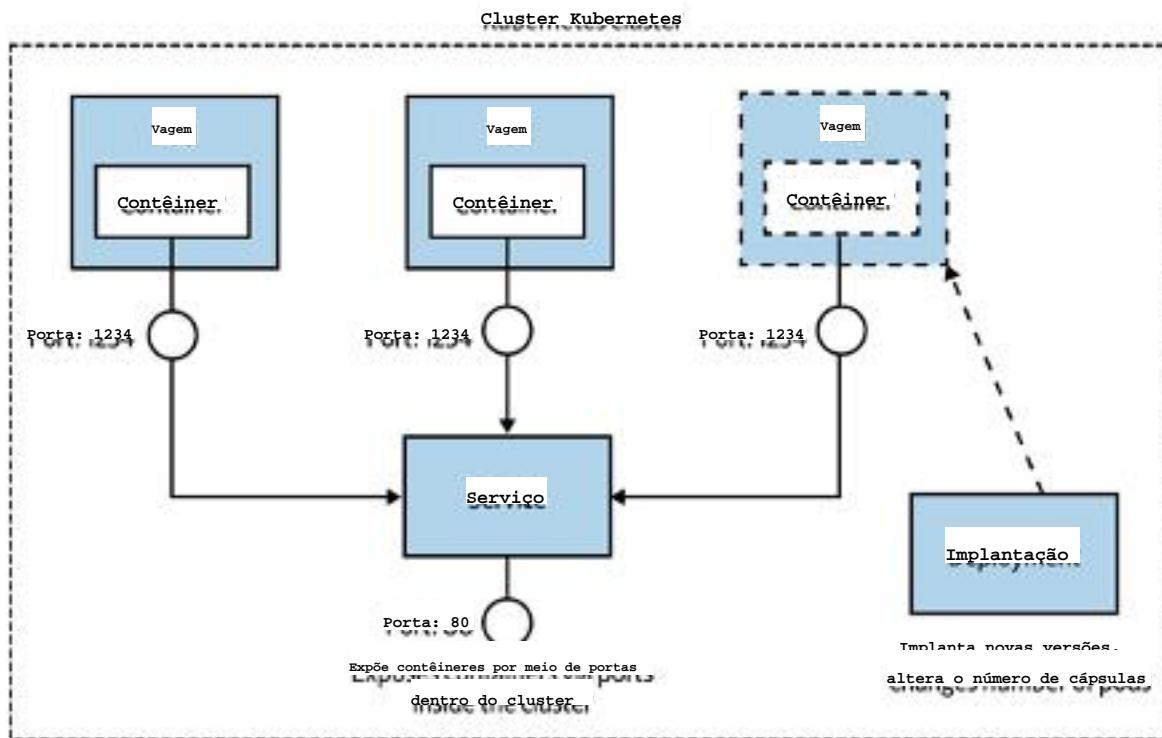


Figura 8-23. Como um pod, um serviço e uma implantação funcionam juntos

A ideia é que uma determinada cápsula possa ser considerada efêmera - ela pode ser fechada inativo por vários motivos, enquanto um serviço como um todo continua vivo. O serviço existe para rotear chamadas de e para os pods e pode lidar com pods sendo desligados ou novos pods sendo lançados. Puramente a partir de uma terminologia ponto de vista, isso pode ser confuso. Falamos de forma mais geral sobre a implantação um serviço, mas no Kubernetes você não implanta um serviço - você implanta pods esse mapa para um serviço. Pode demorar um pouco para você entender isso.

Em seguida, temos um conjunto de réplicas. Com um conjunto de réplicas, você define o estado desejado de um conjunto de cápsulas. É aqui que você diz: "Eu quero quatro dessas cápsulas", e o Kubernetes cuida do resto. Na prática, não se espera mais que você trabalhe com conjuntos de réplicas diretamente; em vez disso, eles são gerenciados para você por meio de uma implantação, o último conceito que veremos. Uma implantação é como você aplica as alterações em seus pods e conjuntos de réplicas. Com uma implantação, você pode fazer coisas como problemas atualizações contínuas (para que você substitua os pods por uma versão mais recente de forma gradual).

moda para evitar tempo de inatividade), reversões, aumento do número de nós, e mais.

Então, para implantar seu microsserviço, você define um pod, que conterá sua instância de microsserviço dentro dela; você define um serviço, que permitirá o Kubernetes saber como seu microsserviço será acessado; e você se inscreve alterações nos pods em execução usando uma implantação. Parece fácil quando eu digo isso, não é? Digamos que eu deixei de fora algumas coisas aqui por uma questão de brevidade.

Multilocação e federação

Do ponto de vista da eficiência, você gostaria de agrupar toda a computação, recursos disponíveis para você em um único cluster Kubernetes e tenha todos as cargas de trabalho são executadas lá de toda a organização. Isso provavelmente seria proporcionam uma maior utilização dos recursos subjacentes, como recursos não utilizados podem ser realocados livremente para quem precisar deles. Isso, por sua vez, deveria reduzir os custos adequadamente.

O desafio é que, embora o Kubernetes seja capaz de gerenciar diferentes microsserviços para diferentes propósitos, tem limitações sobre como "multilocatária" é a plataforma. Diferentes departamentos em sua organização pode querer diferentes graus de controle sobre vários recursos. Esses tipos de controles não foram incorporados ao Kubernetes, uma decisão que parece sensata em termos de tentar manter o escopo do Kubernetes um pouco limitado. Para trabalhar em torno desse problema, as organizações parecem explorar algumas coisas diferentes caminhos.

A primeira opção é adotar uma plataforma construída sobre o Kubernetes que fornece esses recursos - o OpenShift da Red Hat, por exemplo, tem um rico conjunto de controles de acesso e outros recursos que são criados com maiores organizações em mente e pode transformar o conceito de multilocação em algo mais fácil. Além de qualquer implicação financeira do uso desses tipos de plataformas, para que eles funcionem, às vezes você terá que trabalhar com as abstrações fornecidas para você pelo fornecedor que você escolheu, o que significa que seus desenvolvedores não precisam saber

apenas como usar o Kubernetes, mas também como usar o desse fornecedor específico na sua plataforma.

Outra abordagem é considerar um modelo federado, descrito na Figura 8-24.

Com a federação, você tem vários clusters separados, com alguma camada de software que fica na parte superior, permitindo que você faça alterações em todos os clusters se necessário. Em muitos casos, as pessoas trabalhavam diretamente em um cluster, oferecendo a eles uma experiência bastante familiar com o Kubernetes, mas em algumas situações, talvez você queira distribuir um aplicativo em vários clusters, talvez se esses clusters estavam em diferentes geografias e você queria sua aplicação implantado com alguma capacidade de lidar com a perda de um cluster inteiro.

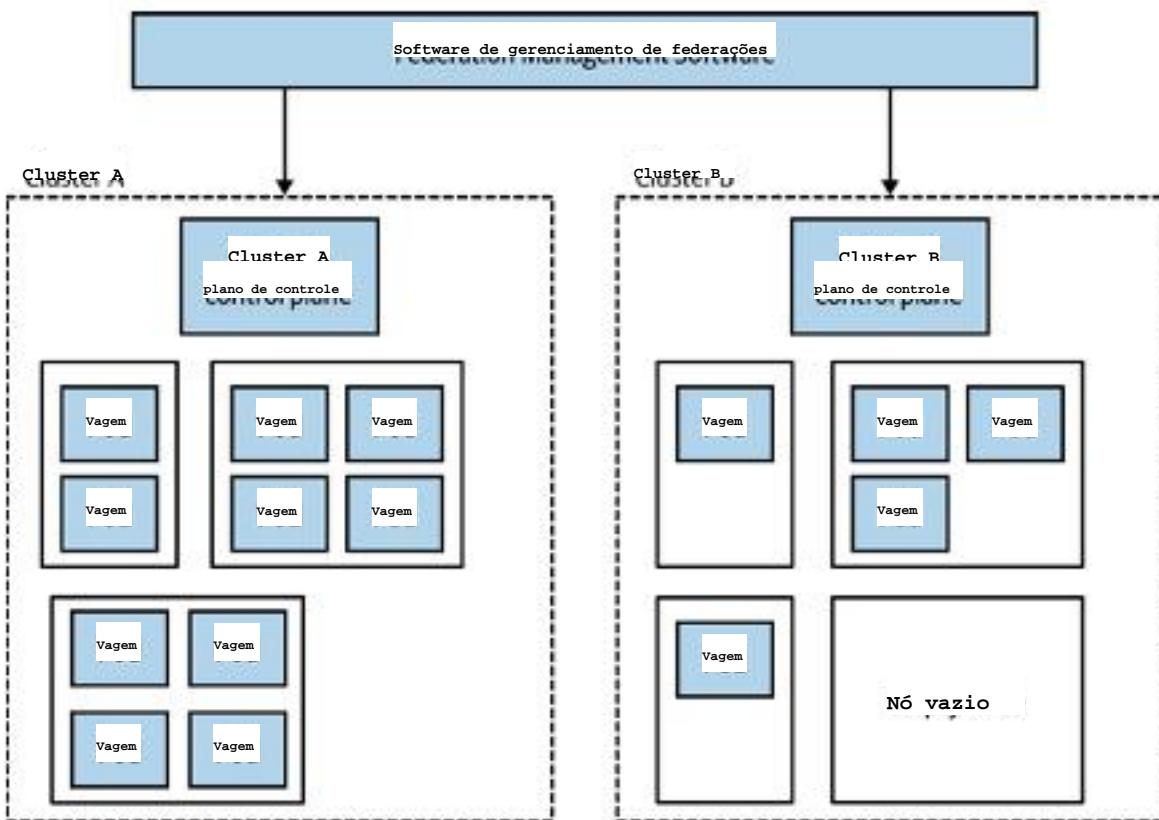


Figura 8-24. Um exemplo de federação no Kubernetes.

A natureza federada torna o agrupamento de recursos mais desafiador. Como vemos em Figura 8-24, o Cluster A é totalmente utilizado, enquanto o Cluster B tem muitos não utilizados. Se quiséssemos executar mais cargas de trabalho no Cluster A, isso seria só possível se pudéssemos fornecer mais recursos, como mover o nó vazio no Cluster B para o Cluster A. Quão fácil será mover o

o nó de um cluster para o outro dependerá da natureza da federação software sendo usado, mas posso imaginar que isso seja uma mudança nada trivial. Lembre-se de que um único nó pode fazer parte de um cluster ou de outro e, portanto, não pode executar pods para o Cluster A e o Cluster B.

É importante notar que ter vários clusters pode ser benéfico quando considere os desafios de atualizar o próprio cluster. Pode ser mais fácil e é mais seguro mover um microserviço para um cluster recém-atualizado do que atualize o cluster no local.

Fundamentalmente, esses são desafios de escala. Para algumas organizações, você nunca tenha esses problemas, pois você está feliz em compartilhar um único cluster. Para outros organizações que buscam obter eficiências em maior escala, essa é certamente uma área que você desejará explorar com mais detalhes. Deve-se notar que há várias visões diferentes de como a federação do Kubernetes deve ser como, e várias cadeias de ferramentas diferentes para gerenciá-las.

O PLANO DE FUNDO POR TRÁS DO KUBERNETES

O Kubernetes começou como um projeto de código aberto no Google, que desenhou seu inspiração dos sistemas anteriores de gerenciamento de contêineres Omega e Borg. Muitos dos principais conceitos do Kubernetes são baseados em conceitos sobre como as cargas de trabalho de contêineres são gerenciadas no Google, embora com um alvo ligeiramente diferente em mente. A Borg opera sistemas em uma grande escala global escala, lidando com dezenas, senão centenas de milhares de contêineres em todo o mundo data centers em todo o mundo. Se você quiser mais detalhes sobre como os diferentes as mentalidades por trás dessas três plataformas do Google se comparam, embora a partir de um Do ponto de vista centrado no Google, eu recomendo "Borg, Omega e Kubernetes" de Brendan Burns et al. como uma boa visão geral.

Enquanto o Kubernetes compartilha algum DNA com Borg e Omega, trabalhando na a grande escala não tem sido o principal fator por trás do projeto. Ambos Nomad e Mesos (cada um deles foi inspirado por Borg) encontraram um nicho em situações em que grupos de milhares de máquinas estão necessário, conforme demonstrado no uso do Mesos pela Apple para Siri ou Roblox uso do Nomad.

O Kubernetes queria pegar ideias do Google, mas fornecer mais experiência mais amigável para desenvolvedores do que a oferecida pela Borg ou Omega. É possível analisar a decisão do Google de investir muito esforço de engenharia na criação de uma ferramenta de código aberto sob uma luz puramente altruísta, e enquanto eu estou claro que essa era a intenção de algumas pessoas, a realidade é que isso é como muito sobre o risco que o Google estava vendo com a concorrência pública espaço em nuvem, especificamente AWS.

No mercado de nuvem pública, o Google Cloud ganhou espaço, mas ainda é um distante terceiro, atrás do Azure e da AWS (que estão na frente), e alguns A análise mostra que está em quarto lugar pela Alibaba Cloud. Apesar a melhoria da participação de mercado, ainda não está nem perto de onde o Google quer será.

Parece provável que uma grande preocupação fosse que o líder claro do mercado, A AWS poderia eventualmente ter um quase monopólio na computação em nuvem

espaço. Além disso, preocupações com relação ao custo da migração de um fornecedor para outro significava que tal posição de domínio de mercado seria difícil mudar. E então vem o Kubernetes, com seu promessa de ser capaz de fornecer uma plataforma padrão para execução cargas de trabalho de contêineres que podem ser executadas por vários fornecedores. A esperança era que isso permitiria a migração de um provedor para outro e evite um futuro exclusivo da AWS.

Assim, você pode ver o Kubernetes como uma contribuição generosa do Google para o setor de TI mais amplo, ou como uma tentativa do Google de permanecer relevante no espaço de nuvem pública em rápida evolução. Não tenho nenhum problema em ver as duas coisas como igualmente verdadeiro.

A Federação de Computação Nativa em Nuvem

A Cloud Native Computing Foundation (CNCF para abreviar) é uma ramificação da a Linux Foundation, sem fins lucrativos. O CNCF se concentra na curadoria do ecossistema de projetos para ajudar a promover o desenvolvimento nativo em nuvem, embora na prática isso significa apoiar Kubernetes e projetos que funcionam com ou se baseiam O próprio Kubernetes. Os projetos em si não são criados ou diretamente desenvolvido pelo CNCF: em vez disso, você pode ver o CNCF como um lugar onde esses projetos que, de outra forma, poderiam ser desenvolvidos isoladamente podem ser hospedados juntos no mesmo local e onde existem padrões comuns e interoperabilidade pode ser desenvolvido.

Dessa forma, o CNCF me lembra o papel do Software Apache Fundamento-AS com o CNCF, um projeto que faz parte do Software Apache. A fundação normalmente implica um nível de qualidade e um apoio comunitário mais amplo. Todos os projetos hospedados pelo CNCF são de código aberto, embora o o desenvolvimento desses projetos pode muito bem ser conduzido por entidades comerciais.

Além de ajudar a orientar o desenvolvimento desses projetos associados, o CNCF também realiza eventos, fornece documentação e materiais de treinamento e define os vários programas de certificação em torno do Kubernetes. O grupo em si tem membros de todo o setor e, embora possa ser difícil

para que grupos menores ou independentes desempenhem um papel importante na organização

em si, o grau de suporte intersetorial (incluindo muitas empresas que são concorrentes entre si) é impressionante.

Como forasteiro, o CNCF parece ter tido grande sucesso em ajudar a se espalhar a palavra sobre a utilidade dos projetos que organiza. Também atuou como um lugar onde a evolução de grandes projetos pode ser discutida abertamente, garantindo uma grande quantidade de informações amplas. O CNCF desempenhou um papel importante na sucesso do Kubernetes - é fácil imaginar que, sem ele, ainda teríamos um paisagem fragmentada nesta área.

Plataformas e portabilidade

Muitas vezes, você ouvirá o Kubernetes ser descrito como uma "plataforma". Não é realmente uma plataforma no sentido de que um desenvolvedor entenderia o termo, no entanto. Fora pronto para uso, tudo o que ele realmente oferece é a capacidade de executar cargas de trabalho de contêineres. A maioria das pessoas que usam o Kubernetes acaba montando sua própria plataforma por instalação de software de suporte, como service meshes, corretores de mensagens, log ferramentas de agregação e muito mais. Em organizações maiores, isso acaba sendo o responsabilidade de uma equipe de engenharia de plataforma, que montou essa plataforma e gerencie-a e ajude os desenvolvedores a usar a plataforma de forma eficaz.

Isso pode ser tanto uma bênção quanto uma maldição. Essa abordagem de escolher e misturar é feita possível devido a um ecossistema de ferramentas bastante compatível (graças em grande parte a o trabalho do CNCF). Isso significa que você pode selecionar suas ferramentas favoritas para tarefas específicas, se você quiser. Mas isso também pode levar à tirania da escolha - nós pode facilmente ficar sobrecarregado com tantas opções. Produtos como Red Hat's OpenShift retira parcialmente essa escolha de nós, pois nos oferece uma solução pronta para... fez uma plataforma com algumas decisões já tomadas para nós.

O que isso significa é que, embora no nível básico, o Kubernetes ofereça uma abstração portátil para execução de contêineres; na prática, não é tão simples quanto pegar um aplicativo que funciona em um cluster e esperar que ele funcione em outro lugar. Seu aplicativo, suas operações e o fluxo de trabalho do desenvolvedor podem confiaremos em sua própria plataforma personalizada. Movendo-se de um cluster Kubernetes para outro, também pode exigir que você reconstrua essa plataforma em sua nova destino. Conversei com muitas organizações que adotaram o Kubernetes

principalmente porque estão preocupados em ficar presos a um único fornecedor, mas essas organizações não entenderam essas nuances - aplicativos baseados em Teoricamente, os Kubernetes são portáteis entre clusters do Kubernetes, mas nem sempre na prática.

Helm, operadores e CRDs, meu Deus!

Uma área de confusão continua no espaço do Kubernetes é como gerenciar a implantação e o ciclo de vida de aplicativos de terceiros e subsistemas. Considere a necessidade de executar o Kafka em seu cluster Kubernetes. Você poderia criar suas próprias especificações de pod, serviço e implantação e executar eles você mesmo. Mas e quanto ao gerenciamento de um upgrade para sua configuração do Kafka? E quanto a outras tarefas comuns de manutenção com as quais você talvez queira lidar, gosta de atualizar um software com estado em execução?

Surgiram várias ferramentas que visam dar a você a capacidade de gerenciar esses tipos de aplicações em um nível de abstração mais sensato. A ideia é que alguém crie algo semelhante a um pacote para o Kafka e você o execute em seu cluster Kubernetes de uma forma mais caixa-preta. Dois dos melhores... soluções conhecidas neste espaço são Operator e Helm. Helm se autodenomina como "o gerenciador de pacotes que faltava" para o Kubernetes, e enquanto o Operator pode gerencie a instalação inicial, parece estar mais focado na instalação em andamento gerenciamento do aplicativo. Surpreendentemente, embora você possa ver o Operador e Como alternativas um ao outro, você também pode usar os dois juntos em algumas situações (Helm para instalação inicial, Operator para ciclo de vida operações).

Uma evolução mais recente nesse espaço é algo chamado recurso personalizado definições, ou CRDs. Com os CRDs, você pode estender as principais APIs do Kubernetes, permitindo que você conecte um novo comportamento ao seu cluster. A coisa boa sobre CRDs é que eles se integram perfeitamente à linha de comando existente interface, controles de acesso e muito mais, para que sua extensão personalizada não pareça como uma adição alienígena. Eles basicamente permitem que você implemente o seu próprio Abstrações do Kubernetes. Pense no pod, no conjunto de réplicas, no serviço e abstrações de implantação que discutimos anteriormente - com CRDs, você pode adicionar você mesmo na mistura.

Você pode usar CRDs para tudo, desde o gerenciamento de pequenos pedaços de configuração para controlar malhas de serviços, como o Istio, ou softwares baseados em cluster, como o Kafka.

Com um conceito tão flexível e poderoso, acho difícil entender onde os CRDs seriam melhor usados e não parece haver um consenso geral entre os especialistas com quem conversei. Esse todo espaço ainda não parece estar se estabilizando tão rápido quanto eu esperava, e não há tanto consenso quanto eu gostaria - uma tendência no Kubernetes ecosistema.

E Knative

O Knative é um projeto de código aberto que visa fornecer fluxos de trabalho no estilo FaaS para desenvolvedores, usando o Kubernetes nos bastidores. Fundamentalmente, o Kubernetes não é muito fácil de desenvolver, especialmente se o compararmos com a usabilidade de coisas como Heroku ou plataformas similares. O objetivo com o Knative é trazer a experiência do desenvolvedor de FaaS para Kubernetes, ocultando a complexidade do Kubernetes de desenvolvedores. Por sua vez, isso deve significar equipes de desenvolvimento são capazes de gerenciar com mais facilidade o ciclo de vida completo de seu software.

Já discutimos as malhas de serviços e mencionamos especificamente o Istio, de volta ao Capítulo 5. Uma malha de serviços é essencial para a execução do Knative. Enquanto teoricamente, o Knative permite que você conecte diferentes malhas de serviço, apenas o Istio é considerado estável no momento (com suporte para outras malhas como Ambassador e Gloo ainda em alfa). Na prática, isso significa que se você quiser para adotar o Knative, você também já precisará ter comprado o Istio.

Com o Kubernetes e o Istio, projetos conduzidos em grande parte pelo Google, foi necessário um muito tempo para chegarem a um estágio em que possam ser considerados estáveis. O Kubernetes ainda teve grandes mudanças após seu lançamento 1.0, e apenas muito recentemente, o Istio, que vai sustentar o Knative, foi completamente rearquitetado. Esse histórico de entrega estável e pronta para produção. projetos me fazem pensar que o Knative pode muito bem levar muito mais tempo para ficar pronto para uso pela maioria de nós. Embora algumas organizações o estejam usando, e você poderia provavelmente também o use, a experiência diz que demorará muito até que alguns ocorrerá uma grande mudança que exigirá uma migração dolorosa. É parcialmente para esta é a razão pela qual eu sugeri que organizações mais conservadoras que são

considerando uma oferta semelhante ao FaaS para sua aparência de cluster Kubernetes
projetos em outros lugares como o OpenFaaS já estão sendo usados na produção por
organizações em todo o mundo e não precisam de um serviço subjacente
malha. Mas se você embarcar no trem Knative agora, não se surpreenda se
você tem um estranho descarrilamento em seu futuro.

Outra observação: foi uma pena ver que o Google decidiu não fazer
Knative, como parte do CNCF-One, só podemos supor que isso ocorre porque o Google
quer direcionar a direção da própria ferramenta. O Kubernetes era confuso
perspectiva para muitos quando lançada, em parte porque refletia a do Google
mentalidade sobre como os contêineres devem ser gerenciados. Ele se beneficiou enormemente de
envolvimento de um conjunto mais amplo da indústria, e é uma pena que, neste momento
pelo menos no estágio, o Google decidiu que não está interessado na mesma área
envolvimento da indústria para a Knative.

O futuro

No futuro, não vejo sinais de que o gigante do Kubernetes
será interrompido em breve, e espero ver mais organizações
implementando seus próprios clusters Kubernetes para nuvens privadas ou fazendo uso
de clusters gerenciados em configurações de nuvem pública. No entanto, acho que o que somos
vendo agora, com os desenvolvedores tendo que aprender a usar o Kubernetes diretamente,
será um pontinho de vida relativamente curta. O Kubernetes é ótimo para gerenciar
cargas de trabalho de contêineres e fornecimento de uma plataforma para que outras coisas possam ser construídas.

No entanto, não é o que poderia ser considerado uma experiência amigável para desenvolvedores.
O próprio Google nos mostrou isso, com o impulso por trás do Knative, e eu acho
continuaremos vendo o Kubernetes escondido sob abstracão de alto nível
camadas. Então, no futuro, espero que o Kubernetes esteja em todo lugar. Você simplesmente não vai
saiá disso.

Isso não quer dizer que os desenvolvedores possam esquecer que estão criando um sistema distribuído
sistema. Eles ainda precisarão entender os inúmeros desafios que esse tipo de
a arquitetura traz. Só que eles não precisarão se preocupar tanto com o
detalhes de como seu software é mapeado para os recursos de computação subjacentes.

Você deve usá-lo?

Então, para aqueles que ainda não são membros totalmente remunerados do Kubernetes club, você deveria entrar? Bem, deixe-me compartilhar algumas diretrizes.

Em primeiro lugar, implementar e gerenciar seu próprio cluster Kubernetes não é para os fracos de coração - é um empreendimento significativo. Muito da qualidade de a experiência que seus desenvolvedores terão ao usar a instalação do Kubernetes será dependem da eficácia da equipe que administra o cluster. Por esse motivo, um número das maiores organizações com quem falei que passaram pelo O Kubernetes on-prem path terceirizou esse trabalho para empresas especializadas empresas.

Melhor ainda, use um cluster totalmente gerenciado. Se você puder fazer uso do público na nuvem e, em seguida, use soluções totalmente gerenciadas, como as fornecidas pelo Google, Azure e AWS. O que eu diria, porém, é que se você for capaz de usar o nuvem pública e, em seguida, considere se o Kubernetes é realmente o que você deseja. Se você está procurando uma plataforma amigável para desenvolvedores para lidar com a implantação e ciclo de vida de seus microserviços e, em seguida, as plataformas FaaS que já temos visto pode ser uma ótima opção. Você também pode ver outros semelhantes ao PaaS ofertas, como Azure Web Apps, Google App Engine ou algumas das menores fornecedores como Zeit ou Heroku.

Antes de decidir começar a usar o Kubernetes, chame alguns de seus administradores e desenvolvedores que o usam. Os desenvolvedores podem começar a executar algo leves localmente, como minikube ou microK8s, dando a eles algo quase uma experiência completa do Kubernetes, mas em seus laptops. As pessoas você verá que o gerenciamento da plataforma pode precisar de um mergulho mais profundo. Katacoda tem alguns ótimos tutoriais on-line para se familiarizar com os principais conceitos e o CNCF ajuda a publicar muitos materiais de treinamento nesse espaço. Certifique-se as pessoas que realmente usarão essas coisas podem brincar com elas antes de você fazer em sua mente.

Não fique preso pensando que você precisa ter o Kubernetes "porque todo mundo está fazendo isso." Essa é uma justificativa igualmente perigosa para escolher Kubernetes como ele é para escolher microserviços. Por melhor que seja o Kubernetes, não é para todos - faça sua própria avaliação. Mas sejamos francos...

você tem um punhado de desenvolvedores e apenas alguns microsserviços,

É provável que o Kubernetes seja um grande exagero, mesmo usando um sistema totalmente gerenciado plataforma.

Entrega progressiva

Ao longo da última década, nos tornamos mais inteligentes na implantação de software para nossos usuários. Surgiram novas técnicas que foram impulsionadas por uma série de casos de uso diferentes e vieram de várias partes diferentes do setor de TI, mas principalmente, eles estavam todos focados em fazer o ato de lançar novas software muito menos arriscado. E se o lançamento de software se tornar menos arriscado, nós pode lançar software com mais frequência.

Há uma série de atividades que realizamos antes de enviar nosso software ao vivo, que podem nos ajudar a resolver problemas antes que eles afetem usuários reais.

Os testes de pré-produção são uma grande parte disso, no entanto, como discutiremos em Capítulo 9, isso só pode nos levar até certo ponto.

Em seu livro *Accelerate*,⁶ Nicole Forsgren, Jez Humble e Gene Kim mostram evidências claras extraídas de uma extensa pesquisa de que alto desempenho as empresas implantam com mais frequência do que suas contrapartes de baixo desempenho e, ao mesmo tempo, têm taxas de falha de mudança muito mais baixas.

A ideia de que você "vai rápido e quebra coisas" realmente não parece se aplicar quando trata-se de enviar software com frequência e ter menos falhas as taxas andam de mãos dadas, e as organizações que perceberam isso têm mudou a forma como eles pensam sobre o lançamento de software.

Essas organizações fazem uso de técnicas como alternância de recursos, canário lançamentos, execuções paralelas e muito mais, que detalharemos nesta seção. Isso A mudança na forma como pensamos sobre o lançamento de funcionalidades se enquadra na o que é chamado de entrega progressiva. A funcionalidade é liberada para os usuários em um de forma controlada; em vez de uma grande implantação, podemos ser inteligentes sobre quem vê qual funcionalidade, por exemplo, lançando uma nova versão do nosso software para um subconjunto de nossos usuários.

Fundamentalmente, o que todas essas técnicas têm em seu cerne é uma simples mudança na forma como pensamos sobre o software de envio. Ou seja, que podemos separar o conceito de implantação a partir do de lançamento.

Separando a implantação da liberação

Jez Humble, coautor de Continuous Delivery, defende a separação

essas duas ideias, e ele faz disso um princípio fundamental para software de baixo risco lançamentos:

A implantação é o que acontece quando você instala alguma versão do seu software em um ambiente específico (o ambiente de produção é frequentemente implícito). A liberação é quando você cria um sistema ou parte dele (para exemplo, um recurso) disponível para os usuários.

Jez argumenta que, ao separar essas duas ideias, podemos garantir que nosso software funciona em sua configuração de produção sem que falhas sejam vistas por nossos usuários. A implantação azul-verde é um dos exemplos mais simples desse conceito em ação - você tem uma versão do seu software ativa (azul) e, em seguida, você implante uma nova versão junto com a versão antiga em produção (verde). Você verifique se a nova versão está funcionando conforme o esperado e, se estiver, você redireciona os clientes para ver a nova versão do seu software. Se você encontrar um problema antes da transição, nenhum cliente é afetado.

Embora as implantações azul-esverdeadas estejam entre os exemplos mais simples disso, princípio, existem várias técnicas mais sofisticadas que podemos usar quando adotamos esse conceito.

Para a entrega progressiva

James Governor, cofundador da empresa de análise industrial focada em desenvolvedores RedMonk, primeiro cunhou o termo entrega progressiva para abranger uma série de

diferentes técnicas sendo usadas neste espaço. Ele passou a descrever entrega progressiva como "entrega contínua" com controle refinado sobre o raio de explosão" - então é uma extensão da entrega contínua, mas também uma

técnica que nos dá a capacidade de controlar o impacto potencial de nossos recém-nascidos software lançado.

Pegando esse tema, Adam Zimman do LaunchDarkly descreve como a entrega progressiva afeta "o negócio". Desse ponto de vista, nós exigem uma mudança no pensamento sobre como as novas funcionalidades chegam aos nossos clientes. Não é mais um lançamento único - agora pode ser uma atividade em fases. É importante ressaltar que no entanto, a entrega progressiva pode capacitar o proprietário do produto, como Adam coloca, "delegando o controle do recurso ao proprietário que está mais próximo responsável pelo resultado." Para que isso funcione, no entanto, o proprietário do produto em questão precisa entender a mecânica da entrega progressiva técnica que está sendo usada, implicando um proprietário de produto um tanto tecnicamente experiente, ou então o apoio de um grupo de pessoas com experiência adequada.

Já abordamos as implantações azul-esverdeadas como uma progressiva técnica de entrega. Vamos dar uma olhada breve em mais alguns.

Alternâncias de recursos

Com alternâncias de recursos (também conhecidas como sinalizadores de recursos), nos escondemos implantados funcionalidade por trás de um botão que pode ser usado para desligar a funcionalidade ou ligado. Isso é mais comumente usado como parte do desenvolvimento baseado em troncos, onde uma funcionalidade que ainda não foi concluída pode ser verificada e implantada, mas ainda escondido dos usuários finais, mas tem muitos aplicativos fora disso. Isso pode ser útil para ativar um recurso em um horário especificado ou desativar um recurso isso está causando problemas.

Você também pode usar as alternâncias de recursos de uma maneira mais refinada, talvez permitindo que um sinalizador tenha um estado diferente com base na natureza do usuário fazendo uma solicitação. Então, você poderia, por exemplo, ter um grupo de clientes que vê um recurso ativado (talvez um grupo de teste beta), enquanto a maioria das pessoas vê o recurso, como desativado, pode ajudá-lo a implementar um canário lançamento, algo que discutiremos a seguir. Existem soluções totalmente gerenciadas para gerenciando alternâncias de recursos, incluindo LaunchDarkly e Split. Impressionante como essas plataformas são, acho que você pode começar com algo muito mais simples.

— basta um arquivo de configuração para começar, então veja essas tecnologias quando você começa a pressionar como deseja usar os botões.

Para um mergulho muito mais profundo no mundo das alternâncias de recursos, posso sinceramente recomendar o artigo de Pete Hodgson, "Feature Toggles (aka Feature Flags)", que detalha muitos detalhes sobre como implementá-los e os muitos diferentes maneiras pelas quais eles podem ser usados.

Canary Release

Errar é humano, mas para realmente estragar as coisas, você precisa de um computador. Todos cometemos erros, e os computadores podem nos permitir cometer erros mais rapidamente e com rapidez maior escala do que nunca. Dado que os erros são inevitáveis (e confiem eu, eles são), então faz sentido fazer coisas que nos permitam limitar o impacto desses erros. As liberações canárias são uma dessas técnicas.

Nomeado em homenagem aos canários levados para as minas como um sistema de alerta precoce para mineiros para avisá-los da presença de gases perigosos, com um canário lançamento: a ideia é que um subconjunto limitado de nossos clientes veja novos recursos ou funcionalidades. Se houver um problema com a implantação, somente essa parte dos nossos clientes são impactados. Se o recurso funcionar para esse grupo de canários, então pode ser distribuído para mais clientes até que todos vejam a nova versão.

Para uma arquitetura de microserviços, uma alternância pode ser configurada em um nível de microserviço individual, ativando (ou desativando) a funcionalidade para solicitações de essa funcionalidade do mundo exterior ou de outros microserviços. Outro A técnica é ter duas versões diferentes de um microserviço rodando lado a lado lateral e use o botão para rotear para a versão antiga ou a nova. Aqui, a implementação canária deve estar em algum lugar no roteamento/rede caminho, em vez de estar em um microserviço.

Quando fiz um lançamento canário pela primeira vez, controlamos o lançamento manualmente. Nós poderíamos configurar a porcentagem do nosso tráfego vendo a nova funcionalidade, e durante um período de uma semana, aumentamos gradualmente isso até que todos vissem a nova funcionalidade. Durante a semana, ficamos de olho em nossas taxas de erro, bug

relatórios e afins. Hoje em dia, é mais comum ver esse processo manuseado de forma automatizada. Ferramentas como o Spinnaker, por exemplo, têm a capacidade de aumentar automaticamente as chamadas com base em métricas, como aumentar a porcentagem de chamadas para uma nova versão de microsserviço se as taxas de erro estiverem em um nível aceitável.

Execução paralela

Com uma versão canária, uma solicitação de uma funcionalidade será atendida por a versão antiga ou a nova. Isso significa que não podemos comparar como os dois versões da funcionalidade lidariam com a mesma solicitação, algo que pode seja importante se você quiser ter certeza de que a nova funcionalidade funciona exatamente da mesma forma que a versão antiga da funcionalidade.

Com uma corrida paralela, você faz exatamente isso - você corre duas diferentes implementações da mesma funcionalidade lado a lado e envie uma solicitação para a funcionalidade de ambas as implementações. Com uma arquitetura de microsserviços, a abordagem mais óbvia pode ser enviar uma chamada de serviço para dois versões diferentes do mesmo serviço e compare os resultados. Uma alternativa é coexistir as duas implementações da funcionalidade dentro da mesma serviço, que muitas vezes pode facilitar a comparação.

Ao executar as duas implementações, é importante perceber que você provavelmente só quer os resultados de uma das invocações. Uma implementação é considerada a fonte da verdade - esta é a implementação em que você confia atualmente, e normalmente é a implementação existente. Dependendo da natureza do funcionalidade que você está comparando com uma execução paralela, talvez seja necessário fornecer esta nuance: pensamento cuidadoso: você não gostaria de enviar dois pedidos idênticos atualiza para um cliente ou paga uma fatura duas vezes, por exemplo!

Eu exploro o padrão de execução paralela com muito mais detalhes no Capítulo 3 do meu livro. Monolith a Microservices. Lá, exploro seu uso para ajudar a migrar a funcionalidade de um sistema monolítico a uma arquitetura de microsserviços, onde queremos garantir que nosso novo microsserviço se comporte da mesma forma que o funcionalidade monolítica equivalente. Em outro contexto, o GitHub faz uso de esse padrão ao retrabalhar partes principais de sua base de código e lançar um

ferramenta de código aberto Scientist para ajudar nesse processo. Aqui, a corrida paralela é feito em um único processo, com o Scientist ajudando a comparar o invocações.

GORJETA

Com implantação azul-esverdeada, alternância de recursos, lançamentos rápidos e execuções paralelas, acabamos de arranhau a superfície do campo de entrega progressiva. Essas ideias podem funcionar bem juntos (já abordamos como você pode usar alternâncias de recursos para implementar um canary rollout, por exemplo), mas você provavelmente quer se candidatar Para começar. Apenas lembre-se de separar os dois conceitos de implantação e lançamento. Em seguida, comece a procurar maneiras de ajudar você a implantar seu software com mais frequência, mas de forma segura. Trabalhe com seu proprietário do produto ou outras partes interessadas da empresa para entender como algumas delas as técnicas podem ajudar você a ir mais rápido, mas também ajudam a reduzir falhas.

Resumo

OK, então cobrimos muito terreno aqui. Vamos recapitular brevemente antes de avançarmos ligado. Em primeiro lugar, vamos nos lembrar dos princípios de implantação que eu descrito anteriormente:

Execução isolada

Execute instâncias de microserviços de forma isolada, onde elas tenham suas recursos de computação próprios e sua execução não pode impactar outros instâncias de microserviços em execução nas proximidades.

Foco na automação

Escolha uma tecnologia que permita um alto grau de automação e adote automação como parte essencial de sua cultura.

Infraestrutura como código

Represente a configuração de sua infraestrutura para facilitar a automação e promover o compartilhamento de informações. Armazene esse código no controle de origem para permitir para que os ambientes sejam recriados.

Busque a implantação sem tempo de inatividade

Aprofunde a implantação independente e garanta a implantação de uma nova versão de um microsserviço pode ser feita sem nenhum tempo de inatividade para os usuários do seu serviço (sejam humanos ou outros microsserviços).

Gestão do estado desejado

Use uma plataforma que mantenha seu microsserviço em um estado definido lancando novas instâncias, se necessário, em caso de interrupção ou tráfego aumenta.

Além disso, compartilhei minhas próprias diretrizes para selecionar a implantação correta plataforma:

1. Se não estiver quebrado, não conserte. 9
2. Desista de todo o controle com o qual você se sente feliz e, em seguida, doe apenas um pouco mais. Se você puder transferir todo o seu trabalho para um bom PaaS como o Heroku (ou plataforma FaaS), então faça isso e seja feliz. Você realmente precisa mexer em cada configuração?
3. A containerização de seus microsserviços não é fácil, mas é realmente uma bom compromisso em relação ao custo de isolamento e tem alguns fantásticos benefícios para o desenvolvimento local, ao mesmo tempo em que oferece um grau de controle sobre o que acontece. Espere o Kubernetes em seu futuro.

Também é importante entender seus requisitos. O Kubernetes pode ser um ótima opção para você, mas talvez algo mais simples também funcione. Não tenha vergonha de escolher uma solução mais simples e também não se preocupe muito sobre transferir o trabalho para outra pessoa - se eu puder empurrar o trabalho para o nuvem pública, então eu farei isso, pois ela permite que eu me concentre no meu próprio trabalho.

Acima de tudo, esse espaço está passando por muita agitação, espero ter te dado alguns insights sobre a principal tecnologia neste espaço, mas também compartilhou alguns princípios que provavelmente sobreviverão à atual safra de tecnologia quente.

O que quer que venha a seguir, espero que você esteja muito mais preparado para aceitá-lo

passo.

No próximo capítulo, vamos nos aprofundar em um tópico que abordamos brevemente

aqui: testando nossos microserviços para garantir que eles realmente funcionem.

1 Kief Morris, Infraestrutura como Código, 2ª edição (Sebastopol: O'Reilly, 2020).

2 Senthil Padmanabhan e Pranav Jha, "WebAssembly no eBay: um caso de uso no mundo real", eBay, 22 de maio de 2019, <https://oreil.ly/Sfv.HT>.

3 Johnathan Ishmael, "Otimizando a tecnologia sem servidor para a BBC Online", Tecnologia e criatividade na BBC (blog). BBC, 26 de janeiro de 2021. <https://oreil.ly/gk.Sdp>.

4 Talvez eu não tenha inventado essa regra.

5 Daniel Bryant, "A Apple reconstrói os serviços de back-end da Siri usando o Apache Mesos", InfoQ. 3 de maio. 2015. <https://oreil.ly/NLUMX>.

6 Nicole Forsgren, Jez Humble e Gene Kim. Accelerate: a ciência da construção e Dimensionando organizações de tecnologia de alto desempenho (Portland, OR: IT Revolution 2018).

7 Essa citação é frequentemente atribuída ao biólogo Paul Ehrlich, mas suas origens reais não são claras.

8 Sam Newman, de Monolith a Microservices (Sebastopol: O'Reilly, 2019).

9 Talvez eu não tenha inventado essa regra.

Capítulo 9. Testando

O mundo dos testes automatizados avançou significativamente desde que comecei escrevendo código, e todo mês parece haver alguma nova ferramenta ou técnica para torná-lo ainda melhor. Mas os desafios permanecem em relação a como efetivamente e teste eficientemente a funcionalidade do nosso código quando ele abrange um código distribuído sistema. Este capítulo detalha os problemas associados aos testes mais precisos sistemas granulados e apresenta algumas soluções para ajudá-lo a garantir que você possa lance sua nova funcionalidade com confiança.

Os testes abrangem muito terreno. Mesmo quando estamos falando apenas sobre testes automatizados, há um grande número a considerar. Com microserviços, adicionamos outro nível de complexidade. Entendendo o que é diferente os tipos de testes que podemos executar são importantes para nos ajudar a equilibrar o que às vezes-forças opostas de colocar nosso software em produção tão rapidamente quanto possível em vez de garantir que nosso software seja de qualidade suficiente. Dado o escopo do teste como um todo, não vou tentar uma ampla exploração de o tópico. Em vez disso, este capítulo se concentra principalmente em analisar como testar de uma arquitetura de microserviços é diferente quando comparada à menos distribuída sistemas como aplicativos monolíticos de processo único.

O local onde os testes são feitos também mudou desde a primeira edição deste livro. Anteriormente, os testes eram realizados predominantemente antes que o software chegasse ao produção. Cada vez mais, porém, procuramos testar nossos aplicativos uma vez eles chegam à produção - confundindo ainda mais os limites entre o desenvolvimento e atividades relacionadas à produção; isso é algo que exploraremos neste capítulo antes de explorar os testes em produção de forma mais completa no Capítulo 10.

Tipos de testes

Como muitos consultores, sou culpado de ocasionalmente usar quadrantes como forma de categorizando o mundo, e eu estava começando a me preocupar que esse livro não funcionaria um. Felizmente, Brian Marick criou um sistema de categorização fantástico para

testes que se encaixam perfeitamente. A Figura 9-1 mostra uma variação do quadrante de Marick de

O livro Agile Testing 1, de Lisa Crispin e Janet Gregory, que ajuda a categorizar os diferentes tipos de testes.



Figura 9-1. Lisa Crispin e Janet Gregory, quadrante de testes de Brian Marick, Agile Testing: A Guia prático para testadores e equipes ágeis, c 2009

Na parte inferior do quadrante, temos testes que estão voltados para a tecnologia, ou seja, testes que ajudam os desenvolvedores a criar o sistema em primeiro lugar.

Testes de propriedades, como testes de desempenho e testes unitários de pequeno escopo, se enquadram em esta categoria; todos são normalmente automatizados. A metade superior do quadrante inclui os testes que ajudam as partes interessadas não técnicas a entender como seu sistema funciona, o que chamamos de testes voltados para negócios. Esses poderiam ser testes de grande escopo, de ponta a ponta, conforme mostrado no quadrante de Teste de Aceitação no canto superior esquerdo; ou teste manual (conforme tipificado pelo teste do usuário feito em um UAT sistema), conforme mostrado no quadrante de testes exploratórios.

Neste ponto, vale ressaltar o fato de que a grande maioria desses testes foca na validação da pré-produção. Especificamente, estamos usando esses testes para garantir que o software tenha qualidade suficiente antes de ser implantado em um ambiente de produção. Normalmente, esses testes serem aprovados (ou reprovados) seriam uma condição de bloqueio para decidir se o software deve ser implantado.

Cada vez mais, estamos vendo o valor de testar nosso software quando, na verdade, entre em um ambiente de produção. Falaremos mais sobre o equilíbrio entre essas duas ideias mais adiante no capítulo, mas por enquanto vale a pena destacar um limite do quadrante de Marick nesse sentido.

Cada tipo de teste mostrado neste quadrante tem um lugar. Exatamente quanto cada teste que você deseja fazer dependerá da natureza do seu sistema, mas a chave Um ponto a entender é que você tem várias opções em termos de como testar seu sistema. Recentemente, a tendência se afastou de qualquer manual de grande escala testando em favor de automatizar o máximo possível de testes repetitivos, e eu certamente concordo com essa abordagem. Se você atualmente realiza grandes quantidades de teste manual, sugiro que você resolva isso antes de prosseguir demais no caminho dos microserviços, pois você não obterá muitos de seus benefícios se você não consegue validar seu software com rapidez e eficiência.

MANUAL EXPLORATÓRIO TESTE TESTE EXPLORATÓRIO MANUAL

Em geral, a mudança de uma arquitetura monolítica para um microserviço a arquitetura terá um impacto mínimo nos testes exploratórios, além de qualquer mudança organizacional mais ampla que possa ocorrer. Como exploraremos em "Rumo a equipes alinhadas ao fluxo", esperaríamos a interface de usuário para o aplicativo também deve ser dividido de acordo com as linhas da equipe. Nesse contexto, a propriedade do teste manual pode muito bem mudar.

A capacidade de automatizar algumas tarefas, como verificar como algo parece, anteriormente estava limitado exclusivamente a testes exploratórios manuais. O amadurecimento das ferramentas que permitem afirmações visuais nos permitiu começar a automatizar tarefas que antes eram feitas manualmente. Você não deveria No entanto, veja isso como um motivo para não fazer testes manuais; em vez disso, você deveria veja isso como uma chance de liberar o tempo dos testadores para se concentrarem em atividades menos repetitivas, testes exploratórios.

Quando bem feito, o teste exploratório manual envolve, em grande parte, descoberta. Reservar um tempo para explorar o aplicativo como usuário final pode descobrir problemas que, de outra forma, não seriam evidentes. O teste manual pode também ser útil em situações em que um teste automatizado é inviável para implementar, talvez por causa do custo de escrever o teste. A automação é sobre a remoção de tarefas repetitivas para liberar os humanos para fazerem mais atividades criativas e ad hoc. Então, pense na automação como uma forma de liberar nossa capacidade intelectual para as coisas que fazemos melhor.

Para os fins deste capítulo, geralmente ignoramos a exploração manual de testes. Isso não quer dizer que esse tipo de teste não seja importante, mas apenas isso o escopo deste capítulo é focar principalmente em como testar microservices. Isso difere de testar aplicações monolíticas mais típicas. Mas quando chega aos testes automatizados, quantos de cada teste queremos? Outro modelo será ajudar-nos a responder a essa pergunta e entender quais são as diferentes vantagens que pode ser.

Escopo do teste

Em seu livro *Succeeding with Agile*, Mike Cohn descreve um modelo chamado de pirâmide de teste para ajudar a explicar quais tipos de testes automatizados são necessários. O modelo divide os testes automatizados em testes unitários, testes de serviço e testes de interface do usuário, como ilustrado na Figura 9-2.

Aumentando o escopo

Mais confiança

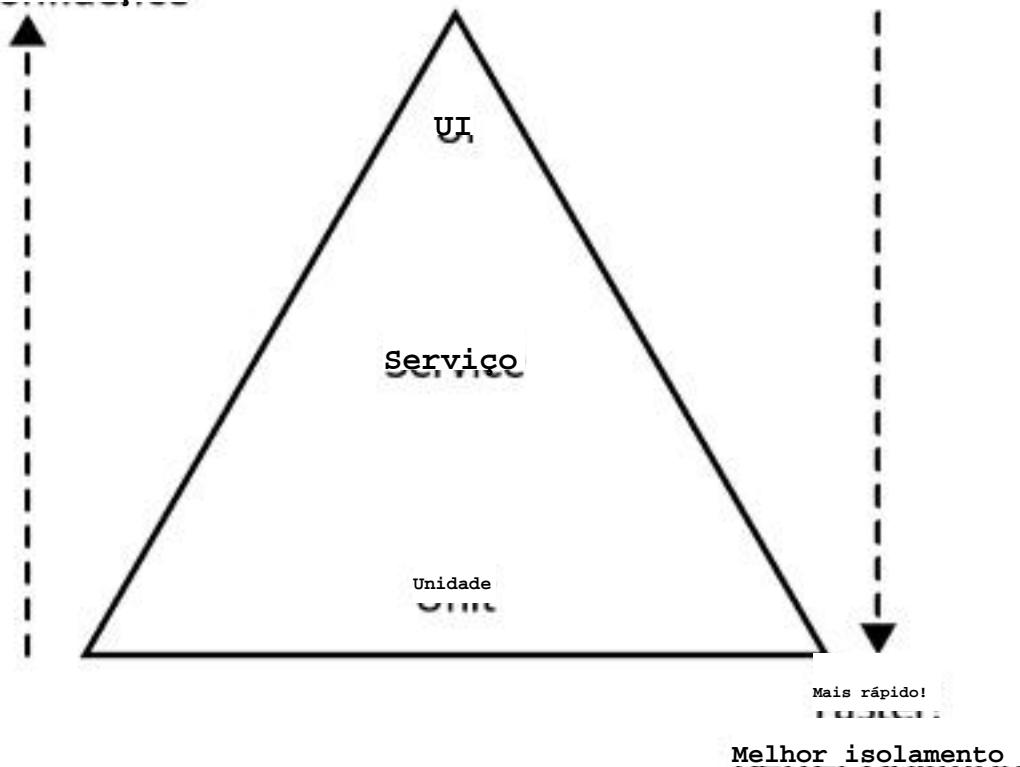


Figura 9-2. Pirâmide de teste de Mike Cohn. Mike Cohn. Obtendo sucesso com o Agile: desenvolvimento de software Usando Scrum, 1^a ed., c 2010

A principal coisa a ser aprendida ao ler a pirâmide é que, à medida que subimos o pirâmide, o escopo do teste aumenta, assim como nossa confiança de que o código é funcional. Por outro lado, o tempo do ciclo de feedback aumenta à medida que os testes demoram mais para serem executados e, quando um teste falha, pode ser mais difícil determinar qual funcionalidade foi interrompida. À medida que descemos a pirâmide, em geral, os testes se tornam muito mais rápidos, então obtemos ciclos de feedback muito mais rápidos. Encontramos funcionalidades quebradas mais rapidamente, nossas construções de integração contínua são mais rápidas, e é menos provável que passemos para uma nova tarefa antes de descobrirmos quebrei alguma coisa. Quando esses testes de escopo menor falham, também tendemos a sair o que falhou, geralmente até a linha exata do código - cada teste é melhor isolado, facilitando a compreensão e a correção de quebras. Por outro lado, Além disso, não temos muita confiança de que nosso sistema como um todo funcione se testamos apenas uma linha de código!

O problema com esse modelo é que todos esses termos significam coisas diferentes para pessoas diferentes. O serviço está especialmente sobre carregado e há muitos definções de um teste unitário disponível. Um teste é um teste unitário se eu testar apenas uma linha de código? Eu diria que é. Ainda é um teste unitário se eu testar várias funções ou classes? Eu diria que não, mas muitos discordariam! Eu custumo ficar com a unidade e o serviço nomes apesar de sua ambigüidade, mas preferem chamar testes de interface de ponta a ponta testes, que farei a partir de agora

Praticamente todas as equipes em que trabalhei usaram nomes diferentes para testes do que aqueles que Cohn usa na pirâmide. O que quer que você os chame, a chave A conclusão é que você desejará testes automatizados funcionais de diferentes escopos para finalidades diferentes.

Dada a confusão, vale a pena ver o que essas diferentes camadas significam. Vejamos um exemplo prático. Na Figura 9-3, temos nosso helpdesk aplicativo e nosso site principal, ambos interagindo com nosso Microserviço do cliente para recuperar, revisar e editar detalhes do cliente. Nossso microserviço do cliente, por sua vez, está conversando com nosso microserviço de fidelidade, onde nossos clientes acumulam pontos comprando CDs de Justin Bieber. Provavelmente, obviamente, isso é uma parte do nosso sistema MusicCorp geral, mas é um bom fatia suficiente para mergulharmos em alguns cenários diferentes que talvez queiramos testar.

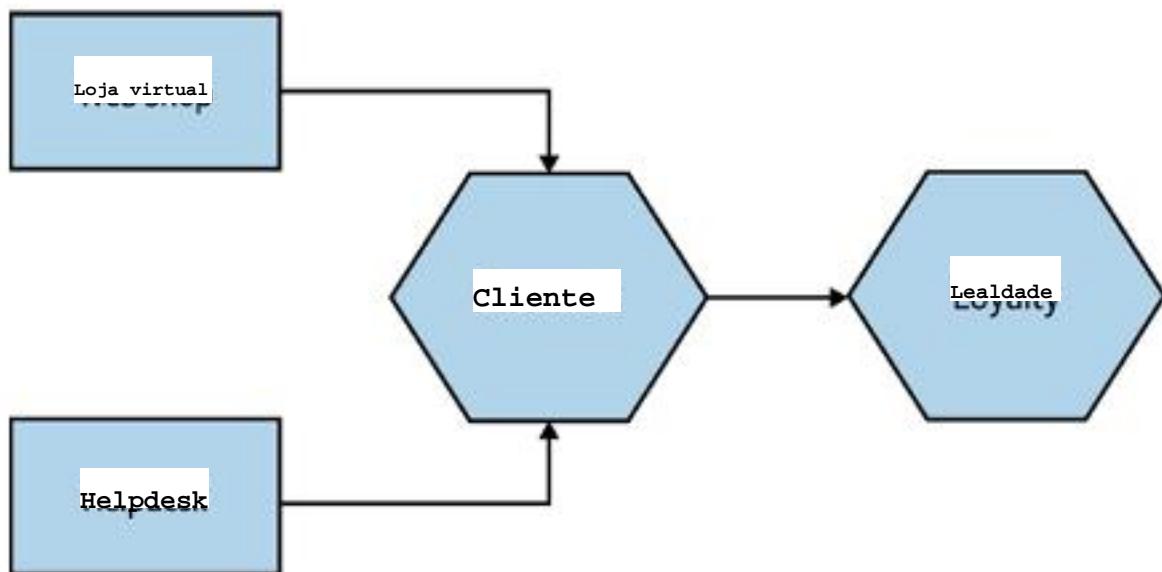


Figura 9-3 Parte da nossa loja de música em teste

Testes unitários

Os testes unitários normalmente testam uma única função ou chamada de método. Os testes gerados como efeito colateral do design orientado a testes (TDD) se enquadram nessa categoria, assim como os tipos de testes gerados por técnicas como testes baseados em propriedades. Não estamos lançando microserviços aqui e estamos limitando o uso de serviços externos, arquivos ou conexões de rede. Em geral, você quer um grande número desses tipos de testes. Feitos corretamente, eles são muito, muito rápidos e estão em hardware moderno. Você pode esperar executar muitos milhares deles em menos de um minuto. Eu sei que muitas pessoas que têm esses testes sendo executados automaticamente quando os arquivos estão alterados localmente, especialmente com idiomas interpretados, isso pode dar muitos ciclos rápidos de feedback.

Os testes unitários nos ajudam, desenvolvedores e, portanto, seriam voltados para a tecnologia, não voltado para negócios, na terminologia de Marick. Eles também estão onde esperamos pegue a maioria dos nossos insetos. Então, em nosso exemplo, quando pensamos sobre o Microserviço para clientes e testes unitários cobririam pequenas partes do código em isolamento, conforme mostrado na Figura 9-4.

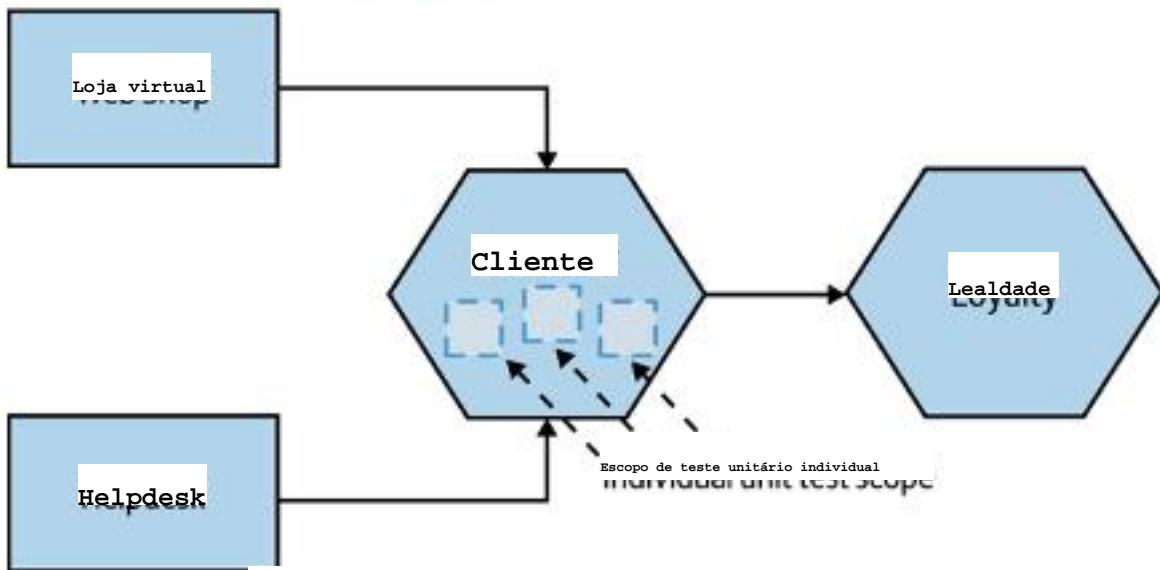


Figura 9-4 Escopo dos testes unitários em nosso sistema de exemplo

O objetivo principal desses testes é nos fornecer um feedback muito rápido sobre se nossa funcionalidade é boa. Os testes unitários também são importantes para apoiar refatoração do código, permitindo-nos reestruturar nosso código à medida que nos protegemos.

o conhecimento de que nossos testes de pequeno escopo nos surpreenderão se fizermos um erro.

Testes de serviço

Os testes de serviço são projetados para contornar a interface do usuário e testar nossa microsserviços diretamente. Em um aplicativo monolítico, talvez estejamos apenas testando uma coleção de classes que fornecem um serviço para a interface do usuário. Para um sistema compreendendo vários microsserviços, um teste de serviço testaria umas das capacidades individuais do microsserviço.

Ao executar testes em um único microsserviço dessa forma, aumentamos a confiança de que o serviço se comportará conforme o esperado, mas ainda mantemos o escopo do teste um pouco isolado. A causa da falha do teste deve ser limitado apenas ao microsserviço em teste. Para alcançar esse isolamento, precisamos para eliminar todos os colaboradores externos para que somente o microsserviço em si esteja incluído no escopo, como mostra a Figura 9-5.

Alguns desses testes podem ser tão rápidos quanto nossos testes unitários de pequeno escopo, mas se você decidir testar em um banco de dados real ou passar pelas redes para o stubbed colaboradores posteriores, os tempos de teste podem aumentar. Esses testes também abrangem mais escopo do que um simples teste unitário, então, quando eles falham, pode ser mais difícil detectar o que está quebrado do que com um teste unitário. No entanto, eles têm consideravelmente menos partes móveis e, portanto, são menos frágeis do que os testes com escopo maior.

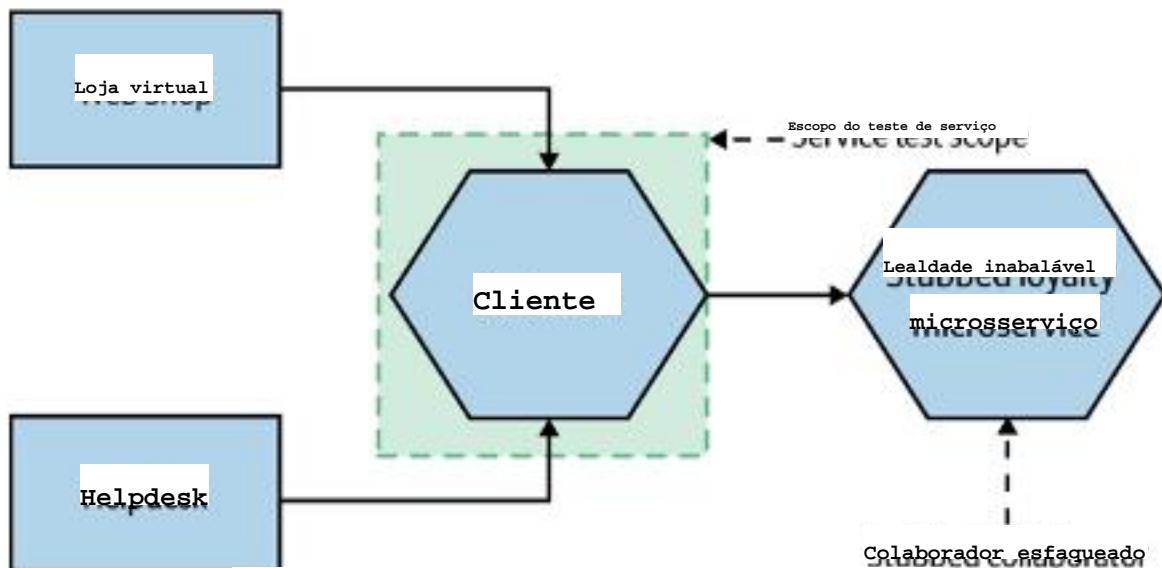


Figura 9-5. Escopo dos testes de serviço em nosso sistema de exemplo.

Testes de ponta a ponta

Testes de ponta a ponta são testes executados em todo o sistema. Muitas vezes eles serão conduzindo uma GUI por meio de um navegador, mas eles poderiam facilmente estar imitando outros tipos de interação do usuário, como o upload de um arquivo.

Esses testes abrangem muitos códigos de produção, como vemos na Figura 9-6. Assim quando eles passam, você se sente bem: você tem um alto grau de confiança de que o código que está sendo testado funcionará na produção. Mas esse escopo aumentado vem com desvantagens e, como veremos em breve, os testes de ponta a ponta podem ser muito complicados para se sair bem em um contexto de microserviços.

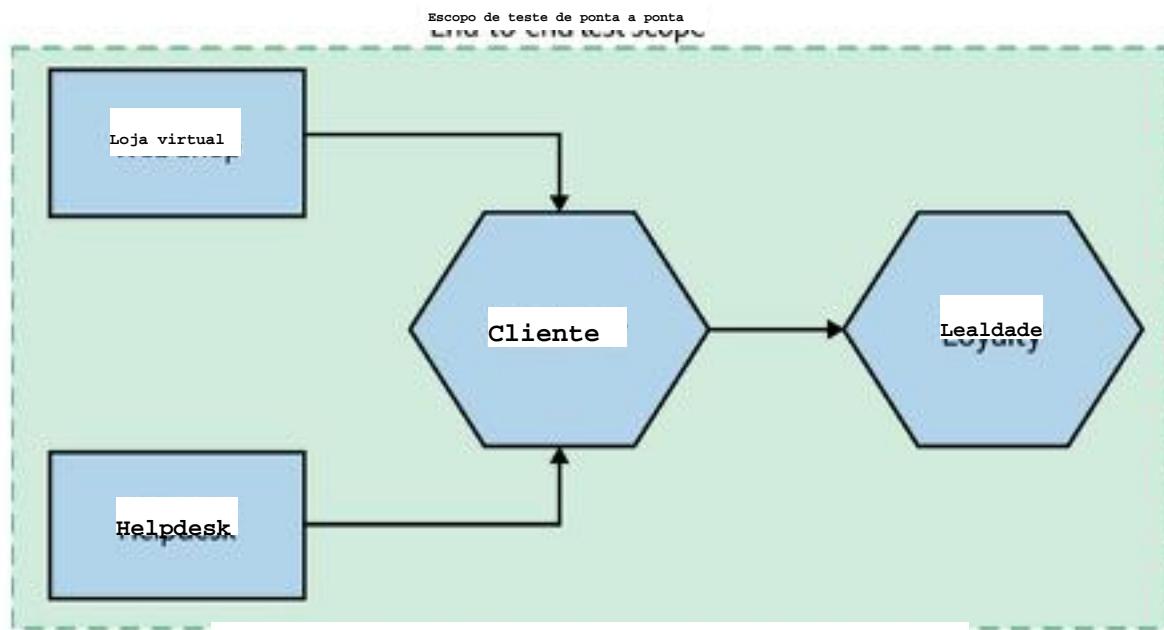


Figura 9-6. Escopo dos testes de ponta a ponta em nosso sistema de exemplo

E QUANTO AOS TESTES DE INTEGRAÇÃO?

Você pode notar que eu não descrevi explicitamente os testes de integração. Isso é de propósito. Descobri que o termo é frequentemente usado por pessoas diferentes para descrever diferentes tipos de testes. Para alguns, um teste de integração pode ser apenas observar as interações entre dois serviços, ou talvez a vinculação entre o código e um banco de dados. Para outros, os testes de integração acabam sendo o mesmo que testes completos de ponta a ponta. Eu tentei usar termos mais explícitos em este capítulo; espero que seja fácil para você combinar o que quer que você se refira como um "teste de integração" para os termos que eu uso aqui.

Compensações

O que estamos buscando com os diferentes tipos de testes que a pirâmide das capas são um equilíbrio sensato. Queremos feedback rápido e queremos confiança que nosso sistema funciona.

Os testes unitários têm um escopo pequeno, então, quando falham, podemos encontrar o problema rapidamente. Eles também são rápidos de escrever e muito rápidos de executar. À medida que nossos testes terminam maior em escopo, temos mais confiança em nosso sistema, mas em nosso feedback

começa a sofrer à medida que os testes demoram mais para serem executados. Eles também são mais caros para escrever e manter.

Freqüentemente, você estará equilibrando quantos de cada tipo de teste você precisa encontrar. aquele ponto ideal. Descobrindo que sua suíte de testes está demorando muito para ser executada? Quando testes de escopo mais amplo, como nosso serviço ou testes de ponta a ponta, falham, escreva um teste unitário de menor escopo para detectar essa quebra mais cedo. Procure substituir alguns testes de escopo maior (e mais lentos) com testes unitários mais rápidos e de menor escopo. Por outro lado, quando um bug chega à produção, pode ser um sinal que você está perdendo um teste.

Então, se todos esses testes têm vantagens e desvantagens, quantos de cada tipo de teste você quer? Uma boa regra geral é que você provavelmente quer uma ordem de magnitude maior testa enquanto você desce a pirâmide. É importante saber que você tem diferentes tipos de testes automatizados, e é importante entender se seu o saldo atual causa um problema!

Eu trabalhei em um sistema monolítico, por exemplo, onde tínhamos 4.000 unidades testes, 1.000 testes de serviço e 60 testes de ponta a ponta. Decidimos que a partir de um ponto de vista do feedback, tivemos muitos serviços e testes de ponta a ponta (os últimos foram os piores criminosos ao impactar os ciclos de feedback), por isso, trabalhamos duro para substituir a cobertura do teste por testes de menor escopo.

Um antipadrão comum é o que geralmente é chamado de cone de neve de teste ou pirâmide invertida. Aqui, há pouco ou nenhum teste de pequeno escopo, com todos os cobertura em testes de grande escopo. Esses projetos geralmente têm testes glacialmente lentos execuções e ciclos de feedback muito longos. Se esses testes forem executados como parte do integração contínua, você não terá muitas compilações e a natureza da construção times significa que a construção pode ficar quebrada por um longo período quando algo quebra.

Implementando testes de serviço

Implementar testes unitários é uma tarefa bastante simples no grande esquema das coisas, e há muita documentação explicando como escrevê-las.

O serviço e os testes de ponta a ponta são os mais interessantes, especialmente no contexto de microsserviços, então é nisso que nos concentraremos a seguir.

Nossos testes de serviço querem testar uma fatia da funcionalidade em todo o mundo microsserviço e somente esse microsserviço. Então, se quiséssemos escrever um serviço teste para o microsserviço Customer da Figura 9-3, implantaríamos um instância do microsserviço do cliente - e, conforme discutido anteriormente, gostaríamos querer eliminar o microsserviço Loyalty para garantir melhor que um teste a quebra pode ser atribuída a um problema com o próprio microsserviço do Cliente.

Conforme exploramos no Capítulo 7, uma vez que verificamos nosso software, um dos primeiros coisas que nossa construção automatizada fará é criar um artefato binário para nosso microsserviço - por exemplo, criando uma imagem de container para essa versão do software. Portanto, implantar isso é bem simples. Mas como lidamos com isso? fingindo os colaboradores posteriores?

Nosso conjunto de testes de serviços precisa eliminar colaboradores posteriores e configure o microsserviço em teste para se conectar aos serviços de stub. Nós então precisa configurar os stubs para enviar respostas para imitar o mundo real microsserviços.

Zombando ou esbarrando

Quando falo em subtrair colaboradores, quero dizer que criamos um microsserviço de esboço que responde com respostas prontas a solicitações conhecidas do microsserviço em teste. Por exemplo, eu poderia dizer que meu palhaco Microsserviço de fidelidade que, quando solicitado o saldo do cliente 123, deve devolver 15.000. O teste não se importa se o esboço é chamado de 0, 1 ou 100 vezes. Uma variação disso é usar uma simulação em vez de um esboço.

Ao usar uma simulação, eu realmente vou mais longe e me certifico de que a ligação foi feita. Se a chamada esperada não é feita, o teste falha. Implementando essa abordagem exige mais inteligência nos falsos colaboradores que criamos e, se usados em excesso, pode fazer com que os testes se tornem quebradiços. Conforme observado, no entanto, um esboço não se importa se é chamado de 0, 1 ou várias vezes.

Às vezes, porém, simulações podem ser muito úteis para garantir que o lado esperado efeitos acontecem. Por exemplo, talvez eu queira verificar se quando crio um cliente, um novo saldo de pontos é configurado para esse cliente. O equilíbrio entre telefonar e zombar de chamadas é delicado e é igualmente complicado testes de serviço como em testes unitários. Em geral, porém, eu uso esboços muito mais do que simulações para testes de serviço. Para uma discussão mais aprofundada sobre essa compensação, faça um olhar sobre o crescimento do software orientado a objetos, guiado por testes de Steve Freeman e Nat Pryce.³

Em geral, raramente uso simulações para esse tipo de teste. Mas ter uma ferramenta que pode implementar simulações e esboços é útil.

Embora eu ache que esboços e simulações são, na verdade, bastante diferenciados, eu saber que a distinção pode ser confusa para alguns, especialmente quando algumas pessoas use outros termos, como falsificações, espiões e bonecos. Gerard Meszaros liga todas essas coisas, incluindo esboços e simulações, "Test Doubles".

Um serviço de esboço mais inteligente

Normalmente, para serviços de esboços, eu mesmo os crio. Eu usei tudo do servidor web Apache ou nginx para contêineres Jetty incorporados ou até mesmo servidores web Python lançados por linha de comando para iniciar servidores stub para tais casos de teste. Provavelmente já reproduzi o mesmo trabalho várias vezes em criando esses esboços. Um antigo colega meu da Thoughtworks, Brandon Byars, potencialmente economizou muito trabalho para muitos de nós com seu esboço/simulação servidor chamado mountebank.

Você pode pensar no Mountebank como um pequeno dispositivo de software, ou seja, programável via HTTP. O fato de estar escrito em NodeJS é completamente opaco para qualquer serviço de chamadas. Quando Mountebank é lançado, você envie comandos pedindo que ele crie um ou mais "impostores", que serão respondida em uma determinada porta com um protocolo específico (atualmente TCP, HTTP, HTTPS e SMTP são suportados). e suas respostas esses impostores deve enviar quando as solicitações forem enviadas. Ele também suporta o estabelecimento de expectativas se você quer usá-lo como uma simulação. Porque uma única instância de mountebank pode

apoie a criação de vários impostores, você pode usá-lo para eliminar vários microsserviços downstream.

O mountebank tem usos fora dos testes funcionais automatizados. Capital

Um fez uso da marcenha para substituir a infraestrutura de simulação existente por seus testes de desempenho em grande escala, por exemplo. 4

Uma limitação da encosta da montanha é que ela não suporta arranhões protocolos de mensagens - por exemplo, se você quiser ter certeza de que um evento foi enviado corretamente (e talvez recebido) por meio de um corretor, você terá que consultar em outro lugar para uma solução. Esta é uma área em que o Pact pode ajudar - isso é algo que veremos mais em breve.

Então, se quisermos executar nossos testes de serviço apenas para nosso microsserviço de cliente, podemos iniciar o microsserviço Customer e uma instância de mountebank que atua como nosso microsserviço de fidelidade na mesma máquina. E se esses testes passem, posso implantar o atendimento ao cliente imediatamente! Ou eu posso? E quanto os serviços que chamam o microsserviço do cliente - o helpdesk e a web fazer compras? Sabemos se fizemos uma mudança que pode quebrá-los? Declaro, esquecemos os testes importantes no topo da pirâmide: os testes de ponta a ponta.

Implementando (aqueles complicados) de ponta a ponta

Testes

Em um sistema de microsserviços, os recursos que expomos por meio de nossas interfaces de usuário são fornecidos por vários microsserviços. O objetivo dos testes de ponta a ponta conforme descrito na pirâmide de Mike Cohn, é impulsionar a funcionalidade por meio das interfaces de usuário com tudo abaixo para nos dar algum feedback sobre a qualidade do sistema como um todo.

Então, para implementar um teste de ponta a ponta, precisamos implantar vários microsserviços juntos e, em seguida, execute um teste em todos eles. Obviamente, isso o teste tem um escopo muito maior, resultando em mais confiança de que nosso sistema funciona! Por outro lado, esses testes podem ser mais lentos e sobreviver

mais difícil de diagnosticar a falha. Vamos nos aprofundar um pouco mais usando nosso anterior exemplo para ver como esses testes podem se encaixar.

Imagine que queremos lançar uma nova versão do microserviço Customer. Queremos implantar nossas mudanças na produção o mais rápido possível, mas estamos preocupado com o fato de podermos ter introduzido uma mudança que poderia quebrar o helpdesk ou a loja virtual. Sem problemas, vamos implantar todos os nossos serviços juntos e execute alguns testes no helpdesk e na loja virtual para ver se introduzimos um bug. Agora, uma abordagem ingênua seria simplesmente adicionar esses testes até o final do nosso pipeline de atendimento ao cliente, como na Figura 9-7.



Figura 9-7 Adicionando nosso estágio de testes de ponta a ponta: a abordagem correta?

Até agora, tudo bem. Mas a primeira pergunta que devemos nos fazer é: qual versão dos outros microserviços que devemos usar? Devemos fazer nossos testes contra as versões de helpdesk e loja virtual que estão em produção? É uma suposição sensata - mas e se uma nova versão do helpdesk ou da loja virtual está na fila para entrar no ar? O que devemos fazer então?

Aqui está outro problema: se tivermos um conjunto de testes de ponta a ponta do Cliente que implante muitos microserviços e execute testes com eles, e quanto ao final testes completos que os outros microserviços executam? Se eles estão testando o mesmo coisa, podemos nos encontrar cobrindo muito do mesmo terreno e pode duplicar grande parte do esforço para implantar todos esses microserviços no primeiro lugar.

Podemos lidar com esses dois problemas com elegância, tendo vários canais para o "fan-in" para um único estágio de teste de ponta a ponta. Aqui, quando um de um número de compilações diferentes é acionado, isso pode resultar em etapas de construção compartilhadas sendo acionada. Por exemplo, na Figura 9-8, uma compilação bem-sucedida para qualquer um dos quatro microserviços acabaria acionando a fase compartilhada de testes de ponta a ponta. Algumas ferramentas de CI com melhor suporte de pipeline de construção permitirão modelos fan-in como isso fora da caixa.

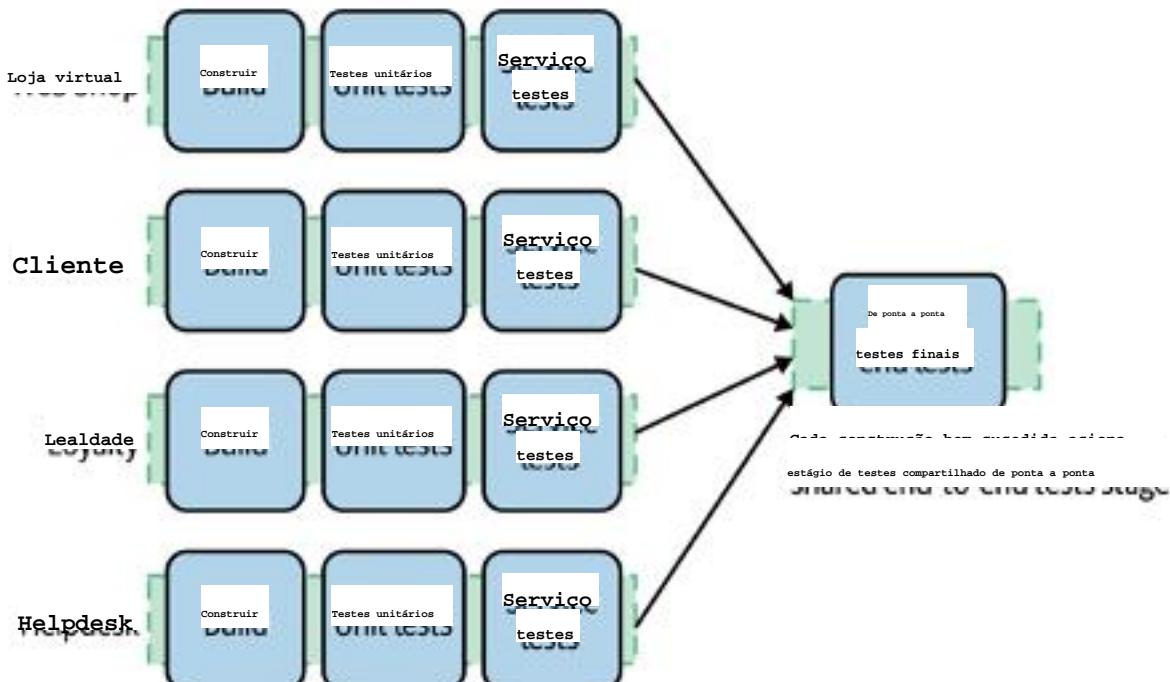


Figura 9-8. Uma forma padrão de lidar com testes de ponta a ponta em todos os serviços

Então, sempre que um de nossos serviços muda, executamos os testes localmente.

serviço. Se esses testes passarem, açãoaremos nossos testes de integração. Ótimo, hein? Bem, infelizmente, existem muitas desvantagens nos testes de ponta a ponta.

Testes de escamação e fragilidade

À medida que o escopo do teste aumenta, também aumenta o número de peças móveis. Esses partes móveis podem introduzir falhas de teste que não mostram que a funcionalidade o teste está quebrado, mas indica que algum outro problema ocorreu. Como um exemplo, se tivermos um teste para verificar se podemos fazer um pedido de um único CD e estamos executando esse teste em quatro ou cinco microserviços, se houver eles estão inativos, poderíamos ter uma falha que não tem nada a ver com a natureza do teste em si. Da mesma forma, uma falha temporária na rede pode fazer com que um teste falhe sem dizer nada sobre a funcionalidade em teste.

Quanto mais partes móveis houver, mais frágeis nossos testes podem ser e eles são menos determinísticos. Se você tem testes que às vezes falham, mas todo mundo basta executá-los novamente porque eles podem passar novamente mais tarde, então você tem testes instáveis. E testes que abrangem vários processos diferentes não são os únicos culpados. Testes que abrangem a funcionalidade que está sendo exercida em vários segmentos (e em

vários processos) também costumam ser problemáticos; uma falha pode significar uma corrida condicão ou tempo limite, ou que a funcionalidade está realmente quebrada. Testes escamosos são o inimigo. Quando falham, não nos dizem muita coisa. Executamos novamente nossas compilações de CI na esperança de que eles passem novamente mais tarde, apenas para ver os check-ins se acumularem, e de repente, nos encontramos com uma grande quantidade de funcionalidades quebradas.

Quando detectamos testes escamosos, é essencial que façamos o possível para removê-los. Caso contrário, começaremos a perder a fé em uma suíte de testes que "sempre falha desse jeito". UMA Uma suíte de testes com testes instáveis pode se tornar uma vítima do que Diane Vaughan chama a normalização do desvio - a ideia de que com o tempo podemos nos tornar assim acostumados a que as coisas estejam erradas. Começamos a aceitá-las como sendo normal e não um problema 5 Essa tendência muito humana significa que precisamos encontrar e eliminar esses testes instáveis o mais rápido possível antes de começarmos a presumir que os testes reprovados estão OK.

Em "Erradicando o não-determinismo em testes", 6 Martin Fowler defende a abordagem de que, se você tiver testes instáveis, você deve rastreá-los e se você não pode consertá-los imediatamente - remova-os da suíte para que você possa tratar eles. Veja se você pode reescrevê-los para evitar testar o código em execução em vários fios. Veja se você pode tornar o ambiente subjacente mais estável. Melhor ainda assim, veja se você pode substituir o teste flaky por um teste de menor escopo que seja menor provável que apresente problemas. Em alguns casos, alterar o software em teste para facilitar o teste também pode ser o caminho certo a seguir.

Quem escreve esses testes de ponta a ponta?

Com os testes executados como parte do pipeline de um microserviço específico. Um ponto de partida sensato é que a equipe proprietária desse serviço os escreva testes (falaremos mais sobre propriedade do serviço no Capítulo 15). Mas se nós consideremos que podemos ter várias equipes envolvidas, e a A etapa de testes agora é efetivamente compartilhada entre as equipes, que escrevem e examinam depois desses testes?

Eu vi vários problemas causados aqui. Esses testes se tornam gratuitos- para todos, com todas as equipes tendo acesso para adicionar testes sem qualquer entendimento da saúde de toda a suíte. Isso geralmente pode resultar em uma explosão de teste

casos, às vezes culminando no teste de cone de neve de que falamos anteriormente. EU também vi situações em que, porque não havia um óbvio real propriedade desses testes, seus resultados são ignorados. Quando eles quebram, todo mundo assume que é problema de outra pessoa, então eles não se importam se os testes estão passando.

Uma solução que vi aqui é designar certos testes de ponta a ponta como sendo a responsabilidade de uma determinada equipe, mesmo que ela possa atravessar microserviços sendo trabalhados por várias equipes diferentes. Eu aprendi pela primeira vez sobre essa abordagem de Emily Bache.⁷ A ideia é que, embora façamos uso de um estágio de "entrada de ventilador" em nosso pipeline, eles então dividiriam o teste de ponta a ponta conjunto em grupos de funcionalidades que pertenciam a diferentes equipes, como veremos na Figura 9-9.

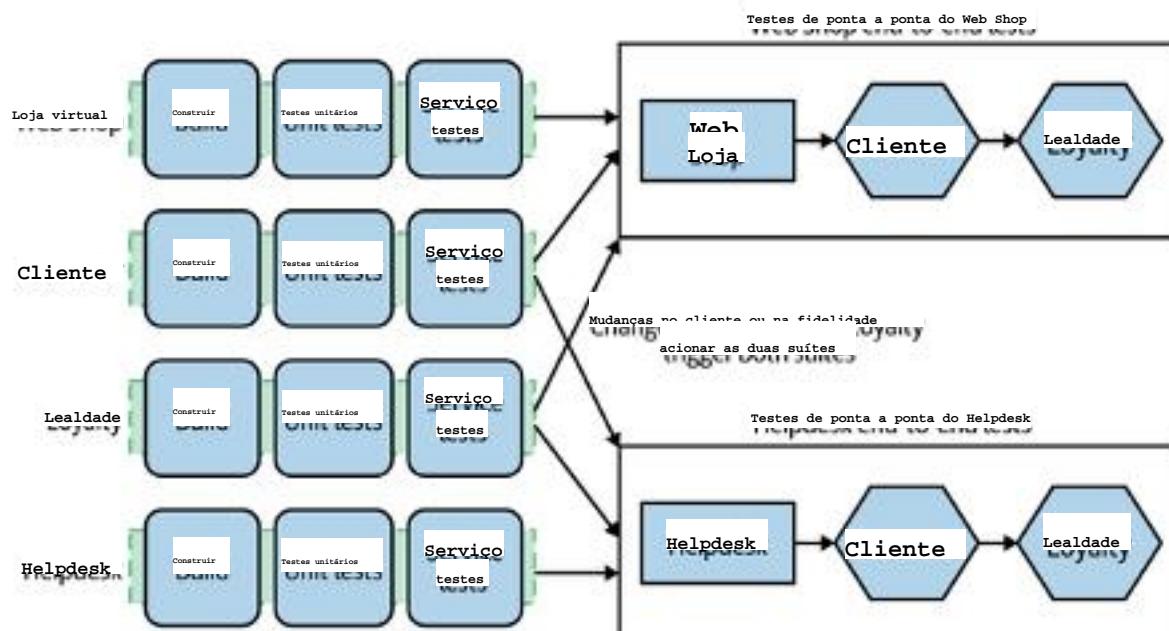


Figura 9-9. Uma forma padrão de lidar com testes de ponta a ponta em todos os serviços

Neste exemplo específico, uma alteração na loja virtual que passa o serviço de teste de ponta a ponta para a equipe proprietária da Loja Virtual. Da mesma forma, qualquer alteração no cliente ou na fidelidade só afetará os testes de ponta a ponta associados ao cliente e à fidelidade. No entanto, se houver uma alteração no Helpdesk, ele só executará os testes de ponta a ponta associados ao Helpdesk. Isso pode nos levar a uma situação em que uma alteração feita no microserviço Loyalty poderia falhar.

ambos os conjuntos de testes de ponta a ponta, potencialmente exigindo as equipes que os possuem duas suítes de teste para perseguir o proprietário do microserviço Loyalty em busca de uma solução. Embora esse modelo tenha ajudado no caso de Emily, como podemos ver, ele ainda tem seu desafios. Fundamentalmente, é problemático ter uma equipe própria responsabilidade por testes em que pessoas de uma equipe diferente podem causar esses testes para quebrar.

Às vezes, as organizações reagem fazendo com que uma equipe dedicada escreva esses testes. Isso pode ser desastroso. A equipe que desenvolve o software se torna cada vez mais distante dos testes de seu código. Os tempos de ciclo aumentam, à medida que os proprietários do serviço acabam esperando que a equipe de teste escreva testes de ponta a ponta para a funcionalidade que acabou de escrever. Como outra equipe escreve esses testes, o A equipe que escreveu o serviço está menos envolvida e, portanto, menos propensa a saiba como executar e corrigir esses testes. Embora, infelizmente, ainda seja um padrão organizacional comum, vejo danos significativos causados sempre que uma equipe está distante de escrever testes para o código que ele escreveu em primeiro lugar.

Acertar esse aspecto é muito difícil. Não queremos duplicar esforços, nem queremos centralizar isso completamente na medida em que a formação de equipes, os serviços estão muito distantes das coisas. Se você puder encontrar uma maneira limpa de atribuir testes de ponta a ponta a uma equipe específica e, em seguida, faça isso. Se não, e se você não puder encontrar uma maneira de remover testes de ponta a ponta e substituí-los por outra coisa, então você provavelmente precisará tratar o conjunto de testes de ponta a ponta como uma base de código compartilhada, mas com propriedade conjunta. As equipes podem fazer o check-in nesta suíte, mas a propriedade da saúde da suíte deve ser compartilhada entre as equipes desenvolvendo os serviços em si. Se você quiser fazer uso extensivo de testes de ponta a ponta com várias equipes, acho que essa abordagem é essencial e No entanto, eu vi isso ser feito apenas raramente e nunca sem problemas. Em última análise, eu sou convencido de que, em um determinado nível de escala organizacional, você precisa se mover longe de testes de ponta a ponta entre equipes por esse motivo.

Por quanto tempo os testes de ponta a ponta devem ser executados?

Esses testes de ponta a ponta podem demorar um pouco. Eu os vi levar até um dia para execute, se não mais, e em um projeto em que trabalhei, foi necessário um conjunto completo de regressão

seis semanas! Raramente vejo equipes realmente organizarem suas suítes de teste de ponta a ponta para reduza a sobreposição na cobertura dos testes ou dedique tempo suficiente para torná-los rápidos.

Essa lentidão, combinada com o fato de que esses testes geralmente podem ser instáveis, pode seja um grande problema. Uma suíte de testes que dura o dia todo e geralmente tem quebras que não têm nada a ver com funcionalidade quebrada é um desastre. Mesmo que seu a funcionalidade está quebrada, você pode levar muitas horas para descobrir em qual ponto em que você provavelmente já teria passado para outras atividades, e o Uma mudança de contexto ao mudar seu cérebro de volta para resolver o problema seria dolorosa.

Podemos melhorar um pouco disso executando testes em paralelo - por exemplo, fazendo uso de ferramentas como o Selenium Grid. No entanto, essa abordagem não é uma substituto para realmente entender o que precisa ser testado eativamente removendo testes que não são mais necessários.

Remover testes às vezes é um exercício difícil, e eu suspeito que aqueles que tente ter muito em comum com pessoas que deseiam remover certos medidas de segurança aeroportuária. Não importa quão ineficazes sejam as medidas de segurança pode ser que qualquer conversa sobre removê-los seja frequentemente combatida com reações instintivas sobre não se importar com a segurança ou o desejo das pessoas terroristas para vencer. É difícil ter uma conversa equilibrada sobre o valor algo acrescenta versus o fardo que isso acarreta. Também pode ser difícil compensação de risco/recompensa. Você será agradecido se remover um teste? Talvez. Mas você certamente será culpado se um teste removido permitir que um bug passe. Quando chega às suítes de teste de maior escopo, no entanto, é exatamente disso que precisamos ser capaz de fazer. Se o mesmo recurso for abordado em 20 testes diferentes, talvez podemos nos livrar de metade deles, pois esses 20 testes levam 10 minutos para serem executados! O que isso requer uma melhor compreensão do risco, que é algo que os humanos são famosa arte ruim. Como resultado, essa curadoria e gerenciamento inteligentes de testes de maior escopo e alta carga acontecem com uma frequência incrivelmente rara. Desejando as pessoas fazerem isso mais não é a mesma coisa que fazer isso acontecer.

O Grande Amontoamento

Os longos ciclos de feedback associados aos testes de ponta a ponta não são apenas um problema quando se trata da produtividade do desenvolvedor. Com uma longa suíte de testes, qualquer

as pausas demoram um pouco para serem consertadas, o que reduz a quantidade de tempo que as pausas de ponta a ponta Pode-se esperar que os testes finais sejam aprovados. Se implantarmos apenas software que tenha passou por todos os nossos testes com sucesso (o que deveríamos!), isso significa menos de nossos serviços chegam ao ponto de serem implantados na produção.

Isso pode levar a um acúmulo. Enquanto um estágio de teste de integração interrompido está sendo realizado corrigido, mais mudanças das equipes iniciais podem se acumular. Além do fato de que isso pode dificultar a fixação da construção, isso significa que o escopo das mudanças será aumentos implantados. A maneira ideal de lidar com isso é não permitir que as pessoas façam o check-in. se os testes de ponta a ponta estão falhando, mas com um longo tempo de suíte de testes, isso geralmente ocorre impraticável. Tente dizer: "Vocês 30 desenvolvedores: não há check-ins até resolvemos isso construção de sete horas!" Permitindo check-ins em um teste interrompido de ponta a ponta A suíte, no entanto, está realmente resolvendo o problema errado. Se você permitir o check-in em um construção quebrada, a construção pode permanecer quebrada por mais tempo, minando sua eficácia como forma de fornecer feedback rápido sobre a qualidade do código. A resposta certa é tornar a suíte de testes mais rápida.

Quanto maior o escopo de uma implantação e maior o risco de um lançamento, o é mais provável que quebremos alguma coisa. Então, queremos ter certeza de que podemos libere mudanças pequenas e bem testadas com frequência. Quando os testes de ponta a ponta ficam lentos diminuindo nossa capacidade de liberar pequenas mudanças, elas podem acabar causando mais danos do que bom.

A metaversão

Com a etapa de testes de ponta a ponta, é fácil começar a pensar, eu sei tudo isso os serviços nessas versões funcionam juntos, então por que não implantar todos eles juntos? Isso rapidamente se torna uma conversa na linha de, Então por que não usar um número de versão para todo o sistema? Para citar Brandon Byars, "Agora você tem problemas com o 2.1.0."

Ao reunir as alterações feitas em vários serviços, nós efetivamente adote a ideia de que mudar e implantar vários serviços ao mesmo tempo é aceitável. Isso se torna a norma; fica OK. Ao fazer isso, cedemos um dos as principais vantagens de uma arquitetura de microserviços: a capacidade de implantar uma serviço por si só, independentemente de outros serviços.

Com muita frequência, a abordagem de aceitar a implantação de vários serviços, juntos se transformam em uma situação em que os serviços se acoplam. Antes serviços longos e bem separados se tornam cada vez mais confusos com outros, e você nunca percebe, porque você nunca tenta implantá-los sozinhos. Você acabe com uma bagunça emaranhada em que você precisa orquestrar a implantação do vários serviços ao mesmo tempo e, como discutimos anteriormente, esse tipo de o acoplamento pode nos deixar em um lugar pior do que estariamos com um único aplicação monolítica.

Isso é ruim.

Falta de testabilidade independente

Voltamos com frequência ao tópico de implantabilidade independente:

uma propriedade importante para facilitar o trabalho das equipes de forma mais autônoma, permitindo que o software seja enviado com mais eficiência. Se suas equipes trabalham de forma independente, conclui-se que eles devem ser capazes de testar de forma independente. Como vimos que testes de ponta a ponta podem reduzir a autonomia das equipes e podem forçar níveis aumentados de coordenação, com os desafios associados que podem trazer.

O impulso em direção à testabilidade independente se estende ao nosso uso da infraestrutura associada aos testes. Muitas vezes, vejo pessoas tendo que fazer uso de ambientes de teste compartilhados nos quais os testes de várias equipes são executado. Esse ambiente geralmente é altamente restrito e qualquer problema pode causar problemas significativos. Idealmente, se você quiser que suas equipes sejam capazes de desenvolver e testar de forma independente, eles devem ter seu próprio teste ambientes também.

A pesquisa resumida no Accelerate descobriu que equipes de alto desempenho

eram mais propensos a "fazer a maioria dos testes sob demanda", sem exigir um

ambiente de teste integrado. "8

Você deve evitar testes de ponta a ponta?

Apesar das desvantagens descritas, para muitos usuários, os testes de ponta a ponta podem ainda ser gerenciados com um pequeno número de microserviços, e nesses casos em que ainda fazem muito sentido. Mas o que acontece com 3, 4, 10 ou 20 serviços? Muito rapidamente, essas suítes de teste ficam extremamente inchadas e, na pior das hipóteses, eles podem resultar em uma explosão cartesiana nos cenários em teste.

Na verdade, mesmo com um pequeno número de microserviços, esses testes se tornam difícil quando você tem várias equipes compartilhando testes de ponta a ponta. Com um suíte de testes compartilhada de ponta a ponta, você enfraquece sua meta de ser independente capacidade de implantação. Sua capacidade como equipe de implantar um microserviço agora exige que uma suíte de testes compartilhada por várias equipes seja aprovada.

Qual é um dos principais problemas que estamos tentando resolver quando usamos os testes de ponta a ponta descritos anteriormente? Estamos tentando garantir que, quando implantando um novo serviço na produção, nossas mudanças não prejudicarão os consumidores. Agora, como abordamos em detalhes em "Contrato estrutural versus contrato semântico "Quebras", ter esquemas explícitos para nossas interfaces de microserviços pode ajudar detectarmos quebras estruturais, e isso pode definitivamente reduzir a necessidade de mais testes complexos de ponta a ponta.

No entanto, os esquemas não conseguem detectar quebras semânticas, ou seja, mudanças em comportamento que causa quebras devido à incompatibilidade com versões anteriores. De ponta a ponta testes absolutamente podem ajudar a detectar essas quebras semânticas, mas o fazem em um ótimo custo. Idealmente, gostaríamos de ter algum tipo de teste que pudesse responder a quebra semântica muda e é executada em um escopo reduzido, melhorando o teste isolamento (e, portanto, velocidade do feedback). É aqui que o contrato testa e entram os contratos voltados para o consumidor.

Testes de contrato e contratos orientados ao consumidor (CDCs)

Com testes de contrato, uma equipe cujo microserviço consome um externo service escreve testes que descrevem como ele espera que um serviço externo se comportar. Isso é menos sobre testar seu próprio microserviço e mais sobre especificando como você espera que um serviço externo se comporte. Um dos principais

Os motivos pelos quais esses testes de contrato podem ser úteis é que eles podem ser comparados quaisquer esboços ou simulações que você esteja usando que representem serviços externos - seu os testes de contrato devem ser aprovados quando você executa seus próprios esboços, assim como deveriam o verdadeiro serviço externo.

Os testes de contrato se tornam muito úteis quando usados como parte de atividades voltadas para o consumidor contratos (CDCs). Os testes contratuais são, na verdade, de forma explícita e programática representação de como o microsserviço do consumidor (upstream) espera o microsserviço produtor (downstream) para se comportar. Com os CDCs, o consumidor A equipe garante que esses testes de contrato sejam compartilhados com a equipe de produtores para permitir que a equipe de produtores garanta que seu microsserviço atenda a essas expectativas. Normalmente, isso é feito com a equipe de produtores downstream execute os contratos de consumo para cada microsserviço consumidor como parte de seu teste suíte que seria executada em todas as compilações. Muito importante, a partir de um feedback de teste ponto de vista, esses testes precisam ser executados apenas contra um único produtor em isolamento, para que possam ser mais rápidos e confiáveis do que os testes de ponta a ponta eles podem substituir.

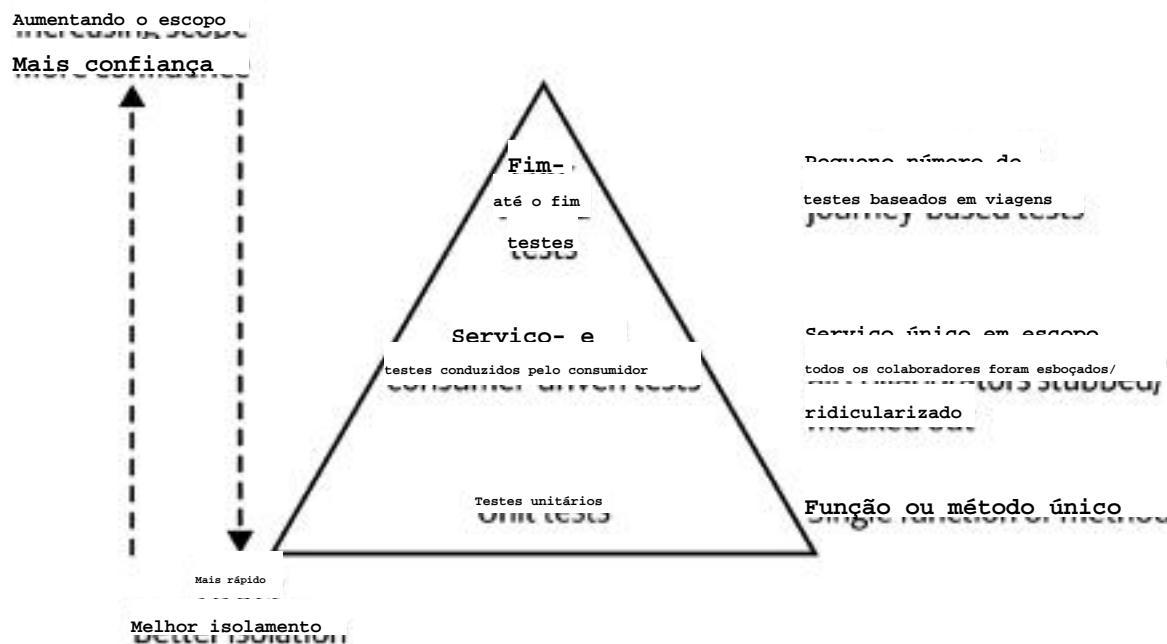
Como exemplo, vamos revisitar nosso cenário anterior. O cliente O microservice tem dois consumidores distintos: o helpdesk e a loja virtual. Ambos os aplicativos consumidores têm expectativas de como o O microsserviço do cliente se comportará. Neste exemplo, você cria um conjunto de testes para cada consumidor: um representando as expectativas do helpdesk em relação ao Microsserviço para clientes e outro conjunto representando as expectativas de a loja virtual tem.

Porque esses CDCs são expectativas de como o cliente é microsserviço deve se comportar, precisamos apenas executar o próprio microsserviço do Cliente, o que significa que temos o mesmo escopo de teste efetivo de nossos testes de serviço. Eles teriam têm características de desempenho semelhantes e exigiriam que executássemos apenas o próprio microsserviço do cliente, com todas as dependências externas separadas fora.

Uma boa prática aqui é ter alguém do produtor e do consumidor. equipes colaboram na criação dos testes, então talvez pessoas da loja virtual e as equipes de helpdesk se unem a pessoas da equipe de atendimento ao cliente.

Indiscutivelmente, os contratos orientados ao consumidor têm tanto a ver com a promoção de linhas claras de comunicação e colaboração, quando necessário, entre microsserviços e as equipes que os consomem. Pode-se argumentar, de fato, que implementar CDCs é apenas tornar mais explícita a comunicação entre as equipes que já devem existir. Na colaboração entre equipes, os CDCs são lembrete explícito da lei de Conway.

Os CDCs estão no mesmo nível da pirâmide de testes dos testes de serviço, embora com um foco muito diferente, conforme mostrado na Figura 9-10. Esses testes estão focados em como um consumidor usará o serviço, e o gatilho, se ele quebrar, é muito diferente quando comparado aos testes de serviço. Se um desses CDCs quebrar durante uma construção do atendimento ao cliente, fica óbvio qual consumidor seria impactado. Neste ponto, você pode corrigir o problema ou então começar a discussão sobre a introdução de uma mudança significativa na forma como discutido em "Como lidar com mudanças entre microsserviços". Então, com CDCs, nós podemos identificar uma alteração significativa antes de nosso software entrar em produção sem precisar usar um teste de ponta a ponta potencialmente caro.



O Pact é uma ferramenta de teste voltada para o consumidor que foi originalmente desenvolvida internamente em realestate.com.au, mas agora é de código aberto. Originalmente apenas para Ruby e focado apenas em protocolos HTTP, o Pact agora suporta vários idiomas e plataformas, como JVM, JavaScript, Python e .NET, e também podem ser usado com interações de mensagens.

Com o Pact, você começa definindo as expectativas do produtor usando um DSL em um dos idiomas suportados. Em seguida, você inicia um servidor Pact local e execute essa expectativa em relação a ela para criar o arquivo de especificação do Pact. O Pacto arquivo é apenas uma especificação formal de JSON; obviamente, você poderia codificar isso manualmente, mas usar o SDK específico da linguagem é muito mais fácil.

Uma propriedade muito boa desse modelo é que o servidor simulado em execução local usado para gerar o arquivo Pact também funciona como um stub local para downstream microsserviços. Ao definir suas expectativas localmente, você está definindo como esse serviço de esboço local deve responder. Isso pode substituir a necessidade de ferramentas como Mountebank (ou suas próprias soluções de estacas ou simulações enroladas à mão).

No lado do produtor, você verifica se essa especificação do consumidor foi atendida usando a especificação JSON Pact para direcionar chamadas em seu microsserviço e verifique as respostas. Para que isso funcione, o produtor precisa ter acesso ao Pacto arquivo. Como discutimos anteriormente em "Mapeando código-fonte e compilações para Microserviços", esperamos que o consumidor e o produtor sejam diferentes constrói. Isso significa que precisamos de alguma forma para esse arquivo JSON, que será gerado para o consumidor, a ser disponibilizado pelo produtor.

Você pode armazenar o arquivo Pact no repositório de artefatos da sua ferramenta de CI/CD, ou então você pode usar o Pact Broker, que permite armazenar várias versões das especificações do seu Pacto. Isso pode permitir que você administre seu negócio voltado para o consumidor testes de contrato contra várias versões diferentes dos consumidores, se você queria testar, digamos, a versão do consumidor em produção e a versão do consumidor que foi criada mais recentemente.

O Pact Broker, na verdade, tem uma série de recursos úteis. Além de atuar como um local onde os contratos podem ser armazenados, você também pode descobrir quando aqueles os contratos foram validados. Além disso, porque o Pact Broker conhece o

relação entre o consumidor e o produtor, é capaz de mostrar quais os microsserviços dependem de quais outros microsserviços.

Outras opções

O Pact não é a única opção para contornar contratos voltados para o consumidor.

O Spring Cloud Contract é um desses exemplos. No entanto, é importante notar que -ao contrário do Pact, que foi projetado desde o início para suportar diferentes tecnologia stacks-Spring Cloud Contract é realmente útil apenas em um ambiente puro Ecossistema JVM.

É sobre conversas

Em Agile, as histórias geralmente são chamadas de espaço reservado para uma conversa.

Os CDCs são exatamente isso. Eles se tornam a codificação de um conjunto de discussões sobre como deve ser a aparência de uma API de serviço e, quando ela falha, ela se torna um ponto de partida para conversar sobre como essa API deve evoluir.

É importante entender que os CDCs exigem boa comunicação e confiança entre o consumidor e o serviço de produção. Se ambas as partes estiverem na mesma equipe (ou são a mesma pessoa!), então isso não deve ser difícil. No entanto, se você está consumindo um serviço fornecido por terceiros, você pode não ter o frequência de comunicação, ou confiança, para fazer os CDCs funcionarem. Nestes em situações em que você pode ter que se contentar com uma integração limitada de escopo maior testes em torno do componente não confiável. Como alternativa, se você estiver criando uma API para milhares de consumidores em potencial, como com uma API de serviço web disponível, você pode ter que desempenhar o papel de consumidor você mesmo (ou talvez trabalhe com um subconjunto de seus consumidores) ao defini-los testes. Quebrar um grande número de consumidores externos é uma péssima ideia, então se qualquer coisa que aumente a importância dos CDCs!

A palavra final

Conforme descrito em detalhes anteriormente neste capítulo, os testes de ponta a ponta têm uma grande número de desvantagens que aumentam significativamente à medida que você adiciona mais movimentos peças em teste. De falar com pessoas que estão implementando microsserviços em grande escala há algum tempo, aprendi que a maioria deles acabou

o tempo elimina totalmente a necessidade de testes de ponta a ponta em favor de outros mecanismos para validar a qualidade de seu software - por exemplo, o uso de esquemas e CDCs explícitos, testes em produção ou talvez alguns dos técnicas de entrega progressiva que discutimos, como liberações canárias.

Você pode ver a execução de testes de ponta a ponta antes da implantação da produção como rodinhas de treinamento. Enquanto você aprende como os CDCs funcionam e melhoram suas técnicas de monitoramento e implantação de produção, essas de ponta a ponta os testes podem formar uma rede de segurança útil, na qual você está negociando o tempo de folga por risco reduzido. Mas à medida que você melhora essas outras áreas e conforme o custo relativo de criar os aumentos de testes de ponta a ponta, você pode começar a reduzir seu confiança em testes de ponta a ponta a ponto de não serem mais necessários. Abandonar os testes de ponta a ponta sem entender completamente o que você perdeu é provavelmente uma má ideia.

Obviamente, você terá uma melhor compreensão do risco de sua própria organização, mais do que eu, mas eu desafiaria você a pensar muito sobre como muitos testes de ponta a ponta que você realmente precisa fazer.

Experiência de desenvolvedor

Um dos desafios significativos que podem surgir à medida que os desenvolvedores encontram a necessidade trabalhar em mais e mais microsserviços é essa a experiência do desenvolvedor podem começar a sofrer, pela simples razão de que estão tentando correr mais e mais microsserviços localmente. Isso geralmente ocorre em situações em que um o desenvolvedor precisa executar um teste de grande escopo que conecte vários não fragmentos microsserviços.

A rapidez com que isso se torna um problema dependerá de vários fatores. Como muitos microsserviços que um desenvolvedor precisa executar localmente, quais pilhas de tecnologia esses microsserviços são escritos em, e o poder da máquina local pode todos desempenham um papel. Algumas pilhas de tecnologia consomem mais recursos em termos de sua pegada inicial, microsserviços baseados em JVM vêm à mente. Sobre o por outro lado, algumas pilhas de tecnologia podem resultar em microsserviços com uma rapidez e uma pegada de recursos mais leve, talvez permitindo que você execute muito mais microsserviços localmente.

Uma abordagem para lidar com esse desafio é, em vez disso, fazer com que os desenvolvedores façam seus trabalhos de desenvolvimento e teste em um ambiente de nuvem. A ideia é que você pode ter muito mais recursos disponíveis para você executar os microserviços que você necessidade. Além do fato de que este modelo exige que você sempre tenha conectividade com seus recursos de nuvem, o outro problema principal é que seu os ciclos de feedback podem sofrer. Se você precisar fazer uma alteração de código localmente e faça o upload da nova versão desse código (ou de um artefato construído localmente) para a nuvem, isso pode adicionar um atraso significativo aos seus ciclos de desenvolvimento e teste, especialmente se você está operando em uma parte do mundo com internet mais restrita conectividade.

O desenvolvimento completo na nuvem é uma possibilidade para resolver o problema de ciclos de feedback; IDEs baseados em nuvem, como o Cloud9, agora de propriedade da AWS, têm mostrado que isso é possível. No entanto, embora algo assim possa ser o futuro para o desenvolvimento, certamente não é o presente para a grande maioria dos nós.

Fundamentalmente, acho que o uso de ambientes em nuvem permite um desenvolvedor para executar mais microserviços para seus ciclos de desenvolvimento e teste é errando o ponto, resultando em mais complexidade do que o necessário, além de custos mais altos. Idealmente, você deve procurar um desenvolvedor que precise executar apenas o microserviços nos quais eles realmente trabalham. Se um desenvolvedor faz parte de uma equipe que possui cinco microserviços, então esse desenvolvedor precisa ser capaz de executá-los microserviços da forma mais eficaz possível e, para um feedback rápido, minha preferência sempre seria que eles funcionassem localmente.

Mas e se os cinco microserviços que sua equipe possui quiserem chamar outros sistemas e microserviços que são de propriedade de outras equipes? Sem eles, o ambiente local de desenvolvimento e teste não funcionará, não é? Aqui novamente, arrancar vem em socorro. Eu deveria ser capaz de defender canhotos locais que imite os microserviços que estão fora do escopo da minha equipe. O único real os microserviços que você deve executar localmente são aqueles em que você está trabalhando. Se você estiver trabalhando em uma organização na qual se espera que trabalhe centenas de microserviços diferentes, bem, então, você tem muito mais problemas para lidar - este é um tópico que exploraremos com mais profundidade em "Strong Versus propriedade coletiva".

Da pré-produção aos testes em produção

Historicamente, a maior parte do foco dos testes tem sido em testar nossos sistemas antes de chegarmos à produção. Com nossos testes, estamos definindo uma série de modelos com os quais esperamos provar se nosso sistema funciona e se comporta como gostaríamos, tanto funcionalmente quanto não funcionalmente. Mas se nossos modelos não são perfeitos, encontraremos problemas quando nossos sistemas forem usados em raiva. Bugs entram em produção, novos modos de falha são descobertos e nossos usuários usam o sistema de maneiras que nunca poderíamos esperar.

Uma reação a isso geralmente é definir mais e mais testes e refinar nossos modelos, para detectar mais problemas com antecedência e reduzir o número de problemas que encontramos com nosso sistema de produção em execução. No entanto, em um determinado momento temos que aceitar que obtemos retornos decrescentes com essa abordagem. Com testando antes da implantação, não podemos reduzir a chance de falha a zero.

A complexidade de um sistema distribuído é tal que pode ser inviável detectar todos os possíveis problemas que possam ocorrer antes de iniciarmos a produção em si.

Genericamente falando, o objetivo de um teste é nos dar feedback sobre se nosso software é de qualidade suficiente ou não. Idealmente, queremos isso feedback o mais rápido possível, e gostaríamos de poder identificar se há um problema com nosso software antes que um usuário final tenha esse problema. Isso é por que muitos testes são feitos antes de lancarmos nosso software.

Limitar-nos a testar somente em um ambiente de pré-produção, no entanto, é nos prejudicando. Estamos reduzindo os locais em que podemos resolver problemas e também estamos eliminando a possibilidade de testar a qualidade do nosso software no local mais importante - onde ele será usado.

Também podemos e devemos procurar aplicar testes em um ambiente de produção. Isso pode ser feito de maneira segura, pode fornecer feedback de maior qualidade do que teste de pré-produção e, como vemos, provavelmente é algo que você já é fazendo, quer você perceba ou não.

Tipos de testes em produção

Há uma longa lista de testes diferentes que podemos realizar na produção, variando do simples ao complexo. Para começar, vamos pensar em algo tão simples quanto uma verificação de ping para garantir que um microsserviço esteja ativo. Simplesmente verificar se uma instância de microsserviço está em execução é um tipo de teste - simplesmente não vemos isso assim, pois é uma atividade normalmente realizada por pessoas de "operações". Mas, fundamentalmente, algo tão simples quanto determinar se um microsserviço está ativo ou não, pode ser visto como um teste que executamos frequentemente em nosso software.

Os testes de fumaça são outro exemplo de testes em produção. Normalmente feito como parte das atividades de implantação, um teste de fumaça garante que o implantado o software está funcionando corretamente. Esses testes de fumaça normalmente serão feitos em o software real em execução antes de ser lançado para os usuários (mais sobre isso, em breve).

Os lançamentos do Canary, que abordamos no Capítulo 8, também são um mecanismo que é indiscutivelmente sobre testes. Lançamos uma nova versão do nosso software para um pequena parte dos usuários para "testar" se ele funciona corretamente. Se isso acontecer, podemos rolar transfira o software para uma parte maior de nossa base de usuários, talvez de forma completa forma automatizada.

Outro exemplo de teste em produção é injetar um comportamento falso no o sistema para garantir que ele funcione conforme o esperado, por exemplo, fazendo um pedido para um cliente falso, ou registrando um novo, (falso) usuário na produção real sistema. Às vezes, esse tipo de teste pode ser rejeitado, porque as pessoas são preocupado com o impacto que isso poderia ter no sistema de produção. Então, se você crie testes como esse, certifique-se de torná-los seguros.

Tornando os testes na produção seguros

Se você decidir fazer testes em produção (e você deveria!), é importante que os testes não causam problemas de produção, nem por introduzirem instabilidade no sistema ou contaminando os dados de produção. Algo tão simples quanto fazer ping em um Uma instância de microsserviço para garantir que esteja ativa provavelmente será uma operação segura -se isso causar instabilidade no sistema, você provavelmente tem problemas muito sérios que

precisa ser abordado, a menos que você tenha criado acidentalmente seu sistema de verificação de saúde um ataque interno de negação de serviço.

Os testes de fumaça geralmente são seguros, pois as operações que eles realizam geralmente são realizadas no software antes de ser lançado. Como exploramos em "Separating Deployment" from Release", separar o conceito de implantação do lançamento pode ser incrivelmente útil. Quando se trata de testes em produção, testes realizados em o software que é implantado na produção, antes de ser lançado, deve ser seguro.

As pessoas tendem a se preocupar mais com a segurança de coisas como a injeção comportamento falso do usuário no sistema. Nós realmente não queremos que o pedido seja enviado ou o pagamento feito. Isso é algo que precisa dos devidos cuidados e atenção, e apesar dos desafios, esse tipo de teste pode ser enorme benéfico. Voltaremos a isso em "Monitoramento semântico".

Tempo médio de reparo acima do tempo médio entre Falhas?

Então, analisando técnicas como implantação azul-esverdeada ou liberação canária, encontramos uma maneira de testar mais perto (ou mesmo na) produção e também construímos ferramentas para nos ajudar a gerenciar uma falha, caso ela ocorra. Usar essas abordagens é reconhecimento tácito de que não podemos identificar e detectar todos os problemas antes de nós na verdade, lance nosso software.

Às vezes, fazendo o mesmo esforço para melhorar a resolução de problemas quando ocorrem pode ser significativamente mais benéfico do que adicionar mais testes funcionais automatizados. No mundo das operações na web, isso geralmente é referido para como compensação entre otimizar o tempo médio entre falhas (MTBF), e otimização do tempo médio de reparo (MTTR).

Técnicas para reduzir o tempo de recuperação podem ser tão simples quanto muito rápidas reversões aliadas a um bom monitoramento (que discutiremos em Capítulo 10). Se conseguirmos identificar um problema na produção mais cedo e reverter cedo, reduzimos o impacto em nossos clientes.

Para diferentes organizações, a compensação entre MTBF e MTTR será variável, e muito disso está na compreensão do verdadeiro impacto do fracasso em um ambiente de produção. No entanto, a maioria das organizações que eu vejo gastando o tempo de criação de suítes de testes funcionais geralmente gasta pouco ou nenhum esforço para melhorar o monitoramento ou recuperação de falhas. Então, embora eles possam reduzir o número dos defeitos que ocorrem em primeiro lugar, eles não conseguem eliminar todos eles, e não estão preparados para lidar com eles se eles aparecerem na produção.

Existem outras compensações além das entre o MTBF e o MTTR. Por exemplo, se você está tentando descobrir se alguém realmente usará seu software, pode fazer muito mais sentido divulgar algo agora, para provar a ideia ou o modelo de negócios antes de criar um software robusto. Em um ambiente onde este é o caso, o teste pode ser um exagero, pois o impacto de não saber se seu software funciona é muito maior do que ter um defeito na produção. Uma ideia funciona muito mais do que ter um defeito na produção. Nestes tipos de situações em que pode ser bastante sensato evitar testes antes da produção ao todo.

Teste multifuncional

A maior parte deste capítulo se concentrou em testar partes específicas do comportamento funcional e como isso difere quando você está testando um baseado em microserviços sistema. No entanto, há outra categoria de teste que é importante para discutir. Requisitos não funcionais é um termo genérico usado para descrever aquelas características que seu sistema exibe que não podem ser simplesmente implementadas como um recurso normal. Eles incluem aspectos como a latência aceitável de uma página da web, o número de usuários que um sistema deve suportar, quanto acessível sua interface do usuário deve ser para pessoas com deficiências, ou quanto segura sua base de dados os dados do cliente devem ser.

O termo não funcional nunca me agradou. Algumas das coisas que obtêm abrangidos por este termo parecem de natureza muito funcional! Um colega anterior de minha, Sarah Taraporewala, cunhou a frase requisitos multifuncionais (CFR) em vez disso, o que eu prefiro muito. Isso fala mais sobre o fato de que esses comportamentos do sistema realmente só surgem como resultado de muitos cortes transversais trabalhar.

Muitos, se não a maioria dos CFRs, só podem ser atendidos na produção. Dito isso, nós pode definir estratégias de teste para nos ajudar a ver se estamos pelo menos avançando para atingindo essas metas. Esses tipos de testes se enquadram no Teste de Propriedades quadrante. Um ótimo exemplo desse tipo de teste é o teste de desempenho, que discutiremos com mais profundidade em breve.

Talvez você queira rastrear alguns CFRs em um nível de microsserviço individual. Para por exemplo, você pode decidir que a durabilidade do serviço que você exige do seu o serviço de pagamento é significativamente maior, mas você está feliz com mais tempo de inatividade para seu serviço de recomendação de música, sabendo que seu núcleo os negócios podem sobreviver se você não conseguir recomendar artistas semelhantes a Metallica por cerca de 10 minutos. Essas compensações acabarão tendo uma grande impacto na forma como você projeta e evolui seu sistema e, mais uma vez, na precisão A natureza granulada de um sistema baseado em microsserviços oferece muito mais chances para fazer essas compensações. Ao analisar os CFRs de um determinado microsserviço ou a equipe pode ter que assumir a responsabilidade, é comum que ela apareça como parte dos objetivos de nível de serviço (SLOs) de uma equipe, um tópico que exploramos mais detalhadamente em "Estamos bem?".

Os testes em torno de CFRs também devem seguir a pirâmide. Alguns testes terão que sejam de ponta a ponta, como testes de carga, mas outros não. Por exemplo, uma vez que você tenha encontrado um gargalo de desempenho em um teste de carga de ponta a ponta, escreva um menor teste de escopo para ajudá-lo a detectar o problema no futuro. Outros CFRs se encaixam mais rápido testa com bastante facilidade. Lembro-me de trabalhar em um projeto para o qual havíamos insistido sobre como garantir que nossa marcação HTML estivesse usando recursos de acessibilidade adequados para ajudar pessoas com deficiência a usar nosso site. Verificando o gerado marcação para garantir que os controles apropriados estivessem lá pudesse ser feitos muito rapidamente, sem a necessidade de viagens de ida e volta de networking.

Com muita frequência, as considerações sobre os CFRs chegam tarde demais. Eu sugiro fortemente examinando seus CFRs o mais cedo possível e revisando-os regularmente.

Testes de desempenho

Vale a pena mencionar explicitamente os testes de desempenho como forma de garantir que alguns de nossos requisitos multifuncionais podem ser atendidos. Ao se decompor

sistemas em microserviços menores, aumentamos o número de chamadas que serão feito além dos limites da rede. Onde anteriormente uma operação poderia envolver uma chamada de banco de dados, agora pode envolver três ou quatro chamadas cruzando os limites da rede para outros serviços, com um número correspondente de chamadas de banco de dados. Tudo isso pode diminuir a velocidade com que nossos sistemas operar. Rastrear as fontes de latência é especialmente importante. Quando você ter uma cadeia de chamadas de várias chamadas síncronas, se alguma parte da cadeia começar agindo lentamente, tudo é afetado, potencialmente levando a uma significativa impacto. Isso faz com que você tenha alguma maneira de testar o desempenho de seus aplicativos ainda mais importante do que poderia ser com um sistema mais monolítico. Frequentemente a razão pela qual esse tipo de teste é adiado é porque inicialmente não há o suficiente do sistema para testar. Eu entendo esse problema, mas com muita frequência isso leva a chutar a lata pela estrada, com testes de desempenho geralmente apenas sendo feito pouco antes de ir ao ar pela primeira vez, se for o caso! Não caia em essa armadilha.

Assim como nos testes funcionais, você pode querer uma mistura. Você pode decidir que quer testes de desempenho que isolam serviços individuais, mas começam com testes que verifique as principais viagens em seu sistema. Você pode ser capaz de fazer de ponta a ponta testes de viagem e simplesmente execute-os em volume.

Para gerar resultados que valham a pena, muitas vezes você precisará executar determinados cenários com aumento gradual do número de clientes simulados. Isso permite que você veja como a latência das chamadas varia com o aumento da carga. Isso significa que os testes de desempenho podem demorar um pouco para serem executados. Além disso, você vai querer o sistema para combinar a produção da forma mais próxima possível, para garantir que os resultados que você veja será indicativo do desempenho que você pode esperar da produção sistemas. Isso pode significar que você precisará adquirir um produto mais parecido com o de produção volume de dados e pode precisar de mais máquinas para corresponder à infraestrutura-tarefas que podem ser desafiadoras. Mesmo que você tenha dificuldade em fazer o desempenho Em um ambiente verdadeiramente semelhante ao de produção, os testes ainda podem ter valor no rastreamento reduza os gargalos. Esteja ciente de que você pode obter falsos negativos - ou até pior, falsos positivos.

Devido ao tempo necessário para executar os testes de desempenho, nem sempre é possível executá-los em cada check-in. É uma prática comum executar um subconjunto todos os dias,

e um conjunto maior a cada semana. Seja qual for a abordagem escolhida, certifique-se de correr faz testes com a maior regularidade possível. Quanto mais tempo você ficar sem correr, desempenho testes, mais difícil pode ser rastrear o culpado. Problemas de desempenho são especialmente difíceis de resolver, portanto, se você puder reduzir o número de compromissos que você precisa analisar para ver um problema recém-introduzido, sua vida será muito mais fácil.

E não deixe de ver também os resultados! Fiquei muito surpreso com o número de equipes que encontrei que trabalharam muito implementando testes e executando-os, mas nunca realmente verifique os números. Muitas vezes, isso ocorre porque as pessoas não sabem o que é um resultado "bom". Você realmente preciso ter alvos. Quando você fornece um microsserviço para ser usado como parte de uma arquitetura mais ampla, é comum ter expectativas específicas de que você se compromete a entregar os SLOs que mencionei anteriormente. Se, como parte disso, você comprometa-se a oferecer um certo nível de desempenho, então faz sentido para qualquer teste automatizado para fornecer feedback sobre se é provável que você se encontre ou não (e espero que exceda) essa meta.

Em vez de metas de desempenho específicas, os testes de desempenho automatizados ainda podem seja muito útil para ajudar você a ver como está o desempenho do seu microsserviço varia conforme você faz alterações. Pode ser uma rede de segurança para pegá-lo se você fizer uma mudança que causa uma degradação drástica no desempenho. Então, uma alternativa para uma meta específica pode ser falhar no teste se o delta no desempenho for de um construir para o próximo varia muito.

O teste de desempenho precisa ser feito em conjunto com a compreensão do real desempenho do sistema (que discutiremos mais no Capítulo 10) e, idealmente, você usaria as mesmas ferramentas em seu ambiente de teste de desempenho para visualizando o comportamento do sistema como aqueles que você usa na produção. Essa abordagem pode tornar muito mais fácil comparar igual com semelhante.

Testes de robustez

Uma arquitetura de microsserviços geralmente é tão confiável quanto seu elo mais fraco, e como resultado, é comum que nossos microsservicos criem mecanismos para permitem que eles melhorem sua robustez para melhorar o sistema.

confiabilidade. Exploraremos mais esse tópico em "Padrões de estabilidade", mas exemplos incluem a execução de várias instâncias de um microsserviço por trás de uma carga balanceador para tolerar a falha de uma instância, ou usar disjuntores para lidar programaticamente com situações em que microsserviços downstream não pode ser contatado.

Em tais situações, pode ser útil ter testes que permitam recriar certas falhas para garantir que seu microsserviço continue operando como um todo. Por sua natureza, esses testes podem ser um pouco mais difíceis de implementar. Para por exemplo, talvez seja necessário criar um tempo limite de rede artificial entre um microsserviço em teste e um esboço externo. Dito isso, eles podem ser vale a pena, especialmente se você estiver criando uma funcionalidade compartilhada que será usado em vários microsserviços, por exemplo, usando um serviço padrão de implementação de malha para lidar com interrupções de circuito.

Resumo

Juntando tudo isso, o que descrevi neste capítulo é holístico abordagem de teste que, esperançosamente, fornece algumas orientações gerais sobre como prossiga ao testar seus próprios sistemas. Para reiterar o básico:

- Optimize para obter feedback rápido e separe os tipos de testes de acordo.
- Evite a necessidade de testes de ponta a ponta que abranjam mais de uma equipe em vez disso, considere usar contratos voltados para o consumidor.
- Use contratos orientados pelo consumidor para fornecer pontos de foco para conversas entre equipes.
- Tente entender a desvantagem entre se esforçar mais em testando e detectando problemas com mais rapidez na produção (otimizando para MTBF versus otimização para MTTR).
- Experimente os testes em produção!

Se você estiver interessado em ler mais sobre testes, eu recomendo o Agile Testes feitos por Lisa Crispin e Janet Gregory (Addison-Wesley), que entre

outras coisas abrangem o uso do quadrante de teste com mais detalhes. Para um mergulhe mais profundamente na pirâmide de teste, junto com alguns exemplos de código e muito mais referências de ferramentas, eu também recomendo "The Practical Test Pyramid" de Ham Voz. 9

Este capítulo se concentrou principalmente em garantir que nosso código funcione antes de chegar à produção, mas também começamos a testar nosso aplicativo uma vez que ele atinge a produção. Isso é algo que precisamos explorar muito mais. Detalhe. Acontece que os microserviços causam uma infinidade de desafios para entender como nosso software está se comportando na produção - um tópico que abordaremos mais tarde com mais profundidade a seguir.

1 Lisa Crispin e Janet Gregory. *Teste ágil: um guia prático para testadores e equipes ágeis* (Upper Saddle River, NJ: Addison-Wesley, 2008).

2 Mike Cohn, *Successing with Agile* (Upper Saddle River, NJ: Addison-Wesley, 2009).

3 Steve Freeman e Nat Pryce. *Desenvolvimento de software orientado a objetos, guiado por testes (superior)* (Saddle River, NJ: Addison-Wesley, 2009).

4 Jason D. Valentino, "Transferindo um dos maiores aplicativos voltados para o cliente da Capital One para a AWS" Capital One Tech, 24 de maio de 2017, <https://oreil.ly/5UM5W>.

5 Diane Vaughan, *A decisão de lançamento do Challenger: tecnologia, cultura e Deviance na NASA* (Chicago: University of Chicago Press, 1996).

6 Martin Fowler, "Erradicando o não determinismo em testes", martinfowler.com, 14 de abril de 2011, <https://oreil.ly/7Ve7e>.

7 Emily Bache, "Teste automatizado de ponta a ponta em uma arquitetura de microserviços - Emily Bache", Conferências da NDC, 5 de julho de 2017, vídeo do YouTube, 56:48, NDC Oslo 2017, <https://oreil.ly/QX3EK>.

8 Nicole Forsgren, Jez Humble e Gene Kim, *Accelerate: a ciência da construção e Dimensionando organizações de tecnologia de alto desempenho* (Portland, OR: IT Revolution, 2018).

9 Ham Vocke, "The Practical Test Pyramid", martinfowler.com, 26 de fevereiro de 2018, <https://oreil.ly/J7lc6>.

Capítulo 10. Do monitoramento ao Observabilidade

Como mostrei até agora, espero que, dividindo nosso sistema em um sistema menor, bom... microsserviços granulados resultam em vários benefícios. Também, como nós também abordado com alguma profundidade, adiciona fontes significativas de nova complexidade. Em não situação é esse aumento de complexidade mais evidente do que quando se trata de compreender o comportamento de nossos sistemas em um ambiente de produção. Muito logo no início, você descobrirá que as ferramentas e técnicas que funcionaram bem para aplicativos monolíticos de processo único relativamente mais simples não funcionam tão bem para sua arquitetura de microsserviços.

Neste capítulo, analisaremos os desafios associados ao monitoramento de nossos arquitetura de microsserviços, e mostrarei que, embora novas ferramentas possam ajudar, fundamentalmente, você pode precisar mudar toda a sua mentalidade quando se trata de descobrindo o que diabos está acontecendo na produção. Também falaremos sobre o maior foco no conceito de observabilidade - entendendo como fazer é possível fazer perguntas ao nosso sistema para que possamos descobrir o que está acontecendo errado.

DOR DE PRODUÇÃO

Você não apreciará verdadeiramente a dor, o sofrimento e a angústia potenciais causados por um arquitetura de microsserviços até que você a execute em produção e atenda ao tráfego real.

Disrupção, pânico e confusão

Imagine a cena: é uma tarde tranquila de sexta-feira e a equipe está olhando Estou ansioso para sair cedo para o pub como forma de começar um fim de semana fora do trabalho. Então, de repente, os e-mails chegam. O site está se comportando mal!

O Twitter está em chamas com as falhas da sua empresa, seu chefe está mastigando suas Desapareça e as perspectivas de um fim de semana tranquilo desaparecerão.

Poucas coisas resumem o problema tão bem quanto o seguinte tweet:

Substituímos nosso monólito por microserviços para que cada interrupção pudesse seja mais como um mistério de assassinato.

Rastrear o que deu errado e o que o causou é nossa primeira porta de telefonar. Mas isso fica difícil se tivermos uma longa lista de suspeitos.

No mundo da aplicação monolítica de processo único, temos pelo menos um lugar muito óbvio para começar nossas investigações. Site lento? É o monólito. Site com erros estranhos? É o monólito. CPU em 100% Monólito. Um cheiro de queimado? Bem, você entendeu. Ter um único ponto de a falha torna a investigação de falhas um pouco mais simples!

Agora vamos pensar em nosso próprio sistema baseado em microserviços. As capacidades oferecemos aos nossos usuários vários microserviços, alguns dos quais comunique-se com ainda mais microserviços para realizar suas tarefas. Ali Há muitas vantagens em tal abordagem (o que é bom, caso contrário, livro seria uma perda de tempo), mas no mundo do monitoramento, temos um problema mais complexo em nossas mãos.

Agora temos vários servidores para monitorar, vários arquivos de log para filtrar, e vários locais onde a latência da rede pode causar problemas. Nosso a área superficial da falha aumentou, assim como as coisas que precisam ser investigado. Então, como abordamos isso? Precisamos entender o que caso contrário, pode ser uma bagunça caótica e emaranhada - a última coisa que qualquer um de nós quer lide com isso em uma sexta-feira à tarde (ou a qualquer hora, chegue a esse ponto!).

Em primeiro lugar, precisamos monitorar as pequenas coisas e fornecer agregação para permitir nós para ver o panorama geral. Então, precisamos ter certeza de que temos ferramentas disponível para nós para dividir e dividir esses dados como parte de nossa investigação. Finalmente, precisamos ficar mais inteligentes sobre a forma como pensamos sobre a saúde do sistema, por adotando conceitos como testes em produção. Vamos discutir cada um deles necessidades deste capítulo. Vamos começar.

Microsserviço único, servidor único

A Figura 10-1 apresenta uma configuração muito simples: um host que está executando um instância de um microsserviço. Agora precisamos monitorá-lo para saber quando algo dá errado, para que possamos consertá-lo. Então, o que devemos procurar?

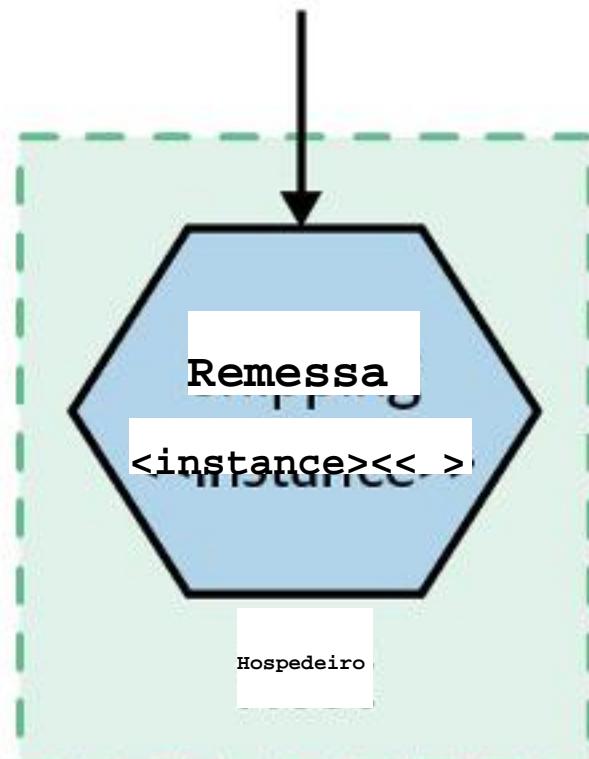


Figura 10-1. Uma única instância de microsserviço em um único host.

Primeiro, queremos obter informações do próprio anfitrião. CPU, memória – tudo dessas coisas podem ser úteis. Em seguida, queremos ter acesso aos registros da própria instância de microsserviço. Se um usuário relatar um erro, devemos ser capazes para ver o erro nesses registros, espero que nos dê uma maneira de descobrir o que deu errado. Neste ponto, com nosso único anfitrião, provavelmente podemos sobreviver com basta fazer login localmente no host e usar ferramentas de linha de comando para ver o registro.

Finalmente, talvez queiramos monitorar o próprio aplicativo, observando-o por conta própria a partir de do lado de fora. No mínimo, monitorar o tempo de resposta do microsserviço é uma boa ideia. Se você tiver um servidor web na frente do seu instância de microsserviço, talvez você possa simplesmente ver os registros da web servidor. Ou talvez você possa ficar um pouco mais avançado, usando algo como um

endpoint de verificação de integridade para ver se o microserviço está ativo e "saudável" (vamos explore o que isso significa mais tarde).

O tempo passa, as cargas aumentam e precisamos escalar

Microsserviço único, vários servidores

Agora temos várias cópias do serviço em execução em hosts separados, como mostrado na Figura 10-2, com solicitações para as diferentes instâncias distribuídas via um平衡ador de carga. As coisas começam a ficar um pouco mais complicadas agora. Nós ainda queremos monitorar todas as mesmas coisas de antes, mas precisamos fazer isso de forma que podemos isolar o problema. Quando a CPU está alta, é um problema? Vendo em todos os hosts, o que indicaria um problema com o serviço em si? Ou é isolado em um único hospedeiro, o que implica que o próprio hospedeiro tem o problema-talvez um processo operacional desonesto?

Neste momento, ainda queremos rastrear as métricas em nível de host, e talvez talvez até mesmo alerta-los quando ultrapassam algum tipo de limite. Mas agora queremos para ver o que eles são em todos os anfitriões, bem como em anfitriões individuais. Em outras palavras, queremos agregá-las e ainda assim poder detalhá-las. Então nós preciso de algo que possa coletar todas essas métricas de todos os anfitriões e permita-nos cortá-los e cortá-los em cubos.

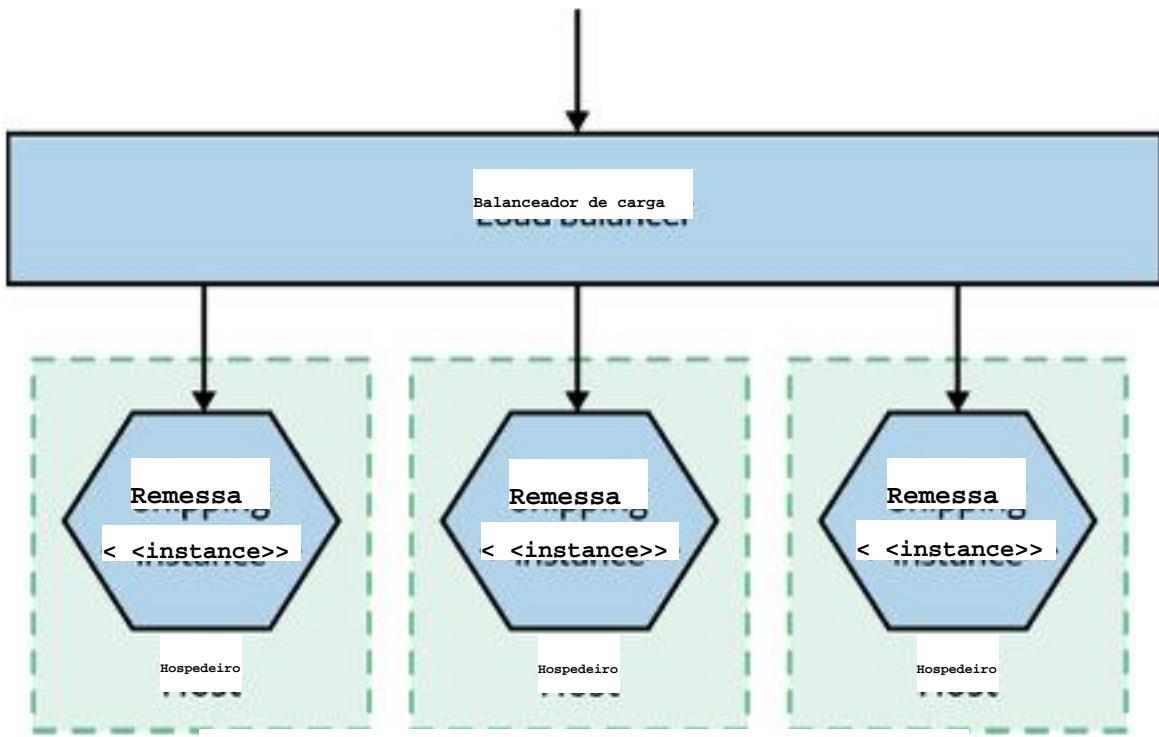


Figura 10-2. Um único serviço distribuído em vários hosts

Então temos nossos registros. Com nosso serviço em execução em mais de um servidor, provavelmente nos cansaremos de entrar em cada caixa para vê-la. Com apenas alguns hosts, no entanto, podemos usar ferramentas como multiplexadores SSH, que nos permitem executar os mesmos comandos em vários hosts. Com a ajuda de um grande monitor, e executando grep "Error" em nosso log de microsserviços, podemos encontrar nosso culpado. EU Quer dizer, isso não é ótimo, mas pode ser bom o suficiente por um tempo. Vai ficar velho. Mas muito rápido.

Para tarefas como rastreamento do tempo de resposta, podemos capturar os tempos de resposta no balanceador de carga para chamadas downstream para microsserviços. No entanto, nós também tem que considerar o que acontece se o balanceador de carga se tornar o gargalo em nossos tempos de resposta de captura de sistema, tanto no balanceador de carga e nos próprios microsserviços podem ser necessários. Neste ponto, nós provavelmente também se preocupa muito mais com a aparência de um serviço saudável, pois configuraremos nosso balanceador de carga para remover nós não saudáveis do nosso aplicação. Espero que, quando chegarmos aqui, tenhamos pelo menos alguma ideia de o que parece ser saudável.

Vários serviços, vários servidores

Na Figura 10-3, as coisas ficam muito mais interessantes. Vários serviços são colaborando para fornecer recursos aos nossos usuários, e esses serviços são executado em vários hosts, sejam eles físicos ou virtuais. Como você encontra o erro que você está procurando em milhares de linhas de registros em vários hosts? Como você determina se um servidor está se comportando mal ou se é um problema sistêmico? E como rastrear um erro encontrado no fundo de uma cadeia de chamadas entre vários anfitriões e descubra o que causou isso?

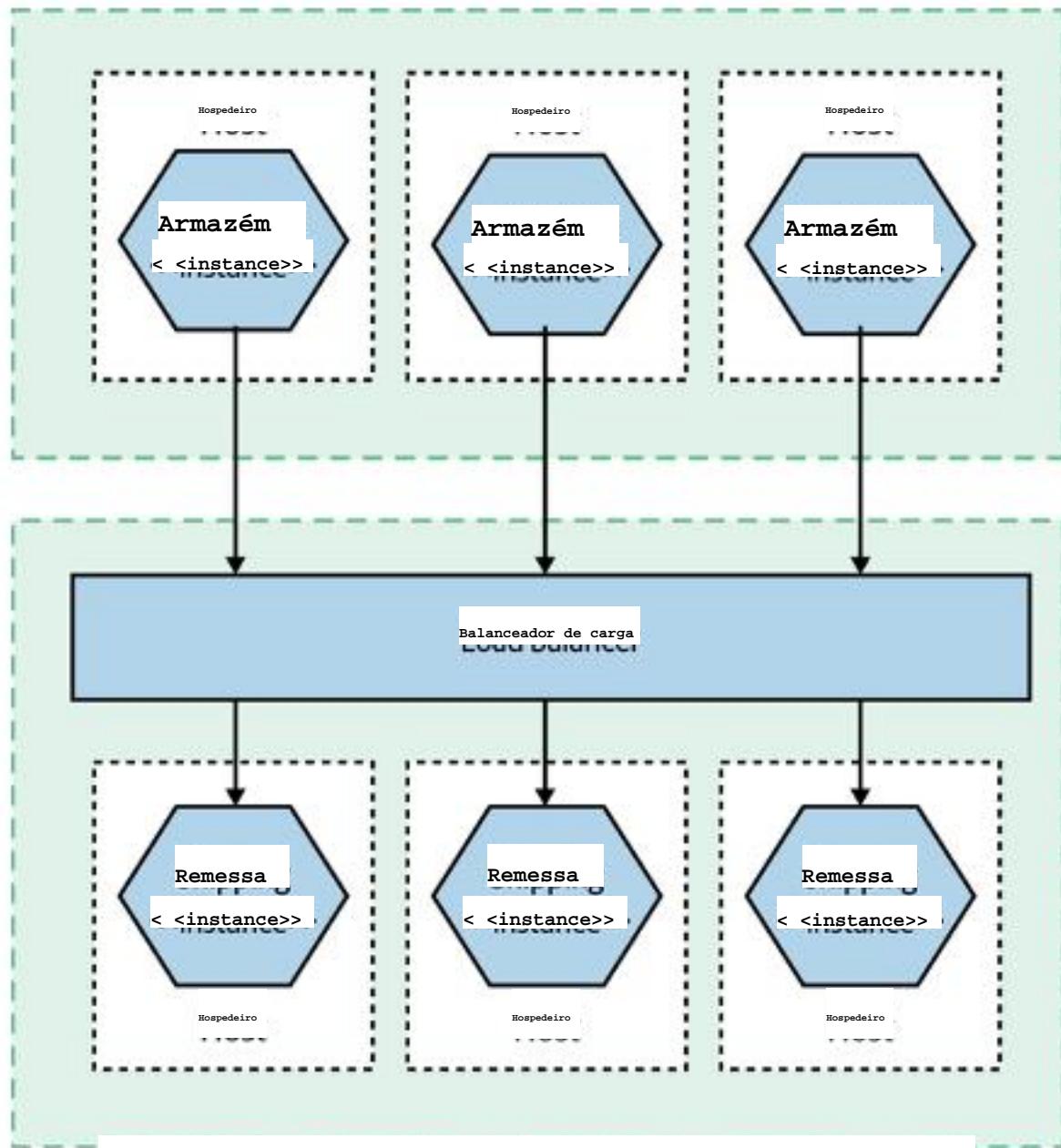


Figura 10-3. Vários serviços de colaboração distribuídos em vários hosts

A agregação de informações, métricas e registros desempenha um papel vital na criação isso acontece. Mas essa não é a única coisa que precisamos considerar. Nós precisamos descubra como filtrar esse enorme fluxo de dados e tenta entender tudo isso. Acima de tudo, trata-se principalmente de uma mudança de mentalidade, de uma paisagem bastante estática. de monitoramento para o mundo mais ativo de observabilidade e testes em produção.

Observabilidade versus monitoramento

Vamos começar a investigar como começar a resolver alguns dos problemas acabamos de descrever, mas antes disso, acho importante que exploraremos um termo que ganhou muita popularidade desde que escrevi a primeira edição deste observabilidade do livro.

Como costuma acontecer, o conceito de observabilidade existe há décadas, mas só recentemente chegou ao desenvolvimento de software. O observabilidade de um sistema é a medida em que você pode entender o estado interno do sistema a partir de saídas externas. Normalmente, requer mais compreensão holística do seu software - vendo-o mais como um sistema do que como do que um conjunto de entidades diferentes.

Na prática, quanto mais observável for um sistema, mais fácil será para nós entender qual é o problema quando algo dá errado. Nosso a compreensão das saídas externas nos ajuda a rastrear o problema subjacente mais rapidamente. O desafio é que, muitas vezes, precisaremos criá-las saídas externas e use diferentes tipos de ferramentas para entender as saídas.

O monitoramento, por outro lado, é algo que fazemos. Nós monitoramos o sistema.

Nós olhamos para isso. As coisas começam a dar errado se você se concentrar apenas no

monitorando a atividade sem pensar no que você espera que atividade a ser alcançada.

Abordagens mais tradicionais de monitoramento fariam você pensar com antecedência sobre o que pode dar errado e definir mecanismos de alerta para informar quando essas coisas aconteceram. Mas à medida que o sistema se torna cada vez mais distribuído, você encontrará problemas que nunca teriam ocorrido com você. Com um altamente sistema observável, você terá uma coleção de saídas externas que você pode interrogar de maneiras diferentes - o resultado concreto de ter um observável sistema é que você pode fazer perguntas ao seu sistema de produção que você nunca Eu teria pensado em perguntar antes.

Portanto, podemos ver o monitoramento como uma atividade - algo com o qual fazemos observabilidade sendo uma propriedade do sistema.

Os pilares da observabilidade? Não tão rápido

Algumas pessoas tentaram resumir a ideia de observabilidade a um alguns conceitos básicos. Alguns se concentraram nos "três pilares" da observabilidade na forma de métricas, registro e rastreamento distribuído. Evento New Relic cunhou o termo MELT (métricas, eventos, registros e rastreamentos), que na verdade não tem entendido, mas pelo menos New Relic está tentando. Embora este modelo simples inicialmente me atraiu muito (e eu adoro uma sigla!), ao longo do tempo Eu realmente me afastei desse pensamento por ser excessivamente redutor, mas também potencialmente não entendendo.

Em primeiro lugar, reduzir uma propriedade de um sistema aos detalhes de implementação dessa forma parece retrógrado para mim. A observabilidade é uma propriedade e há muitas maneiras Talvez eu consiga alcançar essa propriedade. Concentrando-se demais em coisas específicas os detalhes da implementação correm o risco de se concentrar na atividade versus no resultado. É análogo ao mundo atual da TI, onde centenas, senão milhares, de as organizações se apaixonaram pela criação de sistemas baseados em microserviços sem realmente entender o que eles estão tentando alcançar!

Em segundo lugar. sempre há linhas concretas entre esses conceitos? Eu argumentaria que muitos deles se sobrepõem. Posso colocar métricas em um arquivo de registro, se quiser. Da mesma forma, eu pode construir um traço distribuído a partir de uma série de linhas de registro, algo que é comumente feito.

NOTE NOTA

A observabilidade é a medida em que você pode entender o que o sistema está fazendo com base em entradas externas. Registros, eventos e métricas podem ajudar você a tornar as coisas observáveis, mas seja certifique-se de se concentrar em tornar o sistema comprehensível, em vez de usar muitas ferramentas.

Cinicamente, eu poderia sugerir que a promocão dessa narrativa simplista é uma maneira de vender ferramentas para você. Você precisa de uma ferramenta para métricas, uma ferramenta diferente para registros, e mais uma ferramenta para traços! E você precisa enviar todas essas informações de forma diferente! É muito mais fácil vender recursos por meio de um exercício de marcar as caixas ao tentar comercializar um produto, em vez de falar sobre resultados. Como eu

nica, mas estamos em 2021 e estou tentando
seja um pouco mais positivo em meu pensamento. 3
~~Be a bit more positive in my thinking.~~

É discutível todos esses três (ou quatro!) conceitos são, na verdade, apenas específicos
exemplos de um conceito mais genérico. Fundamentalmente, podemos ver qualquer peça de
informações que podemos obter do nosso sistema - qualquer uma dessas saídas externas -
genericamente como um evento. Um determinado evento pode conter uma quantidade pequena ou grande
de informações. Ele pode conter uma taxa de CPU, informações sobre uma falha
pagamento, o fato de um cliente ter feito login ou qualquer outra coisa. Nós
pode projetar a partir desse fluxo de eventos um traço (supondo que possamos correlacioná-los
eventos), um índice pesquisável ou uma agregação de números. Embora em
Atualmente, optamos por coletar essas informações de maneiras diferentes, usando diferentes
ferramentas e protocolos diferentes, nossos conjuntos de ferramentas atuais não devem limitar nossos
pensando em termos da melhor forma de obter as informações de que precisamos.

Quando se trata de tornar seu sistema observável, pense nas saídas que você
necessidade do seu sistema em termos de eventos que você pode coletar e interrogar.
Talvez seja necessário usar ferramentas diferentes para expor diferentes tipos de eventos
agora, mas isso pode não ser o caso no futuro.

Blocos de construção para observabilidade

Então, o que precisamos? Precisamos saber se os usuários do nosso software são
feliz. Se houver um problema, queremos saber sobre ele, idealmente antes de nossos
os usuários encontram um problema sozinhos. Quando ocorre um problema, precisamos trabalhar
Descubra o que podemos fazer para que o sistema volte a funcionar, e uma vez que a poeira
decidiu que queremos ter informações suficientes em mãos para descobrir qual é o
O inferno deu errado e o que podemos fazer para evitar que o problema aconteça novamente.

No restante deste capítulo, veremos como fazer tudo isso acontecer.

Abordaremos vários blocos de construção que podem ajudar a melhorar o
observabilidade da arquitetura do seu sistema.

Agregação de registros

Coletando informações em vários microsserviços, um edifício vital

bloqueio de qualquer solução de monitoramento ou observabilidade

Agregação de métricas

Capturando números brutos de nossos microserviços e infraestrutura para ajudar a detectar problemas, impulsionar o planejamento da capacidade e talvez até mesmo escalar nosso aplicativo.

Rastreamento distribuído

Rastreando um fluxo de chamadas em vários limites de microserviços até descobrir o que deu errado e obtenha informações precisas de latência.

Você está bem?

Analisando orçamentos de erro, SLAs, SLOs e assim por diante para ver como eles podem ser usado como parte de garantir que nosso microserviço atenda às necessidades de seus consumidores.

Alertando

Sobre o que você deve alertar? Qual é a aparência de um bom alerta?

Monitoramento semântico

Pensando de forma diferente sobre a saúde de nossos sistemas e sobre o que deve nos acordar às 3 da manhã.

Testes em produção

Um resumo de vários testes em técnicas de produção.

Vamos começar com talvez a coisa mais simples de começar a trabalhar, mas algo que se pagará muitas vezes: agregação de registros.

Agregação de registros

Com muitos servidores e instâncias de microserviços, mesmo que modestos arquitetura de microserviços, login em máquinas ou multiplexação SSH para

recuperar registros realmente não é suficiente. Em vez disso, estamos procurando usar produtos especializados para pegar nossos registros e disponibilizá-los centralmente.

Os registros rapidamente se tornarão um dos mecanismos mais vitais para ajudá-lo a entender o que está acontecendo em seu sistema de produção. Com mais simples arquiteturas de implantação, nossos arquivos de log, o que colocamos neles e como tratá-los geralmente é uma reflexão tardia. Com um sistema cada vez mais distribuído, eles se tornarão uma ferramenta vital, não apenas ajudando você a diagnosticar o que aconteceu errado quando você descobre que tem um problema, mas também está dizendo que era um problema que precisava de sua atenção em primeiro lugar.

Como discutiremos em breve, há uma variedade de ferramentas neste espaço, mas todas elas operam em grande parte da mesma maneira, conforme descrito na Figura 10-4. Processos (como nossas instâncias de microsserviço) fazem login em seu sistema de arquivos local. Um daemon local O processo coleta periodicamente e encaminha esse registro para algum tipo de loja que pode ser consultado pelos operadores. Um dos aspectos interessantes desses sistemas é que sua arquitetura de microsserviços pode, em grande parte, não ter conhecimento deles. Você não precisa alterar seu código para usar algum tipo de API especial; basta fazer login em um sistema de arquivos local. Você precisa entender os modos de falha em torno desse registro de processo de envio, no entanto, especialmente se você quiser entender as situações nos registros podem se perder.

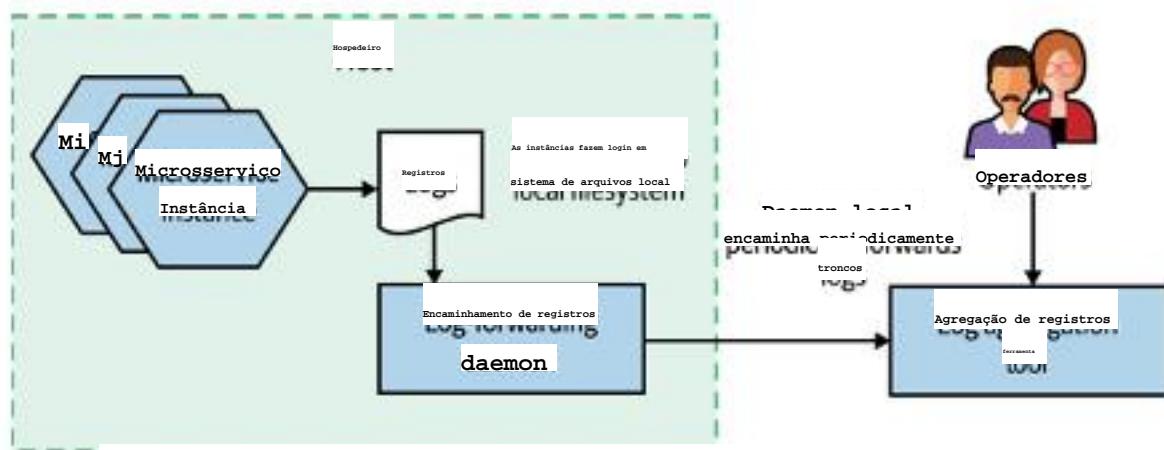


Figura 10-4. Uma visão geral de como os registros são coletados como parte da agregação de registros.

Agora, espero que você tenha percebido que eu tento evitar ser dogmático sobre as coisas. Em vez de apenas dizer que você deve fazer X ou Y, tentei dar contexto e orientar e explicar as nuances de certas decisões, ou seja, tentei

fornecem as ferramentas para fazer a escolha certa para seu contexto. Mas sobre o assunto sobre agregação de registros, chegarei o mais perto possível de dar um conselho único para todos: você deve ver a implementação de uma ferramenta de agregação de registros como um pré-requisito para implementando uma arquitetura de microsserviços.

Minhas razões para esse ponto de vista são duplas. Em primeiro lugar, a agregação de registros é incrivelmente útil. Para aqueles que tratam seus arquivos de log como um dumping motivo para desinformação, isso será uma surpresa. Mas confie em mim, quando feita corretamente, a agregação de registros pode ser incrivelmente valiosa, especialmente quando usada com outro conceito que abordaremos em breve, IDs de correlação.

Em segundo lugar, implementar uma agregação de log, quando comparada com a outra fontes de dor e sofrimento que uma arquitetura de microsserviços pode trazer, não são tão difícil. Se sua organização não conseguir implementar com sucesso um solução simples de agregação de registros, provavelmente encontrará os outros aspectos de um a arquitetura de microsserviços é demais para ser manuseada. Então, considere usar implementação de tal solução como forma de testar a de sua organização prontidão para o resto do horror que se seguirá.

ANTES DE QUALQUER OUTRA COISA

Antes de fazer qualquer outra coisa para criar sua arquitetura de microsserviços, obtenha um registro ferramenta de agregação em funcionamento. Considere isso um pré-requisito para criar um microsserviço arquitetura. Você vai me agradecer mais tarde.

Agora, também é verdade dizer que a agregação de logs tem suas limitações e mais momento em que você pode muito bem querer procurar ferramentas mais sofisticadas para aumentar ou até mesmo substitua parte do que a agregação de registros fornece. Tudo o que foi dito, é ainda é um excelente lugar para começar.

Formato comum

Se você for agrregar seus registros, você vai querer ser capaz de executar consultas entre eles para extrair informações úteis. Para que isso funcione, é importante que você escolhe um formato de registro padrão sensato, caso contrário, suas consultas terminarão

sendo difícil ou talvez impossível de escrever. Você quer a data, a hora, o nome do microserviço, nível de registro e assim por diante em locais consistentes em cada registro.

Alguns agentes de encaminhamento de registros oferecem a capacidade de reformatar registros antes de encaminhando-os para seu repositório central de registros. Pessoalmente, eu evitaria isso sempre que possível. O problema é que a reformatação dos registros pode ser computacional intensivo, a ponto de ver problemas reais de produção causados pela CPU sendo amarrado realizando essa tarefa. É muito melhor alterar os registros conforme eles são escritos pelo seu próprio microserviço. Eu continuaria usando o encaminhamento de registros agentes para fazer a reformatação do log em locais onde eu não consigo alterar o formato de registro de origem, por exemplo, software legado ou de terceiros.

Eu teria pensado que desde que escrevi a primeira edição deste livro ressaltei que era importante reservar um padrão industrial comum para exploração madeireira teria ganhado força, mas isso não parece ter acontecido. Muitas variações parecem existir; elas normalmente envolvem o uso do formato de log de acesso padrão suportado pela web servidores como Apache e nginx e expandindo-os adicionando mais colunas de dados. O principal é que, dentro de sua própria arquitetura de microserviços, você escolha um formato que você padronize internamente.

Se você estiver usando um formato de log bastante simples, estará emitindo apenas linhas simples de texto que contêm informações específicas em locais específicos no registro de linha. No Exemplo 10-1, vemos um formato de exemplo.

Exemplo 10-1. Alguns exemplos de registros

```
15-02-2020 16:00:58 INFORMAÇÕES DO PEDIDO (abc-123) O cliente 2112 fez o pedido
q88827
15-02-2020 16:01:01 INFORMAÇÕES DE PAGAMENTO [abc-123] Pagamento de $20,99 por 988827 por cust
2112
```

A ferramenta de agregação de registros precisará saber como analisar essa string para extrair as informações que talvez queiramos consultar no carimbo de data/hora, microserviço nome ou nível de registro, por exemplo. Neste exemplo, isso é viável, pois esses partes de dados ocorrem em locais estáticos em nossos registros - a data é a primeira coluna, a hora é a segunda coluna e assim por diante. Isso é mais problemático, no entanto, se quisermos encontrar linhas de registro relacionadas a um determinado cliente, o ID do cliente está nas duas linhas de registro, mas é exibido em locais diferentes. Isso é onde podemos começar a pensar em escrever linhas de registro mais estruturadas,

talvez usando um formato JSON, para que possamos encontrar informações como um cliente ou ID do pedido em um local consistente. Novamente, a ferramenta de agregação de registros precisará ser configurado para analisar e extrair as informações necessárias do registro. Outra coisa a observar é que, se você registrar em JSON, isso pode tornar mais difícil seja lido diretamente por um ser humano sem ferramentas adicionais para analisar os valores necessários - simplesmente ler o registro em um visualizador de texto sem formatação pode não ser terrivelmente útil.

Correlacionando linhas de registro

Com um grande número de serviços interagindo para fornecer a qualquer usuário final capacidade, uma única chamada inicial pode acabar gerando várias chamadas a jusante. chamadas de serviço. Por exemplo, vamos considerar um exemplo na MusicCorp, como mostrado na Figura 10-5. Estamos inscrevendo um cliente em nosso novo streaming serviço. O cliente seleciona o pacote de streaming escolhido e clica enviar. Nos bastidores, quando o botão é clicado na interface do usuário, ele pressiona o Gateway que fica no perímetro do nosso sistema. Isso, por sua vez, passa a chamada para o microsserviço de streaming. Este microsserviço se comunica com Pagamento para receber o primeiro pagamento, usa o microsserviço do Cliente para atualizar o fato de que esse cliente agora tem o streaming ativado e envia um e-mail para o cliente usando nosso microsserviço de e-mail confirmando que está agora subscrita.

O que acontece se a chamada para o microsserviço de pagamento acabar gerando um erro estranho? Falaremos detalhadamente sobre como lidar com falhas no Capítulo 12, mas considere a dificuldade de diagnosticar o que aconteceu.

O problema é que o único microsserviço que registra um erro é o nosso Microsserviço de pagamento. Se tivermos sorte, podemos resolver qual solicitação causou o problema, e talvez até possamos examinar os parâmetros do telefonar. Mas não podemos ver esse erro no contexto mais amplo em que ele ocorre. Em este exemplo específico, mesmo se assumirmos que cada interação gera apenas uma única linha de registro, teríamos cinco linhas de log com informações sobre isso fluxo de chamadas. Ser capaz de ver essas linhas de registro agrupadas pode ser incrivelmente útil.

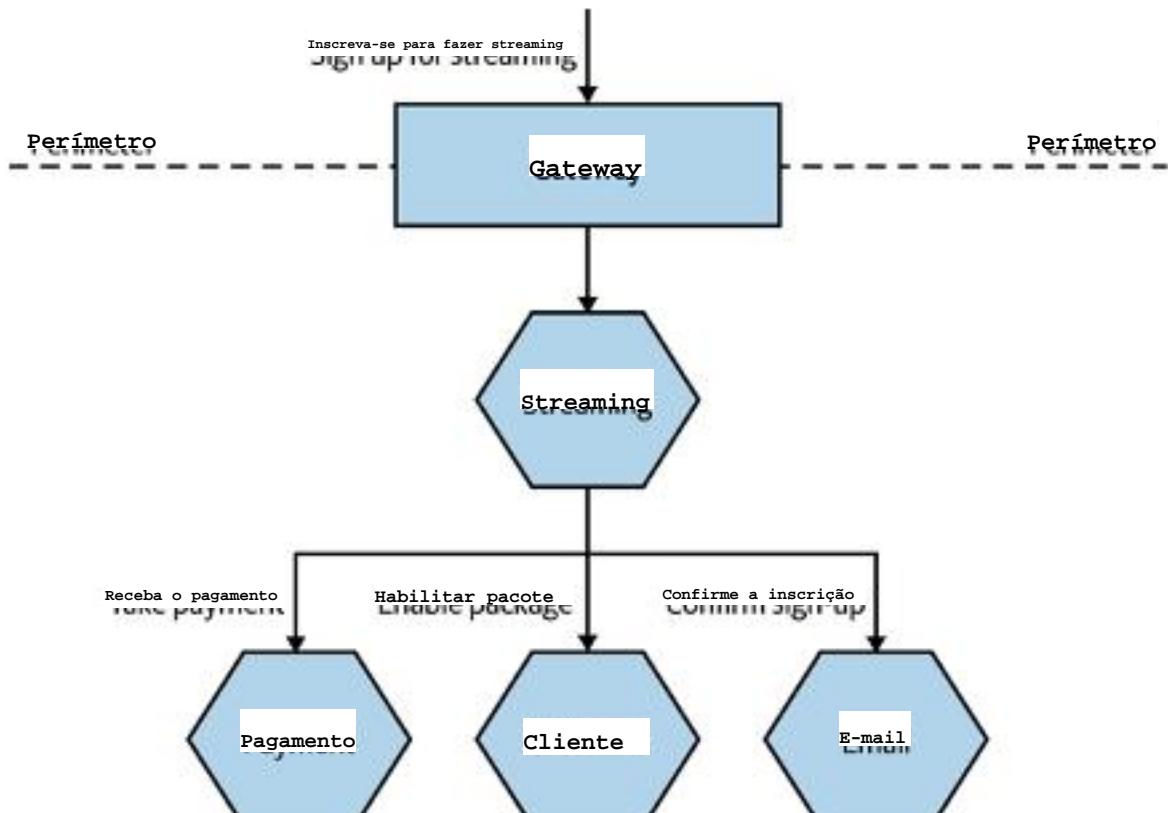


Figura 10-5. Uma série de chamadas em vários microserviços relacionados ao registro de um cliente

Uma abordagem que pode ser útil aqui é usar TDS de correlação – algo mencionamos pela primeira vez no Capítulo 6 ao discutir sagas. Quando a primeira chamada é feito, você gera um ID exclusivo que será usado para correlacionar todos os subsequentes chamadas relacionadas à solicitação. Na Figura 10-6, geramos esse ID na Gateway e, em seguida, é passado como parâmetro para todas as chamadas subsequentes.

O registro de qualquer atividade de um microserviço causada por essa chamada recebida será ser registrado juntamente com o mesmo ID de correlação, que colocamos em um localização consistente em cada linha de registro, conforme mostrado no Exemplo 10-2. Isso faz com que é fácil de extrair todos os registros associados a um determinado ID de correlação em uma data posterior.

Exemplo 10-2. Usando uma ID de correlação em um local fixo em uma linha de registro

```

15-02-2020 16:01:01 INFORMAÇÕES DO GATEWAY [abc-123] Inscreve-se para streaming
15-02-2020 16:01:02 Informações de streaming [abc-123] Cust 773 se inscreve
15-02-2020 16:01:03 INFORMAÇÕES DO CLIENTE [abc-123] Pacote de streaming adicionado
15-02-2020 16:01:03 Informações de e-mail [abc-123] Enviar streaming de boas-vindas
15-02-2020 16:01:03 ERRO DE PAGAMENTO [abc-123] Validar pagamento

```

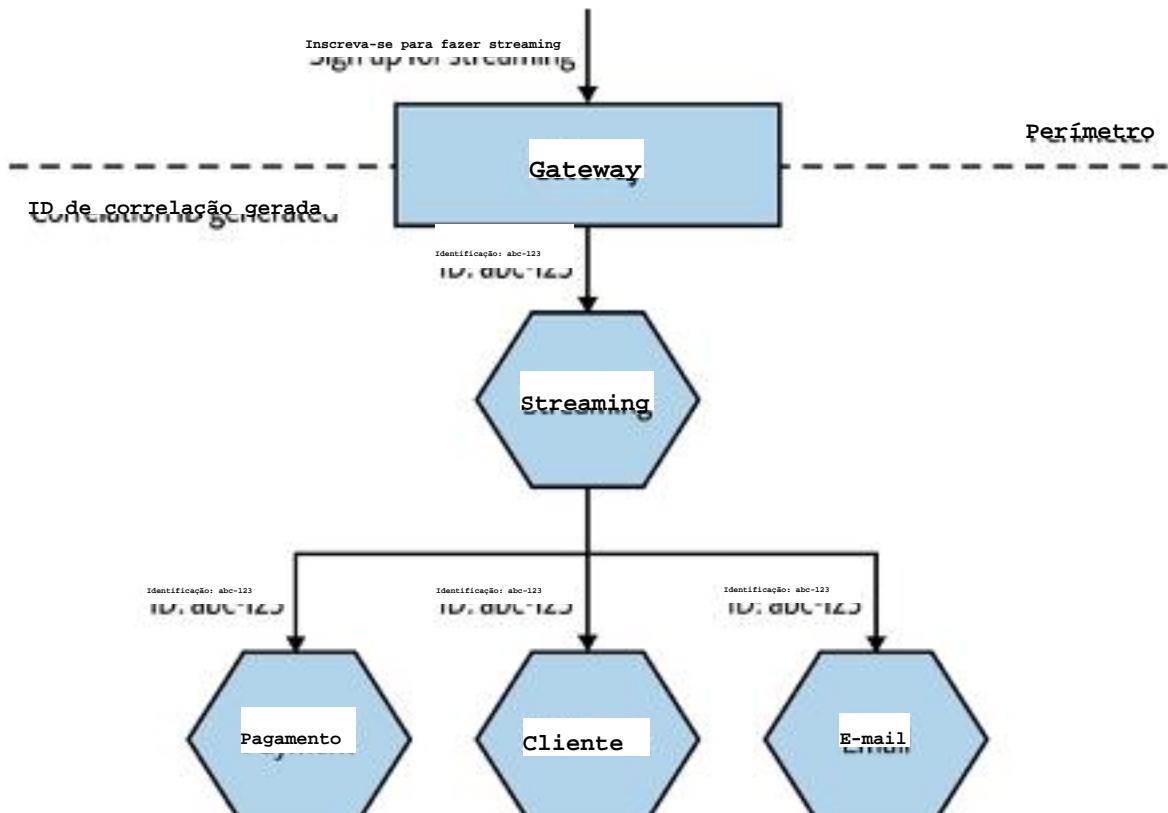


Figura 10-6. Gerando uma ID de correlação para um conjunto de chamadas

Obviamente, você precisará garantir que cada serviço saiba repassar o ID de correlação. É aqui que você precisa padronizar e ser mais forte impondo isso em todo o seu sistema. Mas depois de fazer isso, você pode na verdade, crie ferramentas para rastrear todos os tipos de interações. Essas ferramentas podem ser: útil para rastrear eventos, tempestades ou casos estranhos, ou mesmo em identificando transações especialmente caras, pois você pode imaginar o todo cascata de chamadas.

IDs de correlação em registros são o tipo de coisa que não parece tão útil inicialmente, mas confie em mim, com o tempo, eles podem ser incrivelmente úteis. Infelizmente, pode ser difícil adaptá-los a um sistema. É por esse motivo que é importante enfaticamente que você implemente IDs de correlação no registro já possível. Obviamente, os registros só podem levar você até certo ponto nesse aspecto - alguns tipos de problemas são melhor resolvidos por meio de ferramentas de rastreamento distribuídas, que vamos explore em breve. Mas um ID de correlação simples em um arquivo de log pode ser incrível útil inicialmente, o que significa que você pode adiar o uso de uma ferramenta de rastreamento dedicada até seu sistema é complexo o suficiente para garantir isso.

GORJETA

Depois de obter a agregação de registros, obtenha os IDs de correlação o mais rápido possível. Fácil de fazer em No início e, difíceis de atualizar posteriormente, eles aumentarão drasticamente o valor de seus registros.

Cronometragem

Ao examinar uma lista de linhas de registro, podemos nos enganar pensando que somos vendo uma cronologia precisa que nos ajudará a entender o que aconteceu e em que ordem. Afinal, cada linha de nossos registros inclui uma data e hora - então por que não podemos usar isso como uma forma de determinar a ordem em que as coisas ocorreu? Na sequência de chamadas no Exemplo 10-2, vemos uma linha de registro de Gateway, seguido por registros do Streaming, do e-mail do cliente e, em seguida, do Microsserviço de pagamento. Podemos concluir que esta é a ordem em que as chamadas realmente aconteceram. Infelizmente, nem sempre podemos contar com isso verdadeiro.

As linhas de registro são geradas nas máquinas em que esses microsserviços as instâncias estão em execução. Depois de serem escritos localmente, em algum momento esses registros são encaminhados. Isso significa que os carimbos de data nas linhas de registro são gerados nas máquinas em que os microsserviços estão sendo executados. Infelizmente, nós não podemos garantir que os relógios dessas diferentes máquinas estejam sincronizados. Isso significa que o relógio na máquina em que o microsserviço de e-mail está sendo executado pode estar alguns segundos à frente do relógio na máquina em que o pagamento está executar - isso pode fazer com que pareça que algo aconteceu no e-mail microserviço antes de acontecer no microsserviço de pagamento, mas isso poderia só por causa dessa distorção do relógio.

O problema da distorção do relógio causa todos os tipos de problemas em sistemas distribuídos. Existem protocolos para tentar reduzir a distorção do relógio nos sistemas - a Rede O Time Protocol (NTP) é o exemplo mais usado. O NTP, no entanto, não é garantido que funcione e, mesmo quando funciona, tudo o que pode fazer é reduzir distorção, não o elimine. Se você tiver uma sequência de chamadas que acontecem bem perto juntos, você pode descobrir que até mesmo um segundo de inclinação entre as máquinas é suficiente para que sua compreensão da sequência de chamadas seja totalmente alterada.

Fundamentalmente, isso significa que temos duas limitações quando se trata de tempo em troncos. Não podemos obter informações de tempo totalmente precisas para o fluxo geral de chama, nem podemos entender a causalidade.

Em termos de ajudar a resolver esse problema para que possamos entender a verdadeira ordem. De todas as coisas, Leslie Lamport propôs um sistema de relógio lógico, onde um contador é usado para acompanhar a ordem das chamadas. Você poderia implementar um esquema semelhante, se você quiser, e várias variações desse esquema existem. Pessoalmente, porém, se eu quisesse informações mais precisas sobre a ordem das chamadas, e eu também queria um tempo mais preciso, eu estaria mais inclinado para fazer uso de uma ferramenta de rastreamento distribuída, que abordará os dois problemas para eu. Examinaremos o rastreamento distribuído com mais detalhes posteriormente neste capítulo.

Implementações

Poucos espaços em nosso setor são tão disputados quanto o da agregação de registros e um existe uma grande variedade de soluções neste espaço.

Uma popular cadeia de ferramentas de código aberto para agregação de registros tem sido o uso de um agente de encaminhamento de registros como o Fluentd para enviar registros para o Elasticsearch, usando Kibana como forma de cortar e cortar o fluxo de troncos resultante. O maior desafio com essa pilha tende a ser a sobrecarga de gerenciar o Elasticsearch em si, mas isso pode ser menos problemático se você precisar executar o Elasticsearch para outras finalidades, ou se você fizer uso de um provedor gerenciado. Eu vou soar dois notas adicionais de cautela sobre o uso dessa cadeia de ferramentas. Em primeiro lugar, muitos esforços foram feitos para comercializar o Elasticsearch como um banco de dados. Pessoalmente, isso sempre se sentiu desconfortável comigo. Pegando algo que sempre foi anunciado como um índice de pesquisa e rebatizá-lo como um banco de dados pode ser altamente problemático. Nós implicitamente fazemos suposições sobre como os bancos de dados agem e se comportam, e nós tratá-los adequadamente, considerando-os uma fonte de verdade para dados vitais. Mas por design, um índice de pesquisa não é a fonte da verdade; é uma projeção da realidade. Faço uma pausa para pensar. Embora eu tenha certeza de que muitos desses problemas foram resolvidos, minha própria leitura desses problemas me tornou cauteloso em relação ao uso do Elasticsearch em determinadas situações e, certamente, ao considerá-lo como um banco de dados. Ter um índice de pesquisa que ocasionalmente pode perder dados não é

um problema se você já puder reindexar. Mas tratá-lo como um banco de dados é outra coisa inteiramente. Se eu estivesse usando essa pilha e não pudesse me dar ao luxo de perder o log informações, gostaria de garantir que posso reindexar os registros originais se tudo deu errado.

O segundo conjunto de preocupações tem menos a ver com os aspectos técnicos do Elasticsearch e Kibana e mais sobre os comportamentos da Elastic, a empresa por trás desses projetos. Recentemente, a Elastic tomou a decisão de mudar a licença para o código-fonte do banco de dados principal do Elasticsearch e para uma Licença Pública do Lado do Servidor (SSPL) de código aberto (SSPL). Essa mudança na licença parece ter sido o fato de a Elastic estar frustrada que organizações como a AWS fizeram ofertas comerciais bem-sucedidas com base nessa tecnologia que prejudicou o próprio comercial da Elastic. Além da preocupação de que o SSPL possa ser de natureza "viral" (em forma semelhante à GNU General Public License), essa decisão deixou muitos estão furiosos. Bem mais de mil pessoas contribuíram com código para Elasticsearch sob a expectativa de que eles estavam doando para um código aberto produto. Há a ironia adicional de que o próprio Elasticsearch e, portanto, grande parte da empresa Elastic, como um todo, foi construída com base na tecnologia da Projeto de código aberto Lucene. No momento em que este artigo foi escrito, a AWS tinha previsivelmente comprometido em criar e manter um fork de código aberto de ambos Elasticsearch e Kibana sob o Apache 2.0 de código aberto usado anteriormente licença.

De muitas maneiras, o Kibana foi uma tentativa louvável de criar um código aberto alternativa a opções comerciais caras, como o Splunk. Tão bom quanto o Splunk parece que todos os clientes do Splunk com quem falei também me disseram que pode ser extremamente caro, tanto em termos de taxas de licenciamento quanto de custos de hardware. No entanto, muitos desses clientes veem seu valor. Dito isso, há uma grande variedade de opções comerciais por aí. Pessoalmente, sou um grande fã do Humio, muitas pessoas gostam de usar o Datadog para agregação de registros, e você tem soluções prontas para uso básicas, mas viáveis, para agregação de registros com alguns provedores de nuvem pública, como o CloudWatch for AWS ou o Application Insights para o Azure.

A realidade é que você tem uma grande variedade de opções neste espaço, desde o aberto de origem para comercial, e de auto-hospedado para totalmente hospedado. Se você quiser crie uma arquitetura de microserviços, essa não é uma das coisas que você deve encontrar difícil de resolver.

Deficiências

Os registros são uma maneira fantástica e fácil de obter informações rapidamente do seu sistemas em execução. Continuo convencido de que, para um microserviço em estágio inicial arquitetura, há poucos lugares que retornarão mais o investimento do que registros quando se trata de melhorar a visibilidade de sua aplicação na produção. Eles se tornarão a força vital para a coleta e diagnóstico de informações. Isso disse, você precisa estar ciente de alguns desafios potencialmente significativos com troncos.

Em primeiro lugar, como já mencionamos, devido à distorção do relógio, nem sempre podem ser confiou para ajudá-lo a entender a ordem em que as chamadas ocorreram. Isso inclinação do relógio entre máquinas também significa o tempo preciso de uma sequência O número de chamadas será problemático, potencialmente limitando a utilidade dos registros em rastreando gargalos de latência.

O principal problema com os registros, porém, é que, à medida que você tem mais microserviços e mais chamadas, você acaba gerando MUITOS dados. Carrega grandes quantidades. Isso pode resultar em custos mais altos em termos de exigência de mais hardware, e também pode aumentar a taxa que você paga ao seu provedor de serviços (alguns provedores cobrança por uso). E dependendo de como está sua cadeia de ferramentas de agregação de registros construído, isso também pode resultar em desafios de escalabilidade. Alguma agregação de registros soluções: tente criar um índice ao receber dados de log para fazer consultas mais rápido. O problema é que manter um índice é computacionalmente caro -e quanto mais registros você recebe, e quanto maior o índice cresce, mais isso pode se tornar problemático. Isso resulta na necessidade de ser mais adaptado em o que você registra para reduzir esse problema, o que, por sua vez, pode gerar mais trabalho e o que corre o risco de você adiar o registro de informações que caso contrário, seja valioso. Falei com uma equipe que gerenciou o Elasticsearch cluster para ferramentas de desenvolvedor baseadas em SaaS; descobriu que o maior cluster Elasticsearch, que poderia ser executado com prazer, só seria capaz de lidar com seis

semanas de registro de um de seus produtos, fazendo com que a equipe tenha constantemente para transferir dados para manter as coisas gerenciáveis. Parte da razão pela qual eu gosto Humio é que isso foi algo para o qual seus desenvolvedores criaram, em vez de mantendo um índice, eles se concentram na ingestão eficiente e escalável de dados com algumas soluções inteligentes para tentar reduzir os tempos de consulta.

Mesmo que você tenha uma solução que possa armazenar o volume de registros que você deseja, esses registros podem acabar contendo muitas informações valiosas e confidenciais. Isso significa que talvez você precise limitar o acesso aos registros (o que poderia complicar ainda mais seus esforços para ter a propriedade coletiva de seu microsserviços em produção), e os registros podem ser alvo de mal-intencionados festas. Portanto, talvez seja necessário considerar não registrar certos tipos de informações (como abordaremos em "Seja frugal", se você não armazena dados, não pode ser roubado) para reduzir o impacto do acesso de pessoas não autorizadas.

Agregação de métricas

Assim como acontece com o desafio de analisar registros de diferentes hosts, precisamos analisar em maneiras melhores de coletar e visualizar dados sobre nossos sistemas. Pode ser difícil saiba o que é "bom" quando analisamos métricas para obter mais sistema complexo. Nosso site está vendo quase 50 códigos de erro HTTP 4XX por segundo. Isso é ruim? A carga da CPU no serviço de catálogo aumentou em 20% desde o almoço; algo deu errado? O segredo para saber quando entrar em pânico e quando relaxar é reunir métricas sobre como seu sistema se comporta durante um período de tempo suficientemente longo para que padrões claros surjam.

Em um ambiente mais complexo, provisionaremos novas instâncias do nosso microsserviços com bastante frequência, então queremos que o sistema que escolhemos o faça é muito fácil coletar métricas de novos anfitriões. Vamos querer poder ver uma métrica agregada para todo o sistema, por exemplo, a CPU média bot, também queremos agrregar essa métrica para todas as instâncias de um determinado serviço, ou mesmo para uma única instância desse serviço. Isso significa que vamos precisar ser capaz de associar metadados à métrica para nos permitir inferir isso estrutura.

Outro benefício importante de entender suas tendências é quando se trata de capacidade planejamento. Estamos atingindo nosso limite? Quanto tempo falta até precisarmos de mais anfitriões? Em No passado, quando comprávamos anfitriões físicos, isso geralmente era um trabalho anual. No nova era da computação sob demanda fornecida pela infraestrutura como serviço Fornecedores (IaaS), podemos aumentar ou reduzir a escala em minutos, se não segundos. Isso significa que, se entendermos nossos padrões de uso, podemos ter certeza de que infraestrutura suficiente para atender às nossas necessidades. Quanto mais inteligentes somos no rastreamento nossas tendências e, sabendo o que fazer com elas, mais econômicas e nossos sistemas podem ser responsivos.

Devido à natureza desse tipo de dados, talvez queiramos armazená-los e relatá-los métricas em diferentes resoluções Por exemplo, talvez eu queira uma amostra de CPU para meus servidores na resolução de uma amostra a cada 10 segundos nos últimos 30 minutos, a fim de reagir melhor a uma situação que está acontecendo atualmente. Ligado por outro lado, as amostras de CPU dos meus servidores do mês passado são prováveis necessário apenas para análise geral de tendências, então talvez eu fique feliz em calcular uma amostra média de CPU por hora. Isso geralmente é feito de forma padrão. plataformas de métricas para ajudar a reduzir os tempos de consulta e também reduzir os dados armazenamento. Para algo tão simples quanto uma taxa de CPU, isso pode ser bom, mas o O processo de agregação de dados antigos faz com que perdemos informações. O problema com a necessidade de que esses dados sejam agregados, muitas vezes você precisa decidir o que agregar de antemão - você precisa adivinhar com antecedência o que informações que não há problema em perder.

As ferramentas de métricas padrão podem ser perfeitamente adequadas para entender tendências ou modos de falha simples. Eles podem ser vitais, na verdade. Mas muitas vezes eles não nos ajudam tornam nossos sistemas mais observáveis, pois eles restringem os tipos de perguntas que quero perguntar. As coisas começam a ficar interessantes quando partimos de peças simples de informações como tempo de resposta, CPU ou espaço em disco usadas para pensar mais amplamente sobre os tipos de informações que queremos capturar.

Cardinalidade baixa versus alta

Muitas ferramentas, especialmente as mais recentes, foram construídas para acomodar o armazenamento e a recuperação de dados de alta cardinalidade. Existem várias maneiras para descrever a cardinalidade, mas você pode pensar nela como o número de campos que podem

seja facilmente consultado em um determinado ponto de dados. Quanto mais campos potenciais pudermos ter, maior será a cardinalidade que precisamos suportar.

Fundamentalmente, isso fica mais problemático com bancos de dados de séries temporais para razões que não vou expandir aqui, mas que estão relacionadas à forma como muitas das desses sistemas são construídos.

Por exemplo, talvez eu queira capturar e consultar o nome do microserviço, ID do cliente, ID da solicitação, número de compilação do software e ID do produto, ao longo do tempo. Então eu decido capturar informações sobre o sistema operacional, a arquitetura do sistema, o provedor de nuvem e assim por diante ligado. Eu poderia precisar capturar todas essas informações para cada ponto de dados I coletar. À medida que eu aumento o número de coisas que eu gostaria de consultar, o sistema é construído com esse caso de uso em mente. Como Charity Majors, fundadora da Honeycomb, explica:

Tudo se resume, essencialmente, à métrica. A métrica é um ponto de dados, um número único com um nome e algumas etiquetas de identificação. Todo o contexto que você pode obter, tem que ser inserido nessas etiquetas. Mas a explosão de escrita de escrever todas essas tags é caro devido à forma como as métricas são armazenadas em disco. Armazenar uma métrica é muito barato, mas armazenar uma etiqueta é caro; e armazenar muitas tags por métrica interromperá seu mecanismo de armazenamento rápido.

Na prática, sistemas construídos com baixa cardinalidade em mente terão dificuldades muito se você tentar colocar dados de maior cardinalidade neles. Sistemas como Prometheus, por exemplo, foram construídos para armazenar peças bastante simples de informações, como a taxa de CPU de uma determinada máquina. De muitas maneiras, podemos ver o Prometheus e ferramentas similares como sendo uma ótima implementação do armazenamento e consulta de métricas tradicionais. Mas a falta de capacidade de apoiar dados de maior cardinalidade podem ser um fator limitante. Os desenvolvedores do Prometheus são bastante aberto sobre essa limitação:

Lembre-se de que cada combinação exclusiva de pares de rótulos de valores-chave representa uma nova série temporal, que pode aumentar drasticamente o quantidade de dados armazenados. Não use etiquetas para armazenar dimensões com alta cardinalidade (muitos valores de rótulos diferentes), como IDs de usuário, e-mail endereços ou outros conjuntos ilimitados de valores.

Sistemas que são capazes de lidar com alta cardinalidade são mais capazes de permitir você deve fazer uma série de perguntas diferentes sobre seus sistemas - muitas vezes perguntas a você não sabia que você precisava perguntar de antemão Agora, isso pode ser difícil conceito a ser entendido, especialmente se você estiver gerenciando seu processo único sistema monolítico, felizmente, com ferramentas mais "convencionais". Mesmo aqueles pessoas com sistemas maiores se contentaram com sistemas de baixa cardinalidade, muitas vezes porque eles não tinham escolha. Mas à medida que seu sistema aumenta em complexidade, você precisará melhorar a qualidade das saídas que seu sistema fornece para permitem que você melhore sua observabilidade. Isso significa coletar mais informações e ferramentas que permitem dividir e dividir esses dados.

Implementações

Desde a primeira edição deste livro, Prometheus se tornou um popular livro aberto ferramenta de origem para o uso de métricas de coleta e agregação e, em casos em que eu poderia ter recomendado anteriormente o uso de grafite (que obteve minha recomendação na primeira edição), Prometheus pode ser um sensato substituto. O espaço comercial também se expandiu bastante nessa área, com novos fornecedores e fornecedores antigos construindo ou reequipando os existentes soluções para atingir usuários de microserviços.

Tenha em mente, porém, minhas preocupações em relação à baixa cardinalidade versus alta cardinalidade dados. Sistemas criados para lidar com dados de baixa cardinalidade serão muito difíceis de modernização para suportar armazenamento e processamento de alta cardinalidade. Se você está procurando para sistemas capazes de armazenar e gerenciar dados de alta cardinalidade, permitindo para observação (e questionamento) muito mais sofisticados do seu sistema comportamento, eu sugiro fortemente que você analise Honeycomb ou Lightstep. Embora essas ferramentas sejam frequentemente vistas como soluções para o rastreamento distribuído (que exploraremos mais tarde), eles são altamente capazes de armazenar, filtrar e consultando dados de alta cardinalidade.

SISTEMAS DE MONITORAMENTO E OBSERVABILIDADE SÃO SISTEMAS DE PRODUÇÃO

Com um conjunto crescente de ferramentas para nos ajudar a gerenciar nosso microserviço arquitetura, devemos lembrar que essas ferramentas são elas mesmas sistemas de produção. Plataformas de agregação de registros, ferramentas de rastreamento distribuídas, sistemas de alerta – todos eles são aplicativos de missão crítica que são justos tão vital quanto nosso próprio software, se não mais. O mesmo grau de diligência precisa ser aplicada em termos de manutenção de nossa produção... ferramentas de monitoramento à medida que aplicamos ao software que escrevemos e mantemos.

Devemos também reconhecer que essas ferramentas podem se tornar vetores em potencial de ataques de terceiros. No momento em que este artigo foi escrito, o governo dos EUA e outras organizações ao redor do mundo estão lidando com a descoberta de uma violação no software de gerenciamento de rede da SolarWinds. Apesar a natureza exata da violação ainda está sendo explorada, acredita-se que isso é conhecido como ataque à cadeia de suprimentos. Uma vez instalado no cliente sites (e a SolarWinds é usada por 425 das empresas da Fortune 500 dos EUA), este software permitiu que partes mal-intencionadas obtivessem acesso externo a redes de clientes, incluindo a rede do Tesouro dos EUA....

Rastreamento distribuído

Até agora, tenho falado principalmente sobre coletar informações de forma isolada. Sim, estamos agregando essas informações, mas entendendo o contexto mais amplo em que essas informações foram capturadas podem ser fundamentais. Fundamentalmente, um arquitetura de microserviços é um conjunto de processos que trabalham juntos para realizar algum tipo de tarefa - exploramos as várias maneiras diferentes que podemos coordenar essas atividades no Capítulo 6. Assim, faz sentido, quando se quer para entender como nosso sistema está realmente se comportando em uma produção ambiente, que somos capazes de ver as relações entre nossos microserviços. Isso pode nos ajudar a entender melhor como nosso sistema é se comportar, avaliar o impacto de um problema ou descobrir melhor o que exatamente não está funcionando como esperávamos.

À medida que nossos sistemas se tornam mais complexos, torna-se importante temos uma maneira de ver esses tracos em nossos sistemas. Precisamos ser capazes de extraia esses dados dispersos para nos dar uma visão conjunta de um conjunto de correlacionados chamadas. Como já vimos, fazer algo simples, como colocar a correlação IDs em nossos arquivos de log são um bom começo, mas é uma solução pouco sofisticada, especialmente porque acabaremos tendo que criar nossas próprias ferramentas personalizadas para ajudar visualize, corte e corte esses dados. É aqui que vem o rastreamento distribuído.

Como funciona

Embora as implementações exatas variem de um modo geral, são distribuídas. Todas as ferramentas de rastreamento funcionam de maneira semelhante. A atividade local em um tópico é capturado em um intervalo. Essas extensões individuais são correlacionadas usando algumas extensões exclusivas identificador. Os vãos são então enviados para um coletor central, que é capaz de construa esses intervalos relacionados em um único traço. Na Figura 10-7, vemos um imagem do Honeycomb que mostra um rastreamento em um microsserviço arquitetura.

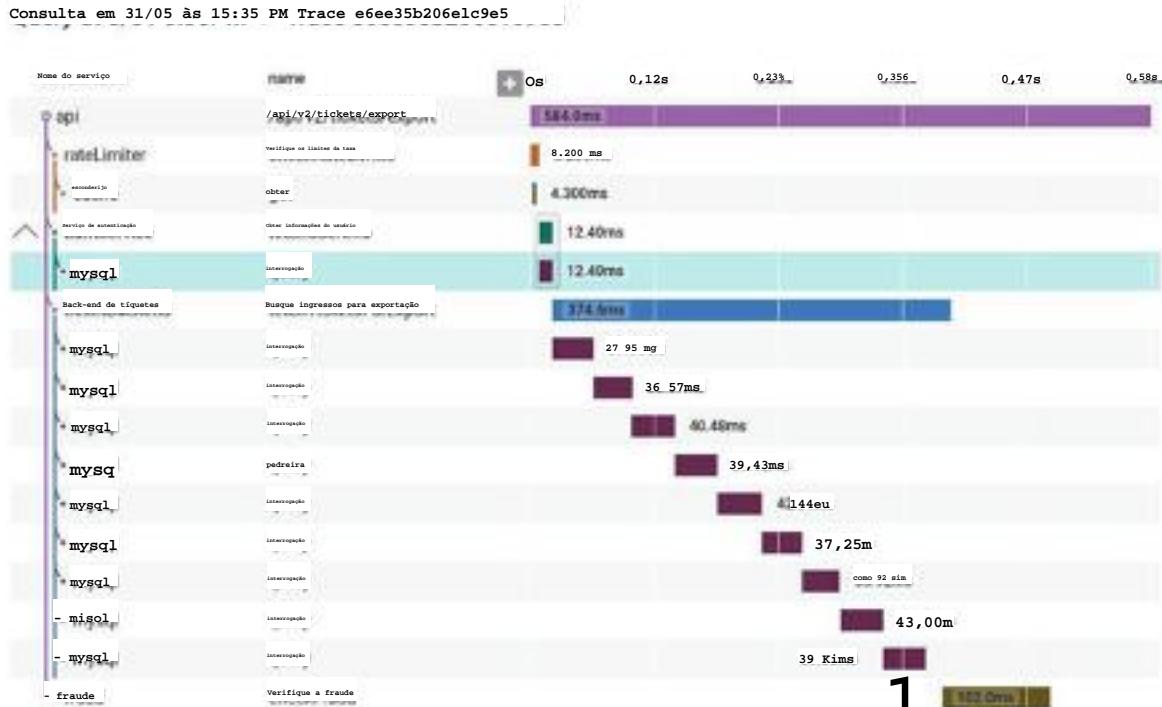


Figura 10-7. Um traço distribuído mostrado no Honeycomb, permitindo que você identifique onde está o tempo sendo gasto em operações que podem abranger vários microsserviços.

Esses intervalos permitem que você colete uma série de informações. Exatamente quais dados você coletar dependerá do protocolo que você está usando, mas no caso da API OpenTracing, cada período contém uma hora de início e término, um conjunto de registros associado ao intervalo e a um conjunto arbitrário de pares de valores-chave para ajudar com consultas posteriores (elas podem ser usadas para enviar coisas como um ID de cliente, um ID do pedido, um nome de host, um número de compilação e similares).

Coletar informações suficientes para nos permitir rastrear chamadas em nosso sistema pode têm um impacto direto no próprio sistema. Isso resulta na necessidade de alguma forma de amostragem, na qual algumas informações são explicitamente excluídas de nossa coleta de tracos para garantir que o sistema ainda possa operar. O desafio é garantindo que as informações corretas sejam descartadas e que ainda coletemos amostras suficientes para nos permitir extrapolar corretamente as observações.

As estratégias de amostragem podem ser muito básicas. O sistema Dapper do Google, que inspirou muitas das ferramentas de rastreamento distribuído que vieram depois, realizou uma amostragem aleatória altamente agressiva. Uma certa porcentagem de chamadas foram amostradas e pronto. Jaeger, por exemplo, capturará apenas 1 em 1.000 chamadas em sua configuração padrão. A ideia aqui é capturar o suficiente informações para entender o que nosso sistema está fazendo, mas não capturar muita coisa informações que o próprio sistema não consegue lidar. Ferramentas como Honeycomb e o Lightstep podem fornecer uma amostragem dinâmica e com mais nuances do que essa simples amostragem aleatória. Um exemplo de amostragem dinâmica pode ser onde você quer mais amostras para determinados tipos de eventos - por exemplo, você pode querer amostrar qualquer coisa que gere um erro, mas ficaria feliz com a amostragem apenas 1 em 100 operações bem-sucedidas se todas forem muito semelhantes.

Implementando o rastreamento de distribuição

Colocar o rastreamento distribuído em funcionamento em seu sistema requer algumas coisas. Em primeiro lugar, você precisa capturar informações de amplitude dentro de seus microserviços. Se você estiver usando uma API padrão, como o OpenTracing ou o API OpenTelemetry mais recente, você pode descobrir que alguns dos terceiros, bibliotecas e estruturas virão com suporte para essas APIs incorporadas e já enviarão informações úteis (por exemplo, capturando automaticamente informações sobre chamadas HTTP). Mas mesmo que o façam, é provável que você ainda

deseja instrumentar seu próprio código, fornecendo informações úteis sobre o que seu microsserviço está funcionando a qualquer momento.

Em seguida, você precisará de alguma forma de enviar essas informações de intervalo para seu coletor - pode ser que você envie esses dados diretamente da sua instância de microsserviço para o coletor central, mas é muito mais comum usar um encaminhamento local agente. Assim como acontece com a agregação de registros, você executa um agente localmente em seu instância de microsserviço, que enviará periodicamente as informações do intervalo para um coletor central. O uso de um agente local normalmente permite um pouco mais recursos avançados, como alterar a amostragem ou adicionar tags adicionais, e também pode armazenar com mais eficiência as informações enviadas.

Por fim, é claro, você precisa de um coletor capaz de receber essas informações e dê sentido a tudo isso.

No espaço de código aberto, a Jaeger surgiu como uma escolha popular para rastreamento distribuído. Para ferramentas comerciais, eu começaria examinando o já mencionei Lightstep e Honeycomb. Eu exorto você, no entanto, a escolher algo comprometido em oferecer suporte à API OpenTelemetry.

OpenTelemetry é uma especificação de API aberta que torna muito mais fácil faça com que códigos como drivers de banco de dados ou estruturas da web saiam da caixa com suporte para rastreamento e também pode facilitar a portabilidade entre fornecedores diferentes no lado da coleção. Com base no trabalho realizado pelo anterior APIs OpenTracing e OpenConsensus, essa API agora tem uma ampla indústria apoio.

Estamos bem?

Falamos muito sobre as coisas que você poderia fazer como operador de um sistema - a mentalidade de que você precisa, as informações que talvez precise coletar. Mas como você sabe se está fazendo demais ou não o suficiente? Como você saiba se você está fazendo um trabalho bom o suficiente ou se seu sistema está funcionando bem suficiente?

Conceitos binários de um sistema "ativo" ou "inativo" começam a ter cada vez menos ou seja, à medida que o sistema se torna mais complexo. Com um único processo sistema monolítico, é mais fácil ver a integridade do sistema em preto e branco

qualidade. Mas e quanto a um sistema distribuído? Se uma instância de um microserviço está inacessível, isso é um problema? Um microserviço é "saudável" se estiver acessível? Que tal uma situação em que nosso microserviço de devoluções está disponível, mas metade da funcionalidade que ele fornece requer o uso do microserviço downstream Inventory, que está experimentando atualmente problemas? Isso significa que consideramos o microserviço Returns saudável ou não? saudável?

À medida que as coisas se tornam mais complexas, torna-se cada vez mais importante fazer um dê um passo atrás e pense nas coisas de um ponto de vista diferente. Pense em um colmeia. Você pode olhar para uma abelha individual e determinar que não é feliz. Talvez tenha perdido uma de suas asas e, portanto, não possa mais voar. Isso é certamente um problema para aquela abelha individual, mas você pode estender a partir disso qualquer observação sobre a saúde da própria colmeia? Não, você precisaria olhar para o saúde da colmeia de uma forma mais holística. Uma abelha estar doente não significa toda a colmeia está doente.

Podemos tentar descobrir se um serviço é saudável decidindo, por exemplo, o que um bom nível de CPU é, ou o que torna um tempo de resposta aceitável. Se nosso sistema de monitoramento detecta que os valores reais estão fora desse nível seguro, podemos acionar um alerta. No entanto, de várias maneiras, esses valores são uma etapa removido do que realmente queremos rastrear, ou seja, é o sistema trabalhando? Quanto mais complexas forem as interações entre os serviços, mais longe, estamos de realmente responder a essa pergunta apenas olhando em uma métrica isolada.

Assim, podemos reunir muitas informações, mas por si só isso não nos ajuda responder à pergunta se o sistema está funcionando corretamente. Para isso, nós preciso começar a pensar um pouco mais em termos de definir o que é aceitável parece que o comportamento. Muito trabalho foi feito na área de confiabilidade do site engenharia (SRE), cujo foco é como podemos garantir que nossos os sistemas podem ser confiáveis e, ao mesmo tempo, permitir mudanças. A partir desse espaço, nós tenha alguns conceitos úteis para explorar.

Strap in, estamos prestes a entrar na sigla city.

Contrato de nível de serviço

Um contrato de nível de serviço (SLA) é um acordo alcançado entre o pessoal que constrói o sistema e as pessoas que usam o sistema. Não descreve apenas o que os usuários podem esperar, mas também o que acontece se o sistema não atingir esse nível de comportamento aceitável. Os SLAs tendem a estar muito no nível "mínimo" de coisas, geralmente a tal ponto que, se o sistema apenas atingir seus objetivos, o usuário final ainda estaria insatisfeito. Como exemplo, a AWS tem um SLA para seu serviço de computação. Isso deixa claro que não há garantia efetiva de disponibilidade para uma única instância do EC2 (uma instância virtual gerenciada pela AWS). A AWS afirma que faz seus melhores esforços para garantir 90% de disponibilidade para uma determinada instância, mas se isso não for alcançado, ela simplesmente não cobra por a hora específica em que a instância não estava disponível. Agora, se o seu EC2 as instâncias constantemente falhavam em atingir 90% de disponibilidade em uma determinada hora, causando instabilidade significativa no sistema, você pode não estar sendo cobrado, mas você também não ficaria muito feliz. Na minha experiência, a AWS na prática alcança muito mais do que o SLA descreve, como costuma acontecer com SLAs.

Objetivos de nível de serviço

Mapear um SLA para uma equipe é problemático, especialmente se o SLA for um pouco amplo e transversal. Em nível de equipe, em vez disso, falamos sobre objetivos de nível de serviço (SLOs). Os SLOs definem em que a equipe se inscreve para fornecer. Alcançar os SLOs de todas as equipes da organização será satisfazer (e provavelmente exceder em muito) os requisitos da organização. Exemplos de SLOs podem incluir coisas como tempo de atividade esperado ou tempos de resposta aceitáveis para uma determinada operação.

É muito simplista pensar que os SLOs são o que uma equipe precisa fazer para a organização para alcançar seus SLAs. Sim, se toda a organização conseguir tudo de seus SLOs, assumiríamos que todos os SLAs também foram alcançados, mas os SLOs podem falar sobre outras metas não descritas no SLA-O, elas podem ser aspiracionais, eles podem estar voltados para dentro (tentando realizar algumas atividades internas) ou para fora (tentando adaptar-se a mudanças externas). Os SLOs geralmente podem refletir algo que a própria equipe deseja conseguir, isso pode não ter relação com um SLA.

Indicadores de nível de serviço

Para determinar se estamos cumprindo nossos SLOs, precisamos coletar dados reais. Isso é o que nossos indicadores de nível de serviço (SLI) são. Um SLI é uma medida de algo que nosso software faz. Pode, por exemplo, ser um tempo de resposta de um processo, um cliente sendo registrado, um erro sendo apresentado ao cliente ou um pedido sendo feito. Precisamos coletar e exibir esses SLIs para fazer temo certeza de que estamos cumprindo nossos SLOs.

Orçamentos de erro

Quando experimentamos coisas novas, injetamos mais instabilidade potencial em nossos sistemas. Dessa forma, o desejo de manter (ou melhorar) a estabilidade do sistema poderia resultar no desencorajamento da mudança. Orçamentos errados são uma tentativa de evitar esse problema é deixar claro quanto erro é aceitável em um sistema.

Se você já optou por um SLO, calcule seu orçamento de erros deve ficar bem claro. Por exemplo, você pode dizer que seu microsserviço precisa estar disponível 99,9% do tempo por trimestre, 24 horas por dia, 7 dias por semana. Isso significa que você pode realmente ficar inativo por 2 horas e 11 minutos por quarto. Em termos desse SLO, esse é o seu orçamento de erro.

Orçamentos de erros ajudam a fornecer uma compreensão clara de quanto bem você está alcançar (ou não) um SLO, permitindo que você tome melhores decisões sobre o que riscos a serem assumidos. Se você estiver bem abaixo do seu orçamento de erros para o trimestre, talvez você esteja bem em lançar esse microsserviço escrito em uma nova programação linguagem. Se você já excedeu seu orçamento de erros, talvez tenha adiado esse lançamento e, em vez disso, concentre mais o tempo da equipe em melhorar a confiabilidade do seu sistema.

Orçamentos errados têm tanto a ver com dar às equipes espaço para experimentar coisas novas coisas como qualquer coisa.

Alertando

De vez em quando (espero que raramente, mas provavelmente mais do que gostaríamos), algo acontecerá em nossos sistemas, o que exigirá que um operador humano seja notificado para agir. Um microsserviço pode ter ficado indisponível inesperadamente,

podemos estar vendo um número maior de erros do que esperávamos, ou talvez todo o sistema ficou indisponível para nossos usuários. Nessas situações, precisamos que as pessoas estejam cientes do que está acontecendo para que possam tentar corrigir coisas.

O problema é que, com uma arquitetura de microserviços, dado o maior número de chamadas, quanto maior o número de processos e a complexidade subjacente infraestrutura, muitas vezes haverá coisas dando errado. O desafio em um ambiente de microserviços está resolvendo exatamente quais tipos de problemas deve fazer com que um humano seja informado e exatamente como ele deve ser informado.

Alguns problemas são piores do que outros

Quando algo dá errado, queremos saber sobre isso. Ou nós? São todos problemas iguais? À medida que as fontes de problemas aumentam, elas podem se tornar é mais importante poder priorizar essas questões para decidir se, e como, um operador humano deve estar envolvido. Muitas vezes, a maior pergunta que eu me encontro perguntar quando se trata de alertar é: "Esse problema deve fazer com que alguém ser acordado às 3 da manhã?"

Fui vi um exemplo desse pensamento enquanto passava algum tempo no Google campus há muitos anos. Na área de recepção de um dos edifícios em Mountain View era uma antiga prateleira de máquinas, lá como uma espécie de exposição. EU notei algumas coisas. Primeiro, esses servidores não estavam em compartimentos de servidores; eram apenas placas-mãe vazias encaixadas no rack. A principal coisa que eu Percebi, porém, que os discos rígidos estavam conectados por velcro. Eu perguntei um dos Googlers por que isso aconteceu. "Oh", disse ele, "os discos rígidos falham, então Por mais que não os queiramos enganados. Nós apenas os arrancamos, os jogamos no compartimento e velcro em um novo."

Os sistemas que o Google construiu presumiram que os discos rígidos falhariam. É otimizado o design desses servidores para garantir que a substituição do disco rígido fosse tão fácil quanto possível. Como o sistema foi construído para tolerar a falha de um disco rígido unidade, embora fosse importante que a unidade fosse substituída eventualmente, chances são uma única falha no disco rígido que não causaria nenhum problema significativo que ficaria visível para o usuário. Com milhares de servidores em um banco de dados do Google centro, seria a tarefa diária de alguém simplesmente caminhar por uma fileira de prateleiras

e substitua os discos rígidos à medida que avançavam. Claro, a falha da unidade foi um problema, mas que poderia ser tratada de forma rotineira. Uma falha no disco rígido foi considerada rotina - não vale a pena ligar para alguém fora do normal... horas de trabalho, mas talvez apenas algo sobre o qual precisassem ser informados durante seu dia normal de trabalho.

À medida que suas fontes de possíveis problemas aumentarem, você precisará melhorar em priorizando o que causa quais tipos de alertas. Caso contrário, você pode muito bem se vê lutando para separar o trivial do urgente.

Fadiga de alerta

Muitas vezes, muitos alertas podem causar problemas significativos. Em 1979, houve um fusão parcial de um reator na usina nuclear de Three Mile Island nos EUA. A investigação sobre o incidente destacou o fato de que os operadores da instalação ficaram muito impressionados com os alertas que estavam vendo. que era impossível determinar qual ação precisava ser tomada. Ali foi um alerta indicando o problema fundamental que precisava ser resolvido, mas isso não era algo óbvio para os operadores, como tantos outros os alertas estavam disparando ao mesmo tempo. Durante a audiência pública sobre o incidente, um dos operadores, Craig Faust, lembrou que "Eu teria gostado ter jogado fora o painel de alarme. Não estava nos dando nada útil "foi muito inadequado para gerenciar um acidente. n9 "was greatly inadequate for managing an accident.

Mais recentemente, vimos esse problema de muitos alertas sendo gerados no acidentes aéreos separados que mataram 346 pessoas no total. O relatório inicial de 10 em separate air crashes that killed 346 people in total. The initial report - mu as questões do Conselho Nacional de Segurança nos Transportes dos EUA (NTSB) desenharam atenção aos alertas confusos que foram acionados em condições reais e foram considerados fatores contribuintes para os acidentes. Do relatório:

A pesquisa de fatores humanos identificou que, para condições não normais, como aqueles que envolvem uma falha no sistema com vários alertas, onde há podem ser necessárias várias ações da tripulação de voo, fornecendo aos pilotos entender quais ações devem ter prioridade é uma necessidade crítica. Isso é particularmente verdadeiro no caso de funções implementadas em vários sistemas de avião devido a uma falha em um sistema dentro de arquiteturas de sistema integradas podem apresentar vários alertas e indicações para a tripulação de voo à medida que cada sistema de interface registra o falha. Portanto, é importante que as interações do sistema e a cabine de comando a interface deve ser projetada para ajudar a direcionar os pilotos para a maior prioridade ação (s).

Então, aqui estamos falando sobre operar um reator nuclear e pilotar uma aeronave. Eu suspeito que muitos de vocês agora estão se perguntando o que isso tem a ver com o sistema que você está construindo. Agora é possível (se não provável) que você não seja construindo sistemas críticos de segurança como esse, mas há muito com o qual podemos aprender esses exemplos. Ambos envolvem sistemas altamente complexos e inter-relacionados nos quais um Um problema em uma área pode causar um problema em outra. E quando geramos muitos alertas, ou não damos aos operadores a capacidade de priorizar o que os alertas devem ser focados em, o desastre pode ocorrer. Esmagando um operador com alertas pode causar problemas reais. Do relatório novamente:

Além disso, pesquisa sobre respostas piloto a respostas múltiplas/simultâneas situações anômalas, juntamente com dados de acidentes, indicam que vários alertas concorrentes podem exceder os recursos mentais disponíveis e foco de atenção estreito, levando a uma priorização atrasada ou inadequada respostas.

Então, pense duas vezes antes de simplesmente enviar mais alertas para um operador: você pode não conseguir o que você quer.

ALARME VERSUS ALERTA

Ao analisar mais amplamente o tópico de alertas, descobri que muitos pesquisas e práticas incrivelmente úteis em vários contextos, muitos dos quais não estavam falando especificamente sobre alertas em sistemas de TI. O termo alarme é comumente encontrado ao examinar este tópico em engenharia e muito mais, enquanto tendemos a usar mais o termo alerta comumente em TI. Falei com algumas pessoas que vêm uma distinção entre esses dois termos, mas, curiosamente, quais distinções as pessoas traçaram entre esses dois termos não parecia consistente. Com base no fato de que a maioria das pessoas parece ver os termos alerta e alarme como praticamente o mesmo, e sobre o fato de que, quando foi feita uma distinção entre o segundo, não era consistente, decidi padronizar o termo alerta para este livro.

Rumo a um melhor alerta

Portanto, queremos evitar muitos alertas, além de alertas de que não são úteis. Quais diretrizes podemos seguir para nos ajudar a criar alertas melhores? Steven Charnock expanda esse assunto em seu artigo "Alarm Design: From Nuclear Power to WebOps,"¹¹ que é uma ótima leitura e um bom ponto de partida ponto para mais leituras nesta área. Do artigo:

O objetivo de [alertas] é direcionar a atenção do usuário para aspectos significativos da operação ou do equipamento que exigem tempo hábil atenção.

Com base no trabalho externo de desenvolvimento de software, temos um conjunto útil de regras de, de todos os lugares, usuários de equipamentos e materiais de engenharia Associação (EEMUA), que apresentou uma descrição tão boa de o que é um bom alerta, como eu vi:

Relevante

Certifique-se de que o alerta seja valioso.

Único

Certifique-se de que o alerta não esteja duplicando outro.

Oportuna

Precisamos receber o alerta com rapidez suficiente para usá-lo.

Priorizado

Forneca ao operador informações suficientes para decidir em que ordem os alertas deve ser tratado.

Compreensível

As informações no alerta precisam ser claras e legíveis.

diagnóstico

Precisa ficar claro o que está errado.

Consultivo

Ajude o operador a entender quais ações precisam ser tomadas.

Focando

Chame a atenção para as questões mais importantes.

Relembrando minha carreira e momentos em que trabalhei na produção apoio, é deprimente pensar em quão raramente tive que lidar com os alertas. Siga qualquer uma dessas regras.

Infelizmente, com muita frequência, as pessoas que fornecem informações aos nossos alertas sistemas e as pessoas que realmente recebem nossos alertas são pessoas diferentes. De Shorrock novamente:

Compreender a natureza do tratamento de alarmes e o design associado a problemas, pode ajudar você, o especialista em seu trabalho, a ser mais informado usuário, ajudando a criar os melhores sistemas de alarme para apoiar seu trabalho.

Uma técnica que pode ajudar a reduzir o número de alertas que disputam nossa atenção envolve mudar a forma como pensamos sobre quais questões exigem que chame-os à atenção dos operadores em primeiro lugar. Vamos explorar esse tópico no próximo.

Monitoramento semântico

Com o monitoramento semântico, estamos definindo um modelo para o que é aceitável a semântica do nosso sistema é. Quais são as propriedades que o sistema deve ter para pensarmos que está operando dentro de meios aceitáveis? Em grande medida, o monitoramento semântico exige uma mudança em nosso comportamento. Em vez de procurar na presença de erros, em vez disso, precisamos estar constantemente fazendo uma pergunta: o sistema está se comportando da maneira que esperamos? Se estiver se comportando corretamente, então isso nos ajuda melhor a entender como priorizar o tratamento dos erros que estamos vendo.

A próxima coisa a descobrir é como você define um modelo para um corretamente sistema de comportamento. Você pode ser super formal com essa abordagem (literalmente, com algumas organizações fazendo uso de métodos formais para isso), mas fazendo algumas declarações de valor simples podem ajudar você a percorrer um longo caminho. Por exemplo, no caso da MusicCorp, o que deve ser verdade para estarmos convencidos de que o sistema está funcionando corretamente? Bem, talvez digamos isso:

- Novos clientes podem se cadastrar para participar.
- Estamos vendendo pelo menos \$20.000 em produtos por hora durante nosso horário de pico.
- Estamos enviando pedidos a uma taxa normal.

Se essas três afirmações puderem ser corretas, então, em termos gerais, sinto que o sistema está funcionando bem o suficiente. Voltando ao nosso anterior discussões sobre SLAs e SLOs, nosso modelo de correção semântica seria

esperávamos exceder em muito nossas obrigações em um SLA, e esperaríamos ter SLOs concretos que nos permitam rastrear esse modelo. Coloque outra maneira, fazendo essas declarações sobre como esperamos que nosso software esteja agindo contribuirão muito para identificar SLOs.

Um dos maiores desafios é chegar a um acordo sobre o que é esse modelo. Como você pode ver, não estamos falando sobre coisas de baixo nível, como "uso do disco". não deve exceder 95%"; estamos fazendo declarações de alto nível sobre nossos sistema. Como operador do sistema ou pessoa que escreveu e testou o microsserviço, você pode não estar em posição de decidir o que esses valores valem as declarações devem ser. Em uma organização de entrega orientada por produtos, é aqui que o proprietário do produto deve entrar, mas pode ser seu trabalho como operador para garantir que a discussão com o proprietário do produto realmente aconteça.

Depois de decidir qual é o seu modelo, tudo se resume a funcionar exclui-se o comportamento atual do sistema atende a esse modelo. De um modo geral, nós tem duas maneiras principais de fazer isso: monitoramento real do usuário e sintético transações. Analisaremos as transações sintéticas em um momento, à medida que elas caírem sob a égide dos testes em produção, mas vamos dar uma olhada no usuário real monitorando primeiro.

Monitoramento real de usuários

Com o monitoramento real do usuário, analisamos o que realmente está acontecendo em nosso sistema de produção e compare isso com nosso modelo semântico. Na MusicCorp, analisaríamos quantos clientes se inscreveram, quantos pedidos enviamos e assim por diante

O desafio do monitoramento real de usuários é que, muitas vezes, as informações de que precisamos não está disponível para nós em tempo hábil. Considere a expectativa de que A MusicCorp deveria vender pelo menos \$20.000 em produtos por hora. Se isso as informações estão trancadas em um banco de dados em algum lugar, talvez não consigamos coletar essas informações e agir de acordo com elas. É por isso que você pode precisar para melhorar a exposição do acesso às informações que você faria anteriormente considere métricas de "negócios" para suas ferramentas de produção. Se você pode emitir uma taxa de CPU para seu armazenamento de métricas, e esse armazenamento de métricas pode ser usado para alertar

nesta condição, então por que você também não pode registrar uma venda e um valor em dólares
nesta mesma loja?

Uma das principais desvantagens do monitoramento real de usuários é que ele é fundamentalmente barulhento. Você está recebendo muitas informações, examinando-as para descobrir se existe um problema que pode ser difícil. Também vale a pena perceber que o usuário real o monitoramento informa o que já aconteceu e, como resultado, você pode não detecte um problema até que ele tenha ocorrido. Se um cliente não conseguir se cadastrar, é um cliente insatisfeito. Com transações sintéticas, outra forma de teste em produção que analisaremos em breve, temos a oportunidade de não apenas reduzir o ruído, mas também detectar problemas antes que nossos usuários percebam eles.

Teste em produção

Não testar em prod é como não praticar com a orquestra completa porque seu solo soou bem em casa. 12

-Cursos de caridade

Como abordamos várias vezes ao longo do livro, a partir de nossa discussão sobre conceitos como implantações canárias em "Canary Release", para nossa análise do ato de equilíbrio em relação aos testes de pré e pós-produção, realizando alguma forma de teste na produção pode ser incrivelmente útil e segura atividade. Analisamos vários tipos diferentes de testes em produção neste livro, e há mais formas além disso, então eu senti que seria útil para resumir alguns dos diferentes tipos de testes em produção já analisamos e também compartilhamos alguns outros exemplos de testes em produção que é comumente usada. Me surpreende quantas pessoas que estão assustados com o conceito de testes em produção já estão fazendo isso sem realmente percebendo que são.

Todas as formas de teste na produção são indiscutivelmente uma forma de "monitoramento" atividade. Estamos realizando essas formas de testes na produção para garantir que nosso sistema de produção está funcionando conforme o esperado e muitas formas de teste na produção pode ser incrivelmente eficaz na resolução de problemas antes de os usuários até notam.

Transações sintéticas

Com transações sintéticas, injetamos comportamento falso do usuário em nossa produção sistema. Esse comportamento falso do usuário tem entradas conhecidas e saídas esperadas. Para MusicCorp, por exemplo, poderíamos criar artificialmente um novo cliente e em seguida, verifique se o cliente foi criado com sucesso. Essas transações seria demitido regularmente, nos dando a chance de resolver problemas o mais rápido possível.

Eu fiz isso pela primeira vez em 2005. Eu fazia parte de uma pequena equipe da Thoughtworks que estava construindo um sistema para um banco de investimento. Durante todo o dia de negociação, ocorreram muitos eventos que representaram mudanças no mercado. Nosso trabalho era reaja a essas mudanças e observe o impacto na carteira do banco. Nós estavam trabalhando com prazos bastante apertados, uma meta amarga era concluir todos os nossos cálculos menos de 10 segundos após a chegada do evento. O sistema em si consistia em cerca de cinco serviços discretos, pelo menos um dos quais era rodando em uma grade de computação que, entre outras coisas, estava se esgotando e não sendo usada Ciclos de CPU em cerca de 250 hosts de desktop na recuperação de desastres do banco centro.

O número de partes móveis no sistema significava que estava ocorrendo muito ruído gerado a partir de muitas das métricas de nível inferior que estávamos reunindo. Nós também não teve a vantagem de escalar gradualmente ou fazer com que o sistema funcionasse por alguns meses para entender o que era "bom" em termos de métricas de baixo nível como taxa de CPU ou tempo de resposta. Nossa abordagem foi gerar eventos falsos para parte do preço do portfólio que não foi registrada nos sistemas downstream. A cada minuto, usamos uma ferramenta chamada Nagios para executar um trabalho de linha de comando que inseriu um evento falso em uma de nossas filas. Nosso sistema o detectou e executou todos os vários cálculos como qualquer outro trabalho, exceto os resultados apareceu no livro "lixo eletrônico", que foi usado apenas para testes. Se for uma reprecificação não foi visto em um determinado período, Nagios relatou isso como um problema.

Na prática, descobri o uso de transações sintéticas para realizar a semântica um monitoramento como esse é um indicador muito melhor de problemas nos sistemas do que alertando sobre as métricas de nível inferior. Eles não substituem a necessidade do No entanto, detalhes de nível inferior: ainda desejaremos essas informações quando precisarmos para descobrir por que uma transação sintética falhou.

Implementando transações sintéticas

No passado, implementar transações sintéticas era uma tarefa bastante difícil.

Mas o mundo evoluiu e os meios para implementá-los estão à nossa disposição

pontas dos dedos! Você está executando testes para seus sistemas, certo? Se não, vá ler

Capítulo 9 e volte. Tudo pronto? Bom!

Se observarmos os testes, temos que testar um determinado serviço de ponta a ponta, ou mesmo todo o nosso sistema de ponta a ponta, temos muito do que precisamos implementar monitoramento semântico. Nossa sistema já expõe os ganchos necessários para o lançamento faça o teste e verifique o resultado. Então, por que não executar um subconjunto desses testes, em uma base contínua, como forma de monitorar nosso sistema?

Há algumas coisas que precisamos fazer, é claro. Primeiro, precisamos ter cuidado sobre os requisitos de dados de nossos testes. Talvez precisemos encontrar um caminho para nossos testes para se adaptar a diferentes dados ativos se eles mudarem com o tempo, ou então definir uma fonte de dados diferente. Por exemplo, poderíamos ter um conjunto de usuários falsos uso na produção com um conjunto conhecido de dados.

Da mesma forma, temos que garantir que não acionemos accidentalmente imprevistos efeitos colaterais. Um amigo me contou uma história sobre uma empresa de comércio eletrônico que accidentalmente executou seus testes em seus sistemas de pedidos de produção. Não funcionou percebia seu erro até que um grande número de máquinas de lavar chegou ao sede.

Teste A/B

Com um teste A/B, você implanta duas versões diferentes da mesma funcionalidade, com os usuários vendo a funcionalidade "A" ou "B". Você é então capaz de veja qual versão da funcionalidade funciona melhor. Isso é comumente usado ao tentar decidir entre duas abordagens diferentes de como algo deve ser feito - por exemplo, você pode tentar dois clientes diferentes formulários de registro para ver qual deles é mais eficaz para impulsionar as inscrições.

Lançamento canário

Uma pequena parte da sua base de usuários pode ver a nova versão da funcionalidade. Se essa nova funcionalidade funcionar bem, você poderá aumentar a parte do

base de usuários que vê a nova funcionalidade a ponto de a nova versão do
a funcionalidade agora é usada por todos os usuários. Por outro lado, se o novo
a funcionalidade não funciona conforme o esperado, você afetou apenas uma pequena parte
da sua base de usuários e pode reverter a alteração ou tentar corrigir qualquer coisa
problema que você identificou.

Corrida paralela

Com uma execução paralela, você executa duas implementações equivalentes diferentes do
a mesma funcionalidade lado a lado. Qualquer solicitação do usuário é encaminhada para ambos
versões e seus resultados podem ser comparados. Então, em vez de direcionar um usuário para
a versão antiga ou a nova, como em uma versão canária, executamos ambas
versões, mas o usuário vê apenas uma. Isso permite uma comparação completa
entre as duas versões diferentes, o que é incrivelmente útil quando queremos
uma melhor compreensão de aspectos como as características de carga de um novo
implementação de algumas funcionalidades importantes.

Testes de fumaça

Usado depois que o software é implantado na produção, mas antes de ser lançado,
testes de fumaça são executados no software para garantir que ele esteja funcionando
apropriadamente. Esses testes normalmente são totalmente automatizados e podem variar de
atividades muito simples, como garantir que um determinado microsserviço esteja ativo e
correndo para realmente executar transações sintéticas completas.

Transações sintéticas

Uma interação completa e falsa do usuário é injetada no sistema. Muitas vezes é muito
próximo ao tipo de teste de ponta a ponta que você poderia escrever.

Engenharia do caos

Um tópico que discutiremos mais no Capítulo 12, a engenharia do caos pode envolver
injeção de falhas em um sistema de produção para garantir que ele seja capaz de lidar
esses problemas esperados. O exemplo mais conhecido dessa técnica é provavelmente
Chaos Monkey, da Netflix, que é capaz de desligar máquinas virtuais em

produção, com a expectativa de que o sistema seja robusto o suficiente para que esses desligamentos não interrompem a funcionalidade do usuário final.

Padronização

Como abordamos anteriormente, você precisará de um dos atos de equilíbrio contínuos realizar é permitir que as decisões sejam tomadas de forma restrita para um único microsserviço versus onde você precisa padronizar todo o sistema. Em minha opinião, monitoramento e observabilidade são uma área em que a padronização pode ser extremamente importante. Com microsserviços colaborando de várias maneiras diferentes para fornecer recursos aos usuários usando com várias interfaces, você precisa visualizar o sistema de forma holística.

Você deve tentar gravar seus registros em um formato padrão. Você definitivamente quer ter todas as suas métricas em um só lugar, e você pode querer ter uma lista de nomes padrão para suas métricas também; seria muito irritante para alguém serviço para ter uma métrica chamada `ResponseTime` e outro para ter uma chamados `RSPTimeSecs`, quando eles significam a mesma coisa.

Como sempre acontece com a padronização, as ferramentas podem ajudar. Como eu disse antes, a chave é facilitar fazer a coisa certa, portanto, ter uma plataforma com muitos dos blocos de construção básicos existentes, como agregação de registros, fazem muito sentido. Cada vez mais, muito disso recai sobre a equipe da plataforma, cujo papel é explore mais detalhadamente no Capítulo 15.

Seleção de ferramentas

Como já abordamos, há potencialmente uma série de ferramentas diferentes para você pode precisar ser usado para ajudar a melhorar a observabilidade do seu sistema. Mas, como já mencionei, este é um espaço emergente rápido e é altamente provável que as ferramentas que usaremos no futuro tenham uma aparência muito diferente do que temos agora. Com plataformas como Honeycomb e Lightstep liderando o caminho em termos de quais ferramentas de observabilidade para microsserviços

Parece que, e com o resto do mercado, até certo ponto, se atualizando eu...
Espero que esse espaço tenha muita rotatividade no futuro.

Portanto, é bem possível que você precise de ferramentas diferentes das que você tem, certo?
agora, se você está apenas adotando microserviços, é possível que
também quero ferramentas diferentes no futuro, pois as soluções neste espaço continuam
para melhorar. Com isso em mente, quero compartilhar algumas ideias sobre critérios
que eu acho que são importantes para qualquer ferramenta nesta área.

Democrática

Se você tem ferramentas que são tão difíceis de trabalhar apenas com operadores experientes
pode fazer uso deles, então você limita o número de pessoas que podem
participar das atividades de produção. Da mesma forma, se você escolher ferramentas que sejam assim
caro a ponto de proibir seu uso em qualquer situação que não seja crítica.
ambientes de produção, então os desenvolvedores não terão exposição a eles
ferramentas até que seja tarde demais.

Escolha ferramentas que considerem as necessidades de todas as pessoas que você deseja usar
eles. Se você realmente deseja mudar para um modelo de propriedade mais coletiva de
seu software, então o software precisa ser usado por todos na equipe.

Certificando-se de que qualquer ferramenta que você escolher também será usada no desenvolvimento
e os ambientes de teste contribuirão muito para tornar essa meta uma realidade.

Fácil de integrar

Obtendo as informações certas da arquitetura de seus aplicativos e dos
sistemas em que você executa são vitais e, como já abordamos, talvez seja necessário
esteja extraindo mais informações do que antes e em formatos diferentes. Fazendo
esse processo, o mais fácil possível, é crucial. Iniciativas como o OpenTracing
ajudaram em termos de fornecimento de APIs padrão que bibliotecas de clientes e
plataformas podem suportar, fazendo integração e portabilidade entre cadeias de ferramentas
mais fácil. De especial interesse, como já discuti, é o novo OpenTelemetry
iniciativa, que está sendo conduzida por um grande número de partidos.

Escolher ferramentas que suportem esses padrões abertos facilitará os esforços de integração.
e talvez também ajude a facilitar a mudança de fornecedor posteriormente.

Forneça contexto

Ao analisar uma informação, preciso que a ferramenta me forneça uma o máximo de contexto possível para me ajudar a entender o que precisa acontecer a seguir. EU que encontrei em uma postagem no blog da Lightstep: 13 What I Learned via a Lightstep Blog Post.

Contexto temporal

Como isso se parece em comparação com um minuto, hora, dia ou mês atrás?

Contexto relativo

Como isso mudou em relação a outras coisas no sistema?

Contexto relacional

Alguma coisa depende disso? Isso depende de outra coisa?

Contexto proporcional

Quão ruim é isso? Tem um escopo grande ou pequeno? Quem é afetado?

Em tempo real

Você não pode esperar anos por essas informações. Você precisa disso agora. Sua definição de É claro que "agora" pode variar um pouco, mas no contexto de seus sistemas, você precisa de informações com rapidez suficiente para ter a chance de detectar um problema antes que um usuário o faça, ou pelo menos de ter as informações em mãos quando alguém reclama. Na prática, estamos falando de segundos, não minutos ou horas.

Adequado para sua balança

Muito do trabalho no espaço da observabilidade de sistemas distribuídos foi inspirado pelo trabalho realizado em sistemas distribuídos de grande escala. Isso, infelizmente pode nos levar a tentar recriar soluções para sistemas de escala muito maior do que a nossa, sem entender as vantagens e desvantagens.

Sistemas com grande escala geralmente precisam fazer concessões específicas para reduzir a funcionalidade de seus sistemas para lidar com a escala que eles operam em. Dapper, por exemplo, teve que fazer uso de um altamente agressivo amostragem aleatória de dados (efetivamente "descartando" muitas informações) para ser capaz de lidar com a escala do Google. Como Ben Sigelman, fundador da LightStep e criador do Dapper, diz:¹⁴

Os microserviços do Google geram cerca de 5 bilhões de RPCs por segundo; construir ferramentas de observabilidade que escalam para 5 bilhões de RPCs/seg, portanto, ferveu começou a criar ferramentas de observabilidade que são profundamente deficientes em recursos. Se sua organização está fazendo mais de 5 milhões de RPCs/seg, que ainda é bastante impressionante, mas você quase certamente não deve usar o que o Google usou: em 1/1000 da escala, você pode comprar recursos muito mais poderosos.

O ideal é que você também queira uma ferramenta que possa ser dimensionada conforme você escala. Mais uma vez, custo e eficácia pode entrar em jogo aqui. Mesmo que sua ferramenta preferida possa escalar tecnicamente para suportar o crescimento esperado do seu sistema, você pode se dar ao luxo de continuar pagando por isso?

O especialista na máquina

Eu falei muito sobre ferramentas neste capítulo, talvez mais do que em qualquer outro capítulo do livro. Isso se deve em parte à mudança fundamental da visualização. O mundo puramente em termos de monitoramento para, em vez disso, pensar em como tornar nossos sistemas mais observáveis; essa mudança de comportamento requer ferramentas para ajudar a apoiá-lo. Seria um erro, no entanto, ver essa mudança como sendo puramente sobre novas ferramentas, como espero já ter explicado. No entanto, com um grande número de fornecedores diferentes competindo por nossa atenção, também temos que ser cauteloso.

Há mais de uma década, tenho visto vários fornecedores reivindicarem algum grau de inteligência em termos de como seu sistema será aquele que detectará mágicamente problemas e nos diga exatamente o que precisamos fazer para corrigir as coisas. Isso parece vêm em ondas, mas com o recente burburinho em torno do aprendizado de máquina (ML) e a inteligência artificial (IA) está apenas aumentando, estou vendo mais afirmações por aí de detecção automatizada de anomalias sendo feita. Tenho dúvidas sobre a eficácia

isso pode ser totalmente automatizado, e mesmo assim seria
é problemático supor que toda a experiência de que você precisa possa ser automatizada
longe.

Grande parte da motivação em torno da IA sempre foi tentar codificar um especialista
conhecimento em um sistema automatizado. A ideia de que podemos automatizar
a experiência pode ser atraente para alguns, mas também é potencialmente perigosa
ideia neste espaço, pelo menos com nosso entendimento atual. Por que automatizar
experiência fora de casa? Então você não precisa investir em ter operadores especializados
execute seu sistema. Não estou tentando fazer alguma observação aqui sobre a reviravolta de
mão de obra causada pelo avanço tecnológico - mais do que as pessoas estão vendendo
essa ideia agora, e as empresas estão comprando essa ideia agora, na esperança de que essas
são problemas totalmente solucionáveis (e automatizáveis). A realidade é que, neste momento,
eles não são.

Trabalhei recentemente com uma startup com foco em ciência de dados na Europa. A startup
estava trabalhando com uma empresa que fornecia hardware de monitoramento de leitos, que
poderia reunir vários dados sobre um paciente. Os cientistas de dados foram
capaz de ajudar a ver padrões nos dados, mostrando grupos ímpares de pacientes que
pode ser determinado correlacionando várias facetas dos dados. Os dados
os cientistas poderiam dizer que "esses pacientes parecem relacionados", mas não tinham consciência de
qual era o significado desse relacionamento. Foi preciso um médico para explicar isso
alguns desses clusters se referiam a pacientes que estavam, em geral, mais doentes
do que outros. Era necessária experiência para identificar o cluster e um diferente
experiência para entender o que esse cluster significava e colocar esse conhecimento em
ação. Voltando ao nosso conjunto de ferramentas de monitoramento e observabilidade, eu
poderia ver essa ferramenta alertando alguém sobre o fato de que "algo parece estranho"
mas saber o que fazer com essas informações ainda exige um certo grau de
experiência.

Embora eu tenha certeza de que recursos como "detecção automatizada de anomalias" podem muito bem
continuar melhorando, temos que reconhecer que, neste momento, o especialista em
O sistema é, e permanecerá por algum tempo, um ser humano. Podemos criar ferramentas que
podemos informar melhor o operador sobre o que precisa ser feito, e podemos fornecer
automação para ajudar o operador a realizar suas decisões de forma mais eficaz
maneira. Mas o ambiente fundamentalmente variado e complicado de um

sistema distribuído significa que precisamos de pessoas qualificadas e apoiadas por operadores. Queremos que nossos especialistas usem sua experiência para perguntar o que é certo. fazer perguntas e tomar as melhores decisões possíveis. Não deveríamos estar perguntando eles devem usar seus conhecimentos para lidar com as deficiências de ferramentas precárias. Nem devemos sucumbir à noção conveniente de que alguma nova ferramenta sofisticada o fará resolva todos os nossos problemas.

Começando

Como eu descrevi, há muito em que pensar aqui. Mas eu quero fornecer um ponto de partida básico para uma arquitetura simples de microsserviços em termos do que e como você deve capturar as coisas.

Para começar, você quer ser capaz de capturar informações básicas sobre o hospeda seus microsserviços em execução na taxa de CPU, E/S e assim por diante certifique-se de que você pode combinar uma instância de microsserviço com o host em que ela está continuando. Para cada instância de microsserviço, você deseja capturar a resposta horários para suas interfaces de serviço e registre todas as chamadas downstream em registros. Obtenha IDs de correlação em seus registros desde o início. Registre outras etapas importantes em seu processos de negócios. Isso exigirá que você tenha pelo menos uma métrica básica e uma cadeia de ferramentas de agregação de registros em vigor.

Eu hesitaria em dizer que você precisa começar com um rastreamento distribuído dedicado ferramenta. Se você mesmo precisar executar e hospedar a ferramenta, isso pode adicionar algo significativo complexidade. Por outro lado, se você puder fazer uso de um totalmente gerenciado oferta de serviços com facilidade, instrumentar seus microsserviços desde o início pode faz muito sentido.

Para operações-chave, considere seriamente a criação de transações sintéticas como uma forma de entender melhor se os aspectos vitais do seu sistema estão funcionando adequadamente. Crie seu sistema com esse recurso em mente.

Tudo isso é apenas a coleta básica de informações. Mais importante ainda, você precisa para garantir que você possa filtrar essas informações para fazer perguntas sobre a corrida sistema. Você é capaz de dizer com confiança que o sistema está correto? trabalhando para seus usuários? Com o tempo, você precisará coletar mais informações

e melhore suas ferramentas (e a maneira como você as usa) para melhorar melhor o observabilidade de sua plataforma.

Resumo

Os sistemas distribuídos podem ser complicados de entender, e muito mais distribuídos, mas difícil é a tarefa de solução de problemas de produção se torna. Quando a pressão está alta, os alertas aumentam e os clientes estão gritando, é importante que você tenha as informações certas disponíveis para descubra o que diabos está acontecendo e o que você precisa fazer para consertá-lo.

À medida que sua arquitetura de microserviços se torna mais complexa, ela se torna menor é fácil saber com antecedência quais problemas podem acontecer. Em vez disso, você vai frequentemente se surpreenda com os tipos de problemas que você encontrará. É assim torna-se essencial para afastar seu pensamento da maioria (passiva) atividade de monitoramento e no sentido de tornar ativamente seu sistema observável.

Isso envolve não apenas alterar potencialmente seu conjunto de ferramentas, mas também se afastar de painéis estáticos a atividades mais dinâmicas de fatiamento e corte em cubos.

Com um sistema simples, o básico ajudará você a percorrer um longo caminho. Obtenha agregação de registros entre desde o início e também obtenha IDs de correlação em suas linhas de registro. Distribuído o rastreamento pode seguir mais tarde, mas fique atento quando chegar a hora de colocar isso está no lugar certo.

Afaste sua compreensão da integridade do sistema ou do microserviço de um estado binário de "feliz" ou "triste"; perceba, em vez disso, que a verdade é sempre mais mais sutil do que isso. Passe de ter todos os pequenos problemas gerando um alerta a pensar de forma mais holística sobre o que é aceitável. Considere fortemente adotando SLOs e alertando com base nesses preceitos para reduzir a fadiga dos alertas e concentre adequadamente a atenção.

Acima de tudo, trata-se de aceitar que nem todas as coisas são cognoscíveis diante de você produção de sucesso. Seja bom em lidar com o desconhecido.

Já abordamos muitas coisas, mas há mais para descobrir aqui. Se você quiser explorar os conceitos de observabilidade com mais detalhes, então eu recomendo Observabilidade Engenharia feita por Charity Majors, Liz Fong-Jones e George Miranda. 15

também recomendo o Site Reliability Engineering 16 e o The Site

Caderno de trabalho de confiabilidade¹⁷ como bons pontos de partida para uma discussão mais ampla em torno de SLOs, SLIs e similares. É importante notar que esses dois últimos livros são muito escritos do ponto de vista de como as coisas são (ou foram) feitas em Google, e isso significa que esses conceitos nem sempre serão traduzidos. Você provavelmente não são do Google e provavelmente não têm problemas do tamanho do Google. Dito isso, ainda há muito a recomendar nesses livros.

No próximo capítulo, teremos uma visão diferente, embora ainda holística, de nossa sistemas e considere algumas das vantagens e desafios exclusivos que arquiteturas refinadas podem fornecer na área de segurança.

1 Honestamente, as vidas dos negros importam (@honest_update). 7 de outubro de 2015. 19h10,

<https://oreil.ly/Z28BA>.

2 O incêndio como causa da interrupção do sistema não é uma ideia totalmente exagerada. Uma vez eu ajudei no rescaldo de uma interrupção de produção causada por uma rede de armazenamento (SAN) pegando fogo. Isso foi necessário vários dias para sabermos que um incêndio ocorreu é uma história para outro dia.

3 Eu não disse que estava tendo sucesso.

4 Leslie Lamport, "Hora, relógios e a ordenação de eventos em um sistema distribuído".

Comunicações da ACM 21, nº 7 (julho de 1978): 558-65, <https://oreil.ly/qzyMh>.

5 Veja a análise que Kyle Kingsbury fez sobre o Elasticsearch 1.1.0 em "Jepsen: Elasticsearch,"

<https://oreil.ly/uO9wU> e no Elasticsearch 1.5.0, em "Jepsen: Elasticsearch 1.5.0,"

<https://oreil.ly/8BC1>.

6 Renato Losio, "A Elastic altera as licenças do Elasticsearch e do Kibana: o AWS forks Both"

InfoQ. 25 de janeiro de 2021. <https://oreil.ly/VdWeD>.

7 Charity Majors, "Metrics: Not the Observability Droids que você está procurando", Honeycomb (blog), 24 de outubro de 2017, <https://oreil.ly/TEETP>.

8 Na maioria das vezes. A AWS agora fornece instâncias simples, o que mexe um pouco com meu cérebro.

9 Comissão Presidencial dos Estados Unidos sobre o acidente em Three Mile Island, A necessidade de Mudança, o legado da TMI: relatório da Comissão Presidencial sobre o acidente aos três Mile Island (Washington DC: A Comissão, 1979).

10 Conselho Nacional de Segurança de Transporte, Relatório de Recomendação de Segurança: Suposições usadas em o processo de avaliação de segurança e os efeitos de vários alertas e indicações no piloto Desempenho (Washington, DC: NTSB, 2019).

11 Steven Shorrock, "Design de alarmes: da energia nuclear aos WebOps", Humanistic Systems (blog), 16 de outubro de 2015. <https://oreil.ly/RCHDL>.

12 Charity Majors (@mipsytipsy), Twitter, 7 de julho de 2019 às 9h48, <https://oreil.ly/4VUAX>.

13 "Observabilidade: uma visão geral completa para 2021", - Lightstep, acessado em 16 de junho de 2021, <https://oreil.ly/alRu>.

14 Ben Sigelman, "Três pilares com zero respostas" - rumo a um novo scorecard para Observabilidade", Lightstep (postagem no blog), 5 de dezembro de 2018, <https://oreil.ly/R3LwC>.

15 curso da caridade Liz Bong-Tones e George Miranda Engenharia de Observabilidade (Sebastopol: O'Reilly, 2022). No momento em que este livro foi escrito, este livro está em lançamento antecipado.

16 Betsy Beyer et al. eds., Engenharia de confiabilidade de sites: como o Google executa sistemas de produção (Sebastopol: O'Reilly, 2016).

17 Betsy Beyer et al., eds., The Site Reliability Workbook (Sebastopol: O'Reilly, 2018).

Capítulo 11. Segurança

Quero começar este capítulo dizendo que não me considero um especialista na área de segurança de aplicativos. Eu pretendo simplesmente ser um consciente incompetente - em outras palavras, quero entender o que não sei e ser ciente dos meus limites. Mesmo quando aprendo mais sobre esse espaço, aprendo que existe ainda mais para saber. Isso não quer dizer que se educar sobre tópicos como isso é inútil - sinto que tudo o que aprendi sobre esse espaço ao longo do os últimos dez anos me tornaram um desenvolvedor e arquiteto mais eficaz.

Neste capítulo, destaco os aspectos da segurança que considero que valem a pena para um desenvolvedor geral, arquiteto ou profissional de operações que trabalha em um arquitetura de microserviços para entender. Ainda é necessário ter o suporte de especialistas na área de segurança de aplicativos, mas mesmo que você tenha acesso a essas pessoas, ainda é importante que você tenha alguma base em esses tópicos. Da mesma forma que os desenvolvedores aprenderam mais sobre testes ou gerenciamento de dados, tópicos antes restritos a especialistas, com um conhecimento geral dos tópicos de segurança pode ser vital em termos de segurança predial em nosso software desde o início.

Ao comparar microservices com arquiteturas menos distribuídas, acontece que somos apresentados a uma dicotomia interessante. Por um lado, nós agora ter mais dados fluindo pelas redes que antes permaneceriam em uma única máquina, e temos uma infraestrutura mais complexa executando nossa arquitetura - nossa área de superfície de ataque é muito maior. Por outro lado, os microservices nos dão mais oportunidades de defender em profundidade e limitar o escopo de acesso, potencialmente aumentando a projeção do nosso sistema enquanto também reduz o impacto de um ataque caso ele ocorra. Esse aparente paradoxo - que os microservices podem tornar nossos sistemas menos seguros e mais seguros - é realmente apenas um delicado ato de equilíbrio. Minha esperança é que até o final disso capítulo, você terminará no lado direito dessa equação.

Para ajudar você a encontrar o equilíbrio certo quando se trata da segurança do seu arquitetura de microservices, abordaremos os seguintes tópicos:

Princípios fundamentais

Conceitos fundamentais que são úteis para adotar quando se busca construir software mais seguro.

As cinco funções da cibersegurança

Identifique, proteja, detecte, responda e recupere – uma visão geral dos cinco principais áreas funcionais para segurança de aplicativos

Fundamentos da segurança de aplicativos

Alguns conceitos fundamentais específicos de segurança de aplicativos e como eles aplicam-se a microserviços, incluindo credenciais e segredos, correções, backups e reconstrução.

Confiança implícita versus confiança zero

Diferentes abordagens de confiança em nosso ambiente de microserviços e como isso afeta as atividades relacionadas à segurança

Protegendo dados

Como protegemos os dados enquanto eles trafegam pelas redes e ficam no disco

Autenticação e autorização

Como o login único (SSO) funciona em uma arquitetura de microserviços, modelos de autorização centralizados versus descentralizados e o papel do Tokens JWT como parte disso.

Princípios fundamentais

Muitas vezes, quando o tópico de segurança de microserviços surge, as pessoas querem começar falando sobre questões tecnológicas razoavelmente sofisticadas, como o uso de Tokens JWT ou a necessidade de TLS mútuo (tópicos que exploraremos posteriormente neste capítulo). No entanto, o problema com a segurança é que você está tão seguro quanto seu aspecto menos seguro. Para usar uma analogia, se você deseja proteger seu

em casa, seria um erro concentrar todos os seus esforços em ter uma porta da frente resistente a picadas, com luzes e câmeras para dissuadir pessoas mal-intencionadas, se você deixa sua porta traseira aberta.

Portanto, existem alguns aspectos fundamentais da segurança de aplicativos que nós precisa analisar, ainda que brevemente, para destacar a infinidade de questões de que você precisa estar ciente de. Veremos como essas questões centrais são intensificadas (ou menos) complexos no contexto de microserviços, mas eles também devem ser geralmente aplicável ao desenvolvimento de software como um todo. Para aqueles de vocês quem quer avançar com todas essas "coisas boas", por favor, certifique-se de não estão se concentrando muito em proteger sua porta da frente enquanto saem da parte de trás porta aberta.

Princípio do menor privilégio

Ao conceder acesso ao aplicativo a indivíduos, sistemas externos ou internos, ou até mesmo em nossos próprios microserviços, devemos prestar muita atenção ao que acesso que concedemos. O princípio do menor privilégio descreve a ideia de que quando concedendo acesso, devemos conceder o acesso mínimo que uma parte precisa ter eliminam a funcionalidade necessária e somente pelo período de tempo em que precisarem dela. O principal benefício disso é garantir que, se as credenciais forem comprometidas por um atacante, essas credenciais darão à parte mal-intencionada acesso tão limitado quanto possível.

Se um microserviço tiver acesso somente de leitura a um banco de dados, então um atacante obter acesso a essas credenciais do banco de dados só obterá acesso somente para leitura, e somente para esse banco de dados. Se a credencial do banco de dados expirar antes que seja comprometida, então a credencial se torna inútil. Esse conceito pode ser estendido para limitar quais microserviços podem ser comunicados por dados específicos festas.

Como veremos mais adiante neste capítulo, o princípio do menor privilégio pode se estender para garantir que os controles de acesso sejam concedidos apenas por períodos limitados, como bem, limitando ainda mais quaisquer resultados ruins que possam ocorrer no caso de um compromisso.

Defesa em profundidade

O Reino Unido, onde eu moro, está cheio de castelos. Esses castelos são um lembrete parcial da história do nosso país (não menos importante, uma lembrança de uma época antes dos Estados Unidos)

O reino estava, bem, unido até certo ponto). Eles nos lembram de uma época em que as pessoas sentiram a necessidade de defender suas propriedades dos inimigos. Às vezes, o percebi que os inimigos eram diferentes - muitos dos castelos perto de onde eu moro em Kent foram projetados para se defender contra invasões costeiras da França. I tanto faz

A razão pela qual os castelos podem ser um ótimo exemplo do princípio da defesa em profundidade.

Ter apenas um mecanismo de proteção é um problema se um atacante encontrar um maneira de violar essa defesa, ou se o mecanismo de proteção se defender contra apenas certos tipos de atacantes. Pense em um forte de defesa costeira cujo único

A parede está voltada para o mar, deixando-a totalmente indefesa contra um ataque por terra. Se você olha para o Castelo de Dover, que fica muito perto de onde eu moro, há várias proteções fáceis de ver. Em primeiro lugar, está em uma grande colina, fazendo uma abordagem ao castelo por terra é difícil. Não tem uma parede, mas duas. Uma brecha na primeira parede ainda exige que um atacante lide com a segunda. E uma vez que você atravessa a parede final, você tem uma grande e imponente manter (torre) para lidar com.

O mesmo princípio deve ser aplicado quando incorporamos proteções em nosso segurança de aplicativos. Ter várias proteções em vigor contra as quais se defender atacantes são vitais. Com arquiteturas de microserviços, temos muito mais lugares onde podemos proteger nossos sistemas. Ao quebrar nossa

funcionalidade separada em diferentes microserviços e limitando o escopo do que

esses microserviços podem fazer isso, já estamos aplicando a defesa em profundidade. Nós podemos

também executar microserviços em diferentes segmentos de rede, aplique com base na rede

proteções em mais lugares e até mesmo faça uso de uma combinação de tecnologia para

construindo e executando esses microserviços de forma que um único exploit de dia zero

pode não afetar tudo o que temos.

Os microserviços oferecem mais capacidade de defesa em profundidade do que equivalentes

aplicativos monolíticos de processo único e, como resultado, eles podem ajudar

as organizações constroem sistemas mais seguros.

TIPOS DE CONTROLES DE SEGURANÇA

No caso das controles de segurança que podemos implementar para proteger nosso sistema, podemos categorizá-los como: 2

Preventivo

Impedir que um ataque aconteça. Isso inclui armazenar segredos.

com segurança, criptografando dados em repouso e em trânsito e implementando mecanismos adequados de autenticação e autorização.

Detetive

Alertando você sobre o fato de que um ataque está acontecendo/aconteceu.

Os firewalls de aplicativos e os serviços de detecção de intrusões são bons exemplos.

Responsivo

Ajudando você a responder durante/após um ataque. Ter um automatizado mecanismo para reconstruir seu sistema, fazendo backups para recuperar dados, e um plano de comunicação adequado em vigor após um incidente pode ser vital.

Será necessária uma combinação dos três para proteger adequadamente um sistema, e você pode ter múltiplos de cada tipo. Voltando ao nosso castelo exemplo, podemos ter várias paredes, que representam várias controles preventivos. Poderíamos ter torres de vigia instaladas e um farol sistema para que possamos ver se um ataque está acontecendo. Finalmente, talvez tenhamos alguns carpinteiros e pedreiros de prontidão, caso precisemos fortalecer as portas ou paredes após um ataque. Obviamente, você é improvável que construirmos castelos para viver, então veremos exemplos de esses controles para uma arquitetura de microserviços mais adiante neste capítulo.

Automação

Um tema recorrente deste livro é a automação. Como temos muitos m...
peças móveis com arquiteturas de microserviços, a automação se torna fundamental para
nos ajudando a gerenciar a crescente complexidade do nosso sistema. Ao mesmo tempo,
temos um objetivo de aumentar a velocidade de entrega, e a automação é
essencial aqui. Os computadores são muito melhores em fazer a mesma coisa repetidamente
mais uma vez do que os humanos - eles fazem isso mais rápido e com mais eficiência do que nós
pode (e com menos variabilidade também). Eles também podem reduzir o erro humano
e facilitar a implementação do princípio do menor privilégio - podemos
atribuir privilégios específicos a scripts específicos, por exemplo.

Como veremos ao longo deste capítulo, a automação pode nos ajudar a recuperar no
após um incidente. Podemos usá-lo para revogar e alternar chaves de segurança e também
use ferramentas para ajudar a detectar possíveis problemas de segurança com mais facilidade. Como
com outros aspectos da arquitetura de microserviços, adotando uma cultura de
a automação o ajudará imensamente quando se trata de segurança.

Incorpore a segurança ao processo de entrega

Como muitos outros aspectos da entrega de software, a segurança é muito frequente
considerada uma reflexão tardia. Historicamente, pelo menos, abordando aspectos de segurança
de um sistema é algo que é feito depois que o código é escrito,
potencialmente levando a uma reformulação significativa posteriormente. A segurança sempre foi
vista como uma espécie de obstáculo para o lançamento de software.

Nos últimos 20 anos, vimos problemas semelhantes com testes, usabilidade e
operações. Esses aspectos da entrega de software eram frequentemente fornecidos em
de forma isolada, geralmente após a conclusão da maior parte do código. Um antigo colega
meu, Jonny Schneider, certa vez comparou a abordagem à usabilidade de
software em termos de "Você quer batatas fritas com isso?" mentalidade. Em outros
palavras, usabilidade é uma reflexão tardia - algo que você espalha por cima do
"refeição principal".

A realidade, é claro, é que um software que não é utilizável, não é seguro, não pode
operar adequadamente na produção e está cheio de bugs, não é de forma alguma
molde ou forme uma "refeição principal" - é uma oferta defeituosa, na melhor das hipóteses. Nós obtivemos
melhor em direcionar os testes para o processo principal de entrega, pois temos

feito com aspectos operacionais (DevOps, alguém?) e usabilidade-segurança. Não deveria ser diferente. Precisamos garantir que os desenvolvedores tenham mais consciência geral das preocupações relacionadas à segurança, de que os especialistas encontram uma maneira de incorporarem-se às equipes de entrega quando necessário, e às ferramentas melhora para nos permitir incorporar ideias relacionadas à segurança em nosso software. Isso pode criar desafios nas organizações que adotam equipes alinhadas ao fluxo, com maiores graus de autonomia em relação à propriedade de seus microserviços. Qual é o papel dos especialistas em segurança? Em "Habilitando equipes", veremos como especialistas, como especialistas em segurança, podem trabalhar para dar suporte a equipes alinhadas e ajudam proprietários de microserviços a criar mais segurança pensando em seu software, mas também para garantir que você tenha a profundidade certa de experiência disponível quando você precisar.

Existem ferramentas automatizadas que podem sondar nossos sistemas em busca de vulnerabilidades, por exemplo, procurando ataques de script entre sites. O Zed Attack Proxy (também conhecido como ZAP) é um bom exemplo. Informado pelo trabalho da OWASP, o ZAP tenta recriar ataques maliciosos em seu site. Existem outras ferramentas que usam estatística e análise para procurar erros comuns de codificação que possam abrir brechas de segurança, como o Brakeman para Ruby; também existem ferramentas como o Snyk, que entre outras coisas podem adquirir dependências de bibliotecas de terceiros que têm vulnerabilidades conhecidas. Onde essas ferramentas podem ser facilmente integradas ao normal CI constrói, integrando-as em seus check-ins padrão é um ótimo lugar para começar. Obviamente, vale a pena notar que muitos desses tipos de ferramentas podem abordar apenas problemas locais, por exemplo, uma vulnerabilidade em uma parte específica do código. Eles não substituem a necessidade de entender a segurança do seu sistema em um nível mais amplo e sistêmico.

As cinco funções da cibersegurança

Com esses princípios fundamentais no fundo de nossas mentes, vamos agora considerar as atividades abrangentes relacionadas à segurança que precisamos realizar. Nós vamos então continuar entendendo como essas atividades variam no contexto de um microserviço arquitetura. O modelo que eu prefiro para descrever o universo de aplicação e segurança vem do Instituto Nacional de Padrões e Tecnologia dos EUA.

(NIST), que descreve um modelo útil de cinco partes para as várias atividades envolvidas na segurança cibernética:

- Identifique quem são seus possíveis atacantes, quais alvos eles são tentando adquirir e onde você está mais vulnerável.
- Proteja seus principais ativos de possíveis hackers.
- Detecte se um ataque aconteceu, apesar de seus melhores esforços.
- Responda quando descobrir que algo ruim aconteceu.
- Recupere-se após um incidente.

Acho que esse modelo é especialmente útil devido à sua natureza holística. É muito fácil de colocar todo o seu esforço para proteger seu aplicativo sem primeiro considerar quais ameaças você pode realmente enfrentar, quanto mais descobrir quais você pode fazer isso se um atacante inteligente passar por suas defesas.

Vamos explorar cada uma dessas funções com um pouco mais de profundidade e ver como a arquitetura de microserviços pode mudar a forma como você aborda essas ideias no比较 com uma arquitetura monolítica mais tradicional.

Identificar

Antes de descobrirmos o que devemos proteger, precisamos descobrir quem podem estar atrás de nossas coisas e do que exatamente eles podem estar procurando. Muitas vezes é difícil nos colocarmos na mentalidade de um atacante, mas é exatamente isso que precisamos fazer isso para garantir que concentremos nossos esforços no lugar certo. Ameaça modelagem é a primeira coisa que você deve observar ao abordar esse aspecto da segurança de aplicativos.

Como seres humanos, somos muito ruins em entender o risco. Muitas vezes nos fixamos em as coisas erradas, ignorando os maiores problemas que podem surgir. Obviamente, isso se estende ao campo da segurança. Nosso entendimento sobre quais riscos de segurança podemos estar expostos geralmente é amplamente influenciado por nossa visão limitada do sistema, nossas habilidades e nossas experiências.

Quando converso com desenvolvedores sobre riscos de segurança no contexto de arquitetura de microserviços, eles imediatamente começam a falar sobre JWTs e TLS mútuo. Eles buscam soluções técnicas para problemas técnicos tenha alguma visibilidade de. Não quero apontar o dedo apenas para os desenvolvedores todos nós temos uma visão limitada do mundo. Voltando à analogia, nós usado anteriormente, é assim que podemos acabar com uma porta frontal incrivelmente segura e uma porta traseira bem aberta.

Em uma empresa em que trabalhei, houve várias discussões sobre a necessidade de instalar câmeras de circuito fechado de TV (CCTV) nas áreas de recepção dos escritórios da empresa em todo o mundo. Isso foi devido a um incidente em que uma pessoa não autorizada obteve acesso à área do front office e depois a rede corporativa. A crença era que um sistema de câmera CCTV não só impediria que outros tentassem a mesma coisa novamente, mas também ajudaria identificar os indivíduos envolvidos após o fato.

O espectro da vigilância corporativa desencadeou uma onda de angústia na empresa relacionada a questões do tipo "big brother". Dependendo de qual lado do a discussão em que você estava, era sobre espionar funcionários (se você eram câmeras "profissionais") ou sobre ficar feliz que intrusos tivessem acesso ao edifícios (se você fosse "anti" câmeras). Deixando de lado a natureza problemática de uma discussão tão polarizada,⁴ um funcionário falou de forma bastante tímida maneira de sugerir que talvez a discussão tenha sido um pouco equivocada, pois estavam faltando alguns problemas maiores, a saber, o fato de que as pessoas pareciam sem se preocupar com o fato de a porta da frente de um dos escritórios principais ter uma fechadura com defeito, e que durante anos as pessoas chegavam de manhã para encontrar a porta desbloqueado.

Essa história extrema (mas verdadeira) é um ótimo exemplo de um problema comum cara ao tentar proteger nossos sistemas. Sem ter tempo para absorver todas as fatores e entender exatamente onde estão seus maiores riscos, você pode muito bem acabar por sentir falta dos lugares onde seu tempo é melhor gasto. O objetivo da ameaça modelar é ajudar você a entender o que um invasor pode querer de seu sistema. O que eles estão procurando? Serão diferentes tipos de agentes maliciosos quer ter acesso a diferentes ativos? A modelagem de ameaças, quando feita corretamente, é principalmente sobre se colocar na mente do atacante, pensando a partir do

de fora para dentro. Essa visão externa é importante e é uma das razões pelas quais ter uma pessoa externa para ajudar a conduzir um exercício de modelagem de ameaças pode ser muito útil.

A ideia central da modelagem de ameaças não muda muito quando analisamos arquiteturas de microserviços, além do fato de que qualquer arquitetura é analisado agora pode ser mais complexo. O que muda é a forma como obtemos o resultado de um modelo de ameaça e coloque-o em ação. Uma das saídas de uma ameaça o exercício de modelagem seria uma lista de recomendações sobre qual segurança os controles precisam ser implementados - esses controles podem incluir coisas como um mudança no processo, uma mudança na tecnologia ou talvez uma modificação no sistema arquitetura. Algumas dessas mudanças podem ser transversais e impactar várias equipes e seus microserviços associados. Outros podem resultar em trabalho mais direcionado. Fundamentalmente, porém, ao assumir uma ameaça modelagem, você também precisa olhar holisticamente, focando essa análise um pequeno subconjunto do seu sistema, como um ou dois microserviços, pode resultar em uma falsa sensação de segurança. Você pode acabar concentrando seu tempo na construção de um porta da frente fantasticamente segura, apenas para deixar a janela aberta.

Para um mergulho mais profundo nesse tópico, posso recomendar a Modelagem de Ameaças:

Designing for Security by Adam Shostack.

Proteger

Depois de identificarmos nossos ativos mais valiosos e mais vulneráveis, precisam garantir que estejam devidamente protegidos. Como observei, microserviços as arquiteturas, sem dúvida, nos dão uma área de superfície de ataque muito maior, e nós portanto, têm mais coisas que podem precisar ser protegidas, mas elas também fornecem uso mais opções para se defender em profundidade. Passaremos a maior parte deste capítulo focando sobre vários aspectos da proteção, principalmente porque esta é a área onde as arquiteturas de microserviços criam os maiores desafios.

Detectar

Com uma arquitetura de microserviços, detectar um incidente pode ser mais complexo. Temos mais redes para monitorar e mais máquinas para ficar de olho. As fontes de informação são bastante aumentadas, o que pode tornar a detecção

problemas ainda mais difíceis. Muitas das técnicas que exploramos em ...
O capítulo 10, como a agregação de registros, pode nos ajudar a reunir informações que
nos ajude a detectar que algo ruim pode estar acontecendo. Além desses,
existem ferramentas especiais, como sistemas de detecção de intrusão, que você pode usar
identifique o mau comportamento. O software para lidar com a crescente complexidade do
nossos sistemas estão melhorando, especialmente no espaço de cargas de trabalho de contêineres
com ferramentas como o Aqua.

Responder

Se o pior aconteceu e você descobriu, o que você deveria fazer?

Desenvolver uma abordagem eficaz de resposta a incidentes é vital para limitar a
danos causados por uma violação. Isso normalmente começa com a compreensão do
escopo da violação e quais dados foram expostos. Se os dados forem expostos
inclui informações de identificação pessoal (PPI), então você precisa seguir
processos de notificação e resposta a incidentes de segurança e privacidade. Isso
pode muito bem significar que você precisa conversar com diferentes partes da sua organização,
e, em algumas situações, você pode ser legalmente obrigado a informar um designado
oficial de proteção de dados quando certos tipos de violação ocorrem.

Muitas organizações agravaram o impacto de uma violação por meio de
tratamento incorreto das consequências, muitas vezes levando ao aumento das penalidades financeiras
além dos danos causados à sua marca e seu relacionamento com
clientes. É importante, portanto, entender não apenas o que você precisa
faça por motivos legais ou de conformidade, mas também o que você deve fazer em
termos de cuidar dos usuários do seu software. O GDPR, por exemplo,
exige que as violações de dados pessoais sejam comunicadas às autoridades competentes
dentro de 72 horas - um cronograma que não parece excessivamente oneroso. Isso não acontece
significa que você não poderia se esforçar para informar as pessoas mais cedo se houver uma violação de seus dados
ocorreu.

Além dos aspectos de comunicação externa da resposta, como você lida
as coisas internamente também são críticas. Organizações que têm uma cultura de culpa
e o medo provavelmente vai piorar após um grande incidente.
As lições não serão aprendidas e os fatores contribuintes não surgirão. Ligado
por outro lado, uma organização que se concentra na abertura e na segurança será

melhor lugar para aprender as lições que garantem que incidentes semelhantes sejam menores

provável que aconteça. Voltaremos a isso em "Blame".

Recuperar

A recuperação se refere à nossa capacidade de colocar o sistema em funcionamento novamente no
após um ataque e também nossa capacidade de implementar o que aprendemos a fazer
garantir que os problemas tenham menos probabilidade de acontecer novamente. Com um microsserviço
arquitetura, temos muitas outras partes móveis, que podem facilitar a recuperação
mais complexo se um problema tiver um impacto amplo. Então, mais adiante neste capítulo, vamos
veja como coisas simples, como automação e backups, podem ajudar você a reconstruir um
sistema de microsserviços sob demanda e faça com que seu sistema volte a funcionar como
o mais rápido possível.

Fundamentos da segurança de aplicativos

OK, agora que temos alguns princípios fundamentais em vigor e alguma noção do
um mundo amplo que as atividades de segurança podem abranger, vamos dar uma olhada em algumas bases
tópicos de segurança no contexto de uma arquitetura de microsserviços, se você quiser
crie um sistema mais seguro: credenciais, patches, backups e reconstrução do sistema.

Credenciais

De um modo geral, as credenciais dão a uma pessoa (ou computador) acesso a alguns
forma de recurso restrito. Isso pode ser um banco de dados, um computador, um usuário
conta ou outra coisa. Com uma arquitetura de microsserviços, em termos de como
se compara a uma arquitetura monolítica equivalente, provavelmente temos a mesma
número de humanos envolvidos, mas temos muito mais credenciais na mistura
representando diferentes microsserviços, máquinas (virtuais), bancos de dados e o
como. Isso pode levar a um certo grau de confusão sobre como restringir (ou não).
restringir (restringir) o acesso e, em muitos casos, isso pode levar a uma abordagem "preguiçosa" na qual
um pequeno número de credenciais com amplos privilégios é usado na tentativa de
simplifique as coisas. Isso, por sua vez, pode levar a mais problemas se as credenciais forem
comprometido.

Podemos dividir o tópico das credenciais em duas áreas principais. Em primeiro lugar, nós ter as credenciais dos usuários (e operadores) do nosso sistema. Estes são geralmente são o ponto mais fraco do nosso sistema e são comumente usados como um ataque vetorializado por partes maliciosas, como veremos em breve. Em segundo lugar, podemos considerar informações secretas que são essenciais para administrar nosso microsservicos. Em ambos os conjuntos de credenciais, temos que considerar questões de rotacão, revogacão e limitação do escopo.

Credenciais do usuário

As credenciais do usuário, como combinações de e-mail e senha, permanecem essenciais de quantos de nós trabalhamos com nosso software, mas eles também são um potencial fraco identifique quando se trata de nossos sistemas serem acessados por pessoas mal-intencionadas.

O Relatório de Investigações de Violação de Dados de 2020 da Verizon descobriu que alguma forma de o roubo de credenciais foi usado em 80% dos casos causados por hackers. Isso inclui situações em que as credenciais foram roubadas por meio de mecanismos como phishing ataques ou onde as senhas foram brutalmente forçadas

Existem alguns conselhos excelentes sobre como lidar adequadamente com as coisas, como senhas - conselhos que, apesar de serem simples e claros de seguir, ainda são conselhos mais recentes do NIST e do Centro Nacional de Segurança Cibernética do Reino Unido.⁶

Este conselho inclui recomendações para usar gerenciadores de senhas e longos senhas, para evitar o uso de regras de senha complexas, e um pouco surpreendentemente, para evitar alterações regulares de senha obrigatórias. Postagem completa de Troy vale a pena ler em detalhes.

Na era atual dos sistemas orientados por API, nossas credenciais também se estendem a gerenciando coisas como chaves de API para sistemas de terceiros, como contas para seu provedor de nuvem pública. Se uma parte mal-intencionada obtiver acesso à sua raiz Conta da AWS, por exemplo, eles poderiam decidir destruir tudo em execução nessa conta. Em um exemplo extremo, esse ataque resultou em uma empresa chamados de Code Spaces saindo do mercado - todos os seus recursos foram executando em uma única conta, com backups e tudo mais. A ironia dos espaços de código oferecendo "Hospedagem Svn sólida, segura e acessível", hospedagem Git e "Gerenciamento de projetos" não está perdido para mim.

Mesmo que alguém consiga suas chaves de API para seu provedor de nuvem e não decide destruir tudo o que você construiu, eles podem decidir girar invente algumas máquinas virtuais caras para executar alguma mineração de bitcoin na esperança que você não notará. Isso aconteceu com um dos meus clientes, que descobriu que alguém gastou mais de \$10K fazendo exatamente isso antes de a conta ser encerrada para baixo. Acontece que os atacantes também sabem como automatizar - existem bots lá fora, que apenas verificará as credenciais e tentará usá-las para iniciar máquinas para mineração de criptomoedas.

Segredos

De um modo geral, segredos são informações críticas que um microsserviços precisam operar e que também sejam sensíveis o suficiente para que exigem proteção contra pessoas mal-intencionadas. Exemplos de segredos que um o microsserviço pode precisar incluir:

- Certificados para TLS
- Chaves SSH
- Pares de chaves de API pública/privada
- Credenciais para acessar bancos de dados

Se considerarmos o ciclo de vida de um segredo, podemos começar a separar os vários aspectos do gerenciamento de segredos que podem exigir segurança diferente, necessidades:

Criação

Como criamos o segredo em primeiro lugar?

Distribuição

Depois que o segredo é criado, como podemos garantir que ele chegue ao lugar certo? (e somente no lugar certo).

O segredo é armazenado de forma a garantir que somente partes autorizadas possam acessá-lo.

Monitoramento

Sabemos como esse segredo está sendo usado?

Rotação

Somos capazes de mudar o segredo sem causar problemas?

Se tivermos vários microserviços, cada um dos quais pode querer ser diferente conjuntos de segredos, precisaremos usar ferramentas para ajudar a gerenciar tudo isso.

O Kubernetes fornece uma solução de segredos integrada. É um pouco limitado em termos de funcionalidade, mas vem como parte de uma instalação básica do Kubernetes, então pode ser bom o suficiente para muitos casos de uso.

Se você estava procurando por uma ferramenta mais sofisticada nesse espaço, a da Hashicorp Vale a pena dar uma olhada no Vault. Uma ferramenta de código aberto com opções comerciais disponível, é um verdadeiro canivete suíço de gerenciamento e manuseio de segredos tudo, desde os aspectos básicos da distribuição de segredos até a geração de tempo-credentials limitadas para bancos de dados e plataformas em nuvem. O Vault tem o adicionado benefício de que a ferramenta de suporte consul-template é capaz de atualizar dinamicamente segredos em um arquivo de configuração normal. Isso significa partes do seu sistema que quero ler segredos de um sistema de arquivos local, não precisa mudar para oferecer suporte a ferramenta de gerenciamento de segredos. Quando um segredo é alterado no Vault, consulte-o modelo pode atualizar essa entrada no arquivo de configuração, permitindo que seu microserviços para alterar dinamicamente os segredos que eles estão usando. Isso é fantástico para gerenciar credenciais em grande escala.

Alguns provedores de nuvem pública também oferecem soluções nesse espaço; por exemplo, O AWS Secrets Manager ou o Azure Key Vault vêm à mente. Algumas pessoas não gostam da ideia de armazenar informações secretas críticas em uma nuvem pública serviço como esse, no entanto, isso se resume ao seu modelo de ameaça. Se for é uma preocupação séria, não há nada que o impeça de executar o Vault em seu provedor de nuvem pública preferido e gerenciando esse sistema você mesmo. Mesmo que

os dados em repouso são armazenados no provedor de nuvem, com o armazenamento apropriado back-end, você pode garantir que os dados sejam criptografados de tal forma que, mesmo que uma parte externa obteve os dados e não conseguiu fazer nada com eles.

Rotação

Idealmente, queremos alternar as credenciais com frequência para limitar os danos a alguém podem fazer isso se obtiverem acesso às credenciais. Se uma pessoa mal-intencionada obtiver acesso para seu par de chaves públicas/privadas da API da AWS, mas essa credencial é alterada uma vez por semana, eles têm apenas uma semana para fazer uso das credenciais. Eles ainda pode causar muitos danos em uma semana, é claro, mas você entendeu. Alguns tipos de atacantes gostam de obter acesso aos sistemas e depois não serem detectados, permitindo que eles coletem dados mais valiosos ao longo do tempo e encontrem maneiras de outras partes do seu sistema. Se eles usaram credenciais roubadas para obter acesso, você pode ser capaz de interrompê-los se as credenciais que eles usam expirarem antes que eles possam fazer muito uso deles.

Um ótimo exemplo de rotação de credenciais de operadora seria a geração de chaves de API com limite de tempo para usar a AWS. Muitas organizações agora geram chaves de API em tempo real para sua equipe, com o par de chaves públicas e privadas sendo válido somente por um curto período de tempo, normalmente menos de uma hora. Isso permite gerar as chaves de API necessárias para realizar qualquer operação é necessário, com a certeza de que, mesmo que seja uma pessoa mal-intencionada posteriormente obtiver acesso a essas chaves, eles não poderão usá-las. Mesmo que você verificou acidentalmente esse par de chaves no GitHub público, não seria use para qualquer pessoa quando expirar.

O uso de credenciais com limite de tempo também pode ser útil para sistemas. O Hashicorp's Vault pode gerar credenciais com limite de tempo para bancos de dados. Em vez disso do que sua instância de microsserviço lendo detalhes da conexão do banco de dados de um armazenamento de configuração ou um arquivo de texto; em vez disso, eles podem ser gerados em tempo real para um instância específica do seu microsserviço.

Passar para um processo de rotação frequente de credenciais, como chaves, pode ser doloroso. Falei com empresas que sofreram incidentes como resultado da rotação de chaves, em que os sistemas param de funcionar quando as chaves são trocadas. Isso é muitas vezes devido ao fato de que pode não estar claro o que está usando um determinado

credencial. Se o escopo da credencial for limitado, o impacto potencial de a rotação é significativamente reduzida. Mas se a credencial tiver amplo uso, funcionará descobrir o impacto de uma mudança pode ser difícil. Isso não é para afastá-lo da rotação, mas só para alertá-lo sobre os riscos potenciais, e continuo convencido de que é a coisa certa a fazer. A maneira mais sensata de avançar provavelmente seria adotar ferramentas para ajudar a automatizar esse processo e, ao mesmo tempo, limitar o escopo de cada conjunto de credenciais ao mesmo tempo.

Revogação

Ter uma política em vigor para garantir que as principais credenciais sejam alternadas em uma base regular pode ser uma forma sensata de limitar o impacto do vazamento de credenciais, mas o que acontece se você souber que uma determinada credencial se enquadra no māos erradas? Você tem que esperar até que uma rotação programada comece para isso? a credencial não será mais válida? Isso pode não ser prático ou sensato. Em vez disso, o ideal é poder revogar automaticamente e potencialmente regenere credenciais quando algo assim acontece.

O uso de ferramentas que permitem o gerenciamento centralizado de segredos pode ajudar aqui, mas isso pode exigir que seus microsservicos possam reler recentemente valores gerados. Se o seu microsserviço estiver lendo diretamente segredos de algo como a loja de segredos do Kubernetes ou o Vault, ele pode ser notificado quando esses valores foram alterados, permitindo que seu microsserviço faça uso dos valores alterados. Como alternativa, se o seu microsserviço ler apenas esses segredos na inicialização, talvez seja necessário reiniciar continuamente o sistema para recarregar essas credenciais. No entanto, se você está trocando credenciais regularmente, é provável é que você já teve que resolver o problema de seus microsservicos serem capaz de reler essas informações. Se você se sentir confortável com a rotação regular dos credenciais, é provável que você já esteja configurado para lidar com emergências revogação também.

DIGITALIZANDO CHAVES

A verificação acidental de chaves privadas em repositórios de código-fonte é uma forma comum de vazamento de credenciais para partes não autorizadas - isto acontece uma quantidade surpreendente. O GitHub verifica automaticamente os repositórios em busca de alguns tipos de segredos, mas você também pode executar seu próprio escaneamento. É ótimo se você pudesse descobrir segredos antes de fazer o check-in, e git-segredos permite que você faça exatamente isso. Ele pode escanear os commits existentes em busca de possíveis segredos, mas ao configurá-lo como um gancho de confirmação, ele pode até mesmo interromper os commits sendo feito. Há também os gitleaks similares, que, além de suportando ganchos de pré-confirmação e verificação geral de confirmações, tem alguns recursos que o tornam potencialmente mais útil como uma ferramenta geral para digitalizando arquivos locais.

Limitando o escopo

Limitar o escopo das credenciais é fundamental para a ideia de abraçar o princípio do menor privilégio. Isso pode se aplicar a todas as formas de credenciais, mas limitando o escopo daquilo a que um determinado conjunto de credenciais dá acesso pode ser incrivelmente útil. Por exemplo, na Figura 11-1, cada instância do microserviço de inventário recebe o mesmo nome de usuário e senha para o banco de dados de suporte. Também fornecemos acesso somente para leitura ao suporte Processo Debezium que será usado para ler dados e enviá-los via Kafka como parte de um processo ETL existente. Se o nome de usuário e a senha do os microserviços estão comprometidos, teoricamente, uma parte externa poderia ganhar acesso de leitura e gravação ao banco de dados. Se eles tivessem acesso ao Debezium credenciais, no entanto, eles só teriam acesso somente para leitura.

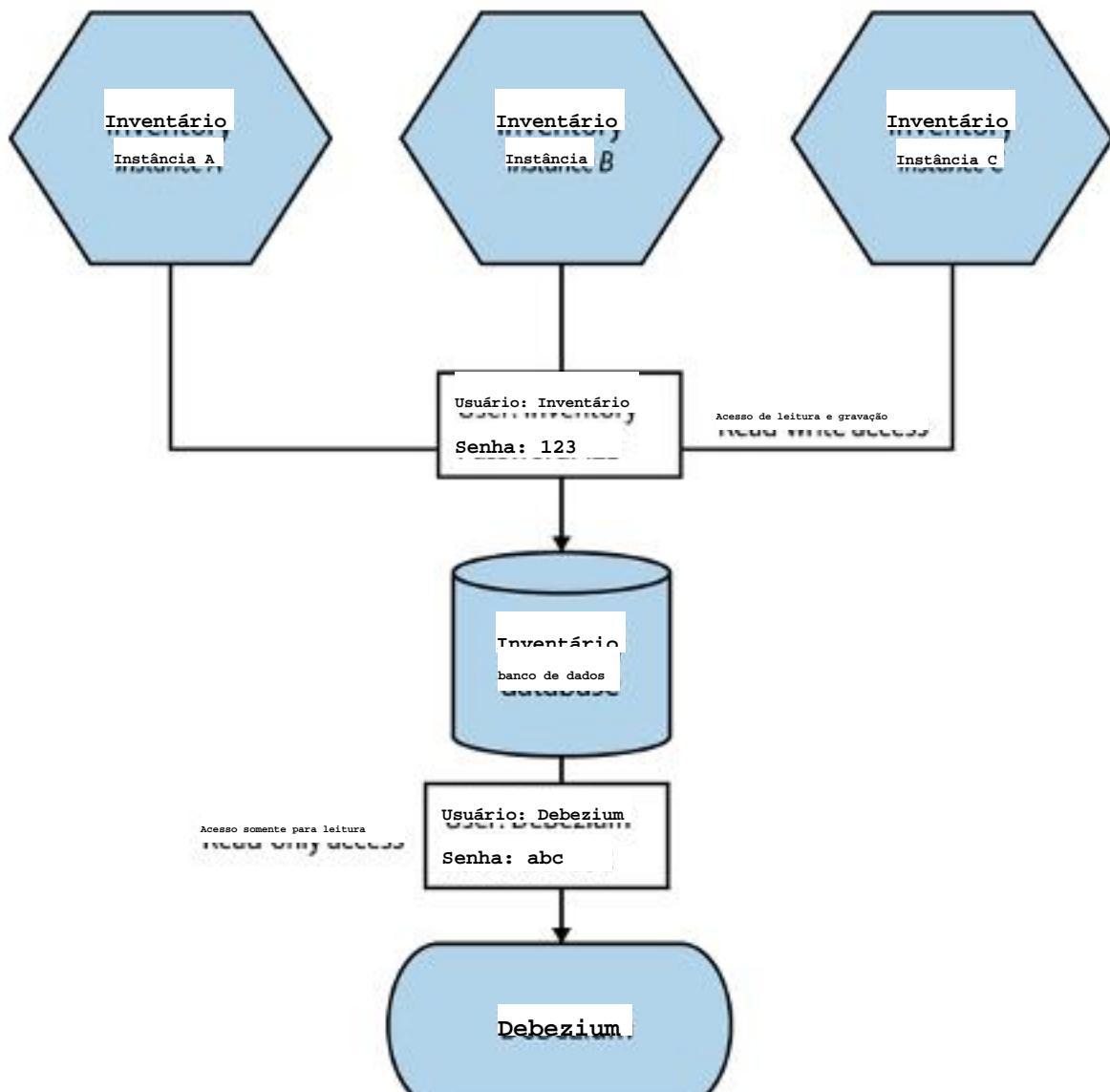


Figura 11-1 Limitando o escopo das credenciais para limitar o impacto do uso indevido

A limitação do escopo pode ser aplicada tanto em termos do que o conjunto de credenciais pode acesso e também quem tem acesso a esse conjunto de credenciais. Na Figura 11-2, mudamos as coisas de forma que cada instância do Inventory tenha uma diferente conjunto de credenciais. Isso significa que podemos alternar cada credencial de forma independente, ou simplesmente revogar a credencial de uma das instâncias, se for o caso que fica comprometido. Além disso, com credenciais mais específicas, pode ser mais fácil descobrir de onde e como a credencial foi obtida. Ali são obviamente outros benefícios decorrentes de ter uma identificação única

nome de usuário para uma instância de microserviço aqui; talvez seja mais fácil rastreá-lo... qual instância causou uma consulta cara, por exemplo...

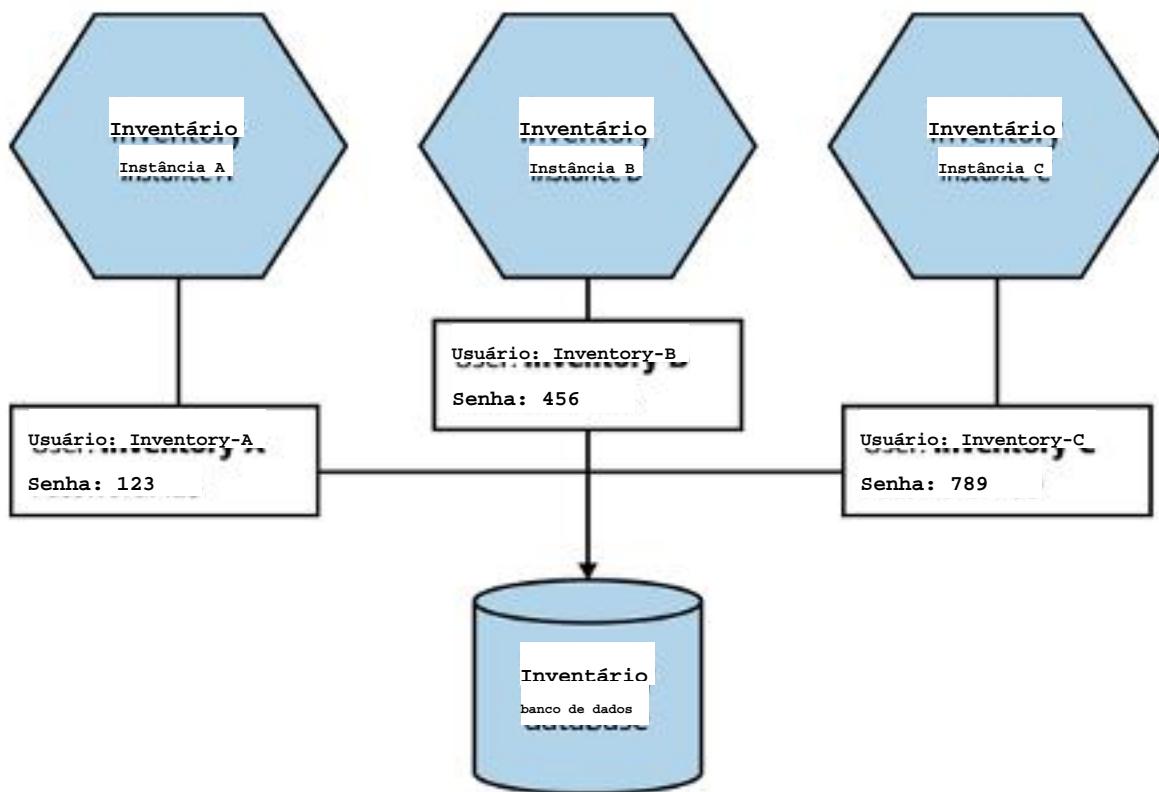


Figura 11-2. Cada instância do Inventory tem suas próprias credenciais de acesso ao banco de dados, limitando ainda mais o acesso

Como já abordamos, o gerenciamento de credenciais refinadas em grande escala pode ser complexo, e se você quisesse adotar uma abordagem como essa, alguma forma de automação seria essencial - lojas secretas como a Vault vêm à mente como maneiras perfeitas de implementar esquemas como esse.

Corrigindo

A violação de dados da Equifax de 2017 é um ótimo exemplo da importância de corrigindo. Uma vulnerabilidade conhecida no Apache Struts foi usada para obter acesso não autorizado aos dados mantidos pela Equifax. Porque Equifax é um crédito Bureau, essa informação era especialmente sensível. No final, descobriu-se que os dados de mais de 160 milhões de pessoas foram comprometidos na violação. A Equifax acabou tendo que pagar um acordo de 700 milhões de dólares.

Meses antes da violação, a vulnerabilidade no Apache Struts havia sido identificado, e uma nova versão foi feita pelos mantenedores corrigindo o problema. Infelizmente, a Equifax não atualizou para a nova versão do software, apesar de estar disponível por meses antes do ataque. Tinha A Equifax atualizou este software em tempo hábil, parece provável que o o ataque não teria sido possível...

A questão de manter o controle dos patches está se tornando mais complexa à medida que implante sistemas cada vez mais complexos. Precisamos nos tornar mais sofisticados na forma como lidamos com esse conceito bastante básico.

A Figura 11-3 mostra um exemplo das camadas de infraestrutura e software que existem abaixo de um cluster típico do Kubernetes. Se você executar tudo isso faça sua própria infraestrutura, você é responsável por gerenciar e corrigir todas essas camadas. Você tem certeza de que está atualizado com o patch? Obviamente, se você puder transferir parte desse trabalho para um provedor de nuvem pública, você também pode descarregar parte desse fardo.

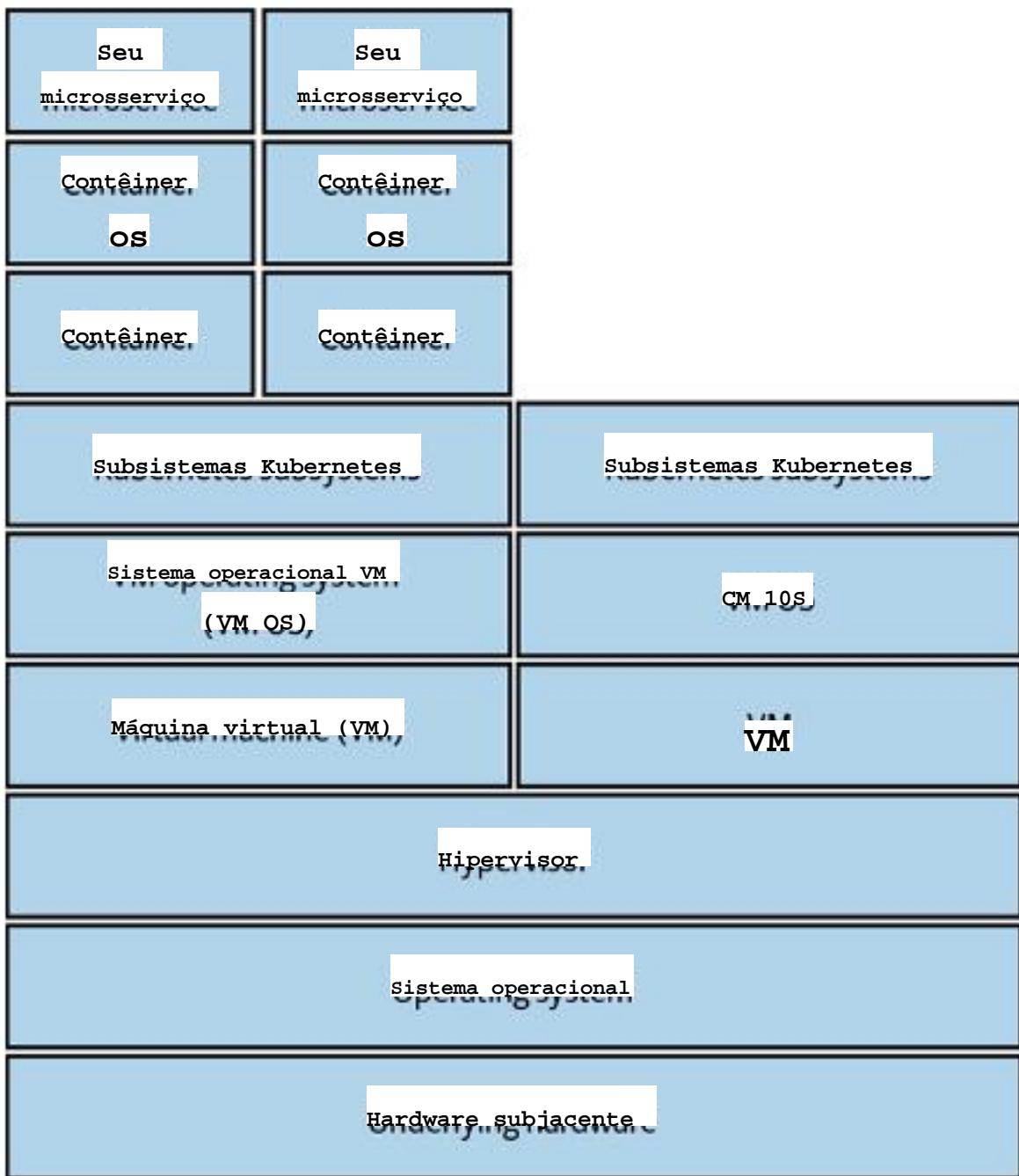


Figura 11-3. As diferentes camadas de uma infraestrutura moderna que exigem manutenção e

aplicação de patches

Se você fosse usar um cluster Kubernetes gerenciado em um dos principais públicos fornecedores de nuvem, por exemplo, você reduziria drasticamente seu escopo de propriedade, como vemos na Figura 11-4.

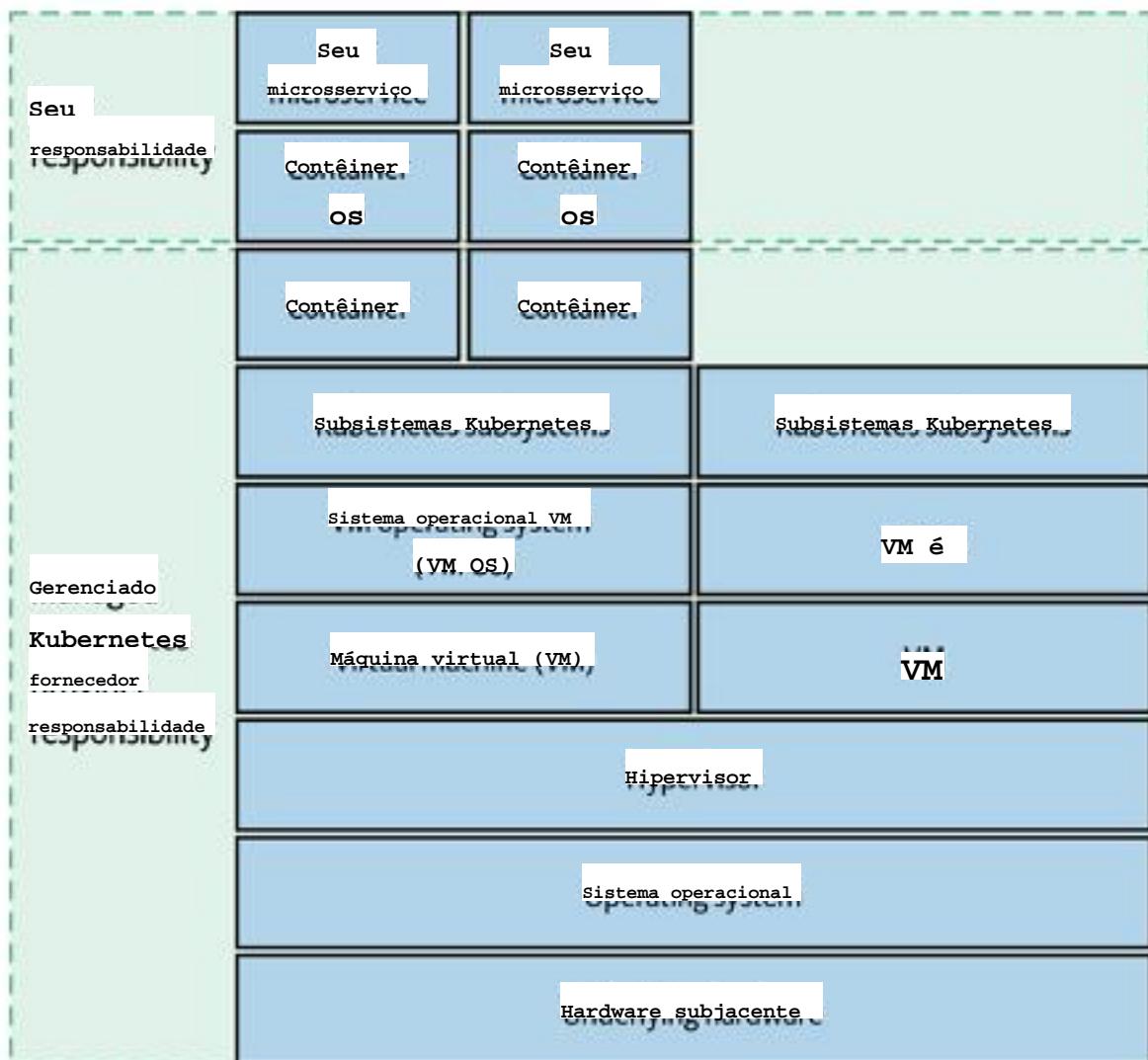


Figura 11-4. A transferência da responsabilidade por algumas camadas dessa pilha pode reduzir a complexidade.

Os contêineres nos oferecem uma bola curva interessante aqui. Tratamos um determinado contêiner instância como imutável. Mas um contêiner contém não apenas nosso software, mas também um sistema operacional. E você sabe de onde veio esse contêiner? Os contêineres são baseados em uma imagem, que por sua vez pode estender outras imagens - tem certeza de que as imagens básicas que você está usando ainda não têm backdoors? Se você não alterou uma instância de contêiner em seis meses, são seis meses de patches do sistema operacional que não foram aplicados. Mantendo além disso, é problemático, e é por isso que empresas como a Aqua fornecem ferramentas para ajudá-lo a analisar seus contêineres de produção em execução para que você possa entender quais problemas precisam ser resolvidos.

No topo desse conjunto de camadas, é claro, está o código do nosso aplicativo. É isso atualizado? Não é apenas o código que escrevemos; e o código de terceiros que escrevemos? usar? Um bug em uma biblioteca de terceiros pode deixar nosso aplicativo vulnerável a ataque. No caso da violação da Equifax, a vulnerabilidade não corrigida foi na verdade, no Struts - uma estrutura web Java.

Em grande escala, descobrindo com quais microserviços estão vinculados às bibliotecas vulnerabilidades conhecidas podem ser incrivelmente difíceis. Esta é uma área na qual eu recomendo fortemente o uso de ferramentas como o escaneamento de código Snyk ou GitHub, que é capaz de escanear automaticamente suas dependências de terceiros e alertar você, se estiver vinculando a bibliotecas com vulnerabilidades conhecidas. Se encontrar alguma, ele pode enviar a você uma solicitação de pull para ajudar a atualizar para as versões mais recentes com patches. Você pode até mesmo incorporar isso ao seu processo de CI e criar um microserviço falhe se for vinculado a bibliotecas com problemas.

Backups

Então, às vezes acho que fazer backups é como usar fio dental, muito mais as pessoas dizem que fazem isso do que realmente fazem. Não sinto a necessidade de reafirmar o argumento a favor de backups é demais aqui, além de dizer: você deve pegar backups, porque os dados são valiosos e você não quer perdê-los.

Os dados são mais valiosos do que nunca, mas às vezes me pergunto se melhorias na tecnologia fizeram com que despriorizássemos os backups. Discos são mais confiáveis do que costumavam ser. É mais provável que os bancos de dados tenham replicação integrada para evitar a perda de dados. Com esses sistemas, podemos convencer nós mesmos que não precisamos de backups. Mas e se ocorrer um erro catastrófico? e todo o seu grupo de Cassandra está extermínado? Ou se for um bug de codificação significa que seu aplicativo realmente exclui dados valiosos? Os backups são como importante como sempre. Então, por favor, faça backup de seus dados críticos.

Com a implantação de nossos microserviços sendo automatizada, não precisamos faça backups completos da máquina, pois podemos reconstruir nossa infraestrutura a partir da fonte código. Portanto, não estamos tentando copiar o estado de máquinas inteiras; em vez disso, direcione nossos backups para o estado mais valioso. Isso significa que nosso foco para backups é limitado a coisas como os dados em nossos bancos de dados, ou talvez nossos

registros de aplicativos. Com a tecnologia certa de sistema de arquivos, é possível clones quase instantâneos em nível de bloco dos dados de um banco de dados sem serem notados interrompendo o serviço.

EVITE O BACKUP DO SCHRODINGER

Ao criar backups, você quer evitar o que eu chamo de backup Schrodinger.⁹ Este é um realmente não sei se é realmente um uso de backup,¹⁰ ou se é apenas um monte de 1s e 0s escritos para o disco. A melhor maneira de evitar esse problema é garantir que o backup seja real, na verdade, restaurando-o. Encontre maneiras de criar uma restauração regular de backups em seu software processo de desenvolvimento, por exemplo, usando backups de produção para criar seus dados de teste de desempenho.

A orientação "antiga" sobre backups é que eles devem ser mantidos fora do local, o que significa que um incidente em seus escritórios ou data center não afetaria seus backups se estivessem em outro lugar. O que significa "fora do site", porém, se o seu aplicativo é implantado em uma nuvem pública? O que importa é que seus backups são armazenados de forma que fiquem o mais isolados possível do seu núcleo sistemas, para que um comprometimento no sistema central também não coloque seus backups em risco. Os Code Spaces, que mencionamos anteriormente, tinham backups mas eles foram armazenados na AWS na mesma conta que foi comprometida. Se seu aplicativo é executado na AWS, você ainda pode armazenar seus backups lá também, mas você deve fazer isso em uma conta separada em recursos de nuvem separados - você pode até querer considerar colocá-los em uma região de nuvem diferente para mitigar o risco de um problema em toda a região, ou você pode até mesmo armazená-los com outro provedor.

Portanto, certifique-se de fazer backup de dados críticos, mantenha esses backups em um sistema separado do seu ambiente de produção principal e garanta os backups na verdade, funcionam restaurando-os regularmente.

Reconstruir

Podemos fazer o nosso melhor para garantir que uma pessoa mal-intencionada não tenha acesso ao nosso sistemas, mas o que acontece se eles acontecerem? Bem, muitas vezes a coisa mais importante

o que você pode fazer no início é colocar o sistema em funcionamento novamente, mas de forma que você tenha removido o acesso da parte não autorizada. Isso No entanto, nem sempre é simples. Lembro-me de uma de nossas máquinas ser hackeado há muitos anos por um rootkit. Um rootkit é um pacote de software que é projetado para ocultar as atividades de uma parte não autorizada, e é uma técnica comumente usada por atacantes que querem permanecer sem serem detectados, permitindo que eles hora de explorar o sistema. Em nosso caso, descobrimos que o rootkit foi alterado comandos principais do sistema, como ls (listando arquivos) ou ps (para mostrar listagens de processos) para ocultar tracos do atacante externo. Nós vimos isso apenas quando estávamos capaz de verificar os hashes dos programas em execução na máquina em comparação com o pacotes oficiais. No final, basicamente tivemos que reinstalar todo o servidor do zero

A capacidade de simplesmente eliminar a existência de um servidor e reconstruí-lo totalmente pode ser incrivelmente eficaz não apenas na sequência de um ataque conhecido, mas também em termos de reduzir o impacto de atacantes persistentes. Você pode não estar ciente da presença de uma parte maliciosa em seu sistema, mas se você estiver rotineiramente reconstruindo seus servidores e alternando credenciais, você pode estar drasticamente limitando o impacto do que eles podem fazer sem que você perceba.

Sua capacidade de reconstruir um determinado microsserviço, ou até mesmo um sistema inteiro, vem até a qualidade de sua automação e backups. Se você puder implantar e configure cada microsserviço do zero com base nas informações armazenadas no controle de origem, você começou bem. Claro, você precisa combinar isso com um processo sólido de restauração de backup dos dados. Assim como acontece com os backups, a melhor maneira de garantir a implantação e configuração automatizadas de seu o funcionamento dos microsserviços é fazer isso muitas vezes, e a maneira mais fácil de conseguir isso é apenas para usar o mesmo processo para reconstruir um microsserviço que você usa para cada implantação. Obviamente, essa é a maioria das implantações baseadas em contêineres os processos funcionam. Você implanta um novo conjunto de contêineres executando a nova versão do seu microsserviço e desligue o conjunto antigo. Tornando isso normal O procedimento faz com que uma reconstrução seja quase um não-evento.

Há uma ressalva aqui, especialmente se você estiver implantando em um contêiner plataforma como o Kubernetes. Você pode estar se afastando com frequência e reimplantando instâncias de contêiner, mas o contêiner subjacente

plataforma em si? Você tem a capacidade de reconstruir isso do zero? Se você é usando um provedor Kubernetes totalmente gerenciado, criar um novo cluster pode não é muito difícil, mas se você mesmo instalou e gerencia o cluster, então isso pode ser uma quantidade não trivial de trabalho.

GORJETA

Ser capaz de reconstruir seu microserviço e recuperar seus dados de forma automatizada ajuda você se recupera após um ataque e também tem a vantagem de fazer suas implantações mais fáceis em todos os setores, com benefícios positivos para desenvolvimento, teste e atividades de operações de produção.

Confiança implícita versus confiança zero

Nossa arquitetura de microserviços consiste em muita comunicação entre coisas. Usuários humanos interagem com nosso sistema por meio de interfaces de usuário. Esses usuários fazem chamadas para microservices, e os microservices acabam chamando ainda mais microservices. Quando se trata de segurança de aplicativos, nós precisamos considerar a questão da confiança entre todos esses pontos de contato. Como fazer estabelecermos um nível aceitável de confiança? Exploraremos esse tópico em breve em termos de autenticação e autorização de humanos e microservices, mas antes disso, devemos considerar alguns modelos fundamentais em torno da confiança.

Confiamos em tudo o que está rodando em nossa rede? Ou nós vemos tudo com suspeita? Aqui, podemos considerar duas mentalidades: confiança implícita e zero confiança.

Confiança implícita

Nossa primeira opção pode ser simplesmente presumir que qualquer chamada para um serviço é feita a partir de dentro do nosso perímetro são implicitamente confiáveis.

Dependendo da sensibilidade dos dados, isso pode ser bom. Alguns organização tentam garantir a segurança no perímetro de suas redes, e, portanto, presumem que não precisam fazer mais nada quando dois

os serviços estão conversando juntos. No entanto, se um atacante penetrar em seu rede, todo o inferno poderia começar. Se o atacante decidir interceptar e leia os dados que estão sendo enviados, altere os dados sem você saber, ou mesmo em algumas circunstâncias fingem ser a coisa com a qual você está falando, você pode não sei muito sobre isso.

Essa é de longe a forma mais comum de confiança dentro do perímetro que vejo em organizações. Eu não estou dizendo que isso é uma coisa boa! Para a maioria dos organizações que vejo usando esse modelo, me preocupo que o modelo de confiança implícito seja não é uma decisão consciente; ao contrário, as pessoas não estão cientes dos riscos na primeira lugar.

Confiança zero

"Jill, rastreamos a ligação - ela vem de dentro da casa!"

-Quando um estranho liga

Ao operar em um ambiente de confiança zero, você deve presumir que está operando em um ambiente que já foi comprometido - os computadores com os quais você está falando poderiam ter sido comprometidos, a entrada as conexões podem ser de partes hostis, os dados que você está escrevendo podem ser lido por pessoas ruins. Paranóico? Sim! Bem-vindo ao Zero Trust.

Confiança zero, fundamentalmente, é uma mentalidade. Não é algo que você possa mágicamente implemente usando um produto ou ferramenta; é uma ideia e essa ideia é que, se você opere sob a suposição de que você está operando em um ambiente hostil em que os malfeitores já poderiam estar presentes, então você tem que realizar precauções para garantir que você ainda possa operar com segurança. Com efeito, o conceito de "perímetro" não tem sentido com confiança zero (por esse motivo, zero a confiança geralmente também é conhecida como "computação sem perímetro").

Como você presume que seu sistema foi comprometido, todas as chamadas recebidas de outros microsserviços devem ser avaliados adequadamente. Isso é realmente um cliente? Eu deveria confiar? Da mesma forma, todos os dados devem ser armazenados com segurança e toda criptografia teclas seguradas com segurança e, como devemos presumir que alguém está ouvindo, tudo dados confidenciais em trânsito em nosso sistema precisam ser criptografados.

Curiosamente, se você implementou adequadamente uma mentalidade de confiança zero, você pode comece a fazer coisas que parecem bem estranhas:

[Com confiança zero], você pode realmente fazer certos acessos contra-intuitivos decisões e, por exemplo, permitir conexões com serviços internos de a internet porque você trata sua rede "interna" da mesma forma confiável como a internet (ou seja, nem um pouco).

-Jan Schaumann 11

O argumento de Jan aqui é que, se você assumir que nada dentro da sua rede é para ser confiável, e essa confiança precisa ser restabelecida, você pode ser muito mais flexível quanto ao ambiente em que seu microsserviço vive - você não é esperando que o ambiente mais amplo seja seguro. Mas lembre-se, confiança zero não é algo que você liga com um interruptor. É um princípio subjacente de como você decide fazer coisas. Tem que orientar sua tomada de decisão sobre como você constrói e evolua seu sistema - será algo que você precisará invista constantemente para obter as recompensas.

É um espectro

Não quero dizer que você tenha uma escolha difícil entre implícito e zero. confiança. Até que ponto você confia (ou não) em outras partes do seu sistema pode mudar com base na sensibilidade das informações acessadas. Você pode decidir, por exemplo, adotar um conceito de confiança zero para qualquer microsserviços lidam com PII, mas fique mais tranquilo em outras áreas. Mais uma vez, o custo de qualquer implementação de segurança deve ser justificado (e impulsionado) por sua modelo de ameaça. Permita que você compreenda suas ameaças e as ameaças associadas o impacto impulsiona sua tomada de decisão sobre se vale ou não a confiança zero é para você.

Como exemplo, vamos dar uma olhada na MedicalCo, uma empresa com a qual trabalhei gerenciou dados confidenciais de saúde pertencentes a indivíduos. Todas as informações foi considerado classificado com base em uma abordagem bastante sensata e direta abordagem:

Público

Dados que podem ser compartilhados livremente com qualquer parte externa. Esta informação está efetivamente no domínio público.

Privado

Informações que devem estar disponíveis somente para usuários conectados. Acesso a essas informações podem ser ainda mais limitadas devido às restrições de autorização. Isso pode incluir coisas como qual plano de seguro um cliente era ligado.

Segredo

Informações incrivelmente confidenciais sobre indivíduos que poderiam ser acessadas por pessoas que não sejam o indivíduo em questão, apenas em casos extremamente específicos de pessoas que precisam saber essas informações em certas situações. Isso inclui informações sobre os dados de saúde de um indivíduo.

Os microserviços foram então categorizados com base nos dados mais confidenciais que eles usavam e teve que ser executado em um ambiente correspondente (zona) com controles correspondentes, como vemos na Figura 11-5. Um microserviço teria que ser executado na zona combinando os dados mais confidenciais que utilizou. Por exemplo, um microserviço em execução na zona pública pode usar somente dados públicos. Por outro lado, um microserviço que usava dados públicos e privados tinha que ser executado na zona privada, e um microserviço que acessava informações secretas sempre tinha que ser executado na zona secreta.

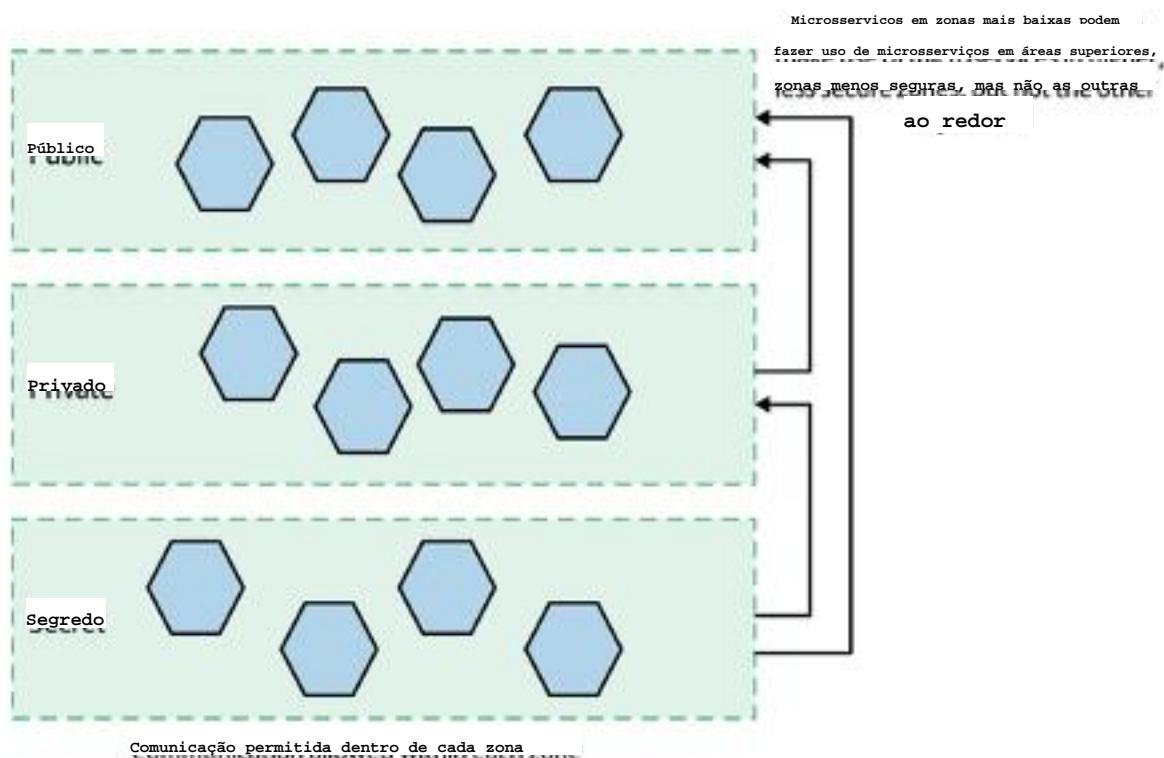


Figura 11-5. Implantando microserviços em diferentes zonas com base na sensibilidade dos dados

Os microserviços dentro de cada zona podiam se comunicar uns com os outros, mas eram incapazes de acessar diretamente dados ou funcionalidades na parte inferior das zonas mais seguras. Os microserviços nas zonas mais seguras podem alcançar até mesmo funcionalidades que só existem nas zonas menos seguras.

Aqui, a MedicalCo se deu a flexibilidade de variar sua abordagem em cada zona. A zona pública menos segura pode operar em algo mais próximo de um ambiente de confiança implícito, enquanto a zona secreta pressupõe confiança zero. Provavelmente, se a MedicalCo adotasse uma abordagem de confiança zero em toda a sua sistema, ter microserviços implantados em zonas separadas não seria obrigatório, pois todas as chamadas entre microserviços exigiriam mais autenticação e autorização. Dito isso, pensando em defesa em profundidade uma vez novamente, não consigo deixar de pensar que ainda consideraria essa abordagem zoneada, dada a sensibilidade dos dados!

Protegendo dados

À medida que dividimos nosso software monolítico em microserviços, nossos dados circula em nossos sistemas mais do que antes. Não flui simplesmente redes; ele também fica no disco. Ter dados mais valiosos espalhados por mais lugares podem ser um pesadelo quando se trata de proteger nosso aplicativo, se não são cuidadosos. Vamos analisar com mais detalhes como podemos proteger nossos dados, pois se move pelas redes e fica em repouso.

Dados em trânsito

A natureza das proteções que você tem dependerá em grande parte da natureza dos protocolos de comunicação que você escolheu. Se você estiver usando HTTP, para exemplo, seria natural considerar o uso de HTTP com camada de transporte Segurança (TLS), um tópico que abordaremos mais na próxima seção; mas se você está usando protocolos alternativos, como comunicação por meio de uma mensagem corretor, talvez seja necessário analisar o suporte dessa tecnologia específica para protegendo dados em trânsito. Em vez de analisar os detalhes de uma grande variedade de tecnologia neste espaço, acho que, em vez disso, é importante considerar mais genericamente, as quatro principais áreas de interesse quando se trata de proteger dados em trânsito e para ver como essas preocupações podem ser tratadas com HTTP como um exemplo. Espero que não seja muito difícil para você mapear essas ideias para qualquer protocolo de comunicação que você esteja escolhendo.

Na Figura 11-6, podemos ver as quatro principais preocupações dos dados em trânsito.

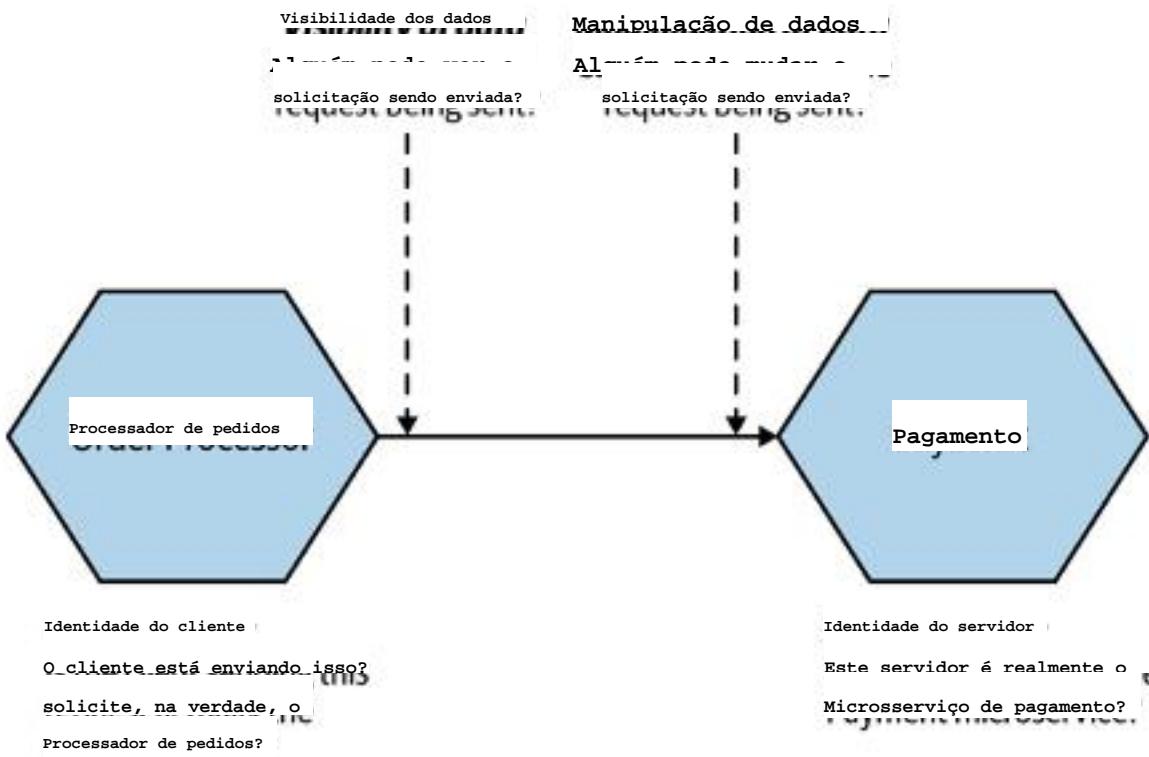


Figura 11-6. As quatro principais preocupações quando se trata de dados em trânsito

Vamos analisar cada preocupação com um pouco mais de detalhes.

Identidade do servidor

Uma das coisas mais simples de verificar é se o servidor com o qual você está falando é exatamente quem afirma ser. Isso é importante porque uma pessoa mal-intencionada teoricamente, poderia se passar por um endpoint e aspirar quaisquer dados úteis que você envia. A validação da identidade do servidor sempre foi uma preocupação no internet pública, levando à pressão por um uso mais difundido de HTTPS e até certo ponto, podemos nos beneficiar do trabalho realizado para ajudar a proteger o internet pública quando se trata de gerenciar endpoints HTTP internos.

com TLS. 12 Com a maioria das comunicações pela Internet pública, devido à vários vetores de ataque em potencial (Wi-Fi não seguro, envenenamento por DNS, e afins), é vital garantir que, quando acessamos um site, seja realmente o site que afirma ser. Com HTTPS, nosso navegador pode ver o certificado para esse site e verifique se ele é válido. É uma segurança altamente sensata

Mecanismo - "HTTPS Everywhere" tornou-se um grito de guerra para o público internet, e por um bom motivo.

É importante notar que alguns protocolos de comunicação que usam HTTP sob o hood pode tirar proveito do HTTPS - para que possamos executar facilmente SOAP ou gRPC por HTTPS sem problemas. O HTTPS também nos oferece proteções adicionais sobre e acima, simplesmente confirmando que estamos falando com quem esperamos. Nós vamos chegar a isso em breve.

Identidade do cliente

Quando nos referimos à identidade do cliente nesse contexto, estamos nos referindo ao microsserviço fazendo uma chamada, então estamos tentando confirmar e autenticar o identidade do microsserviço upstream. Vamos ver como autenticar humanos (usuários!) um pouco mais tarde.

Podemos verificar a identidade de um cliente de várias maneiras. Poderíamos perguntar isso o cliente nos envia algumas informações na solicitação, informando quem eles são. Um exemplo pode ser usar algum tipo de segredo compartilhado ou um certificado do lado do cliente para assinar a solicitação. Quando o servidor precisa verificar a identidade do cliente, queremos para ser o mais eficiente possível - eu vi algumas soluções (incluindo aquelas fornecidos por fornecedores de gateway de API (fornecedores) que envolvem a necessidade de o servidor fazer chamadas aos serviços centrais para verificar a identidade do cliente, o que é muito louco quando você considere as implicações de latência.

Tenho dificuldade em pensar em uma situação na qual eu verificararia a identidade do cliente sem também verificar a identidade do servidor - para verificar ambos, você normalmente faria acabam implementando alguma forma de autenticação mútua. Com mútuo autenticação, ambas as partes se autenticam mutuamente. Então, na Figura 11-6, o O Processador de Pedidos autentica o microsserviço de pagamento e o pagamento o microsserviço autentica o processador de pedidos

Podemos fazer isso por meio do uso de TLS mútuo; nesse caso, tanto o cliente e o servidor faz uso de certificados. Na internet pública, verificando o a identidade de um dispositivo cliente geralmente é menos importante do que verificar o identidade do ser humano usando esse dispositivo. Dessa forma, o TLS mútuo raramente é usado. Em

nossa arquitetura de microserviços, no entanto, especialmente onde podemos estar operando em um ambiente de confiança zero, isso é muito mais comum.

Historicamente, o desafio de implementar esquemas como o TLS mútuo estava trabalhando com ferramentas. Hoje em dia, isso é menos problemático. Ferramentas como o Vault podem fazer distribuindo certificados com muito mais facilidade e querendo simplificar o uso de. O TLS mútuo é um dos principais motivos para as pessoas implementarem o serviço malhas, que exploramos em "Service Meshes and API Gateways".

Visibilidade dos dados

Quando enviamos dados de um microserviço para outro, alguém pode ver os dados? Para obter algumas informações, como o preço dos álbuns de Peter Andre, nós pode não se importar muito, pois os dados já estão no domínio público. Sobre o por outro lado, alguns dados podem incluir PII, que precisamos ter certeza de que é protegido.

Quando você usa HTTPS simples e antigo ou TLS mútuo, os dados não ficam visíveis para partes intermediárias - isso ocorre porque o TLS criptografa os dados que estão sendo enviados. Isso pode ser problemático se você quiser explicitamente que os dados sejam enviados de forma aberta exemplo, proxies reversos como Squid ou Varnish são capazes de armazenar HTTP em cache respostas, mas isso não é possível com HTTPS.

Manipulação de dados

Podemos imaginar várias situações em que a manipulação de dados seja enviada pode alterar incorretamente a quantidade de dinheiro enviada, por exemplo.

Portanto, na Figura 11-6, precisamos garantir que o atacante em potencial não consiga altere a solicitação que está sendo enviada ao Pagamento pelo Processador de Pedidos.

Normalmente, os tipos de proteções que tornam os dados invisíveis também garantirão que os dados não podem ser manipulados (o HTTPS faz isso, por exemplo). No entanto, poderíamos decidir enviar dados ao ar livre, mas ainda queremos garantir que não possam ser manipulado. Para HTTP, uma dessas abordagens é usar uma mensagem baseada em hash código de autenticação (HMAC) para assinar os dados que estão sendo enviados. Com o HMAC, o hash é gerado e enviado com os dados, e o receptor pode verificar o hash contra os dados para confirmar que os dados não foram alterados.

Dados em repouso

Mentir sobre dados é uma responsabilidade, especialmente se forem confidenciais. Espero que tenhamos feito tudo o que pudemos para garantir que os invasores não violem nossa rede e também que eles não podem violar nossos aplicativos ou sistemas operacionais para obter acesso aos dados subjacentes. No entanto, precisamos estar preparados caso eles façam defesa em profundidade é fundamental.

Muitas das violações de segurança de alto perfil que ouvimos falar envolvem dados em repouso sendo adquiridos por um atacante e esses dados sendo legíveis pelo atacante. Isso acontece porque os dados foram armazenados em um formato não criptografado ou porque o mecanismo usado para proteger os dados tinha uma falha fundamental. Os mecanismos pelos quais os dados em repouso podem ser protegidos são muitos e variados, mas há algumas coisas gerais a serem lembradas.

Escolha o mais conhecido

Em alguns casos, você pode transferir o trabalho de criptografar dados para o software existente - por exemplo, usando o suporte integrado do seu banco de dados para criptografia. No entanto, se você achar necessário criptografar e descriptografar dados em seu sistema próprio, certifique-se de usar um sistema conhecido e testado implementações. A maneira mais fácil de baguncar a criptografia de dados é tentar implementar seus próprios algoritmos de criptografia ou até mesmo tentar de outra pessoa. Qualquer que seja a linguagem de programação que você usa, você terá acesso a implementações revisadas e regularmente corrigidas de criptografia conceituada algoritmos. Use aqueles! E assine as listas de endereço/listas consultivas para a tecnologia que você escolhe para garantir que você esteja ciente das vulnerabilidades, como eles foram encontrados, para que você possa mantê-los corrigidos e atualizados.

Para proteger senhas, você deve absolutamente usar uma técnica chamada hashing de senha salgado. Isso garante que as senhas nunca sejam mantidas texto simples e que mesmo que um atacante force brutalmente uma senha com hash, ele não pode então ler automaticamente outras senhas. 13

Criptografia mal implementada pode ser pior do que não ter nenhuma, pois a falsa sensação de segurança (perdoe o trocadilho) pode levar você a tirar os olhos da bola.

Escolha seus alvos

Assumir que tudo deve ser criptografado pode simplificar um pouco as coisas.

Não há suposições sobre o que deve ou não ser protegido.

No entanto, você ainda precisará pensar sobre quais dados podem ser colocados em arquivos de log para ajuda na identificação de problemas e na sobrecarga computacional de criptografar tudo pode se tornar muito oneroso e exigir mais poder hardware como resultado. Isso é ainda mais desafiador quando você está se inscrevendo migrações de banco de dados como parte dos esquemas de refatoração. Dependendo do alterações que estão sendo feitas, os dados podem precisar ser descriptografados, migrados, e criptografado novamente.

Ao subdividir seu sistema em serviços mais refinados, você pode

identifique um armazenamento de dados inteiro que possa ser criptografado por atacado, mas isso é improvável. Limitar essa criptografia a um conjunto conhecido de tabelas é sensato. abordagem.

Seja frugal

À medida que o espaço em disco se torna mais barato e os recursos dos bancos de dados melhoram, a facilidade com que grandes quantidades de informações podem ser capturadas e armazenadas está melhorando rapidamente. Esses dados são valiosos, não apenas para empresas eles mesmos, que cada vez mais veem os dados como um ativo valioso, mas igualmente os usuários que valorizam sua própria privacidade. Os dados que pertencem a um indivíduo ou que possa ser usado para obter informações sobre um indivíduo deve ser o dados com os quais temos muito cuidado.

No entanto, e se tornássemos nossas vidas um pouco mais fáceis? Por que não esfregar tanto informações quanto possível que possam ser pessoalmente identificáveis, e faça-o o mais rápido possível quanto possível? Ao registrar uma solicitação de um usuário, precisamos armazenar o endereço IP inteiro para sempre, ou poderíamos substituir os últimos dígitos por x? Faça precisamos armazenar o nome, a idade, o sexo e a data de nascimento de alguém para fornecer-lhes ofertas de produtos, ou é sua faixa etária e código postal informação suficiente?

As vantagens de ser frugal com a coleta de dados são múltiplas. Primeiro, se você não o guarde, ninguém pode roubá-lo. Em segundo lugar, se você não o armazenar, ninguém (por exemplo, um

agência governamental) também pode solicitá-lo!

A frase alemã Datensparsamkeit representa esse conceito. Originando da legislação alemã de privacidade, ele encapsula o conceito de armazenar apenas o máximo de informações que sejam absolutamente necessárias para realizar as operações comerciais ou cumprir as leis locais.

Obviamente, isso está em tensão direta com o movimento de armazenar mais e mais informações, mas perceber que essa tensão ainda existe é um bom começo!

Tudo gira em torno das chaves

A maioria das formas de criptografia envolve o uso de alguma chave em conjunto com um algoritmo adequado para criar dados criptografados. Para descriptografar os dados, então pode ser lido, as partes autorizadas precisarão acessar uma chave, seja a mesma chave ou uma chave diferente (no caso de criptografia de chave pública). Então, onde estão? Suas chaves estão armazenadas? Agora, se estou criptografando meus dados porque estou preocupado sobre alguém roubando todo o meu banco de dados e eu armazeno a chave que uso no mesmo banco de dados, então eu realmente não consegui muito! Portanto, precisamos guardar as chaves em outro lugar. Mas onde?

Uma solução é usar um dispositivo de segurança separado para criptografar e descriptografar dados. Outra é usar um cofre de chaves separado que seu serviço possa acessar quando precisa de uma chave. O gerenciamento do ciclo de vida das chaves (e o acesso à mudança) eles) pode ser uma operação vital, e esses sistemas podem lidar com isso para você. É aqui que o cofre da HashiCorp também pode ser muito útil.

Alguns bancos de dados incluem até mesmo suporte embutido para criptografia, como SQL Criptografia de dados transparente do servidor, que visa lidar com isso em um modo transparente. Mesmo que seu banco de dados de escolha inclua tal suporte, pesquise como as chaves são tratadas e entenda se a ameaça você está se protegendo contra o fato de estarem sendo mitigados.

Novamente, essas coisas são complexas. Evite implementar sua própria criptografia e faça uma boa pesquisa!

GORJETA

Criptografe dados quando você os vê pela primeira vez. Descriptografe somente sob demanda e garanta que os dados nunca existam armazenado em qualquer lugar.

Criptografe backups

Os backups são bons. Queremos fazer backup de nossos dados importantes. E pode parecer como um ponto óbvio, mas se os dados forem sensíveis o suficiente para que queiramos que sejam criptografados em nosso sistema de produção em execução, então provavelmente também desejaremos para garantir que todos os backups dos mesmos dados também sejam criptografados!

Autenticação e autorização

Autenticação e autorização são conceitos fundamentais quando se trata de pessoas e coisas que interagem com nosso sistema. No contexto da segurança, autenticação é o processo pelo qual confirmamos que uma parte é quem ela dizem que são. Normalmente autenticamos um usuário humano fazendo com que ele digite seu nome de usuário e senha. Assumimos que somente o usuário real tem acesso a essas informações e, portanto, a pessoa que insere essas informações deve ser eles. Outros sistemas mais complexos também existem, é claro. Nossos telefones agora vamos usar nossa impressão digital ou rosto para confirmar que somos quem dizemos ser. Geralmente, quando estamos falando abstratamente sobre quem ou o que está sendo autenticado, nos referimos a essa parte como a principal.

A autorização é o mecanismo pelo qual mapeamos de um diretor para ação que estamos permitindo que eles façam. Muitas vezes, quando um diretor é autenticado, receberemos informações sobre eles que nos ajudarão a decidir o que devemos deveria deixá-los fazer. Podemos, por exemplo, saber qual departamento ou escritório eles trabalham em uma informação que nosso sistema pode usar para decidir o que o diretor pode e não pode fazer.

A facilidade de uso é importante - queremos facilitar o acesso de nossos usuários nosso sistema. Não queremos que todos precisem fazer login separadamente para acessar microserviços diferentes, usando um nome de usuário e senha diferentes para cada.

um. Portanto, também precisamos ver como podemos implementar o login único (SSO) em um ambiente de microserviços.

Autenticação de serviço a serviço

Anteriormente, discutimos o TLS mútuo, que, além de proteger dados em trânsito também nos permite implementar uma forma de autenticação. Quando um cliente fala com um servidor usando TLS mútuo, o servidor é capaz de autenticar o cliente, e o cliente é capaz de autenticar o servidor - esta é uma forma de autenticação de serviço a serviço. Outros esquemas de autenticação podem ser usados além do TLS mútuo. Um exemplo comum é o uso de chaves de API, em que o cliente precisa usar a chave para fazer o hash de uma solicitação de forma que o servidor seja capaz de verificar se o cliente usou uma chave válida.

Autenticação humana

Estamos acostumados a que os humanos se autentiquem com o familiar combinação de nome de usuário e senha. Cada vez mais, porém, isso está sendo usado como parte de uma abordagem de autenticação multifatorial, na qual um usuário pode precisar de mais do que um conhecimento (um fator) para se autenticarem. A maioria comumente, isso assume a forma de autenticação multifatorial (MFA), 14 onde mais de um fator é necessário. O MFA envolveria mais comumente o uso de uma combinação normal de nome de usuário e senha, além de fornecer pelo menos um fator adicional.

Os diferentes tipos de fatores de autenticação cresceram nos últimos anos de códigos enviados por SMS e links mágicos enviados por e-mail a dedicados aplicativos móveis como Authy e dispositivos de hardware USB e NFC como o YubiKey. Os fatores biométricos também são mais comumente usados agora, como usuários ter mais acesso ao hardware que suporta coisas como impressão digital ou rosto reconhecimento. Embora o MFA tenha se mostrado muito mais seguro como abordagem geral, e muitos serviços públicos a apoiam, ela não se popularizou um esquema de autenticação de mercado de massa, embora eu espere que isso mude. Para gerenciando a autenticação dos principais serviços que são vitais para o funcionamento de seu

software ou permitir o acesso a informações especialmente confidenciais (por exemplo, fonte de código aberto), eu consideraria o uso do MFA obrigatório.

Implementações comuns de login único

Uma abordagem comum para autenticação é usar algum tipo de login único (SSO) para garantir que um usuário só precise se autenticar uma vez por sessão, mesmo que durante essa sessão eles possam acabar interagindo com vários serviços ou aplicativos downstream. Por exemplo, quando você faz login com sua conta do Google, você está logado no Google Calendar, Gmail e Google Docs, mesmo que sejam sistemas separados.

Quando um diretor tenta acessar um recurso (como uma interface baseada na web), eles são direcionados para se autenticar com um provedor de identidade. A identidade o provedor pode solicitar que eles forneçam um nome de usuário e senha ou pode exigir o uso de algo mais avançado, como o MFA. Quando o provedor de identidade estiver satisfeito com o fato de o diretor ter sido autenticado, ele fornece informações ao prestador de serviços, permitindo que ele decida se deve conceder-lhes acesso ao recurso.

Esse provedor de identidade pode ser um sistema hospedado externamente ou algo assim dentro de sua própria organização. O Google, por exemplo, fornece um OpenID Connect o provedor de identidade. Para empresas, porém, é comum ter seu próprio provedor de identidade próprio, que pode estar vinculado ao diretório da sua empresa. Um serviço de diretório pode ser algo como o Lightweight Protocol de acesso a diretórios (LDAP) ou Active Directory. Esses sistemas permitem que você armazene informações sobre os diretores, como as funções que eles desempenham na organização. Freqüentemente, o serviço de diretório e o provedor de identidade são iguais, enquanto outras vezes estão separados, mas vinculados. Okta, por exemplo, é um provedor de identidade SAML hospedado que lida com tarefas como duas autenticação fatorial, mas pode se vincular aos serviços de diretório da sua empresa como a fonte da verdade.

Portanto, o provedor de identidade fornece ao sistema informações sobre quem é o diretor é, mas o sistema decide o que esse diretor pode fazer.

O SAML é um padrão baseado em SOAP e é conhecido por ser bastante complexo para trabalhar com. apesar das bibliotecas e ferramentas disponíveis para apoiá-la. e desde a primeira edição deste livro caiu rapidamente em desuso.¹⁵ OpenID Connect é um padrão que surgiu como uma implementação específica do OAuth 2.0, baseado na forma como o Google e outros lidam com o SSO. Usa de forma mais simples chamadas REST e, em parte, devido à sua relativa simplicidade e generalização suporte, é o mecanismo dominante para o SSO do usuário final e ganhou incursões significativas nas empresas.

Gateway de login único

Poderíamos decidir lidar com o redirecionamento e o aperto de mão com o provedor de identidade em cada microserviço, para que qualquer um não autenticado a solicitação de uma parte externa é tratada adequadamente. Obviamente, isso poderia significar muitas funcionalidades duplicadas em nossos microservices. Um compartilhado a biblioteca poderia ajudar, mas teríamos que ter cuidado para evitar o acoplamento que pode vêm de código compartilhado (consulte "DRY e os perigos da reutilização de código" em um Microservice World" (para mais informações). Uma biblioteca compartilhada também não ajudaria se tinha microservices escritos em diferentes pilhas de tecnologia.

Em vez de fazer com que cada serviço gerencie o aperto de mão com nossa identidade provedor, uma abordagem mais comum é usar um gateway para atuar como proxy, sentado entre seus serviços e o mundo exterior (conforme mostrado na Figura 11-7). A ideia é que possamos centralizar o comportamento para redirecionar o usuário e realize o aperto de mão em apenas um lugar.

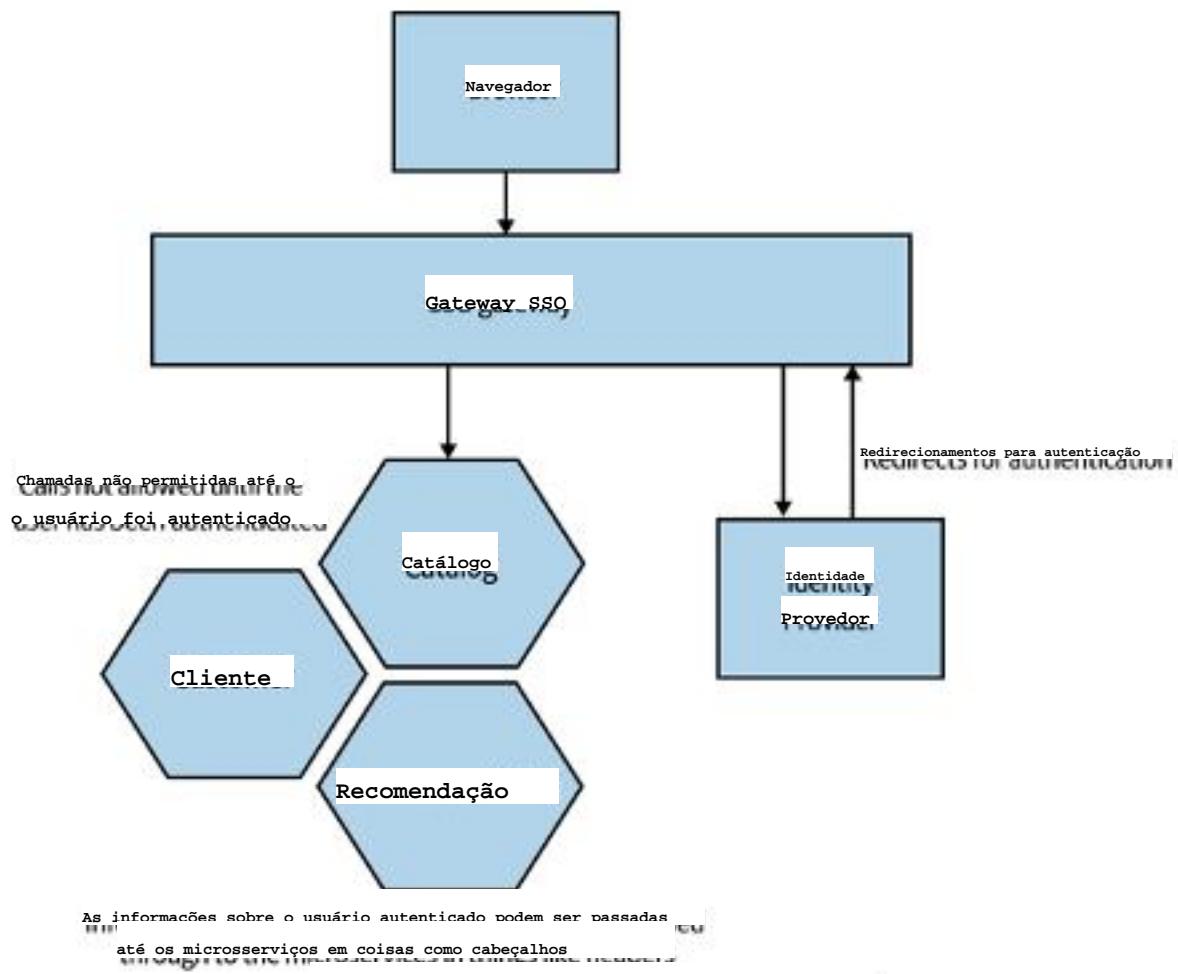


Figura 11-7. Usando um gateway para lidar com o SSO

No entanto, ainda precisamos resolver o problema de como o serviço downstream recebe informações sobre diretores, como nome de usuário ou quais funções eles jogam. Se você estiver usando HTTP, poderá configurar seu gateway para preencher cabeçalhos com essas informações. O Shibboleth é uma ferramenta que pode fazer isso para você, e eu já vi isso ser usado com o servidor web Apache para lidar com integração com provedores de identidade baseados em SAML, com grande efeito. Uma alternativa, que veremos com mais detalhes em breve, é criar um JSON Web Token (JWT) contendo todas as informações sobre o principal; isso tem uma série de benefícios, incluindo ser algo que podemos ultrapassar com mais facilidade de microserviço em microserviço.

Outra consideração ao usar um gateway de login único é que, se tivermos decidido transferir a responsabilidade pela autenticação para um gateway, pode ser mais difícil raciocinar sobre como um microserviço se comporta quando analisado em

isolamento. Lembre-se de que, no Capítulo 9, exploramos alguns dos desafios de reproduzir ambientes semelhantes aos de produção? Se você decidir usar um gateway, certifique-se de que seus desenvolvedores possam lançar seus serviços por trás de um sem precisar muito trabalho.

Um último problema com essa abordagem é que ela pode levar você a um falso sentido de segurança. Mais uma vez, gosto de voltar à ideia de defesa em profundidade - a partir de perímetro de rede para sub-rede, firewall, máquina, sistema operacional e hardware subjacente. Você tem a capacidade de implementar medidas de segurança em todos esses pontos. Eu vi algumas pessoas colocarem todos os ovos na mesma cesta, contando com o gateway para lidar com cada etapa para eles. E todos nós sabemos o que acontece quando temos um único ponto de falha...

Obviamente, você poderia usar esse gateway para fazer outras coisas. Por exemplo, você também poderia decidir encerrar o HTTPS nesse nível, executar a detecção de intrusão, e assim por diante. Mas tenha cuidado. As camadas de gateway tendem a assumir mais e mais funcionalidade, o que por si só pode acabar sendo um ponto de acoplamento gigante. E quanto mais funcionalidade algo tiver, maior será a superfície de ataque.

Autorização refinada

Um gateway pode ser capaz de fornecer granulação grossa bastante eficaz de autenticação. Por exemplo, isso pode impedir que qualquer usuário não conectado acessando o aplicativo de helpdesk. Supondo que nosso gateway possa extrair atributos sobre o principal; como resultado da autenticação, ele pode ser capaz de tomar decisões mais sutis. Por exemplo, é comum colocar pessoas em grupos ou atribuir-lhes funções. Podemos usar essas informações para entender o que eles podem fazer. Portanto, para o aplicativo de helpdesk, podemos permitir o acesso somente para diretores com uma função específica (por exemplo, EQUIPE). Além de permitir (ou proibindo) o acesso a recursos ou endpoints específicos, no entanto, precisamos deixar o resto para o próprio microserviço; ele precisará fazer suas próprias decisões sobre quais operações permitir...

Voltar ao nosso aplicativo de helpdesk: permitimos que algum membro da equipe veja algum e todos os detalhes? O mais provável é que tenhamos funções diferentes no trabalho. Por exemplo, um diretor do grupo CALL CENTER pode ter permissão para ver qualquer parte do

informações sobre um cliente, exceto seus detalhes de pagamento. Este diretor também pode emitir reembolsos, mas esse valor pode ser limitado.

Alguém que tenha a função CALL_CENTER_TEAM LEADER, no entanto, pode ser capaz de emitir reembolsos maiores.

Essas decisões precisam ser locais para o microsserviço em questão. Eu vi as pessoas usam os vários atributos fornecidos pelos provedores de identidade de forma horrível maneiras, usando funções muito refinadas, como

CALL_CENTER_50_DOLLAR_REFUND, onde eles acabam colocando informações específico para uma parte da funcionalidade de microsserviço em seu diretório serviços. É um pesadelo de manter e oferece muito pouco espaço para nossos serviços devem ter seu próprio ciclo de vida independente, pois, de repente, uma parte do informações sobre como um serviço se comporta vivem em outro lugar, talvez em um sistema gerenciado por uma parte diferente da organização,

Garantir que o microsserviço tenha as informações necessárias para avaliar melhor solicitações de autorização granuladas merecem uma discussão mais aprofundada - vamos revisitar isso quando olhamos para os JWTs em um momento

Em vez disso, favoreça funções granulares baseadas em como sua organização funciona. Voltando aos primeiros capítulos, lembre-se de que somos construindo software para combinar com o funcionamento de nossa organização. Então, use suas funções em dessa forma também.

O problema do deputado confuso

Fazer com que um diretor se autentique com o sistema como um todo usando algo como um gateway SSO é simples o suficiente, e isso pode ser suficiente para controlar acesso a um determinado microsserviço. Mas o que acontece se esse microsserviço então precisa fazer chamadas adicionais para concluir uma operação? Isso pode nos deixar aberto a um tipo de vulnerabilidade conhecido como problema do deputado confuso. Isso ocorre quando uma parte inicial engana uma parte intermediária para fazer coisas não deveria estar funcionando. Vejamos um exemplo concreto na Figura 11-8, que ilustra o site de compras on-line da MusicCorp. Nossa JavaScript baseado em navegador A interface do usuário fala com o microsserviço Web Shop do lado do servidor, que é um tipo de backend para front-end. Exploraremos isso com mais profundidade em "Pattern: Backend

para Frontend (BFF)", mas, no momento, pense nisso como um servidor componente que executa agregação e filtragem de chamadas para um externo específico interface (no nosso caso, nossa interface de usuário JavaScript baseada em navegador). Chamadas feitas entre o navegador e a Web Shop pode ser autenticado usando o OpenID Conectar. Até agora, tudo bem.

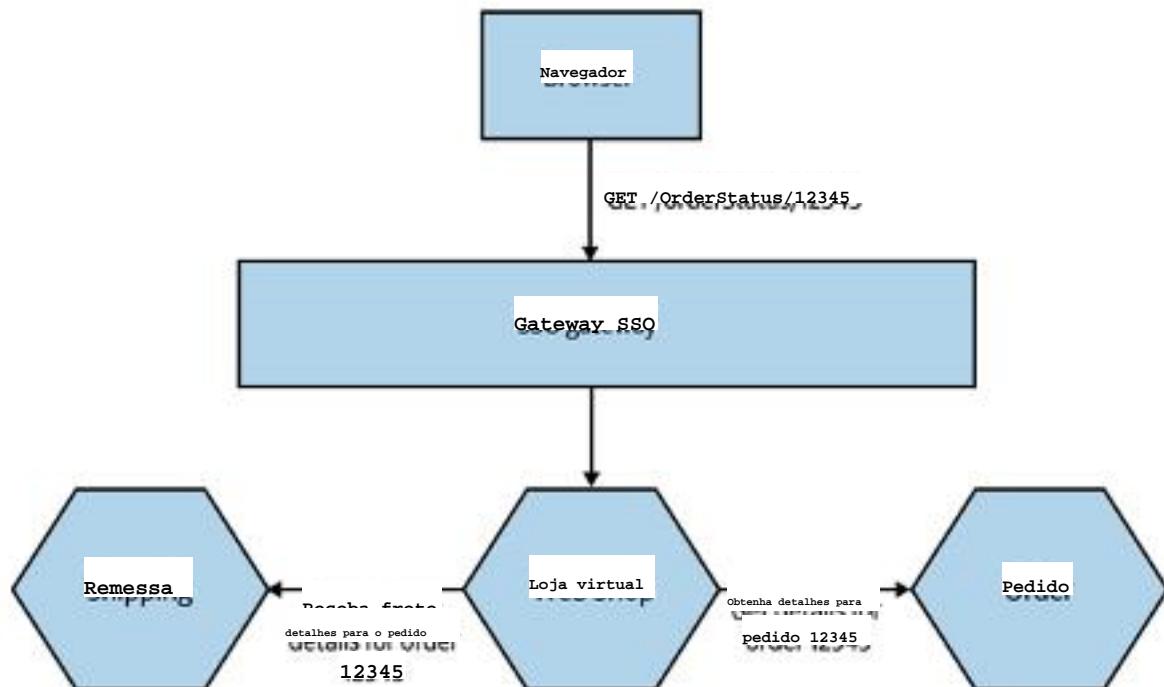


Figura 11-8 Um exemplo em que um deputado confuso pode entrar em cena.

Quando um usuário está logado, ele pode clicar em um link para ver os detalhes de um pedido. Para exibir as informações, precisamos retirar o pedido original do usuário, solicite o serviço, mas também queremos consultar as informações de envio do pedido. Então, quando um cliente conectado clica em um link para OrderStatus/12345, essa solicitação é encaminhada para a Web Shop, que então precisa fazer chamadas para os microserviços de pedido e envio, a jusante solicitando os detalhes de pedido 12345.

Mas esses serviços downstream devem aceitar as chamadas da Web Shop? Poderíamos adotar uma postura de confiança implícita: porque a chamada veio de dentro de nosso perímetro, está tudo bem. Poderíamos até usar certificados ou chaves de API para confirmar que realmente é a loja virtual solicitando essas informações. Mas isso é suficiente? Por exemplo, um cliente que está logado nas compras on-line

o sistema pode ver os detalhes de sua conta pessoal, E se o cliente pudesse enganar a interface do usuário da loja virtual para solicitar os detalhes de outra pessoa, talvez fazendo uma ligação com suas próprias credenciais de login?

Neste exemplo, o que impede o cliente de solicitar pedidos que sejam
não deles? Uma vez logados, eles podem começar a enviar solicitações para outros pedidos
que não são deles para ver se eles poderiam obter informações úteis. Eles poderiam começar
adivinhando IDs de pedidos para ver se eles poderiam extrair informações de outras pessoas.
Fundamentalmente, o que aconteceu aqui é que, embora tenhamos autenticado
para o usuário em questão, não estamos fornecendo autorização suficiente. O que nós
querer fazer parte do nosso sistema para poder julgar que uma solicitação para ver o usuário
Os detalhes de A só podem ser concedidos se for o Usuário A pedindo para vê-los. Onde faz
Mas essa lógica está viva?

Autorização centralizada e de upstream

Uma opção para evitar o problema confuso do deputado é realizar todas
autorização necessária assim que a solicitação for recebida em nosso sistema. Em
Figura 11-8, isso significaria que nosso objetivo seria autorizar a solicitação
no próprio gateway SSO ou na Web Shop. A ideia é que, pelo
quando as chamadas são enviadas para o microsserviço de Pedido ou Envio, assumimos
que as solicitações são permitidas.

Essa forma de autorização inicial implica efetivamente que estamos aceitando
alguma forma de confiança implícita (em oposição à confiança zero) - o transporte e
Os microsserviços de pedidos devem presumir que eles estão recebendo apenas solicitações que
eles estão autorizados a cumprir. O outro problema é que uma entidade upstream para
exemplo, um gateway, ou algo semelhante - precisa ter conhecimento do que
funcionalidade que os microsserviços downstream fornecem, e ela precisa saber como
para limitar o acesso a essa funcionalidade.

Idealmente, porém, queremos que nossos microsserviços sejam tão independentes quanto
possível, para facilitar ao máximo a realização de alterações e a implantação de novas
funcionalidade. Queremos que nossos lançamentos sejam o mais simples possível - queremos
capacidade de implantação independente. Se o ato de implantação agora envolver ambos
implantando um novo microsserviço e aplicando alguns relacionados à autorização

configuração para um gateway upstream, então isso não parece terrível....., “independente” para mim.

Portanto, gostaríamos de decidir se a ligação é ou não deve ser autorizado no mesmo microsserviço em que a funcionalidade sendo solicitadas vidas.. Isso torna o microsserviço mais independente e também nos dá a opção de implementar confiança zero, se quisermos.

Autorização descentralizada

Considerando os desafios da autorização centralizada em microsserviços ambiente, gostaríamos de levar essa lógica para o microsserviço downstream.

O microsserviço Order é onde está a funcionalidade para acessar o pedido detalha vidas, então faria sentido lógico que esse serviço decidisse se a chamada é válida. Nesse caso específico, porém, o microsserviço Order precisa informações sobre qual humano está fazendo a solicitação. Então, como podemos conseguir isso? informações para o microsserviço Order?

No nível mais simples, poderíamos simplesmente exigir que o identificador da pessoa fazendo com que a solicitação seja enviada ao microsserviço do Pedido. Se estiver usando HTTP, para Por exemplo, poderíamos simplesmente colocar o nome de usuário em um cabeçalho. Mas nesse caso, o que impede que uma pessoa mal-intencionada insira qualquer nome antigo no solicita obter as informações de que precisam? Idealmente, queremos uma maneira de garantir que a solicitação está realmente sendo feita em nome de um usuário autenticado e que podemos transmitir informações adicionais sobre esse usuário, por exemplo, os grupos em que o usuário pode se enquadrar.

Historicamente, há uma variedade de maneiras diferentes de lidar com isso (incluindo técnicas como afirmações SAML aninhadas, que, sim, são como dolorosos (por mais dolorosos que pareçam), mas recentemente a solução mais comum para isso um problema específico tem sido usar JSON Web Tokens..

Tokens Web JSON

Os JWTs permitem que você armazene várias afirmações sobre um indivíduo em uma string que pode ser transmitido. Este token pode ser assinado para garantir que a estrutura do

o token não foi manipulado e, opcionalmente, também pode ser criptografado para fornecer garantias criptográficas sobre quem pode ler os dados. Apesar Os JWTs podem ser usados para trocas genéricas de informações onde é importante para garantir que os dados não tenham sido adulterados, eles são mais comuns usado para ajudar a transmitir informações para auxiliar na autorização.

Uma vez assinados, os JWTs podem ser facilmente transmitidos por meio de uma variedade de protocolos e opcionalmente, os tokens podem ser configurados para expirar após um determinado período de tempo. Eles são amplamente suportados, com vários provedores de identidade que suporte à geração de JWTs e um grande número de bibliotecas para usar JWTs dentro do seu próprio código.

Formato

A carga útil principal de um JWT é uma estrutura JSON, que, em termos gerais, pode conter o que você quiser. Podemos ver um exemplo de token no Exemplo 11-1. O padrão JWT descreve alguns campos especificamente nomeados ("públicos"). reivindicações") que você deve usar se elas se relacionarem com você. Por exemplo, exp define a data de validade de um token. Se você usar esses campos de reivindicação pública corretamente, há uma boa chance de que as bibliotecas que você usa possam fazer uso de elas reivindicando adequadamente um token. Por exemplo, se o campo exp declarar que o token já expirou. Mesmo que você não use todos esses tokens públicos reivindicações, vale a pena estar ciente do que são para garantir que você não acabe usando-os para seus próprios usos específicos do aplicativo, pois isso pode causar alguns comportamento estranho nas bibliotecas de suporte.

Exemplo 11-1. Um exemplo da carga útil JSON de um JWT

```
{  
  "sub": "123",  
  "nome": "Sam Newman",  
  "exp": 1606741736,  
  "grupos": "admin, beta"  
}
```

No Exemplo 11-2, vemos o token do Exemplo 11-1 codificado. Esse token é na verdade apenas uma única string, mas é dividida em três partes delineadas com um " ." - o cabeçalho, carga útil e assinatura..

Exemplo 11-2. O resultado da codificação de uma carga JWT

```
EYJHBGCI0IJIUZi1NiIsInR5CC16IKpxVcJ9. ①  
eyJzdWIiOiIxMjM1LCJuYW1lIjoiU2FtIE5ld2lhbiIsImV4cCI6MTYwNjc0MTczNiwiZ3J...  
②  
Z9HMH0DGs60I0P5bVVSVfixeDxJjGovQEtlNUi__iE_0 ③  
① O cabeçalho  
② A carga útil (truncada)  
③ A assinatura
```

Para o benefício do exemplo aqui, eu dividi a linha em cada parte, mas em na realidade, isso seria uma única string sem quebras de linha. O cabeçalho contém informações sobre o algoritmo de assinatura que está sendo usado. Isso permite que o programa decodificando o token para suportar diferentes esquemas de assinatura. A carga útil é onde armazenamos informações sobre as afirmações que o token está fazendo - isso é apenas o resultado da codificação da estrutura JSON no Exemplo 11-1. A assinatura é usada para garantir que a carga útil não tenha sido manipulada e também pode ser usada para garantir que o token foi gerado por quem você acha que foi (supondo que o token esteja assinado com uma chave privada).

Como uma string simples, esse token pode ser facilmente transmitido por meio de diferentes protocolos de comunicação - como cabeçalho em HTTP (na Autorização header), por exemplo, ou talvez como um pedaço de metadado em uma mensagem. Isso. Uma string codificada pode, é claro, ser enviada por meio de protocolo de transporte criptografado - para exemplo, TLS sobre HTTP, caso em que o token não estaria visível para pessoas observando a comunicação.

Usando tokens

Vamos dar uma olhada em uma maneira comum de usar tokens JWT em um microsserviço arquitetura. Na Figura 11-9, nosso cliente faz login normalmente e uma vez autenticados, geramos algum tipo de token para representar seus logados sessão (provavelmente um token OAuth), que é armazenada no dispositivo cliente. Solicitações subsequentes desse dispositivo cliente chegam ao nosso gateway, o que gera um token JWT que será válido durante a solicitação. É esse JWT token que é então passado para os microsserviços downstream. Eles são capazes de

valide o token e extraia as reivindicações da carga útil para determinar qual tipo de autorização é apropriada.

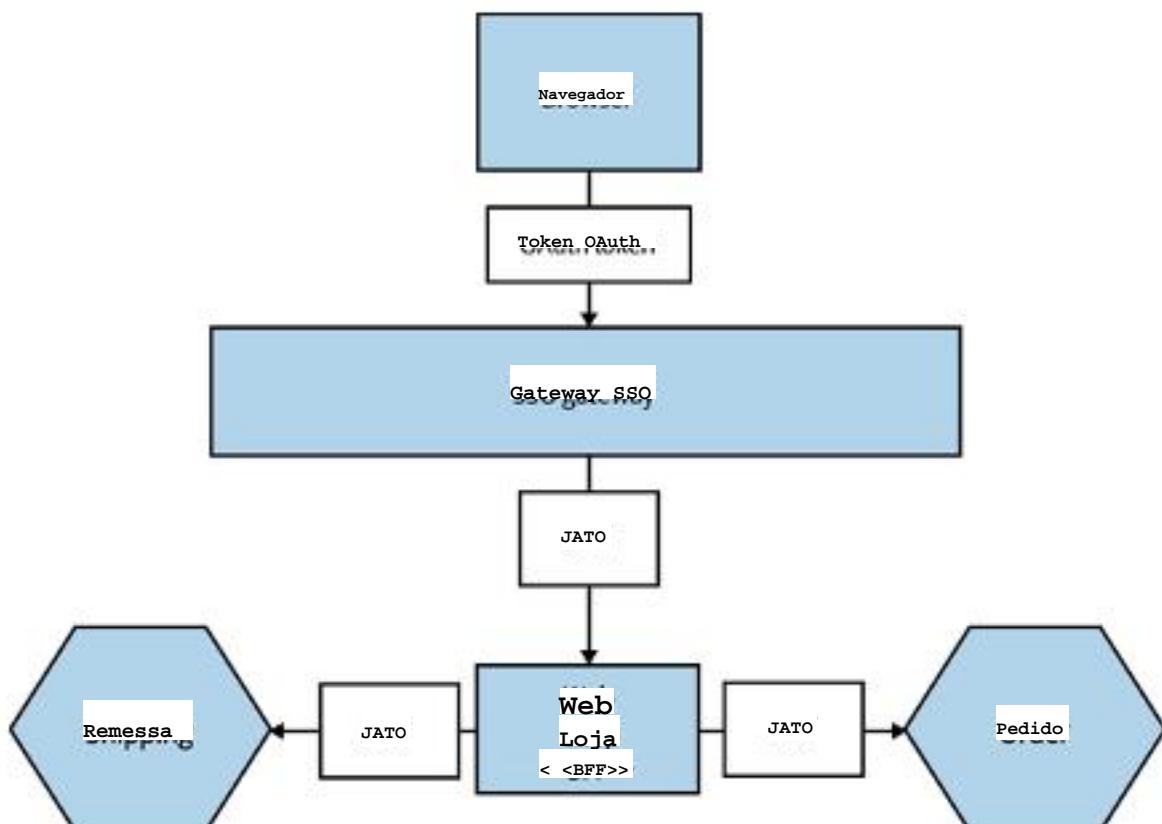


Figura 11-9. Um token JWT é gerado para uma solicitação específica e passado para o downstream microsserviços

Uma variação dessa abordagem é gerar um token JWT quando o usuário inicialmente se autentica com o sistema e depois tem esse JWT token armazenado no dispositivo cliente. Vale a pena considerar, porém, que tal token deverá ser válido durante a sessão conectada; como já discutimos, gostaríamos de limitar o período de validade do sistema-credentials geradas para reduzir as chances de serem mal utilizadas e para reduzir o impacto se precisarmos alterar as chaves usadas para gerar o token codificado. Gerar um token JWT por solicitação parece seja a solução mais comum para esse problema, como mostramos na Figura 11-9. Ter algum tipo de troca de tokens feita no gateway também pode fazer com que muito mais fácil adotar o uso de tokens JWT sem precisar alterar nenhuma parte do fluxo de autenticação que envolve a comunicação com o dispositivo cliente — se você já tem uma solução de SSO em funcionamento, ocultando o fato de que

Os tokens JWT são usados até mesmo a partir do fluxo principal de autenticação do usuário. uma mudança tão menos disruptiva.

Portanto, com a geração adequada de tokens JWT, nossos microsserviços downstream são capaz de obter todas as informações necessárias para confirmar a identidade do usuário fazer a solicitação, bem como informações adicionais, como grupos ou funções em que o usuário está. A validade desse token também pode ser verificada pelo microsserviço simplesmente verificando também a assinatura do token JWT. Em comparação com as soluções anteriores neste espaço (como SAML aninhado) afirmações), os tokens JWT tornaram o processo de descentralização autorização em uma arquitetura de microsserviços muito mais simples.

Desafios

Há alguns problemas relacionados aos tokens JWT que vale a pena ter em mente. A primeira é a questão das chaves. No caso de tokens JWT assinados, para verificar um assinatura, o receptor de um token JWT precisará de algumas informações que precisarão ser comunicados fora da banda - normalmente uma chave pública. Todos os questões de gerenciamento de chaves se aplicam neste caso. Como o microsserviço chega a chave pública? O que acontece se a chave pública precisar mudar? O Vault é um exemplo de uma ferramenta que pode ser usada por um microsserviço para recuperar (e lida com a rotação de (s) chaves públicas e já foi projetado para funcionar de forma altamente ambiente distribuído. É claro que você poderia simplesmente codificar uma chave pública em um arquivo de configuração para o microsserviço receptor, mas você teria então o problema de lidar com a mudança da chave pública.

Em segundo lugar, obter a expiração correta de um token pode ser complicado, se for longo. os tempos de processamento estão envolvidos. Considere uma situação em que um cliente tenha fez um pedido. Isso inicia um conjunto de processos assíncronos que poderiam leva horas, se não dias, para ser concluído, sem qualquer envolvimento subsequente de o cliente (recebendo o pagamento, enviando e-mails de notificação, recebendo o item embalado e enviado, etc.). Portanto, você precisa gerar um token com um período de validade correspondente? A questão aqui é, em que ponto está tendo um um token de vida mais longa é mais problemático do que não ter nenhum token? Eu falei com um poucas equipes que lidaram com esse problema. Alguns geraram um especial token de vida mais longa que tem como objetivo funcionar somente nesse contexto específico; outros

acabei de parar de usar o token em um determinado ponto do fluxo. Eu ainda não analisou exemplos suficientes desse problema para determinar a solução certa aqui, mas é um problema que você deve conhecer.

Finalmente, em algumas situações, você pode acabar precisando de muitas informações no Token JWT de que o próprio tamanho do token se torna um problema. Embora isso A situação é rara, acontece. Há vários anos, eu estava conversando com uma equipe sobre o uso de um token para gerenciar a autorização para um aspecto específico de seu sistema que gerenciava o gerenciamento de direitos de música. A lógica em torno disso era incrivelmente complexo - meu cliente descobriu que, para qualquer faixa, poderia potencialmente precisar de até 10.000 entradas em um token para lidar com os diferentes cenários. Percebemos, porém, que pelo menos nesse domínio, era apenas um caso de uso específico que precisava dessa grande quantidade de informações, enquanto a maior parte do sistema poderia se contentar com um token simples com menos campos. Em tal situação, fazia sentido lidar com os direitos mais complexos processo de autorização de gerenciamento de uma maneira diferente, essencialmente usando o Token JWT para a autorização inicial "simples" e, em seguida, para fazer uma subsequente pesquisa em um armazenamento de dados para buscar os campos adicionais conforme necessário. Isso significava a maior parte do sistema poderia simplesmente funcionar com os tokens.

Resumo

Como espero ter articulado neste capítulo, construir um sistema seguro não se trata de fazendo uma coisa. É necessária uma visão holística do seu sistema, usando alguns tipo de exercício de modelagem de ameaças, para entender que tipo de controles de segurança precisam ser implementados.

Ao pensar nesses controles, uma combinação é essencial para criar uma solução segura sistema. Defesa em profundidade não significa apenas ter várias proteções lugar; isso também significa que você tem uma abordagem multifacetada para construir um lugar mais sistema seguro.

Também retornamos novamente a um tema central do livro: ter um sistema decomposto em serviços mais refinados nos oferece muito mais opções para resolvendo problemas. Ter microserviços não só pode reduzir potencialmente o impacto de qualquer violação, mas também nos permite considerar compensações.

entre a sobrecarga de abordagens mais complexas e seguras, onde os dados estão abordagem sensível e mais leve quando os riscos são menores.

Para uma visão mais ampla da segurança de aplicativos em geral, recomendo o Agile

Segurança de aplicativos por Laura Bell et al. 17
Application security by Laura Bell et al.

A seguir, veremos como podemos tornar nossos sistemas mais confiáveis, à medida que avançamos ao tópico da resiliência.

1. Por favor, não vamos fazer tudo sobre o Brexit.

2. Por mais que tente, não consigo encontrar a fonte original desse esquema de categorização.

3. Eu recomendo *Jonny's Understanding Design Thinking, Lean e Agile* (O'Reilly) para mais de seus insights.

4. Era mais um argumento passivo-agressivo, muitas vezes sem a parte "passiva".

5. Adam Shostack, Modelagem de ameaças: projetando para a segurança (Indianápolis: Wiley, 2014).

6. Troy Hunt, "Passwords Evolved: Authentication Guidance for the Modern Era", 26 de julho de 2017.
<https://oreil.ly/T7PJM>.

7. Neil McAllister, "Code Spaces continua ativo para sempre depois que o atacante ataca sua Amazon-Hosted Data", The Register, 18 de junho de 2014, <https://oreil.ly/mn7PC>.

8. Algumas pessoas estão preocupadas com o fato de os segredos serem armazenados em texto sem formatação. Seja ou não este é um problema para você, depende muito do seu modelo de ameaça. Para que os segredos sejam lidos, um invasor precisaria ter acesso direto aos sistemas principais que executam seu cluster, momento em que é discutível que seu cluster já esteja irremediavelmente comprometido.

9. Eu definitivamente inventei esse termo, mas também acho altamente provável que eu não seja o único.

10. Assim como Niels Bohr argumentou que o gato de Schrödinger estava vivo e morto até você realmente abriu a caixa para verificar.

11. Jan Schaumann (@jschauma), Twitter, 5 de novembro de 2020 às 16:22, <https://oreil.ly/QaCm2>.

12. O "S" em "HTTPS" costumava se relacionar com o antigo Secure Socket Layer (SSL), que foi substituído pelo TLS por vários motivos. Confusamente, o termo SSL ainda persiste mesmo quando o TLS está realmente sendo usado. A biblioteca OpenSSL, por exemplo, é amplamente usada para implementar o TLS. E quando você receber um certificado SSL, ele será, na verdade, para o TLS. Nós não facilitamos as coisas para nós mesmos, não é?

13. Não criptografamos senhas em repouso, pois a criptografia significa que qualquer pessoa com a chave certa pode leia a senha novamente.

14. Anteriormente, falaríamos sobre autenticação de dois fatores (2FA). MFA é o mesmo conceito, mas

apresenta a ideia de que muitas vezes agora permitimos que nossos usuários forneçam um fator adicional a partir de um

variedade de dispositivos, como um token seguro, um aplicativo de autenticação móvel ou talvez biometria.
Você pode considerar o 2FA como um subconjunto do MFA.

15 Eu não posso receber crédito por isso!

16 O site do JWT tem uma excelente visão geral de quais bibliotecas apóiam quais reivindicações públicas - é uma ótimo recurso em geral para tudo relacionado ao JWT.

17. Laura Bell et al., Segurança ágil de aplicativos (Sebastopol: O'Reilly, 2017),

Capítulo 12. Resiliência

À medida que o software se torna uma parte cada vez mais vital da vida de nossos usuários, precisamos para melhorar continuamente a qualidade do serviço que oferecemos. A falha do software pode têm um impacto significativo na vida das pessoas, mesmo que o software não caia na categoria de "segurança crítica", da mesma forma que coisas como aeronaves os sistemas de controle funcionam. Durante a pandemia de COVID-19, que estava em andamento em na época em que escrevo, serviços como compras de supermercado on-line deixaram de ser um conveniência de se tornar uma necessidade para muitas pessoas que não conseguiram deixem suas casas.

Neste contexto, muitas vezes temos a tarefa de criar um software que é cada vez mais confiável. As expectativas de nossos usuários mudaram termos do que o software pode fazer e quando deve estar disponível. Os dias o fato de ter que oferecer suporte a software apenas durante o horário de expediente é cada vez mais raro, e há uma tolerância decrescente ao tempo de inatividade devido à manutenção.

Conforme abordamos no início deste livro, há uma série de razões pelas quais arquiteturas de microserviços estão sendo escolhidas por organizações em todo o mundo. Mas, para muitos, as perspectivas de melhorar a resiliência de seus as ofertas de serviços são citadas como um dos principais motivos.

Antes de entrarmos nos detalhes de como uma arquitetura de microserviços pode permitir resiliência, é importante dar um passo atrás e considerar o que realmente é resiliência é. Acontece que, quando se trata de melhorar a resiliência do nosso software, adotar uma arquitetura de microserviços é apenas parte do quebra-cabeça.

O que é resiliência?

Usamos o termo resiliência em muitos contextos diferentes e em muitos diferentes maneiras. Isso pode causar confusão sobre o significado do termo e também pode resultam em pensarmos muito estreitamente sobre o campo. Fora dos limites da TI, há uma área mais ampla de engenharia de resiliência, que analisa o conceito

de resiliência, pois se aplica a uma série de sistemas, do combate a incêndios ao tráfego aéreo, controle, sistemas biológicos e salas de operação. Desenhando neste campo, David D. Woods tentou categorizar os diferentes aspectos da resiliência para nos ajudar a pensar mais amplamente sobre o que realmente é resiliência significa. Esses quatro conceitos são:

Robustez

A capacidade de absorver a perturbação esperada.

Recuperação

A capacidade de se recuperar após um evento traumático

Extensibilidade elegante

Como lidamos bem com uma situação inesperada

Adaptabilidade sustentada

A capacidade de se adaptar continuamente a ambientes em mudança, partes interessadas, e demandas

Vamos examinar cada um desses conceitos sucessivamente e examinar como essas ideias pode (ou não) se traduzir em nosso mundo de criação de microsserviços arquiteturas.

Robustez

Robustez é o conceito pelo qual construímos mecanismos em nosso software e processos para acomodar os problemas esperados. Temos um avançado compreensão dos tipos de perturbações que podemos enfrentar, e colocamos medidas em vigor para que, quando esses problemas surgirem, nosso sistema possa lidar com eles. No contexto de nossa arquitetura de microsserviços, temos um host inteiro das perturbações que poderíamos esperar: um host pode falhar, uma conexão de rede pode expirar, um microsserviço pode não estar disponível. Nós podemos melhorar o robustez de nossa arquitetura de várias maneiras para lidar com essas perturbações, como a ativação automática de um hospedeiro substituto,

realizar novas tentativas ou lidar com a falha de um determinado microsserviço de forma simples maneira.

No entanto, a robustez vai além do software. Isso pode se aplicar às pessoas. Se você tiver uma única pessoa de plantão para o seu software, o que acontece se essa pessoa receber está doente ou não está acessível no momento de um incidente? Isso é uma coisa bastante fácil de considerar, e a solução pode ser ter uma pessoa de plantão de apoio.

Robustez, por definição, requer conhecimento prévio - estamos adotando medidas instalado para lidar com perturbações conhecidas. Esse conhecimento pode ser baseado sobre previsão: poderíamos nos basear em nossa compreensão do sistema de computador estamos construindo, seus serviços de apoio e nosso pessoal para considerar o que pode dar errado. Mas a robustez também pode vir de uma visão retrospectiva - podemos melhorar a robustez do nosso sistema depois de algo que não esperávamos acontece. Talvez nunca tenhamos considerado o fato de que nosso sistema de arquivos global poderia ficar indisponível, ou talvez tenhamos subestimado o impacto de nossos representantes de atendimento ao cliente não estão disponíveis fora do horário de trabalho.

Um dos desafios para melhorar a robustez do nosso sistema é que à medida que aumentamos a robustez de nossa aplicação, introduzimos mais complexidade no nosso sistema, que pode ser a fonte de novos problemas. Digamos que você está migrando sua arquitetura de microsserviços para o Kubernetes, porque você quer que ele gerencie o estado desejado para suas cargas de trabalho de microsserviços. Você pode ter melhorado alguns aspectos da robustez do seu aplicativo como resultado, mas você também introduziu novos pontos problemáticos em potencial. Como portanto, qualquer tentativa de melhorar a robustez de um aplicativo deve ser considerado, não apenas em termos de uma simples análise de custo/benefício, mas também em termos de se você está feliz ou não com o sistema mais complexo que você encontrará têm como resultado disso.

A robustez é uma área na qual os microsserviços oferecem uma série de opções, e muito do que se segue neste capítulo se concentrará no que você pode fazer em seu software para melhorar a robustez do sistema. Apenas lembre-se de que não. Essa é apenas uma faceta da resiliência como um todo, mas também há uma série de outras robustez não relacionada ao software que talvez você precise considerar.

Recuperação

O quanto bem nos recuperamos de interrupções é uma parte fundamental da construção de um sistema resiliente. Com muita frequência, vejo pessoas concentrando seu tempo e energia em tentando eliminar a possibilidade de uma interrupção, apenas para estar totalmente despreparado quando uma interrupção realmente ocorre. De qualquer forma, faça o possível para se proteger contra as coisas ruins que você acha que podem acontecer - melhorando o sistema robustez, mas também entenda que, à medida que seu sistema cresce em escala e complexidade, eliminando qualquer problema potencial, torna-se insustentável.

Podemos melhorar nossa capacidade de nos recuperar de um incidente colocando coisas em coloque com antecedência. Por exemplo, ter backups em vigor pode nos permitir melhor recuperação após a perda de dados (supondo que nossos backups sejam testados, claro!). Melhorar nossa capacidade de recuperação também pode incluir ter um manual que podemos seguir após uma interrupção do sistema: As pessoas fazem entendem qual é o papel deles quando ocorre uma interrupção? Quem será o ponto pessoa para lidar com a situação? Com que rapidez precisamos permitir que nossos usuários sabe o que está acontecendo? Como nos comunicaremos com nossos usuários? Tentando pensar com clareza sobre como lidar com uma interrupção enquanto a interrupção está acontecendo será problemático devido ao estresse inherente e ao caos da situação.

Ter um plano de ação acordado em vigor em antecipação a esse tipo de problema pode ajudá-lo a se recuperar melhor.

Extensibilidade elegante

Com recuperação e robustez, estamos lidando principalmente com o esperado. Estamos implementando mecanismos para lidar com os problemas que podemos prever. Mas o que acontece quando somos surpreendidos? Se não estivermos preparados para surpresa pelo fato de que nossa visão esperada do mundo pode estar errada acabamos com um sistema frágil. À medida que nos aproximamos dos limites do que nós esperamos que nosso sistema seja capaz de lidar com isso, as coisas desmoronam - não conseguimos executar adequadamente.

Organizações mais planas, onde a responsabilidade é distribuída na organização, em vez de realizada centralmente, geralmente estará mais bem preparada para negociar com surpresa. Quando o inesperado ocorre, se as pessoas estão restritas em que

eles têm que fazer, se tiverem que aderir a um conjunto estrito de regras, sua capacidade de lidar com a surpresa será reduzida criticamente.

Muitas vezes, em uma tentativa de otimizar nosso sistema, podemos, como um efeito colateral infeliz aumentar a fragilidade do nosso sistema. Veja a automação como exemplo.

A automação é fantástica - ela nos permite fazer mais com as pessoas que temos, mas também pode nos permitir reduzir as pessoas que temos, à medida que mais pode ser feito com automação. No entanto, essa redução de pessoal pode ser preocupante.

A automação não aguenta surpresas: nossa capacidade de estender com elegância nossa habilidades, experiência e responsabilidade de lidar com essas situações à medida que elas surgem. O sistema, para lidar com surpresas, vem de ter pessoas no lugar certo

Adaptabilidade sustentada

Ter adaptabilidade sustentada exige que não sejamos complacentes. Como David Woods diz: "Não importa o quanto bem tenhamos feito antes, não importa o quanto bem-sucedidos que fomos, o futuro pode ser diferente e talvez não sejamos bem adaptados. Podemos ser precários e frágeis diante desse novo futuro." O fato de ainda não termos sofrido uma interrupção catastrófica não significa que isso não pode acontecer. Precisamos nos desafiar para garantir que estamos constantemente adaptando o que fazemos como organização para garantir o futuro resiliência. Feito corretamente, um conceito como engenharia do caos, que vamos explore brevemente mais adiante neste capítulo - pode ser uma ferramenta útil para ajudar a construir adaptabilidade sustentada.

A adaptabilidade sustentada geralmente requer uma visão mais holística do sistema. Isso é, paradoxalmente, onde um impulso em direção a equipes menores e autônomas com o aumento da responsabilidade local e focada pode acabar com a perda de vista do imagem maior. Conforme exploraremos no Capítulo 15, há um ato de equilíbrio entre otimização global e local quando se trata de organização dinâmica, e esse equilíbrio não é estático. Nesse capítulo, veremos o papel de equipes focadas e alinhadas ao fluxo que possuem os microserviços de que precisam oferecem funcionalidade voltada para o usuário e aumentam os níveis de responsabilidade para fazer isso acontecer. Também analisaremos o papel de capacitar equipes, que apoie essas equipes alinhadas ao fluxo na realização de seu trabalho e na capacitação

as equipes podem ser uma grande parte de ajudar a alcançar uma adaptabilidade sustentada em um nível organizacional.

Criando uma cultura que priorize a criação de um ambiente no qual as pessoas pode compartilhar informações livremente, sem medo de retribuição, é vital incentivar aprendendo após um incidente. Ter a largura de banda para realmente examinar essas surpresas e extrair os principais conhecimentos requerem tempo, energia e pessoas todas as coisas que reduzirão os recursos disponíveis para você entregar características no curto prazo. Decidir adotar a adaptabilidade sustentada é em parte sobre como encontrar o ponto de equilíbrio entre entrega de curto prazo e adaptabilidade a longo prazo.

Trabalhar em prol da adaptabilidade sustentada significa que você está procurando descubra o que você não sabe. Isso requer investimento contínuo, não um... fora das atividades transacionais - o termo sustentado é importante aqui. É sobre tornando a adaptabilidade sustentada uma parte essencial de sua estratégia organizacional e cultura.

E arquitetura de microserviços

Conforme discutimos, podemos ver uma maneira pela qual uma arquitetura de microserviços pode nos ajudar a alcançar a propriedade de robustez, mas não é suficiente se você quero resiliência.

De forma mais ampla, a capacidade de oferecer resiliência é uma propriedade que não é da software em si, mas das pessoas que constroem e executam o sistema. Dado o foco deste livro, muito do que segue neste capítulo se concentrará principalmente sobre o que uma arquitetura de microserviços pode ajudar a oferecer em termos de resiliência que é quase totalmente limitado a melhorar a robustez das aplicações.

O fracasso está em toda parte

Entendemos que as coisas podem dar errado. Os discos rígidos podem falhar. Nossa software pode falhar. E como qualquer um que tenha lido as falácias da computação distribuída Posso dizer que a rede não é confiável. Podemos fazer o nosso melhor para tentar limitar o causas do fracasso, mas em uma certa escala, o fracasso se torna inevitável. Difícil

os drives, por exemplo, estão mais confiáveis agora do que nunca, mas serão, acabam quebrando. Quanto mais discos rígidos você tiver, maior a probabilidade de falha de uma unidade individual em um determinado dia; a falha se torna uma estatística certeza em grande escala.

Mesmo para aqueles de nós que não pensam em uma escala extrema, se pudermos abraçar a possibilidade de fracasso, estaremos melhor. Por exemplo, se pudermos lidar com a falha de um microserviço normalmente, então conclui-se que também podemos fazer em-faça atualizações de um serviço, pois é muito mais fácil lidar com uma interrupção planejada do que um não planejado.

Também podemos gastar um pouco menos do nosso tempo tentando impedir o inevitável e um pouco passamos mais tempo lidando com isso graciosamente. Estou surpreso com quantas organizações implementam processos e controles para tentar impedir o fracasso de ocorrendo, mas não pense em realmente torná-la mais fácil de recuperar do fracasso em primeiro lugar. Entendendo as coisas que provavelmente falharão é fundamental para melhorar a robustez do nosso sistema.

Partir do pressuposto de que tudo pode e vai falhar leva você a pensar de forma diferente sobre como você resolve problemas. Relembre a história do Google servidores que discutimos no Capítulo 10? Os sistemas do Google foram construídos de tal forma uma forma de que, se uma máquina falhasse, isso não levaria à interrupção do serviço-melhorando a robustez do sistema como um todo. O Google vai mais longe tentando melhorar a robustez de seus servidores de outras maneiras - tem pode continuar operando se o data center sofrer uma interrupção. Como você deve se lembrar de Capítulo 10, os discos rígidos desses servidores foram conectados com Velcro em vez de do que parafusos para facilitar a substituição de unidades, ajudando o Google a obter o a máquina está funcionando rapidamente quando uma unidade falha e, por sua vez, ajuda nisso componente do sistema se recupera de forma mais eficaz.

Então, deixe-me repetir: em grande escala, mesmo que você compre o melhor kit, o mais caro hardware, você não pode evitar o fato de que as coisas podem e irão falhar. Portanto, você precisa presumir que o fracasso pode acontecer. Se você construir esse pensamento em tudo o que você faz e planeja para o fracasso, você pode fazer concessões informadas. Se você sabe que seu sistema pode lidar com o fato de que um servidor pode e falhará,

pode haver retornos decrescentes ao gastar mais e mais dinheiro em máquinas individuais. Em vez disso, ter um número maior de máquinas mais baratas (talvez usando componentes mais baratos e um pouco de velcro!) como o Google fez com muitos faz muito mais sentido.

Quanto é demais?

Abordamos o tópico dos requisitos interfuncionais no Capítulo 9. Compreender os requisitos multifuncionais envolve considerar aspectos como durabilidade dos dados, disponibilidade de serviços, taxa de transferência e aceitação latência das operações. Muitas das técnicas abordadas neste capítulo falam sobre abordagens para implementar esses requisitos, mas só você sabe exatamente quais podem ser os requisitos em si. Então, mantenha o seu requisitos em mente enquanto você lê.

Ter um sistema de escalonamento automático capaz de reagir ao aumento da carga ou ao a falha de nós individuais pode ser fantástica, mas pode ser um exagero para um sistema de relatórios que precisa ser executado apenas duas vezes por mês, onde estar inativo por um dia ou dois não é grande coisa. Da mesma forma, descobrir como fazer zero-implantações em tempo de inatividade para eliminar a interrupção do serviço podem fazer sentido para seu sistema de comércio eletrônico on-line, mas para seu conhecimento de intranet corporativa base, provavelmente é um passo longe demais.

Quanta falha você pode tolerar ou quanto rápido seu sistema precisa ser é conduzido pelos usuários do seu sistema. Essas informações, por sua vez, ajudam você entender quais técnicas farão mais sentido para você. Dito isso, seus usuários nem sempre conseguirão articular quais são seus requisitos exatos. Portanto, você precisa fazer perguntas para ajudar a extraír as informações corretas e ajudá-los a entender os custos relativos de fornecer diferentes níveis de serviço.

Como mencionei anteriormente, esses requisitos multifuncionais podem variar de serviço a serviço, mas eu sugeriria definir uma cruz geral funcionais e, em seguida, substituindo-os para casos de uso específicos. Quando chega para considerar se e como escalar seu sistema para lidar melhor com a carga ou falha, comece tentando entender os seguintes requisitos:

Tempo de resposta/latência

Quanto tempo devem durar várias operações? Pode ser útil medir isso com diferentes números de usuários para entender como o aumento da carga será impactar o tempo de resposta. Dada a natureza das redes, você sempre tem valores atípicos, portanto, defina metas para um determinado percentil das respostas monitoradas pode ser útil. A meta também deve incluir o número de conexões/usuários simultâneos que você espera que seu software gerencie. Então você pode dizer: "Esperamos que o site tenha uma resposta do 90º percentil tempo de 2 segundos ao lidar com 200 conexões simultâneas por segundo."

Disponibilidade

Você pode esperar que um serviço fique inativo? Isso é considerado um serviço 24 horas por dia, 7 dias por semana? Algumas pessoas gostam de observar períodos de inatividade aceitáveis quando medindo a disponibilidade, mas quão útil isso é para alguém ligando para seu serviço? Ou eu deveria poder contar com sua resposta de serviço ou eu não deveria? Medir períodos de inatividade é realmente mais útil a partir de um ângulo de reportagem histórica.

Durabilidade dos dados

Quanta perda de dados é aceitável? Por quanto tempo os dados devem ser mantidos? É altamente provável que isso mude caso a caso. Por exemplo, você pode optar por manter os registros das sessões do usuário por um ano ou menos para economizar espaço, mas seus registros de transações financeiras podem precisar ser mantidos para muitos anos.

Pegando essas ideias e articulando-as como objetivos de nível de serviço (SLOs), que abordamos no Capítulo 10, pode ser uma boa maneira de consagrá-los requisitos como parte essencial do seu processo de entrega de software.

Funcionalidade degradante

Uma parte essencial da construção de um sistema resiliente, especialmente quando a funcionalidade está espalhada por vários microsserviços diferentes que podem ser para cima ou para baixo, é a capacidade de degradar a funcionalidade com segurança. Vamos imaginar um página da web padrão em nosso site de comércio eletrônico. Para reunir as várias partes desse site, talvez precisemos de vários microsserviços para desempenhar um papel. Um o microsserviço pode exibir os detalhes sobre o item que está sendo oferecido para venda. Outro pode mostrar o preço e o nível do estoque. E provavelmente seremos mostrando também o conteúdo do carrinho de compras, o que pode ser mais um microsserviço. Se um desses serviços cair e isso resultar em toda a página da web estando indisponível, então, sem dúvida, criamos um sistema que é menos resiliente do que um que exige que apenas um serviço esteja disponível.

O que precisamos fazer é entender o impacto de cada interrupção e nos exercitar como degradar adequadamente a funcionalidade. Do ponto de vista comercial, nós gostaria que nosso fluxo de trabalho de recebimento de pedidos fosse o mais robusto possível, e nós pode ficar feliz em aceitar alguma degradação da funcionalidade para garantir que isso ainda funciona. Se os níveis de estoque não estiverem disponíveis, podemos tomar a decisão de ainda prossiga com a venda e resolva os detalhes mais tarde. Se o carrinho de compras o microsserviço não está disponível, provavelmente estamos com muitos problemas, mas poderíamos ainda mostre a página da web com a listagem. Talvez nós apenas escondamos as compras carregue ou substitua-o por um ícone dizendo "Volte em breve!"

Com um aplicativo monolítico de processo único, não temos muitas decisões para fazer. A saúde do sistema é, até certo ponto, binária neste contexto - o processo é para cima ou para baixo. Mas com uma arquitetura de microsserviços, precisamos considerar uma situação muito mais sutil. A coisa certa a fazer em qualquer situação é frequentemente não é uma decisão técnica. Talvez saibamos o que é tecnicamente possível quando o carrinho de compras está fora do ar, mas, a menos que entendamos o contexto comercial, não entenderemos qual ação devemos tomar. Por exemplo, talvez nós feche o site inteiro, ainda permita que as pessoas naveguem pelo catálogo de itens ou substitua a parte da interface que contém o controle do carrinho por um número de telefone para fazendo um pedido. Mas para cada interface voltada para o cliente que usa várias microsserviços, ou cada microsserviço que depende de vários downstream colaboradores, você precisa se perguntar: "O que acontece se isso cair?" e saiba o que fazer.

Ao pensar na criticidade de cada uma de nossas capacidades em termos de requisitos multifuncionais, estaremos muito melhor posicionados para saber o que nós podemos fazer. Agora vamos considerar algumas coisas que podemos fazer do ponto de vista técnico de vista para garantir que, quando ocorrer uma falha, possamos lidar com isso com elegância.

Padrões de estabilidade

Existem alguns padrões que podemos usar para garantir que, se algo acontecer dê errado, não cause efeitos de ondulação desagradáveis. É essencial você compreenda essas ideias, e você deve considerar seriamente fazer uso delas em seu sistema para garantir que um cidadão ruim não traga o mundo inteiro caindo ao redor de suas orelhas. Em um momento, daremos uma olhada em algumas chaves medidas de segurança que você deve considerar, mas antes disso, gostaria de compartilhar um breve história para descrever o tipo de coisa que pode dar errado.

Há muitos anos, fui líder técnico em um projeto para a AdvertCorp. AdvertCorp (o nome e os detalhes da empresa são alterados para proteger a inocente!) forneceu anúncios classificados on-line por meio de um site muito popular. O site em si lidou com volumes bastante altos e gerou uma boa quantidade de renda para o negócio. O projeto em que eu estava trabalhando foi encarregado de consolidando vários serviços existentes que foram usados para fornecer serviços similares funcionalidade para diferentes tipos de anúncios. Funcionalidade existente para os diferentes tipos de anúncios estavam lentamente sendo migrados para o novo sistema, nós estavam construindo, com vários tipos diferentes de anúncios ainda veiculados a partir de serviços mais antigos. Para tornar essa transição transparente para o cliente final, nós interceptou todas as chamadas para os diferentes tipos de anúncios em nosso novo sistema, desviando eles para os sistemas antigos, quando necessário, conforme descrito na Figura 12-1. Isso é na verdade, um exemplo de padrão de figo estrangulador, que discutimos brevemente em "Padrões de decomposição úteis".

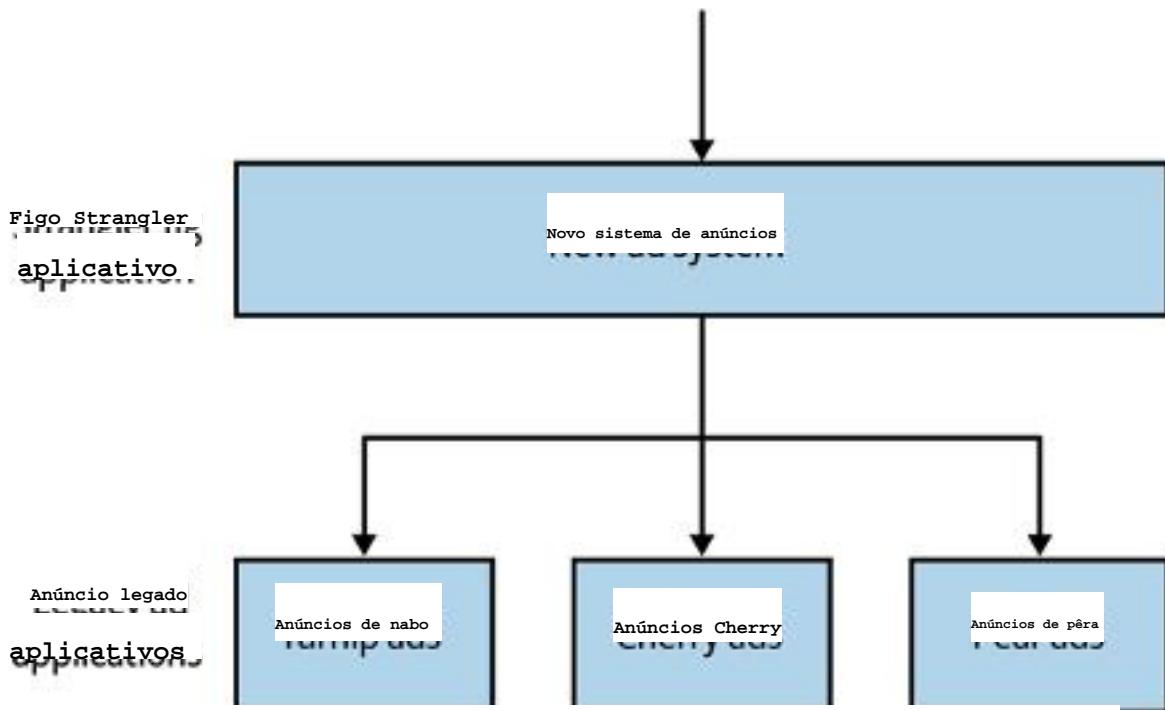


Figura 12-1. Um padrão de figo estrangulador usado para direcionar chamadas para sistemas legados mais antigos.

Tínhamos acabado de transferir o produto de maior volume e maior lucro para o novo sistema, mas muitos dos demais anúncios ainda estavam sendo veiculados por um número de aplicativos mais antigos. Em termos do número de pesquisas e do dinheiro ganho com esses aplicativos, havia uma cauda muito longa - muitos deles aplicativos mais antigos recebiam pequenas quantidades de tráfego e geravam pequenas quantias de receita. O novo sistema estava ativo há algum tempo e foi se comportando muito bem, lidando com uma carga não insignificante. Nesse momento, devemos têrm atendido cerca de 6.000 a 7.000 solicitações por segundo durante o pico, e embora a maior parte disso estivesse muito armazenada em cache por proxies reversos na frente de nossos servidores de aplicativos, as pesquisas por produtos (a maioria aspecto importante do site) estavam em sua maioria sem cache e exigiam um servidor completo ida e volta.

Certa manhã, pouco antes de atingirmos nosso pico diário na hora do almoço, o sistema começou se comportando lentamente e depois começou a falhar. Tivemos algum nível de monitoramento em nosso novo aplicativo principal, o suficiente para nos dizer que cada um de nossos aplicativos os nós estavam atingindo um pico de 100% da CPU, bem acima dos níveis normais, mesmo em pico. Em um curto período de tempo, todo o site caiu.

Conseguimos rastrear o culpado e trazer o site de volta ao ar. Descobriu-se ser um dos sistemas de anúncios downstream, que, por causa disso Um estudo de caso anônimo, que diremos, foi responsável por anúncios relacionados a nabos. O serviço de anúncios de nabo, um dos serviços mais antigos e menos ativamente mantidos, tinha começado a responder muito lentamente. Responder muito lentamente é uma das piores modos de falha que você pode experimentar. Se um sistema simplesmente não está lá, você encontra saia muito rapidamente. Quando está lento, você acaba esperando por um tempo antes de desistir, esse processo de espera pode retardar todo o sistema inativo, causa contêncio de recursos e, como aconteceu em nosso caso, resulta em um falha em cascata. Mas seja qual for a causa da falha, criamos um sistema que estava vulnerável a um problema posterior que causava uma cascata falha em todo o sistema. Um serviço de downstream, sobre o qual tínhamos pouco controle, foi capaz de derrubar todo o nosso sistema.

Enquanto uma equipe analisava os problemas com o sistema de nabo, o resto de nós comecei a analisar o que havia dado errado em nosso aplicativo. Encontramos alguns problemas, que são descritos na Figura 12-2. Estávamos usando um HTTP pool de conexões para lidar com nossas conexões a jusante. Os tópicos do próprio pool tinha tempos limite configurados para quanto tempo eles esperariam quando fazendo a chamada HTTP downstream, o que é bom. O problema era que o todos os trabalhadores estavam demorando um pouco para se afastar devido à lentidão a jusante serviço. Enquanto esperavam, mais solicitações foram enviadas para a piscina pedindo tópicos de trabalho. Sem trabalhadores disponíveis, esses pedidos por si só foram suspensos. Descobrimos que a biblioteca do pool de conexões que estávamos usando teve um tempo limite para esperar por trabalhadores, mas isso foi desativado por padrão! Isso levou a uma enorme acúmulo de tópicos bloqueados. Nossa aplicativo normalmente tinha 40 simultâneos conexões a qualquer momento. No espaço de cinco minutos, essa situação fez com que atingíssemos um pico de cerca de 800 conexões, derrubando o sistema.

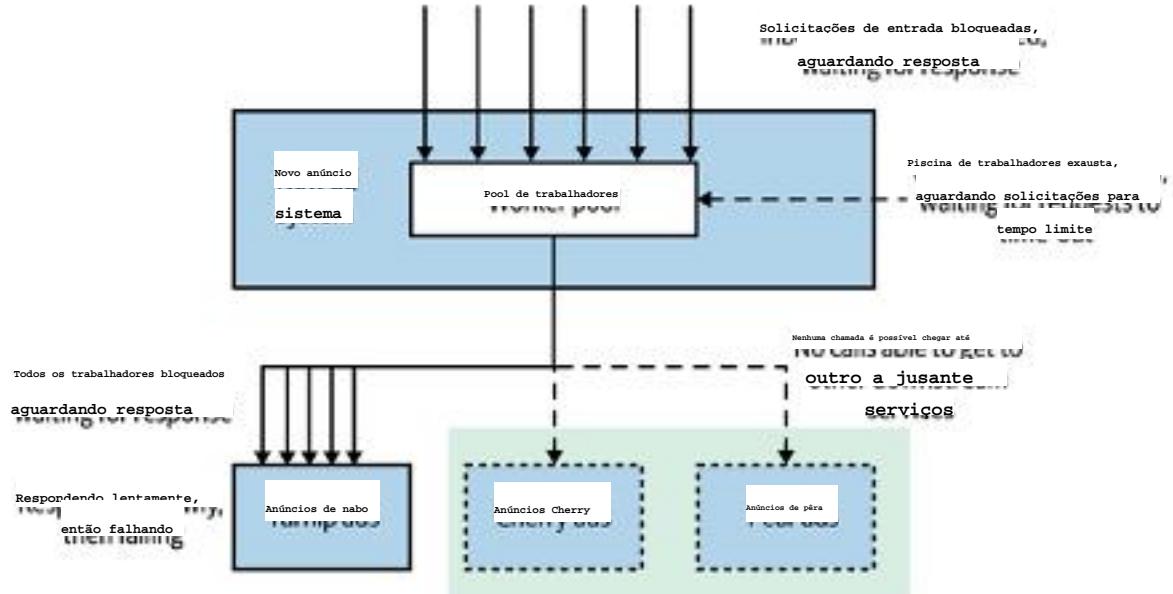


Figura 12-2. Um resumo dos problemas que causaram a interrupção.

O pior foi que o serviço de downstream com o qual estávamos conversando representou uma funcionalidade que menos de 5% de nossa base de clientes usava, e gerou ainda menos receita do que isso. Quando você começar, nós descobriu da maneira mais difícil que sistemas que agem lentamente são muito mais difíceis de lidar com sistemas que simplesmente falham rapidamente. Em um sistema distribuído, a latência mata.

Mesmo que tivéssemos definido corretamente os intervalos na piscina, também estávamos compartilhando um único pool de conexões HTTP para todas as solicitações de saída. Isso significava que um serviço lento de downstream poderia esgotar o número de disponíveis trabalhadores por si só, mesmo que todo o resto fosse saudável. Por fim, ficou claro devido aos frequentes intervalos e erros cometidos pelo serviço downstream ...

A pergunta não era saudável, mas, apesar disso, continuamos enviando tráfego para ela. Em nossa situação, isso significava que estávamos realmente piorando uma situação ruim, como o serviço de downstream não teve chance de se recuperar. Acabamos implementando três soluções para evitar que isso aconteça novamente: acertar nossos intervalos, implementando anteparos para separar diferentes pools de conexões e implementar um disjuntor para evitar o envio de chamadas para um sistema não saudável em primeiro lugar.

Intervalos

É fácil ignorar os intervalos, mas em um sistema distribuído eles são importantes para acertar. Quanto tempo posso esperar antes de desistir de uma ligação para um serviço de downstream? Se você esperar muito tempo para decidir que uma chamada falhou, pode desacelerar todo o sistema. Faça um intervalo muito rápido e você considerará um chamada que pode ter funcionado como falhada. Não tenha nenhum intervalo, e um serviço downstream estar inativo pode paralisar todo o sistema.

No caso da AdvertCorp, tivemos dois problemas relacionados ao tempo limite. Em primeiro lugar, nós faltou um tempo limite no pool de solicitações HTTP, o que significa que ao perguntar para que um trabalhador fizesse uma solicitação HTTP downstream, o thread de solicitação seria bloqueio para sempre até que um trabalhador fique disponível. Em segundo lugar, quando finalmente tínhamos um trabalhador HTTP disponível para fazer uma solicitação ao sistema de anúncios de nabos, nós estávamos esperando muito tempo antes de desistir da ligação. Então, como na Figura 12-3 mostra que precisávamos adicionar um novo tempo limite e alterar um existente.

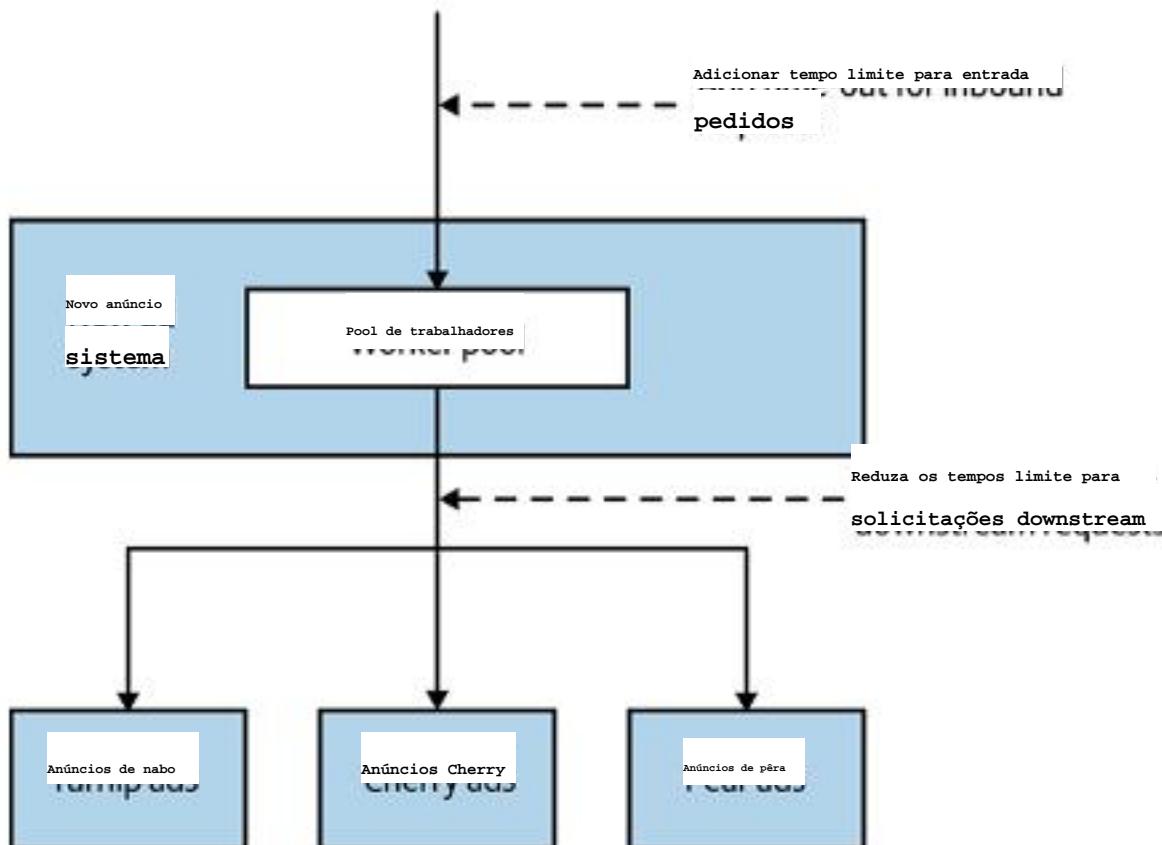


Figura 12-3. Alterando os tempos limite no sistema AdvertCorp

Os tempos limite para as solicitações HTTP downstream foram definidos para 30 segundos. -então, esperaríamos 30 segundos por uma resposta do sistema de nabo antes

desistindo. O problema é que, no contexto mais amplo em que essa chamada estava sendo feita, esperar tanto tempo não fazia sentido. Os anúncios relacionados ao nabo estavam sendo solicitados como resultado de um de nossos usuários acessando nosso site usando um navegador. Mesmo Quando isso acontecia, ninguém esperava 30 segundos para que uma página fosse carregada. Pense sobre o que acontece se uma página da web não for carregada após 5, 10 ou talvez 15 segundos. O que você faz? Você atualiza a página. Então estávamos esperando 30 segundos para que o sistema de anúncios de nabos responda, mas bem antes disso, a original solicitação não era mais válida porque o usuário acabara de se atualizar, causando uma solicitação adicional de entrada. Isso, por sua vez, fez com que outra solicitação fosse recebida o sistema de anúncios, e assim por diante.

Ao observar o comportamento normal do sistema de anúncios de nabos, pudemos ver que normalmente esperaríamos uma resposta em muito menos de um segundo, então esperar 30 segundos foi um exagero. Além disso, tínhamos um alvo para renderizar uma página para o usuário dentro de 4-6 segundos. Com base nisso, fizemos o intervalo muito mais agressivo, configurando-o para 1 segundo. Também colocamos um tempo limite de 1 segundo ao esperar que um trabalhador HTTP esteja disponível. Isso significava que, na pior das hipóteses caso, esperaríamos esperar cerca de 2 segundos para obter informações do sistema de nabo.

GORJETA

Os intervalos são incrivelmente úteis. Coloque intervalos em todas as chamadas fora do processo e escolha um padrão tempo limite para tudo. Registre quando ocorrem intervalos, veja o que acontece e mude eles de acordo. Veja os tempos de resposta saudáveis "normais" para seus serviços posteriores, e use isso para orientar onde você define o limite de tempo limite.

Definir um tempo limite para uma única chamada de serviço pode não ser suficiente. O que acontece se esse tempo limite estiver acontecendo como parte de um conjunto mais amplo de operações que você pode querer desistir mesmo antes que o tempo limite ocorra? No caso de A AdvertCorp, por exemplo, não adianta esperar pelos preços mais recentes do nabo se houver uma boa chance de o usuário já ter desistido de perguntar. Em tal situação, pode fazer sentido ter um tempo limite para a operação geral e desistir se esse tempo limite for ultrapassado. Para que isso funcione, resta a hora atual pois a operação precisaria ser passada rio abaixo. Por exemplo, se o

a operação geral para renderizar uma página teve que ser concluída em 1.000 ms, e por o tempo em que ligamos para o serviço de publicidade de nabos downstream de 300 ms teve já passou, precisaríamos então ter certeza de que não esperaríamos mais do que 700 ms para que o resto das chamadas sejam concluídas.

ADVERTÊNCIA

Não pense apenas no tempo limite para uma única chamada de serviço; pense também em um tempo limite para a operação geral e aborte a operação se esse orçamento geral de tempo limite for excedido.

Tentativas novamente

Alguns problemas com chamadas downstream são temporários. Os pacotes podem ser recebidos mal colocados, ou os gateways podem ter um aumento estranho na carga, causando um tempo limite. Muitas vezes, repetir a chamada pode fazer muito sentido. Voltando ao que acabamos de falou sobre, com que frequência você atualizou uma página da web que não carregou, apenas descobrir que a segunda tentativa funcionou bem? Essa é uma nova tentativa em ação.

Pode ser útil considerar que tipo de falha de chamada downstream deve até mesmo ser tentado novamente. Se estiver usando um protocolo como HTTP, por exemplo, você poderá voltar algumas informações úteis nos códigos de resposta que podem ajudar você a determinar se uma nova tentativa é necessária. Se você recebeu de volta um 404 Not Found, é improvável que tente novamente seja uma ideia útil. Por outro lado, um 503 Service Unavailable ou um 504 O tempo limite do gateway pode ser considerado um erro temporário e pode justificar

uma nova tentativa.

Você provavelmente precisará de um atraso antes de tentar novamente. Se o tempo limite inicial ou o erro foi causado pelo fato de o microsserviço downstream estar sob carregue e, em seguida, bombardeá-lo com solicitações adicionais pode muito bem ser uma má ideia.

Se você vai tentar novamente, você precisa levar isso em consideração ao considerar seu limite de tempo limite. Se o limite de tempo limite para uma chamada downstream for definido até 500 ms, mas você permite até três novas tentativas com um segundo entre cada tenta novamente, então você pode acabar esperando por até 3,5 segundos antes de desistir.

Conforme mencionado anteriormente, ter um orçamento para quanto tempo uma operação é permitida

tomar pode ser uma ideia útil - você pode não decidir fazer a terceira (ou mesmo, segundo) tente novamente se você já excedeu o orçamento geral de tempo limite. No por outro lado, se isso estiver acontecendo como parte de uma operação não voltada para o usuário, esperar mais para fazer algo pode ser totalmente aceitável.

Anteparas

Em *Release It!* 4, Michael Nygard apresenta o conceito de antepara como maneira de se isolar do fracasso. No transporte marítimo, uma antepara faz parte do navio que pode ser selado para proteger o resto do navio. Então, se o navio saltar um vazamento, você pode fechar as portas do anteparo. Você perde parte do navio, mas o resto permanece intacto.

Em termos de arquitetura de software, existem muitas anteparas diferentes que podemos considerar. Voltando à minha própria experiência com a AdvertCorp, na verdade perdeu a chance de implementar uma antepara em relação ao downstream chamadas. Deveríamos ter usado diferentes pools de conexão para cada downstream conexão. Dessa forma, se um pool de conexões se esgotar, o outro, as conexões não seriam afetadas, como vemos na Figura 12-4.

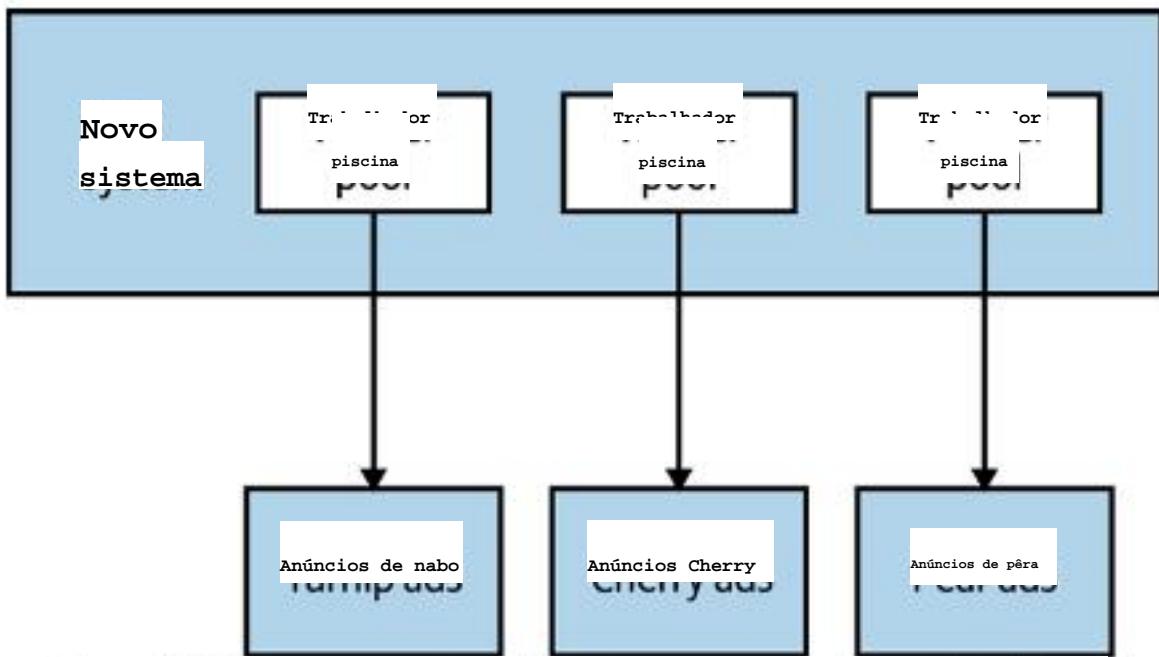


Figura 12-4. Usando um pool de conexões por serviço de downstream para fornecer anteparas

A separação de preocupações também pode ser uma forma de implementar anteparos. Provocando separando a funcionalidade em microserviços separados, reduzimos a chance de um interrupção em uma área afetando outra.

Veja todos os aspectos do seu sistema que podem dar errado, tanto dentro do seu microserviços e entre eles. Você tem anteparos instalados? Eu faria sugiro começar com pools de conexão separados para cada downstream. conexão, no mínimo. Você pode querer ir mais longe, no entanto, e considere também o uso de disjuntores, que veremos em breve.

De muitas maneiras, as anteparas são os padrões mais importantes que observamos até agora. Os intervalos e os disjuntores ajudam você a liberar recursos quando eles estão ficando restritos, mas as anteparas podem garantir que não se tornem restringido em primeiro lugar. Eles também podem dar a você a capacidade de rejeitar solicitações em determinadas condições para garantir que os recursos não se igualem mais saturado; isso é conhecido como redução de carga. Às vezes, rejeitando uma solicitação é a melhor maneira de impedir que um sistema importante se torne sobrecarregado e sendo um gargalo para vários serviços upstream.

Disjuntores

Em sua própria casa, existem disjuntores para proteger seus dispositivos elétricos de picos de poder. Se ocorrer um pico, o disjuntor é queimado, protegendo seus eletrodomésticos caros. Você também pode desativar manualmente um disjuntor para cortar a energia de parte da sua casa, permitindo que você trabalhe com segurança no sistema elétrico. Em outro padrão do *Release It!*, Nygard mostra como a mesma ideia pode fazer maravilhas como mecanismo de proteção para nosso software.

Podemos pensar em nossos disjuntores como um mecanismo automático para selar um antepara, não apenas para proteger o consumidor do problema a jusante, mas também para potencialmente proteger o serviço downstream de mais chamadas que possam estar tendo um impacto adverso. Considerando os perigos do fracasso em cascata, eu recomendo a obrigatoriedade de disjuntores para todos os seus downstream síncronos chamadas. Você também não precisa escrever o seu próprio, nos anos desde que escrevi

na primeira edição deste livro, as implementações de disjuntores se tornaram amplamente disponível.

Voltando à AdvertCorp, considere o problema que tivemos com o nabo. O sistema está respondendo muito lentamente antes de finalmente retornar um erro. Mesmo que se tivéssemos acertado os intervalos, esperaríamos muito tempo antes de chegarmos ao erro. E então tentaríamos novamente na próxima vez que uma solicitação chegassem, e esperar. Já era ruim o suficiente que o serviço de downstream estivesse funcionando mal, mas também estava diminuindo a velocidade de todo o sistema.

Com um disjuntor, após um certo número de solicitações para o downstream o recurso falhou (devido a um erro ou a um tempo limite), o disjuntor

enquanto o disjuntor está em seu estado queimado (aberto), como você pode ver na Figura 12-5.

Depois de um determinado período de tempo, o cliente envia algumas solicitações para ver se o serviço downstream se recuperou e se estiver suficientemente saudável responde: ele reinicia o disjuntor.

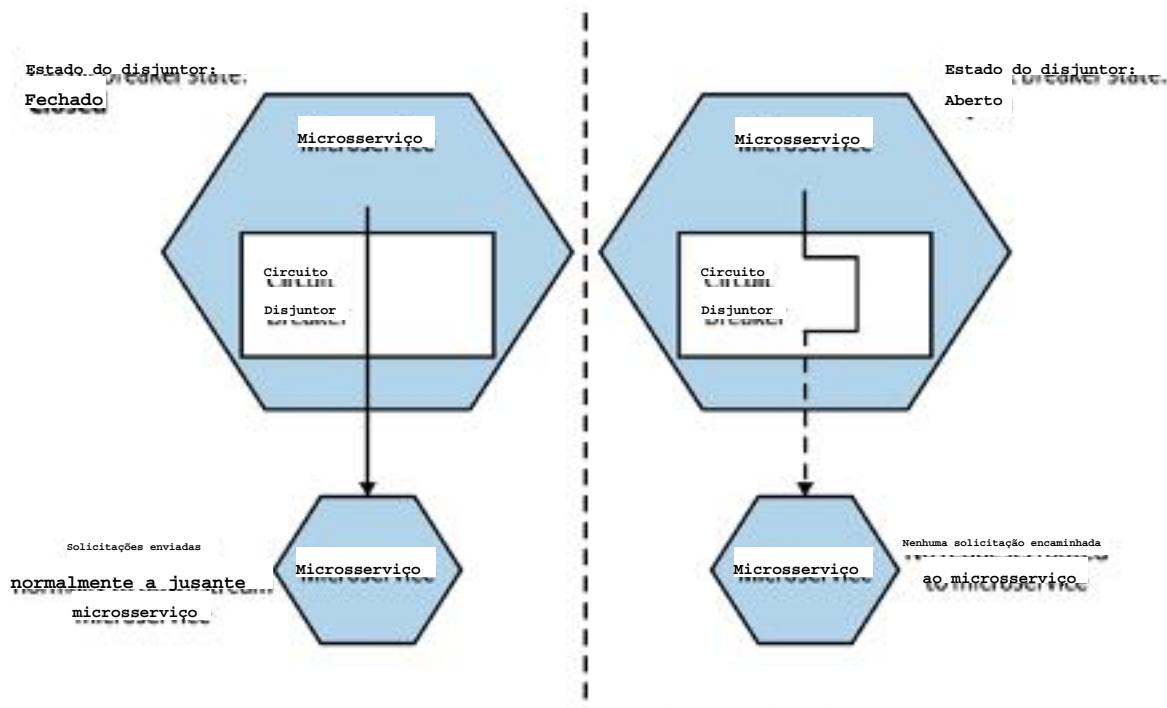


Figura 12-5 Uma visão geral dos disjuntores

A forma como você implementa um disjuntor depende do que é uma solicitação "falhada". Significa... mas quando eu os implementei para conexões HTTP, eu,

geralmente considerado falha em significar um tempo limite ou um subconjunto do 5XX. Códigos de retorno HTTP. Dessa forma, quando um recurso downstream está expirando ou retornando erros, após atingir um determinado limite, paramos automaticamente enviando tráfego e começando a falhar rapidamente. E podemos começar automaticamente de novo quando as coisas estão saudáveis.

Acertar as configurações pode ser um pouco complicado. Você não quer explodir o disjuntor muito rápido, nem você quer demorar muito para explodí-lo.

Da mesma forma, você realmente quer ter certeza de que o serviço downstream é saudável novamente antes de enviar tráfego. Assim como nos intervalos, eu escolheria alguns padrões sensatos e mantenha-os em todos os lugares e, em seguida, altere-os para casos específicos.

Enquanto o disjuntor está queimado, você tem algumas opções. Uma é entrar na fila faça as solicitações e tente novamente mais tarde. Para alguns casos de uso, isso pode ser apropriado, especialmente se você estiver realizando algum trabalho como parte de um trabalho assíncrono. Se essa chamada estiver sendo feita como parte de uma chamada síncrona cadeia, no entanto, provavelmente é melhor falhar rapidamente. Isso pode significar propagar um erro na cadeia de chamadas ou uma degradação mais sutil da funcionalidade.

No caso da AdvertCorp, agrupamos as chamadas downstream para o legado sistemas com disjuntores, como mostra a Figura 12-6. Quando esses circuitos Os disjuntores explodiram, atualizamos programaticamente o site para mostrar que No momento, não podia exibir anúncios de, digamos, nabos. Mantivemos o resto do site trabalhando e comunicando claramente aos clientes que havia um problema restrito a uma parte do nosso produto, tudo de forma totalmente automatizada.

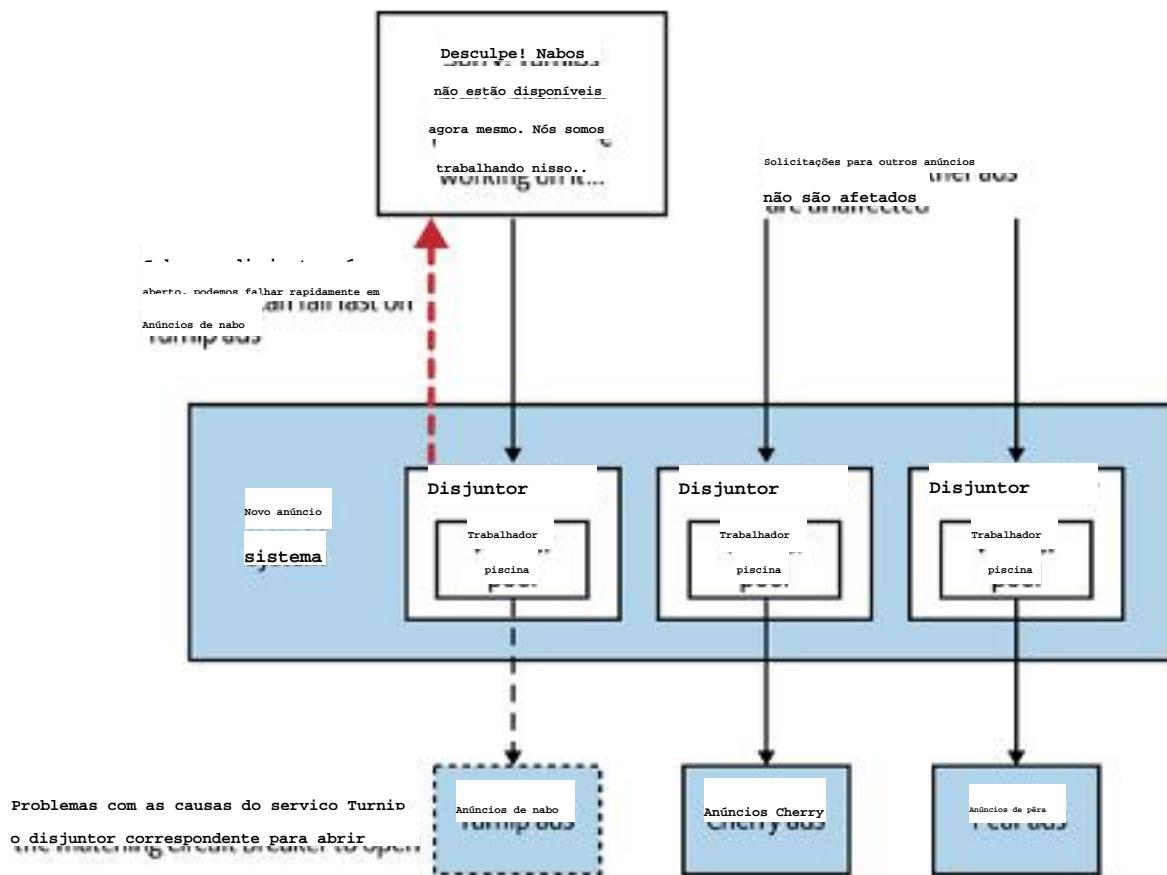


Figura 12-6. Adicionando disjuntores à AdvertCorp

Conseguimos dimensionar nossos disjuntores para que tivéssemos um para cada um dos sistemas legados posteriores - isso se alinhava bem com o fato de que tínhamos decidido ter diferentes grupos de solicitantes para cada serviço de downstream.

Se tivermos esse mecanismo instalado (como acontece com os disjuntores em nosso home), poderíamos usá-los manualmente para tornar mais seguro fazer nosso trabalho. Para exemplo, se quiséssemos desativar um microsserviço como parte da rotina manutenção, poderíamos abrir manualmente todos os disjuntores do upstream para que os consumidores falhem rapidamente enquanto o microsserviço está off-line. Quando estiver de volta, podemos fechar os disjuntores e tudo deve voltar ao normal.

Programando o processo para abrir e fechar manualmente um disjuntor como parte do um processo de implantação automatizado pode ser uma próxima etapa sensata.

Os disjuntores ajudam nosso aplicativo a falhar rapidamente - e falhar rapidamente é sempre melhor do que falhar lentamente. Os disjuntores nos permitem falhar antes de desperdiçar tempo (e recursos) valiosos esperando por um downstream insalubre.

microsserviço para responder. Em vez de esperar até tentarmos usar o
Se o microsserviço downstream falhar, poderíamos verificar o status do nosso circuito
disjuntores mais cedo. Se um microsserviço no qual confiaremos como parte de uma operação for
atualmente indisponível, podemos abortar a operação antes mesmo de começarmos.

Isolamento

Quanto mais um microsserviço depende de outro microsserviço ser
disponível, quanto mais a saúde de um afeta a capacidade do outro de fazer sua
trabalho. Se pudermos usar uma tecnologia que permita que um servidor downstream fique offline,
por exemplo, por meio do uso de middleware ou algum outro tipo de chamada
sistema de buffer, microsserviços upstream têm menos probabilidade de serem afetados por
interrupções, planejadas ou não, de microsserviços downstream.

Há outro benefício em aumentar o isolamento entre os serviços. Quando
os serviços são isolados uns dos outros, muito menos coordenação é necessária
entre proprietários de serviços. Quanto menor a coordenação necessária entre as equipes,
mais autonomia essas equipes têm, pois são capazes de operar e evoluir seus
serviços mais livremente.

O isolamento também se aplica em termos de como passamos do lógico para o
físico. Considere dois microsserviços que parecem estar totalmente isolados
um do outro. Eles não se comunicam entre si de forma alguma. UMA
O problema com um deles não deve impactar o outro, certo? Mas e se ambos
os microsserviços estão sendo executados no mesmo host e em um dos microsserviços
começa a usar toda a CPU, fazendo com que esse host tenha problemas?

Considere outro exemplo. Dois microsserviços cada um tem seu próprio,
banco de dados logicamente isolado. Mas os dois bancos de dados são implantados no mesmo
infraestrutura de banco de dados. Uma falha nessa infraestrutura de banco de dados afetaria
ambos os microsserviços.

Quando consideramos como queremos implantar nossos microsserviços, também queremos
esforçar-se para garantir um certo grau de isolamento de falhas para evitar problemas como esse.
Por exemplo, garantir que os microsserviços sejam executados em hosts independentes
com seu próprio sistema operacional e recursos de computação cercados é um
etapa sensata - isso é o que alcançamos quando executamos instâncias de microsserviços

em sua própria máquina virtual ou contêiner. Esse tipo de isolamento, no entanto, pode têm um custo.

Podemos isolar nossos microserviços uns dos outros com mais eficiência executando-os em máquinas diferentes. Isso significa que precisamos de mais infraestrutura, e ferramentas para gerenciar essa infraestrutura. Isso tem um custo direto e também pode aumentar a complexidade do nosso sistema, expondo novos caminhos de potencial falha. Cada microserviço pode ter seu próprio banco de dados totalmente dedicado àquele serviço, mas isso é mais infraestrutura para gerenciar. Nós poderíamos usar middleware para fornecer desacoplamento temporal entre dois microserviços, mas agora temos um corretor com o qual nos preocupar.

O isolamento, como muitas das outras técnicas que analisamos, pode ajudar a melhorar a robustez de nossos aplicativos, mas é raro que isso aconteça de forma gratuita. Decidindo sobre as compensações aceitáveis entre isolamento versus custo e o aumento da complexidade, como muitas outras coisas, pode ser vital.

Redundância

Ter mais de algo pode ser uma ótima maneira de melhorar a robustez de um componente. Ter mais de uma pessoa que sabe como fazer a produção ou o banco de dados funciona parece sensato, caso alguém saia da empresa ou seja fora de licença. Ter mais de uma instância de microserviço faz sentido, pois ele permite que você tolere a falha de uma dessas instâncias e ainda tenha uma chance de fornecer a funcionalidade necessária.

Descobrir quanta redundância você precisa e onde, dependerá de como bem, você entende os possíveis modos de falha de cada componente, o impacto de essa funcionalidade não estar disponível e o custo de adicionar a redundância.

Na AWS, por exemplo, você não recebe um SLA para o tempo de atividade de um único EC2 instância (máquina virtual). Você tem que trabalhar partindo do pressuposto de que pode e morrerá em você. Portanto, faz sentido ter mais de um. Mas indo além disso, as instâncias do EC2 são implantadas em zonas de disponibilidade (dados virtuais centros), e você também não tem garantias quanto à disponibilidade de uma zona de disponibilidade única, o que significa que você gostaria que a segunda instância estivesse em uma zona de disponibilidade diferente para distribuir o risco.

Ter mais cópias de algo pode ajudar na hora de implementar redundância, mas também pode ser benéfica quando se trata de escalar nosso aplicativos para lidar com o aumento da carga. No próximo capítulo, veremos exemplos de escalabilidade do sistema e veja como escalar para redundância ou escalabilidade para carga pode ser diferente.

Middleware

Em "Message Brokers", analisamos o papel do middleware na forma de corretores de mensagens para ajudar na implementação da solicitação-resposta e do evento interações baseadas. Uma das propriedades úteis da maioria dos corretores de mensagens é sua capacidade de fornecer entrega garantida. Você envia uma mensagem para um parte a jusante, e a corretora garante entregá-la, com algumas ressalvas que exploramos anteriormente. Internamente, para fornecer essa garantia, a mensagem o software do corretor terá que implementar coisas como novas tentativas e tempos limite em seu nome - os mesmos tipos de operações estão sendo realizados como você faria tem que fazer você mesmo, mas eles estão sendo feitos em software escrito por especialistas com um foco profundo nesse tipo de coisa. Ter pessoas inteligentes trabalhando para você geralmente é uma boa ideia.

Agora, no caso do nosso exemplo específico com a AdvertCorp, usando middleware para gerenciar a comunicação entre solicitação e resposta com o sistema de nabos a jusante pode não ter ajudado muito. Nós gostaríamos ainda não estamos recebendo respostas de nossos clientes. O único benefício potencial seria que estariamos aliviando a disputa de recursos em nosso próprio sistema, mas isso apenas mudaria para um número crescente de solicitações pendentes sendo retidas no corretor. Pior ainda, muitos desses pedidos solicitam o último nabo os preços podem estar relacionados a solicitações de usuários que não são mais válidas.

Uma alternativa poderia ser inverter a interação e usar o middleware para que o sistema de nabos transmitisse os últimos anúncios de nabo, e poderíamos então consumi-los. Mas se o sistema de nabos a jusante tivesse um problema, ainda assim não seria capaz de ajudar o cliente a procurar os melhores preços de nabo. Então, usando middleware, como corretores de mensagens, para ajudar a liberar um pouco de robustez preocupações podem ser úteis, mas não em todas as situações.

Idempotência

Em operações idempotentes, o resultado não muda após a primeira aplicação, mesmo que a operação seja aplicada posteriormente várias vezes. Se as operações são idempotentes, podemos repetir a chamada várias vezes sem impacto adverso. Isso é muito útil quando queremos reproduzir mensagens que não temos certeza se foram processados, uma forma comum de se recuperar de erro.

Vamos considerar uma simples chamada para adicionar alguns pontos como resultado de um dos nossos clientes fazendo um pedido. Podemos fazer uma ligação com o tipo de carga mostrado no Exemplo 12-1.

Exemplo 12-1. Creditar pontos em uma conta

```
<credit>
  <amount>100</amount>
  </account><forAccount>1234
</credit>
```

Se essa chamada for recebida várias vezes, adicionariamos 100 pontos múltiplos vezes. Do jeito que está, portanto, esse chamado não é idempotente. Com um pouco mais de informações, no entanto, permitimos que o banco de pontos torne essa chamada idempotente, conforme mostrado no Exemplo 12-2.

Exemplo 12-2. Adicionando mais informações ao crédito de pontos para fazer isso idempotente

```
<credit>
  <quantidade>100</amount>
  <fo></account>Ou conta > 1234
  <reason>
    <forPurchase>4567</forPurchase>
  </reason>
</credit>
```

Sabemos que esse crédito está relacionado a um pedido específico, 4567. Supondo que nós poderíamos receber apenas um crédito para um determinado pedido, poderíamos aplicar esse crédito novamente sem aumentar o número total de pontos.

Esse mecanismo funciona da mesma forma com a colaboração baseada em eventos e pode ser especialmente útil se você tiver várias instâncias do mesmo tipo de

serviço de assinatura de eventos. Mesmo se armazenarmos quais eventos foram processado, com algumas formas de entrega de mensagens assíncronas, pode haver pequenas janelas nas quais dois trabalhadores podem ver a mesma mensagem. Por processando os eventos de forma idempotente, garantimos que isso não nos causará quaisquer problemas.

Algumas pessoas se envolvem bastante com esse conceito e presumem que isso significa que chamadas subsequentes com os mesmos parâmetros não podem ter nenhum impacto, o que então nos deixa em uma posição interessante. Nós realmente ainda gostaríamos de gravar o fato de que uma chamada foi recebida em nossos registros, por exemplo. Queremos gravar o tempo de resposta da chamada e coleta desses dados para monitoramento. O ponto chave aqui está que é a operação comercial subjacente que estamos considerando idempotente, não todo o estado do sistema.

Alguns dos verbos HTTP, como GET e PUT, são definidos no HTTP especificação para ser idempotente, mas para que esse seja o caso, eles confiam em sua serviço que lida com essas chamadas de forma idempotente. Se você começar a fazer esses verbos não idempotentes, mas os chamadores acham que podem executá-los com segurança repetidamente, você pode se meter em uma bagunça. Lembre-se, só porque você é usar HTTP como um protocolo subjacente não significa que você obtenha tudo para grátis!

Divulgando seu risco

Uma forma de escalar a resiliência é garantir que você não coloque todos os ovos uma cesta. Um exemplo simplista disso é garantir que você não tenha vários serviços em um host, onde uma interrupção afetaria vários serviços. Mas vamos considerar o que significa hospedeiro. Na maioria das situações atuais, um "host" é na verdade um conceito virtual. E se eu tiver todos os meus serviços ativados? hosts diferentes, mas todos esses hosts são, na verdade, hosts virtuais, executados na mesma caixa física? Se essa caixa cair, eu poderia perder vários serviços. Algumas plataformas de virtualização permitem que você garanta que seus hosts sejam distribuído em várias caixas físicas diferentes para reduzir a chance de isso está acontecendo.

Para plataformas de virtualização internas, é uma prática comum ter o partição raiz da máquina virtual mapeada para uma única SAN (área de armazenamento rede). Se essa SAN cair, ela poderá derrubar todas as VMs conectadas. SANS são grandes, caros e projetados para não falhar. Dito isso, eu tive um grande, SANS caras falham comigo pelo menos duas vezes nos últimos 10 anos, e todas as vezes os resultados foram bastante sérios.

Outra forma comum de separação para reduzir falhas é garantir que nem todas seus serviços estão sendo executados em um único rack no data center ou que seu os serviços são distribuídos em mais de um data center. Se você estiver usando um provedor de serviços subiacente, é importante saber se um SLA é oferecido e para planejar adequadamente. Se você precisar garantir que seus serviços estejam inativos por enquanto mais de quatro horas por trimestre, mas seu provedor de hospedagem só pode garantir um tempo máximo de inatividade de oito horas por trimestre, você precisa altere o SLA ou então encontre uma solução alternativa.

A AWS, por exemplo, é dividida em regiões, que você pode considerar distintas nuvens. Cada região, por sua vez, é dividida em duas ou mais zonas de disponibilidade, como discutido anteriormente. Essas zonas de disponibilidade são o equivalente da AWS a um dado centro. É essencial ter serviços distribuídos em várias disponibilidades zonas, pois a AWS não oferece nenhuma garantia sobre a disponibilidade de uma única nó ou até mesmo uma zona de disponibilidade inteira. Para seu serviço de computação, ele oferece apenas um tempo de atividade de 99,95% em um determinado período mensal da região como um todo, então você vai querer distribuir suas cargas de trabalho em várias disponibilidades zonas dentro de uma única região. Para algumas pessoas, isso não é bom o suficiente, e eles também administram seus serviços em várias regiões.

Deve-se notar, é claro, que porque os provedores fornecem um SLA "Garantia", eles tenderão a limitar sua responsabilidade! Se eles errarem seus alvos resulta na perda de clientes e de uma grande quantia de dinheiro, você pode descobrir você mesmo pesquisando contratos para ver se consegue recuperar alguma coisa deles Portanto, sugiro fortemente que você entenda o impacto de um fornecedor que não cumpre suas obrigações com você e descubra se você precisa ter um plano B (ou C) no seu bolso. Mais de um cliente com quem trabalhei teve uma plataforma de hospedagem de recuperação de desastres com um fornecedor diferente, por exemplo, para garantir que eles não estivessem muito vulneráveis aos erros de uma empresa.

Teorema CAP

Gostaríamos de ter tudo, mas infelizmente sabemos que não podemos. E quando chega a sistemas distribuídos, como aqueles que construímos usando microserviços arquiteturas, temos até uma prova matemática que nos diz que não podemos. Você pode muito bem ter ouvido falar sobre o teorema CAP, especialmente em discussões sobre os méritos de vários tipos diferentes de armazenamentos de dados. Em sua essência, isso nos diz que em um sistema distribuído, temos três coisas que podemos negociar com cada uma outros: consistência, disponibilidade e tolerância à partição. Especificamente, o teorema nos diz que podemos manter dois em um modo de falha.

Consistência é a característica do sistema pela qual obteremos o mesmo resposta se formos para vários nós. Disponibilidade significa que cada solicitação recebe uma resposta. A tolerância de partição é a capacidade do sistema de lidar com o fato de que a comunicação entre suas partes às vezes é impossível.

Desde que Eric Brewer publicou sua conjectura original, a ideia ganhou uma prova matemática. Não vou mergulhar na matemática da prova em si, pois não só não é esse tipo de livro, mas também posso garantir que obteria está errado. Em vez disso, vamos usar alguns exemplos práticos que nos ajudarão a entender que, no fundo, o teorema do CAP é uma destilação de um conjunto lógico de raciocínio.

Vamos imaginar que nosso microserviço de inventário seja implantado em dois data centers separados, conforme mostrado na Figura 12-7. Apoiando nossa instância de serviço em cada data center há um banco de dados, e esses dois bancos de dados conversam entre si para tentar sincronizar dados entre eles. As leituras e gravações são feitas por meio do nó de banco de dados local, e a replicação é usada para sincronizar os dados entre os nós.

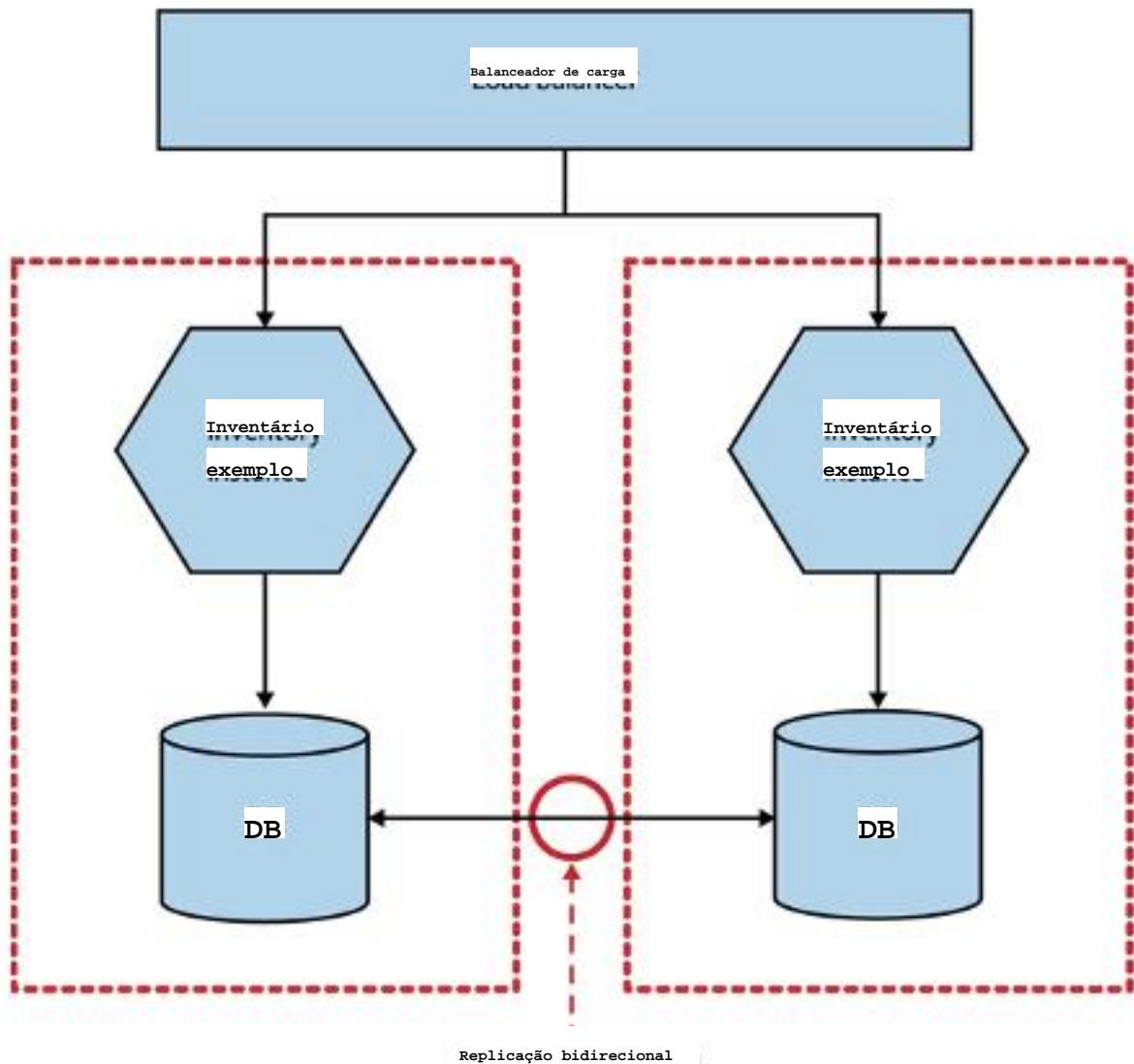


Figura 12-7. Usando replicação multiprimary para compartilhar dados entre dois nós do banco de dados

Agora vamos pensar no que acontece quando algo falha. Imagine isso... algo tão simples quanto o link de rede entre os dois data centers é interrompido... trabalhando, A sincronização nesse momento falha. Gravações feitas no primário o banco de dados em DC1 não se propagarão para DC2 e vice-versa. A maioria dos bancos de que suportam essas configurações também suportam algum tipo de técnica de enfileiramento para garantir que possamos nos recuperar disso depois, mas o que acontece no Enquanto isso?

Sacrificando a consistência

Vamos supor que não encerramos totalmente o microsserviço de inventário...

Se eu fizer uma alteração agora nos dados no DC1, o banco de dados no DC2 não verá isso. Isso significa que todas as solicitações feitas ao nosso nó de inventário no DC2, consulte dados potencialmente obsoletos. Em outras palavras, nosso sistema ainda está disponível nesse sentido ambos os nós são capazes de atender às solicitações e mantivemos o sistema funcionando apesar da partição, mas perdemos a consistência; não conseguimos manter tudo três características. Isso geralmente é chamado de sistema AP, devido à sua disponibilidade e tolerância de partição.

Durante essa partição, se continuarmos aceitando gravações, aceitaremos o fato de que em algum momento no futuro, eles precisarão ser resincronizados. Quanto mais longo o a partição dura, quanto mais difícil essa resincronização pode se tornar.

A realidade é que, mesmo que não tenhamos uma falha de rede entre nossos nós do banco de dados, a replicação dos dados não é instantânea. Conforme mencionado anteriormente, sistemas que ficavam felizes em ceder consistência para manter a tolerância à partição e diz-se que a disponibilidade é eventualmente consistente: ou seja, esperamos que em algum momento no futuro, todos os nós verão os dados atualizados, mas isso não acontecerá acontecem ao mesmo tempo, então temos que conviver com a possibilidade de os usuários verem coisas antigas dados.

Sacrificando a disponibilidade

O que acontece se precisarmos manter a consistência e quisermos abandonar algo outra coisa em vez disso? Bem, para manter a consistência, cada nó do banco de dados precisa saber a cópia dos dados que ele tem é a mesma do outro nó do banco de dados. Agora, na partição, se os nós do banco de dados não conseguirem se comunicar entre si, eles não poderão coordenar para garantir a consistência. Não podemos garantir a consistência, então nossa única opção é recusar-se a responder à solicitação. Em outras palavras, nós sacrificaram a disponibilidade. Nosso sistema é consistente e tolerante a partções, ou CP. Nesse modo, nosso serviço teria que descobrir como se degradar funcionalidade até que a partição seja recuperada e os nós do banco de dados possam ser resincronizado.

A consistência entre vários nós é muito difícil. Há poucas coisas (talvez nada) mais difícil em sistemas distribuídos. Pense nisso por um momento

Imagine que eu queira ler um registro do nó do banco de dados local. Como eu faço para saber que está atualizado? Eu tenho que ir e perguntar ao outro nodo. Mas eu também tenho que peça a esse nó do banco de dados que não permita que ele seja atualizado durante a leitura completa; em outras palavras, preciso iniciar uma leitura transacional. Vários nós do banco de dados para garantir a consistência. Mas, em geral, as pessoas não fazem leituras transacionais, não é? Porque as leituras transacionais são lentas. Eles exigem bloqueios. Uma leitura pode bloquear um sistema inteiro.

Como já discutimos, os sistemas distribuídos precisam esperar falhas. Considere nossa leitura transacional em um conjunto de nós consistentes. Eu pergunto a nó remoto para bloquear um determinado registro enquanto a leitura é iniciada. Eu concluo o leia e peço ao nó remoto que libere o bloqueio, mas agora não consigo falar com ele. O que acontece agora? As fechaduras são muito difíceis de acertar, mesmo em uma única sistema de processo e são significativamente mais difíceis de implementar bem em um sistema distribuído.

Lembra quando falamos sobre transações distribuídas no Capítulo 6? A principal razão pela qual eles são desafiadores é por causa desse problema de garantir consistência em vários nós.

Obter a consistência correta de vários nós é tão difícil que eu gostaria muito, muito sugiro que, se você precisar, não tente inventá-lo sozinho. Em vez disso, escolha um dado serviço de loja ou bloqueio que oferece essas características. Consul, por exemplo, que discutimos em "Registros dinâmicos de serviços", implementa um forte armazenamento consistente de valores-chave projetado para compartilhar a configuração entre vários nós. Junto com "Amigos não permitem que amigos escrevam suas próprias criptomoedas" deveria ser "Amigos não permitem que amigos escrevam seus próprios dados consistentes distribuídos". Se você acha que precisa escrever seu próprio armazenamento de dados CP, leia todas as artigos sobre o assunto primeiro, depois obtenha um doutorado e, em seguida, espere gastar alguns anos errando. Enquanto isso, vou usar algo da prateleira. Isso faz isso por mim, ou, mais provavelmente, tentando muito construir, eventualmente. Em vez disso, sistemas AP consistentes.

Sacrificando a tolerância à partição?

Podemos escolher dois, certo? Então, temos nosso sistema AP eventualmente consistente.

Temos nosso sistema de CP consistente, mas difícil de construir e escalar. Por que não um Sistema CA? Bem, como podemos sacrificar a tolerância à partição? Se nosso sistema não tem tolerância de partição, não pode ser executado em uma rede. Em outras palavras, é preciso ser um único processo operando localmente. Os sistemas CA não existem em sistemas distribuídos.

AP ou CP?

O que é certo, AP ou CP? Bem, a realidade é que isso depende. Como as pessoas construindo o sistema, sabemos que a compensação existe. Sabemos que os sistemas AP escalam com mais facilidade e são mais simples de construir, e sabemos que um sistema CP exigirá mais trabalho devido aos desafios no suporte distribuído consistência. Mas talvez não entendamos o impacto comercial dessa compensação. Para nosso sistema de inventário, se um registro estiver desatualizado por cinco minutos, é que OK? Se a resposta for sim, um sistema AP pode ser a resposta. Mas e quanto o saldo retido por um cliente em um banco? Isso pode estar desatualizado? Sem sabendo o contexto em que a operação está sendo usada, não podemos saber o coisa certa a fazer. Conhecer o teorema CAP ajuda você a entender que essa compensação existe e quais perguntas fazer.

Não é tudo ou nada

Nosso sistema como um todo não precisa ser AP ou CP. Nosso catálogo para A MusicCorp pode ser AP, já que não nos preocupamos muito com um disco obsoleto. Mas podemos decidir que nosso serviço de inventário precisa ser CP, pois não quero vender a um cliente algo que não temos e depois pedir desculpas.

mais tarde.

Mas os serviços individuais nem precisam ser CP ou AP.

Vamos pensar em nosso microsserviço Points Balance, onde armazenamos registros de quantos pontos de fidelidade nossos clientes acumularam. Nós poderíamos decidirmos que não nos importamos se o saldo que mostramos para um cliente está obsoleto, mas que, quando se trata de atualizar uma balança, precisamos que ela seja consistente para garantir que os clientes não usem mais pontos do que os que têm disponíveis. É isso.

microservice CP, AP ou ambos? Realmente, o que fizemos foi empurrar o... compensações em torno do teorema CAP até o microserviço individual capacidades.

Outra complexidade é que nem a consistência nem a disponibilidade são tudo ou nada. Muitos sistemas nos permitem uma compensação muito mais sutil. Por exemplo, com a Cassandra, posso fazer diferentes compensações para chamadas individuais. Então, se eu preciso de consistência estrita, posso realizar uma leitura que bloqueia até todas as réplicas responderam, confirmado que o valor é consistente ou até um quórum específico das réplicas responderam, ou mesmo apenas um único nó. Obviamente, se eu bloquear esperando que todas as réplicas sejam reportadas e uma delas não esteja disponível, eu estarei bloqueando por um longo tempo. Por outro lado, se eu quisesse que minha leitura respondesse como o mais rápido possível, talvez eu espere para receber uma resposta de apenas um único node-in. Nesse caso, existe a possibilidade de que essa seja uma visão inconsistente da minha dados.

Freqüentemente, você verá postagens sobre pessoas "vencendo" o teorema do CAP. Eles não tenho. O que eles fizeram foi criar um sistema no qual alguns recursos são CP e alguns são AP. A prova matemática por trás do teorema CAP segura.

E o mundo real

Muito do que falamos são os bits e bytes do mundo eletrônico armazenado na memória. Falamos sobre consistência de uma forma quase infantil: imaginamos que, dentro do escopo do sistema que criamos, podemos parar o mundo e faça com que tudo faça sentido. E, no entanto, muito do que construímos é apenas um reflexo do mundo real, e não podemos controlar isso, não é?

Vamos revisitar nosso sistema de inventário. Isso mapeia itens físicos do mundo real. Contamos em nosso sistema quantos álbuns temos no MusicCorp armazéns. No início do dia, tínhamos 100 cópias de Give Blood by the Freios. Nós vendemos um. Agora temos 99 cópias. Fácil, certo? Mas o que acontece se, quando o pedido estiver sendo enviado, alguém colocar uma cópia do álbum no chão e ele é pisado e quebrado? O que acontece agora? Nossos sistemas digamos que há 99 cópias na prateleira, mas na verdade existem apenas 98.

E se, em vez disso, fizéssemos nosso sistema de inventário AP e, ocasionalmente, tivéssemos para entrar em contato com um usuário mais tarde e dizer a ele que um de seus itens está realmente fora de estoque? Isso seria a pior coisa do mundo? Certamente seria muito mais fácil de construir e escalar e garantir que esteja correto.

Temos que reconhecer que não importa o quanto consistentes nossos sistemas possam ser em si mesmos, eles não podem saber tudo o que acontece, especialmente quando mantemos registros do mundo real. Este é um dos principais razões pelas quais os sistemas AP acabam sendo a escolha certa em muitas situações. À parte devido à complexidade da construção de sistemas de CP, eles não conseguem resolver todos os nossos problemas de qualquer forma.

ANTIFRAGILIDADE

Na primeira edição, falei sobre o conceito de antifrágil, como popularizado por Nassim Taleb. Este conceito descreve como os sistemas na verdade se beneficiam de falhas e transtornos e foi destacado como sendo inspiração de como algumas partes da Netflix operavam, especificamente com respeito a conceitos como engenharia do caos. Ao olhar de forma mais ampla no conceito de resiliência, porém, percebemos que a antifragilidade é apenas uma subconjunto do conceito de resiliência. Quando consideramos os conceitos de extensibilidade elegante e adaptabilidade sustentada, que introduzimos mais cedo, isso fica claro....

Acho que quando a antifragilidade se tornou, brevemente, um conceito badalado em TI, fez isso em um cenário em que pensávamos estreitamente sobre resiliência onde pensávamos apenas em robustez e talvez em recuperação, mas ignorou o resto. Com o campo da engenharia de resiliência agora ganhando mais reconhecimento e tracção, parece apropriado ir além do termo antifrágil, ao mesmo tempo que garante que destaquemos algumas das ideias por trás disso, que são realmente parte da resiliência como um todo.

Engenharia do Caos

Desde a primeira edição deste livro, outra técnica que ganhou mais atenção é engenharia do caos. Nomeado em homenagem às práticas usadas na Netflix, pode ser uma abordagem útil para ajudar a melhorar sua resiliência, seja em termos de garantindo que seus sistemas sejam tão robustos quanto você pensa que são, ou então como parte de uma abordagem para a adaptabilidade sustentada do seu sistema.

Originalmente inspirado no trabalho que a Netflix estava fazendo internamente, o termo caos a engenharia tem dificuldades um pouco devido a explicações confusas sobre o que é significa. Para muitos, significa "executar uma ferramenta no meu software e ver o que ela diz" algo que provavelmente não ajudou pelo fato de muitos dos mais vocais os proponentes da engenharia do caos geralmente estão vendendo ferramentas para rodar seu software para fazer, bem, engenharia do caos.

Obter uma definição clara do que a engenharia do caos significa para seus praticantes é difícil. A melhor definição (pelo menos na minha opinião) que encontrei é isso:

A Engenharia do Caos é a disciplina de experimentar em um sistema em a fim de criar confiança na capacidade de resistência do sistema condições turbulentas na produção.

-Princípios da Engenharia do Caos

Agora, o sistema de palavras aqui está fazendo muito trabalho. Alguns verão isso estreitamente como os componentes de software e hardware. Mas no contexto de engenharia de resiliência, é importante que vejamos o sistema como sendo o totalidade das pessoas, dos processos, da cultura e, sim, do software e infraestrutura necessária para a criação de nosso produto. Isso significa que devemos ver engenharia do caos de forma mais ampla do que apenas "vamos desligar algumas máquinas e veja o que acontece."

Dias de jogo

Bem antes de a engenharia do caos ter seu nome, as pessoas organizavam o Game Day exercícios para testar a preparação das pessoas para determinados eventos. Planejado em antecipado, mas idealmente lançado de surpresa (para os participantes), isso dá a você a chance de testar seu pessoal e seus processos na sequência de um teste realista, mas situação fictícia. Durante meu tempo no Google, isso era bastante comum.

ocorrência de vários sistemas, e eu certamente acho que muitas organizações poderia se beneficiar de fazer esses tipos de exercícios regularmente. O Google vai além de testes simples para imitar a falha do servidor e como parte de seu DiRT (Disaster) Exercícios de recuperação (Teste de Recuperação): simulou desastres em grande escala, como terremotos.⁶

Os Game Days podem ser usados para investigar fontes suspeitas de fracasso no sistema. Em seu livro, "Learning Chaos Engineering", Russ Miles compartilha o exemplo de um exercício do Game Day que ele facilitou que foi projetado para examinar, em parte, a dependência excessiva de um único membro da equipe chamado Bob. Para No Dia do Jogo, Bob foi sequestrado em uma sala e incapaz de ajudar a equipe durante a interrupção simulada. Bob estava observando, porém, e acabou tendo intervindo quando a equipe, em sua tentativa de corrigir problemas com o sistema "falso", acabou entrando na produção por engano e estava no processo de destruindo dados de produção. Só podemos supor que muitas aulas foram aprendido após esse exercício.

Experimentos de produção

A escala em que a Netflix opera é bem conhecida, assim como o fato de a Netflix é baseado inteiramente na infraestrutura da AWS. Esses dois fatores significam que tem que aceitar bem o fracasso. A Netflix percebeu que planejar o fracasso é na verdade, saber que seu software lidará com essa falha quando ela ocorrer é duas coisas diferentes. Para esse fim, a Netflix realmente incita o fracasso em garantir que seus sistemas são tolerantes a falhas ao executar ferramentas em seus sistemas.

A mais famosa dessas ferramentas é o Chaos Monkey, que durante certas horas of the day desliga máquinas aleatórias em produção. Sabendo que isso pode e acontecer na produção significa que os desenvolvedores que criam os sistemas realmente precisam estar preparados para isso. Chaos Monkey é apenas uma parte do Exército Símio de bots fracassados da Netflix. Chaos Gorilla é usado para eliminar uma toda a zona de disponibilidade (o equivalente da AWS a um data center), enquanto O Latency Monkey simula uma conectividade de rede lenta entre máquinas. Para muitos, o teste definitivo para saber se seu sistema é realmente robusto pode ser liberando seu próprio Exército Símio em sua infraestrutura de produção.

Da robustez ao mais

Aplicada em sua forma mais restrita, a engenharia do caos pode ser uma atividade útil em termos de melhorar a robustez de nossa aplicação. Lembre-se, robustez no contexto da resiliência, engenharia significa até que ponto nosso sistema pode lidar com os problemas esperados. A Netflix sabia que não podia confiar em nada a máquina virtual está disponível em seu ambiente de produção, por isso foi construída Chaos Monkey para garantir que seu sistema sobreviva a esse problema esperado.

Se você usar ferramentas de engenharia do caos como parte de uma abordagem para questione continuamente a resiliência do seu sistema, no entanto, ele pode ter muito maior aplicabilidade. Usando ferramentas neste espaço para ajudar a responder ao "e se" perguntas que você possa ter, questionando continuamente sua compreensão, podem têm um impacto muito maior. O Chaos Toolkit é um projeto de código aberto para ajudam você a executar experimentos em seu sistema, e ele se mostrou muito popular. De forma confiável, a empresa fundada pelos criadores do Chaos Toolkit oferece um uma gama mais ampla de ferramentas para ajudar na engenharia do caos em geral, embora talvez o fornecedor mais conhecido nesse espaço seja a Gremlin.

Lembre-se de que executar uma ferramenta de engenharia do caos não o torna resiliente.

Culpa

Quando as coisas dão errado, há muitas maneiras de lidar com isso. Obviamente, em logo depois, nosso foco é fazer com que as coisas voltem a funcionar, o que é sensato. Depois disso, muitas vezes, vêm as recriminações. Há uma posição padrão para procurar algo ou alguém para culpar. O conceito de A "análise da causa raiz" implica que há uma causa raiz. É surpreendente a frequência queremos que a causa raiz seja humana.

Há alguns anos, quando eu trabalhava na Austrália, a Telstra, a principal A empresa de telecomunicações (e o monopólio anterior) tiveram uma grande interrupção que afetou ambas as vozes e serviços de telefonia. Este incidente foi especialmente problemático devido ao escopo e duração da interrupção. A Austrália tem muitas áreas rurais muito isoladas comunidades e interrupções como essa tendem a ser especialmente graves. Quase

imediatamente após a interrupção, o COO da Telstra divulgou uma declaração dizendo que claro que eles sabiam exatamente o que havia causado o problema:⁸

"Nós removemos esse módulo, infelizmente, o indivíduo que foi gerenciar esse problema não seguiu o procedimento correto, e ele reconectou os clientes ao nó com defeito, em vez de transferindo-os para os outros nove nós redundantes que ele deveria ter transferiu pessoas para, disse a Sra. McKenzie a repórteres na terça-feira tarde.

"Pedimos desculpas em toda a nossa base de clientes. Isso é embarracoso erro humano."

Então, primeiro, observe que essa declaração saiu horas após a interrupção. E ainda assim A Telstra já havia desempacotado o que deve ser um sistema muito complexo de se conhecer exatamente o que era culpar uma pessoa. Agora, se é verdade que uma pessoa cometer um erro pode realmente colocar uma empresa de telecomunicações inteira de joelhos, você pensaria que diria mais sobre a empresa de telecomunicações do que sobre o indivíduo. Além disso, a Telstra sinalizou claramente para sua equipe na época que estava muito feliz em apontar o dedo e atribua a culpa.⁹

O problema de culpar as pessoas após incidentes como esse é que o que comece como um repasse de dinheiro a curto prazo acaba criando uma cultura de medo, onde as pessoas não estarão dispostas a se apresentar para lhe dizer quando as coisas correm mal. Como resultado, você perderá a oportunidade de aprender com falha, preparando-se para que os mesmos problemas aconteçam novamente. Criando uma organização na qual as pessoas têm a segurança de admitir quando cometem erros feito é essencial para criar uma cultura de aprendizagem e, por sua vez, pode ajudar muito para criar uma organização que seja capaz de criar um software mais robusto, além dos benefícios óbvios de criar um lugar mais feliz para trabalhar.

Voltando à Telstra, com a causa da culpa claramente estabelecida no investigaçāo aprofundada que foi realizada poucas horas depois do interrupção, claramente não esperamos interrupções subsequentes, certo? Infelizmente, A Telstra sofreu uma série de interrupções subsequentes. Mais erro humano? Talvez A Telstra achou que sim, após a série de incidentes, o COO renunciou.

Para uma visão mais informada sobre como criar uma organização onde você possa obter tire o melhor proveito dos erros e também crie um ambiente melhor para sua equipe, "Diálogos Post-Mortem's and a Just Culture", de John Allspaw, é uma ótima ponto de partida. 10

Em última análise, como já destaquei várias vezes neste capítulo, a resiliência requer uma mente questionadora - um impulso para examinar constantemente o fraquezas em nosso sistema. Isso requer uma cultura de aprendizado e, muitas vezes, o melhor aprendizado pode surgir na sequência de um incidente. Portanto, é vital que você garanta que, quando o pior acontecer, você faça o possível para criar um ambiente no qual você pode maximizar as informações coletadas no consequências para reduzir as chances de isso acontecer novamente.

Resumo

À medida que nosso software se torna mais vital para a vida de nossos usuários, o drive no sentido de melhorar a resiliência do software que criamos aumenta. Como vimos neste capítulo, porém, não podemos alcançar a resiliência apenas por pensando em nosso software e infraestrutura; também temos que pensar em nossas pessoas, processos e organizações. Neste capítulo, analisamos os quatro conceitos básicos de resiliência, conforme descrito por David Woods:

Robustez

A capacidade de absorver a perturbação esperada.

Recuperação

A capacidade de se recuperar após um evento traumático.

Extensibilidade elegante

Como lidamos bem com uma situação inesperada

Adaptabilidade sustentada

A capacidade de se adaptar continuamente a ambientes em mudança, partes interessadas, e demandas

Olhando de forma mais restrita para os microserviços, eles nos oferecem várias maneiras de onde podemos melhorar a robustez de nossos sistemas. Mas isso melhorou. A robustez não é gratuita, você ainda precisa decidir quais opções usar. Chaves, padrões de estabilidade, como disjuntores, tempos limite, redundância, isolamento, idempotência e similares são todas as ferramentas à sua disposição, mas você precisa decidir quando e onde usá-los. Além desses conceitos estreitos, porém, também precisamos estar constantemente atentos ao que não sabemos.

Você também precisa descobrir quanta resiliência deseja - e isso é quase sempre algo definido pelos usuários e proprietários de negócios do seu sistema. Como tecnólogo, você pode ser responsável por como as coisas são feitas, mas sabendo qual resiliência é necessária exigirá uma comunicação próxima, boa e frequente com usuários e proprietários de produtos.

Vamos voltar à citação de David Woods do início do capítulo, que usamos ao discutir a adaptabilidade sustentada:

Não importa o quanto bem tenhamos feito antes, não importa o quanto bem-sucedidos sejam já estivemos, o futuro pode ser diferente e talvez não estejamos bem adaptado. Podemos ser precários e frágeis diante desse novo futuro.

Fazer as mesmas perguntas repetidamente não ajuda você a entender se você estiver preparado para um futuro incerto. Você não sabe o que não sabe. Adotando o conhecimento, uma abordagem na qual você está constantemente aprendendo, constantemente questionar é fundamental para construir resiliência.

Um tipo de padrão de estabilidade que analisamos, a redundância, pode ser muito eficaz. Essa ideia segue muito bem em nosso próximo capítulo, onde veremos maneiras diferentes de escalar nossos microserviços, o que, além de nos ajudar lidar com mais carga também pode ser uma forma eficaz de nos ajudar a implementar redundância em nossos sistemas e, portanto, também melhoramos a robustez de nossos sistemas.

¹ David D. Woods, "Quatro conceitos para resiliência e as implicações para o futuro da Engenharia de Resiliência", Engenharia de Confiabilidade e Segurança do Sistema 141 (setembro de 2015): 5-9, doi.org/10.1016/j.ress.2015.03.018.

2 Jason Bloomberg, "Inovação: o outro lado da resiliência", Forbes, 23 de setembro de 2014.

<https://oreil.ly/aySmU>.

3 Consulte "Google Uncloaks Once-Secret Server", de Stephen Shankland, para obter mais informações sobre isso,

incluindo uma visão geral interessante de por que o Google acha que essa abordagem pode ser superior à tradicional Sistemas UPS.

4 Michael T. Nygard, *solté isso! Projete e implante software pronto para produção*, 2ª ed.

(Raleigh: Pragmatic Bookshelf, 2018).

5 A terminologia de um disjuntor "aberto", ou seja, as solicitações não podem fluir, pode ser confusa, mas

vem de circuitos elétricos. Quando o disjuntor está "aberto", o circuito está interrompido e está em corrente não pode fluir. Fechar o disjuntor permite que o circuito seja concluído e a corrente flua uma vez novamente.

6 Kripa Krishnan, "Weathering the Unexpected", conquista a 10ª posição nº 9 (2012).

<https://oreil.ly/BCS07>.

7 Russ Miles, *Learning Chaos Engineering* (Sebastopol: O'Reilly, 2019).

8 Kate Aubusson e Tim Biggs, "Uma grande interrupção do celular da Telstra atinge todo o país com chamadas e

Data Affected", Sydney Morning Herald, 9 de fevereiro de 2016. <https://oreil.ly/4cBcy>.

9 Escrevi mais sobre o incidente da Telstra na época "Telstra, Human Error and Blame Culture".

<https://oreil.ly/OXqUO>. Quando escrevi aquela postagem no blog, escueci de perceber que a empresa trabalhou para a Telstra como cliente; meu então empregador realmente lidou com a situação de forma extrema. Bem, apesar de ter sido incômodo algumas horas quando alguns dos meus comentários foram recebidos captado pela imprensa nacional.

10 John Allspaw, "Blameless Post-Mortems and a Just Culture", Code as Craft (blog), Etsy, maio

22, 2012. <https://oreil.ly/7LamI>.

Capítulo 13. Dimensionamento

"Você vai precisar de um barco maior."

-- Chefe Brody, Jaws

Quando escalamos nossos sistemas, fazemos isso por um dos dois motivos. Em primeiro lugar, nos permite melhorar o desempenho do nosso sistema, talvez nos permitindo para lidar com mais carga ou melhorando a latência. Em segundo lugar, podemos escalar nosso sistema para melhorar sua robustez. Neste capítulo, veremos um modelo para descreva os diferentes tipos de dimensionamento e, em seguida, veremos detalhadamente como cada tipo de escalabilidade pode ser implementado usando uma arquitetura de microserviços. No final deste capítulo, você deve ter uma série de técnicas para lidar os problemas de escala que podem surgir em seu caminho.

Para começar, porém, vamos dar uma olhada nos diferentes tipos de escalabilidade que você pode usar querer se inscrever.

Os quatro eixos do dimensionamento

Não existe uma maneira correta de escalar um sistema, pois a técnica usada será depende do tipo de restrição que você possa ter. Temos uma série de diferentes tipos de escalabilidade que podemos usar para ajudar no desempenho, robustez, ou talvez ambas. Um modelo que usei com frequência para descrever os diferentes tipos de escala é o Scale Cube de The Art of Scalability, 1 que quebra reduzindo-se em três categorias que, no contexto de sistemas de computador abrangem decomposição funcional, duplicação horizontal e particionamento de dados. O valor desse modelo é que ele ajuda você a entender que você pode escalar um sistema ao longo de um, dois ou todos esses três eixos, dependendo de suas necessidades. No entanto, especialmente em um mundo de infraestrutura virtualizada, sempre achei que esse modelo não tinha um quarto eixo de escala vertical, embora isso pudesse têm a propriedade um tanto infeliz de fazer com que não seja mais um cubo. No entanto, acho que é um conjunto útil de mecanismos para determinarmos como

melhor escalar nossas arquiteturas de microserviços. Antes de analisarmos esses tipos de escalando detalhadamente, juntamente com os prós e contras relativos, um breve resumo está em pedido:

Escala vertical

Em poucas palavras, isso significa adquirir uma máquina maior.

Duplicação horizontal

Ter várias coisas capazes de fazer o mesmo trabalho.

Particionamento de dados

Dividindo o trabalho com base em algum atributo dos dados, por exemplo, grupo de clientes

Decomposição funcional

Separação do trabalho com base no tipo, por exemplo, decomposição de microserviços.

Entender qual é a melhor combinação dessas técnicas de escalabilidade é
apropriado se resumirá fundamentalmente à natureza do problema de escala que
você está enfrentando. Para explorar isso com mais detalhes, veja exemplos de como
esses conceitos podem ser implementados para a MusicCorp. também examine sua adequação para uma empresa do mundo real, a FoodCo. 2 FoodCo
fornecerá entrega de alimentos diretamente para clientes em vários países em todo o mundo,

Escala vertical

Algumas operações só podem se beneficiar de mais trabalho. Obtendo uma caixa maior com
CPU mais rápida e melhor E/S geralmente podem melhorar a latência e a taxa de transferência, permitindo
você para processar mais trabalho em menos tempo. Então, se seu aplicativo não está indo rápido
o suficiente para você ou não consegue lidar com solicitações suficientes, por que não comprar uma maior
máquina?

No caso da FoodCo, um dos desafios que ela enfrenta é aumentar
escreve a contenção em seu banco de dados primário. Normalmente, a escala vertical é uma

opção absoluta para escalar rapidamente as gravações em um banco de dados relacional e de fato, a FoodCo já atualizou a infraestrutura de banco de dados múltipla vezes. O problema é que a FoodCo realmente já levou isso tão longe está confortável com. O dimensionamento vertical funcionou por muitos anos, mas dado o projeções de crescimento da empresa, mesmo que a FoodCo pudesse adquirir uma máquina maior, isso provavelmente não resolverá o problema a longo prazo.

Historicamente, quando o escalonamento vertical exigia a compra de hardware, isso a técnica era mais problemática. O tempo de espera para comprar hardware significava que isso não era algo que pudesse ser abordado levianamente, e se transformasse descobriu que ter uma máquina maior não resolvia seus problemas, então você tinha provavelmente gastou muito dinheiro que você não precisava. Além disso, era típico para sobredimensionar a máquina de que você precisava devido ao incômodo de obter um orçamento aprovações, esperando a chegada da máquina e assim por diante, o que, por sua vez, levou a capacidade significativa não utilizada em data centers.

A mudança para a virtualização e o surgimento da nuvem pública têm No entanto, ajudou imensamente com essa forma de escalonamento.

Implantação

A implementação variará dependendo da infraestrutura que você está executando ligado. Se estiver executando em sua própria infraestrutura virtualizada, talvez você possa simplesmente redimensionar a VM para usar mais do hardware subjacente - isso é algo que deve ser rápido e razoavelmente livre de riscos de implementar. Se a VM for tão grande quanto a o hardware subjacente pode lidar com isso, essa opção é, obviamente, um obstáculo talvez tenha que comprar mais hardware. Da mesma forma, se você estiver correndo sozinho servidores bare metal e não tenho nenhum hardware sobressalente por aí, ou seja maior do que o que você está usando atualmente, então, novamente, você está vendo ter que comprar mais máquinas.

Em geral, se eu chegassem ao ponto em que teria que comprar um novo infraestrutura para experimentar o escalonamento vertical, devido ao aumento do custo (e do tempo) para que isso tenha um impacto, eu poderia muito bem pular essa forma de dimensionamento para o Em vez disso, veja a duplicação horizontal, que veremos a seguir.

Mas o surgimento da nuvem pública também nos permitiu alugar facilmente, em um por hora (e, em alguns casos, em um prazo ainda mais curto), totalmente máquinas gerenciadas por meio de fornecedores de nuvem pública. Além disso, a nuvem principal os fornecedores oferecem uma variedade maior de máquinas para diferentes tipos de problemas.

Sua carga de trabalho consome mais memória? Presenteie-se com um AWS U-Instância metálica de 24 TB1, que fornece 24 TB de memória (sim, você leu isso certo). Agora, o número de cargas de trabalho que podem realmente precisar de tanto a memória parece muito rara, mas você tem essa opção. Você também tem máquinas adaptado também para altos usos de E/S, CPU ou GPU. Se sua solução existente for já está na nuvem pública, essa é uma forma tão trivial de dimensionamento tentar que, se você está atrás de uma vitória rápida, é um pouco óbvio.

Principais benefícios

Na infraestrutura virtualizada, especialmente em um provedor de nuvem pública, a implementação dessa forma de escalonamento será rápida. Grande parte da solução alternativa escalar aplicativos se resume à experimentação - ter uma ideia sobre algo que pode melhorar seu sistema, fazer a mudança e medir o impacto. As atividades que são rápidas e razoavelmente isentas de riscos de experimentar são sempre vale a pena fazer desde o início. E a escala vertical se encaixa perfeitamente aqui.

Também é importante notar que o dimensionamento vertical pode facilitar a execução de outros tipos de dimensionamento. Como um exemplo concreto, movendo sua infraestrutura de banco de dados para uma máquina maior pode permitir que ela hospede os bancos de dados logicamente isolados para microsserviços recém-criados como parte da decomposição funcional.

É improvável que seu código ou banco de dados precise de alterações para usar o maior infraestrutura subjacente, assumindo o sistema operacional e os chipsets permanecem os mesmos. Mesmo que sejam necessárias alterações em seu aplicativo para fazer uso da mudança de hardware, eles podem estar limitados a coisas como aumentar a quantidade de memória disponível para seu tempo de execução por meio de sinalizadores de tempo de execução.

Limitações

À medida que ampliamos as máquinas em que estamos executando, nossas CPUs geralmente não na verdade, ficam mais rápidos; só temos mais núcleos. Esta foi uma mudança em relação ao últimos 5 a 10 anos. Costumava ser que cada nova geração de hardware

oferecem grandes melhorias na velocidade do relógio da CPU, o que significa que nossos programas se tornaram grandes saltos no desempenho. As melhorias na velocidade do relógio diminuíram drasticamente desligado, porém, e, em vez disso, temos mais e mais núcleos de CPU para jogar. O problema é que muitas vezes nosso software não foi escrito para tirar proveito de hardware multicore. Isso pode significar que a mudança de seu aplicativo de um sistema de 4 a 8 núcleos pode oferecer pouca ou nenhuma melhoria, mesmo que seu sistema existente está vinculado à CPU Alterando o código para aproveitar as vantagens de hardware multicore pode ser uma tarefa significativa e pode exigir uma mudança completa nos idiomas de programação.

Ter uma máquina maior também provavelmente fará pouco para melhorar a robustez. Uma um servidor maior e mais novo pode ter maior confiabilidade, mas, em última análise, se isso acontecer a máquina está parada, então a máquina está parada. Ao contrário das outras formas de dimensionamento veremos que é improvável que a escala vertical tenha muito impacto na melhoria a robustez do seu sistema.

Finalmente, à medida que as máquinas ficam maiores, elas ficam mais caras, mas nem sempre de uma forma que seja igualada pelo aumento dos recursos disponíveis para você. Às vezes, isso significa que pode ser mais econômico ter um número maior de máquinas pequenas, em vez de um número menor de máquinas grandes.

Duplicação horizontal

Com a duplicação horizontal, você duplica parte do seu sistema para lidar com mais cargas de trabalho. Os mecanismos exatos variam - veremos as implementações a duplicação curta, mas fundamentalmente horizontal, exige que você tenha uma maneira de distribuir o trabalho entre essas duplicatas.

Assim como na escala vertical, esse tipo de escala está na extremidade mais simples do espectro e geralmente é uma das coisas que vou tentar desde o início. Se você é monolítico o sistema não consegue lidar com a carga, gire várias cópias dela e veja se ajuda!

Implementações

Provavelmente a forma mais óbvia de duplicação horizontal que vem à mente está usando um平衡ador de carga para distribuir solicitações em várias cópias.

de sua funcionalidade, como vemos na Figura 13-1, onde estamos平衡ando a carga em várias instâncias do microsserviço Catalog da MusicCorp. Carregar os recursos do balanceador são diferentes, mas você esperaria que todos tivessem alguns mecanismos para distribuir a carga entre os nós e detectar quando um nó é indisponível e removê-lo do pool do balanceador de carga. Do consumidor do ponto de vista, o balanceador de carga é uma implementação totalmente transparente - podemos vê-la como parte do limite lógico do microsserviço a esse respeito. Historicamente, os平衡adores de carga seriam considerados principalmente em termos de hardware dedicado, mas isso há muito deixou de ser comum - vez disso, mais平衡amento de carga é feito no software, geralmente executado no cliente lado.

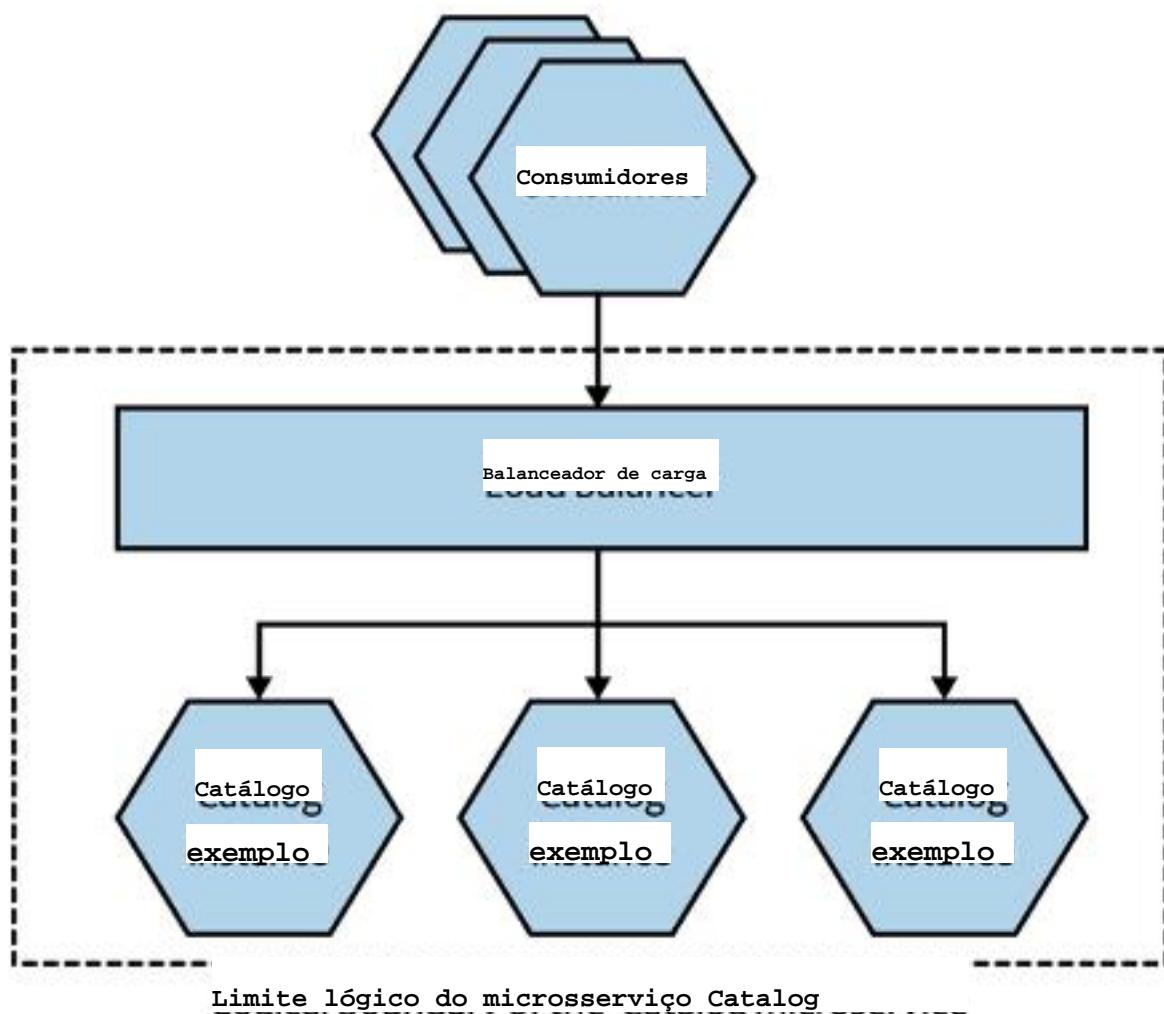


Figura 13-1. O microsserviço Catalog implantado como várias instâncias, com um balanceador de carga para solicitações de spread

padrão, conforme detalhado em Padrões de Integração Corporativa. 3 Na Figura 13-2, nós veja novas músicas sendo enviadas para a MusicCorp. Essas novas músicas precisam ser transcodificadas em arquivos diferentes para serem usados como parte do novo MusicCorp. Oferta de streaming Temos uma fila de trabalho comum na qual esses trabalhos estão colocados e um conjunto de instâncias do Song Transcoder todas consumidas da fila. As diferentes instâncias estão competindo pelos empregos. Para aumentar o taxa de transferência do sistema, podemos aumentar o número de Song Transcoder instâncias.

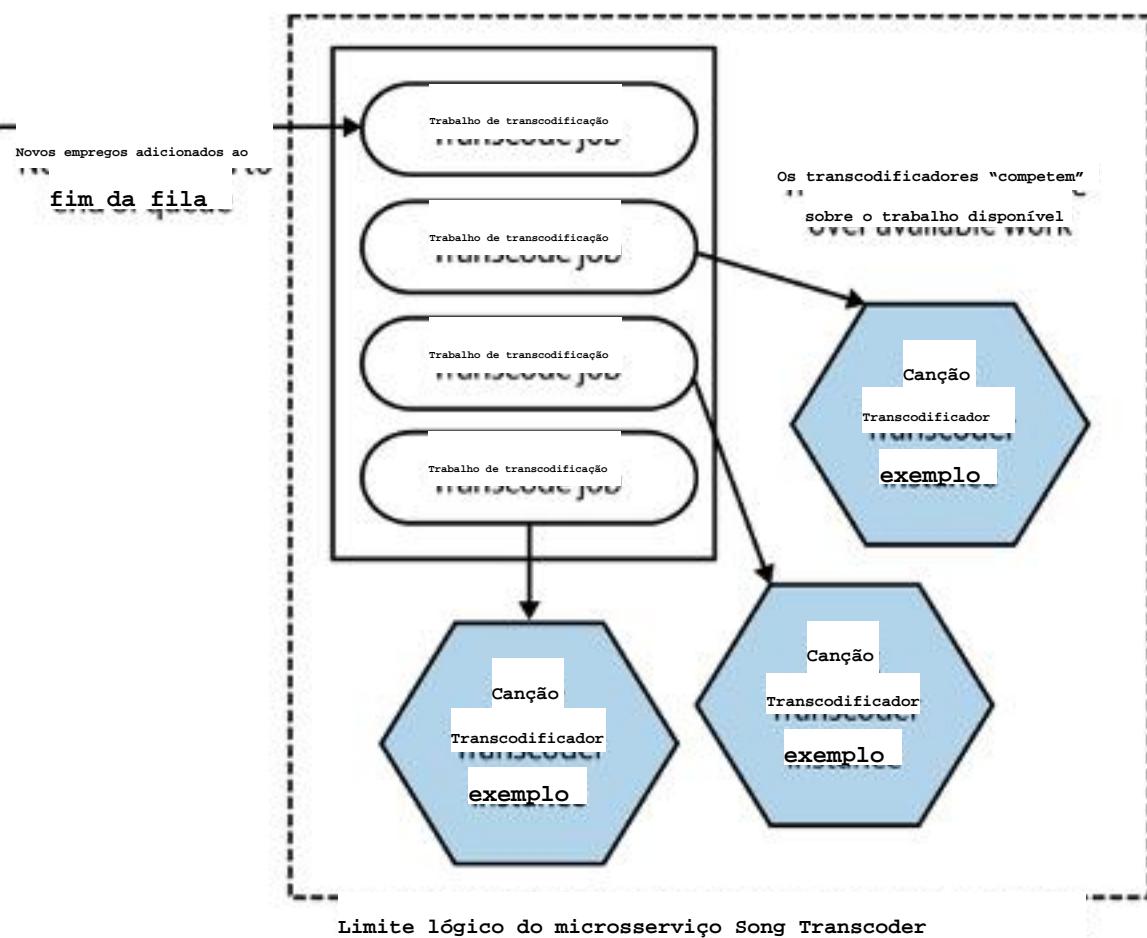


Figura 13-2. A transcodificação para streaming está sendo ampliada usando o padrão de consumo concorrente.

No caso da FoodCo, uma forma de duplicação horizontal foi usada para reduzir a carga de leitura no banco de dados primário por meio do uso de leitura réplicas, como vemos na Figura 13-3. Isso reduziu a carga de leitura no nó primário do banco de dados, liberando recursos para lidar com gravações e tem

funcionou de forma muito eficaz, pois grande parte da carga no sistema principal foi lidado. Essas leituras podem ser facilmente redirecionadas para essas réplicas de leitura, e é comum usar um balanceador de carga em várias réplicas de leitura.

O roteamento para o banco de dados primário ou para uma réplica de leitura é tratado internamente no microserviço. Isso é transparente para os consumidores. O microserviço para saber se uma solicitação que eles enviam acaba atingindo o banco de dados primário ou de réplica de leitura.

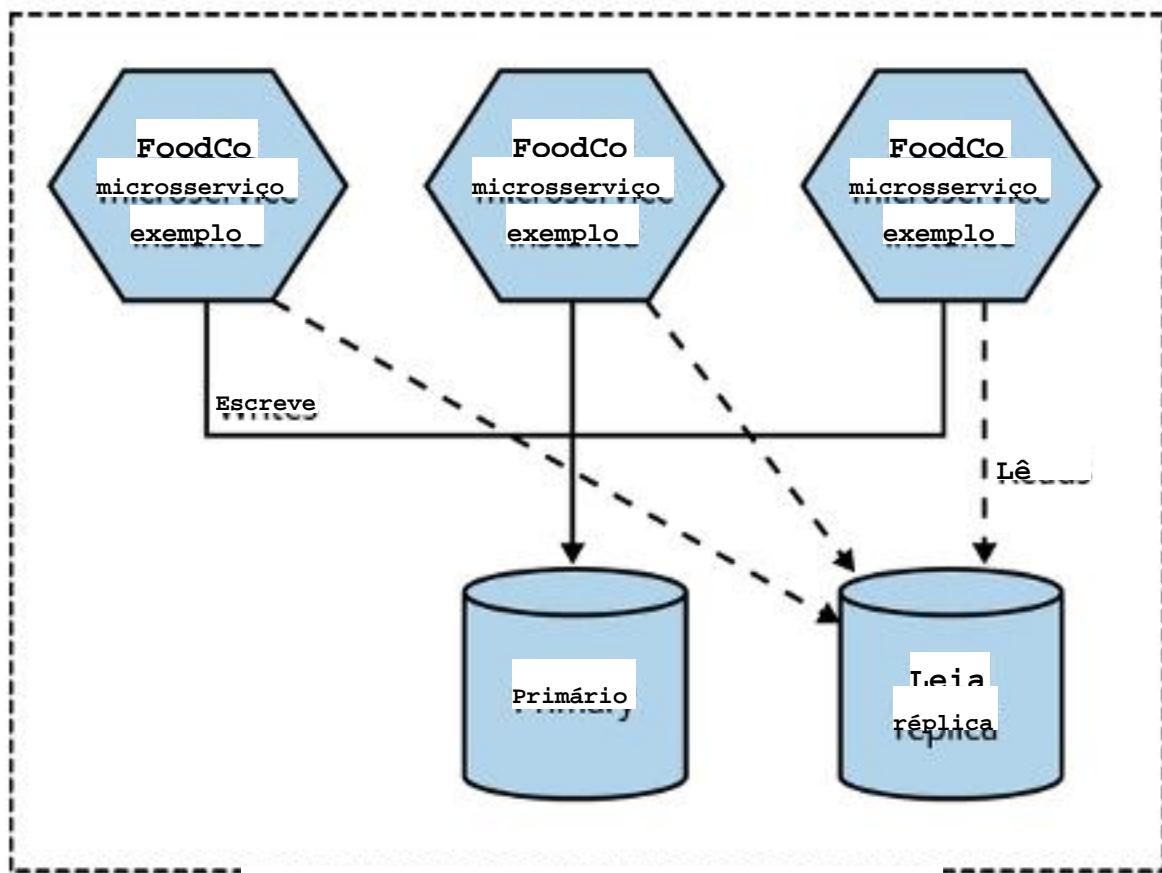


Figura 13-3. FoodCo fazendo uso de réplicas de leitura para escalar o tráfego de leitura

Principais benefícios

A duplicação horizontal é relativamente simples. É raro que o aplicativo precise ser atualizado, pois o trabalho para distribuir a carga geralmente pode ser feito em outro lugar, por exemplo, por meio de uma fila em execução em uma mensagem corretor ou talvez em um平衡ador de carga. Se a escala vertical não estiver disponível para mim, essa forma de dimensionamento normalmente é a próxima coisa que eu examinaria.

Supondo que o trabalho possa ser facilmente distribuído entre as duplicatas, é elegante forma de distribuir a carga e reduzir a disputa pela computação bruta recursos.

Limitações

Como acontece com praticamente todas as opções de dimensionamento que veremos, a duplicação horizontal requer mais infraestrutura, o que, obviamente, pode custar mais dinheiro. Pode ser um instrumento um pouco contundente também - você pode executar várias cópias completas do seu aplicação monolítica, por exemplo, mesmo que apenas parte desse monólito seja na verdade, enfrentando problemas de escala.

Grande parte do trabalho aqui é na implementação de seus mecanismos de distribuição de carga. Eles podem variar desde os mais simples, como balanceamento de carga HTTP, até os mais complexo, como usar um agente de mensagens ou configurar a leitura do banco de dados réplicas. Você está contando com esse mecanismo de distribuição de carga para fazer seu trabalho entender como funciona e com quaisquer limitações específicas a escolha será fundamental.

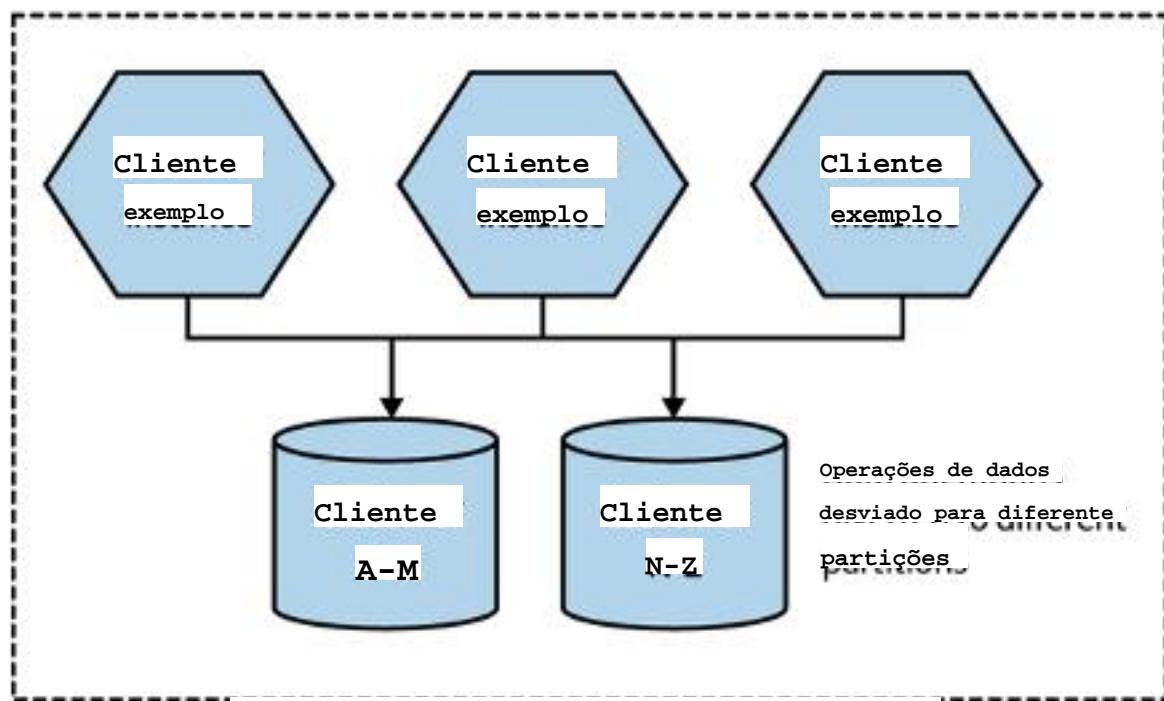
Alguns sistemas podem impor requisitos adicionais à distribuição de carga mecanismo. Por exemplo, eles podem exigir que cada solicitação seja associada com a mesma sessão de usuário é direcionado para a mesma réplica. Isso pode ser resolvido por meio do uso de um平衡ador de carga que requer uma carga de sessão fixa balanceamento, mas isso pode, por sua vez, limitar quais mecanismos de distribuição de carga você poderia considerar. É importante notar que sistemas que exigem carga pega josa balanceamentos como esse são propensos a outros problemas e, em geral, eu evitaria sistemas de construção que tenham esse requisito.

Particionamento de dados

Tendo começado com as formas mais simples de escalonamento, agora estamos entrando em território mais difícil. O particionamento de dados exige que distribuamos a carga com base em algum aspecto dos dados, talvez distribuindo a carga com base no usuário, por exemplo.

Implantação

A forma como o particionamento de dados funciona é que pegamos uma chave associada ao carga de trabalho e aplicamos uma função a ela, e o resultado é a partição (às vezes chamado de fragmento) para o qual distribuiremos o trabalho. Na Figura 13-4, temos dois partições, e nossa função é bem simples - enviamos a solicitação para um banco de dados se o nome da família começar com A a M, ou em um banco de dados diferente se o nome da família comece com N a Z. Agora, este é realmente um mau exemplo de algoritmo de particionamento (veremos por que isso acontecerá em breve), mas espero que seja simples o suficiente para ilustrar a ideia.



Limite lógico do microserviço do cliente

Figura 13-4. Os dados do cliente são particionados em dois bancos de dados diferentes

Neste exemplo, estamos particionando no nível do banco de dados. Um pedido para o microserviço do cliente pode atingir qualquer instância de microserviço. Mas quando nós realizar uma operação que exija o banco de dados (leituras ou gravações), essa solicitação é direcionada para o nó do banco de dados apropriado com base no nome do cliente. No caso de um banco de dados relacional, o esquema de ambos os bancos de dados os nós seriam idênticos, mas o conteúdo de cada um se aplicaria apenas a um subconjunto dos clientes.

O particionamento no nível do banco de dados geralmente faz sentido se o banco de dados
a tecnologia que você está usando suporta esse conceito nativamente, como você pode então
transfira esse problema para uma implementação existente. Poderíamos, no entanto,
em vez disso, particione no nível da instância de microsserviço, como vemos na Figura 13-5.
Aqui, precisamos ser capazes de descobrir, a partir da solicitação de entrada, qual partição
a solicitação deve ser mapeada em nosso exemplo, isso está sendo feito via
alguma forma de proxy. No caso do nosso modelo de particionamento baseado no cliente, se
o nome do cliente está nos cabecalhos da solicitação, então isso seria
suficiente. Essa abordagem faz sentido se você quiser um microsserviço dedicado
instâncias para particionamento, o que pode ser útil se você estiver usando in-
armazenamento em cache de memória. Isso também significa que você pode escalar cada partição em ambos os
nível de banco de dados e nível de instância de microsserviço.

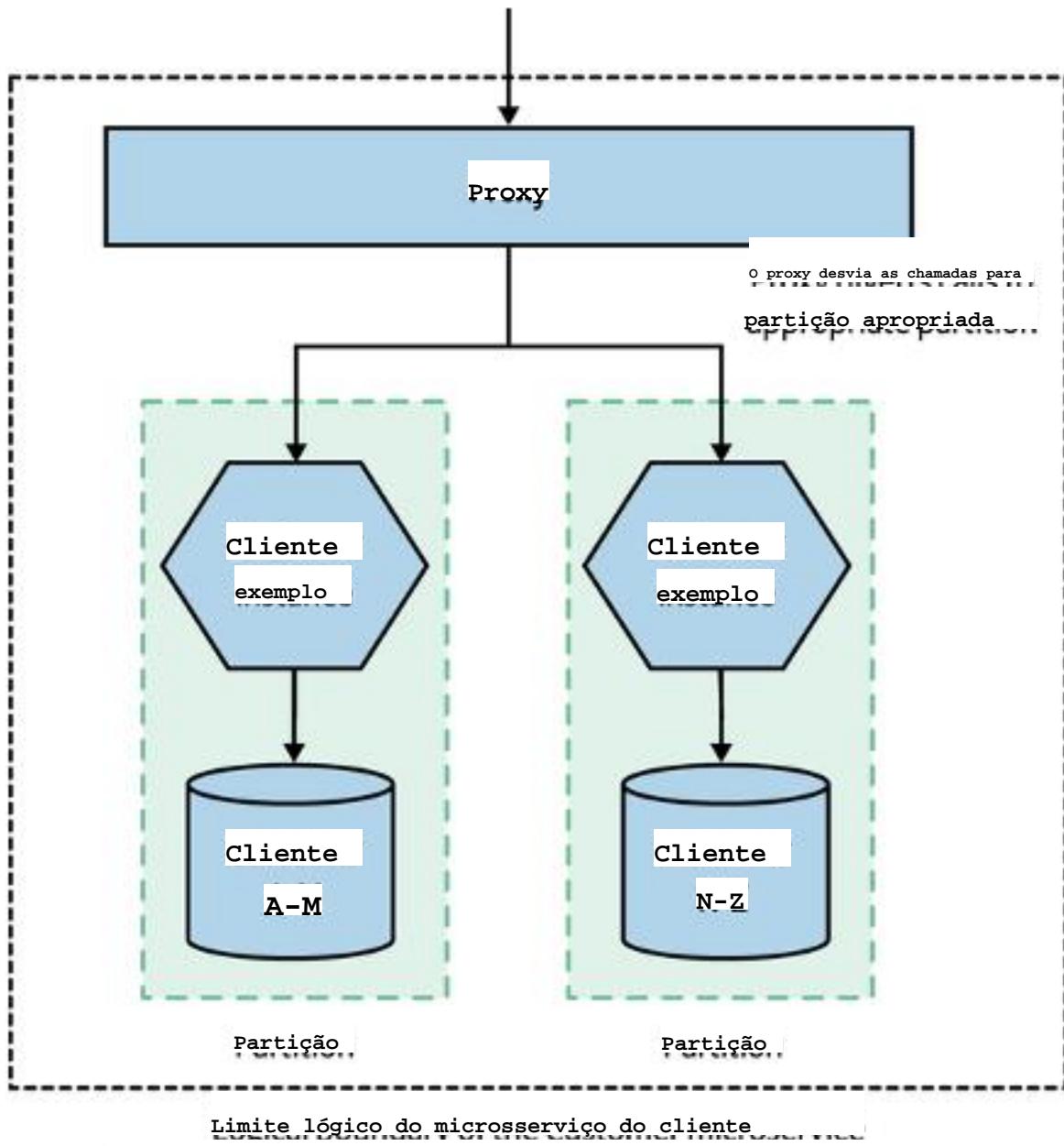


Figura 13-5. As solicitações são direcionadas para a instância de microserviço apropriada.

Assim como no exemplo de réplicas de leitura, gostaríamos que esse dimensionamento fosse feito de tal forma que os consumidores do microserviço não estejam cientes disso. Detalhe da implementação. Quando um consumidor faz uma solicitação ao cliente microserviço na Figura 13-5, gostaríamos que sua solicitação fosse dinâmica, roteado para a partição correta. O fato de termos implementado dados de particionamento deve ser tratado como um detalhe de implementação interna do microserviço em questão - isso nos dá liberdade para alterar o particionamento esquema, ou talvez substitua completamente o particionamento.

Outro exemplo comum de particionamento de dados é fazer isso em uma área geográfica base. Você pode ter uma participação por país ou uma por região.

Para a FoodCo, uma opção para lidar com a contenção em seu banco de dados principal é dados de participação com base no país. Então, os clientes em Gana acessam um banco de dados e clientes em Jersey atingiram outro. Este modelo não faria sentido para a FoodCo devido a vários fatores. O principal problema é que a FoodCo tem planos de continue a expansão geográfica e espera impulsionar a eficiência sendo capaz de atender várias localidades geográficas do mesmo sistema. A ideia de ter que criar continuamente novas partições para cada país seria aumentar drasticamente o custo de se mudar para novos países.

Na maioria das vezes, o particionamento será feito pelo subsistema em que você confia. Por exemplo, o Cassandra usa partições para distribuir leituras e gravações entre os nós em um determinado "anel", e Kafka suporta a distribuição de mensagens em tópicos particionados.

Principais benefícios

O particionamento de dados se adapta muito bem às cargas de trabalho transacionais. Se o seu sistema tem restrições de gravação, por exemplo, o particionamento de dados pode oferecer grandes quantidades de melhorias.

A criação de várias partições também pode facilitar a redução do impacto e escopo das atividades de manutenção. A implantação de atualizações pode ser feita por participação e operações que, de outra forma, exigiriam o tempo de inatividade pode ter um impacto reduzido, pois afetará apenas uma única participação. Por exemplo, ao particionar em torno de regiões geográficas, operações que podem o resultado na interrupção do serviço pode ser feito no momento menos impactante de dia, talvez nas primeiras horas da manhã. O particionamento geográfico pode também ser muito útil se você precisar garantir que os dados não possam deixar certas jurisdições garantindo que os dados associados aos cidadãos da UE permaneçam armazenados dentro da UE, por exemplo.

O particionamento de dados pode funcionar bem com a duplicação horizontal - cada participação pode consistir em vários nós capazes de lidar com esse trabalho.

Limitações

Vale ressaltar que o particionamento de dados tem utilidade limitada em termos de melhorando a robustez do sistema. Se uma partição falhar, essa parte de suas solicitações falhará. Por exemplo, se sua carga for distribuída uniformemente em quatro partições, e uma partição falhar, 25% de suas solicitações terminarão falhando. Isso não é tão ruim quanto um fracasso total, mas ainda é muito ruim. Isso é por que, conforme descrito anteriormente, é comum combinar o particionamento de dados com um técnica como duplicação horizontal para melhorar a robustez de um determinado partição.

Conseguir a chave de partição correta pode ser difícil. Na Figura 13-5, usamos um esquema de particionamento bastante simples, onde particionamos a carga de trabalho com base em o sobrenome do cliente. Clientes com um nome de família começando com A-M vão para a partição 1 e os clientes com um nome que começa com N-Z vão para partição 2. Como indiquei quando compartilhei esse exemplo, isso não é bom estratégia de particionamento. Com o particionamento de dados, queremos uma distribuição uniforme de carregar. Mas não podemos esperar uma distribuição uniforme com o esquema que descrevi. Na China, por exemplo, historicamente houve um número muito pequeno de sobrenomes, e até hoje estima-se que haja menos de 4.000. O 100 sobrenomes mais populares, que representam mais de 80% da população, inclina-se fortemente para os nomes de família que começam com N-Z em mandarim. Isso é um exemplo de um esquema de escala que dificilmente fornecerá uma distribuição uniforme de carga, e em diferentes países e culturas pode dar uma grande variedade resultados.

Uma alternativa mais sensata pode ser particionar com base em um ID exclusivo fornecido para cada cliente quando ele se inscreveu. É muito mais provável que isso nos dê uma distribuição uniforme da carga e também lidaria com a situação em que alguém muda de nome.

A adição de novas partições a um esquema existente geralmente pode ser feita sem a ajuda muitos problemas. Por exemplo, adicionar um novo nó a um anel de Cassandra não exigem qualquer rebalanceamento manual dos dados; em vez disso, o Cassandra tem um sistema integrado suporte para distribuição dinâmica de dados entre os nós. Kafka também faz é bastante fácil adicionar novas partições após o fato, embora as mensagens já estejam em uma partição não se moverá, mas produtores e consumidores podem ser dinâmicos notificado.

As coisas ficam mais complicadas quando você percebe que seu esquema de particionamento simplesmente não é adequado ao propósito, como no caso de nosso esquema baseado em nomes de família descrito mais cedo. Nessa situação, você pode ter um caminho doloroso pela frente. Eu me lembro conversando com um cliente há muitos anos que acabou tendo que tomar o principal sistema de produção off-line por três dias para alterar o esquema de particionamento para seu banco de dados principal.

Também podemos encontrar um problema com consultas. Pesquisar um registro individual é fácil, pois posso simplesmente aplicar a função de hash para descobrir em qual instância os dados deve estar ativado e, em seguida, recuperá-lo do fragmento correto. Mas e quanto consultas que abrangem os dados em vários nós, por exemplo, encontrando todos os clientes com mais de 18 anos? Se você quiser consultar todos os fragmentos, você precisa consultar cada fragmento individual e junte na memória ou então tenha um armazenamento de leitura alternativo onde os dois conjuntos de dados estão disponíveis. Consultando frequentemente entre fragmentos é tratado por um mecanismo assíncrono, usando cache resultados. O Mongo usa tarefas de mapeamento/redução, por exemplo, para realizar essas consultas.

Como você pode ter inferido desta breve visão geral, escalar bancos de dados para escrever é onde as coisas ficam muito complicadas e onde as capacidades de vários bancos de dados realmente começam a se diferenciar. Costumo ver pessoas mudando a tecnologia de banco de dados quando eles começam a atingir os limites da facilidade. Eles podem escalar seu volume de gravação existente. Se isso acontecer com você, compre uma caixa maior geralmente é a maneira mais rápida de resolver o problema, mas no plano de fundo, talvez você queira examinar outros tipos de bancos de dados que podem atender melhor às suas necessidades. Com a riqueza de diferentes tipos de bancos de dados disponíveis, selecionar um novo banco de dados pode ser uma atividade assustadora, mas como ponto de partida, eu recomendo completamente o NoSQL agradavelmente conciso Distilled,4 que oferece uma visão geral dos diferentes estilos de NoSQL.

bancos de dados disponíveis para você a partir de armazenamentos altamente relacionais, como gráficos bancos de dados para armazenamentos de documentos, armazenamentos de colunas e armazenamentos de valores-chave.

Fundamentalmente, o particionamento de dados é mais trabalhoso, especialmente se for provável exigir alterações extensivas nos dados do seu sistema existente. O aplicativo. No entanto, o código provavelmente será apenas ligeiramente afetado.

Decomposição funcional

Com a decomposição funcional, você extrai a funcionalidade e permite que ela seja dimensionado de forma independente. Extraíndo funcionalidade de um sistema existente e criar um novo microsserviço é quase o exemplo canônico de funcional decomposição. Na Figura 13-6, vemos um exemplo do MusicCo em que a funcionalidade do pedido é extraída do sistema principal para nos permitir escalar essa funcionalidade separadamente das demais.

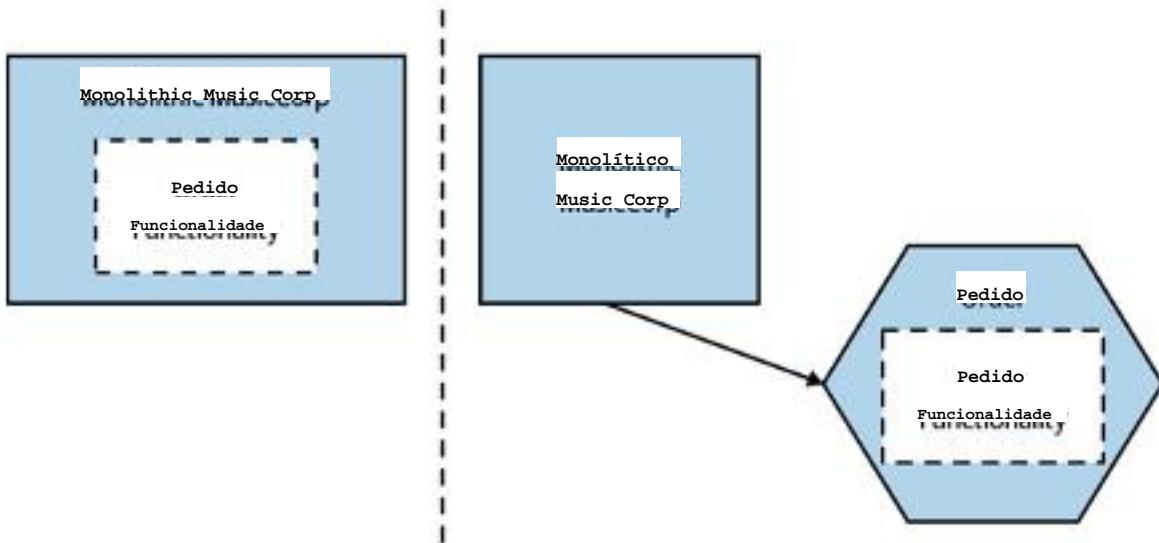


Figura 13-6. O microsserviço Order é extraído do sistema MusicCorp existente.

No caso da FoodCo, essa é sua direção para o futuro. A empresa tem escala vertical esgotada, duplicação horizontal usada na medida do possível, particionamento de dados com desconto - o que resta é começar a avançar. decomposição funcional Os principais dados e cargas de trabalho estão sendo removidos do o sistema central e o banco de dados principal para fazer essa mudança acontecer. Alguns rápidos vitórias foram identificadas, incluindo dados associados a entregas e menus sendo movido do banco de dados principal para microsserviços dedicados. Isso tem o benefício adicional de criar oportunidades para o crescimento da FoodCo equipe de entrega para começar a se organizar em torno da propriedade desses novos microsserviços.

Implantação

Eu não vou me debruçar muito sobre esse mecanismo de escala, como já abordamos os fundamentos dos microsserviços detalhadamente neste livro. Para mais

discussão detalhada de como podemos fazer uma mudança como essa acontecer, consulte

Capítulo 3.

Principais benefícios

O fato de termos dividido diferentes tipos de cargas de trabalho significa que podemos agora dimensionar corretamente a infraestrutura subjacente necessária para nosso sistema.

A funcionalidade decomposta que é usada apenas ocasionalmente pode ser desativada quando não é necessário. Uma funcionalidade que tem apenas requisitos de carga modestos poderia ser implantado em máquinas pequenas. Por outro lado, a funcionalidade que é atualmente restrito poderia ter mais hardware usado, talvez combinando a decomposição funcional com um dos outros eixos de escala, como como executar várias cópias do nosso microsserviço.

Essa capacidade de dimensionar corretamente a infraestrutura necessária para executar essas cargas de trabalho nos dá mais flexibilidade para otimizar o custo da infraestrutura precisa executar o sistema. Esta é uma das principais razões pelas quais grandes provedores de SaaS fazem uso tão intenso de microsserviços, como ser capaz de encontrar o equilíbrio certo de os custos de infraestrutura podem ajudar a impulsionar a lucratividade.

Por si só, a decomposição funcional não tornará nosso sistema mais forte robusto, mas pelo menos abre a oportunidade para construirmos um sistema que pode tolerar uma falha parcial da funcionalidade, algo que exploramos mais detalhes no Capítulo 12.

Supondo que você tenha seguido o caminho dos microsserviços para a decomposição funcional, você terá mais oportunidades de usar diferentes tecnologias que podem ser escaladas o microsserviço decomposto. Por exemplo, você pode mover o funcionalidade para uma linguagem de programação e um tempo de execução mais eficientes para o tipo de trabalho que você está fazendo, ou talvez você possa migrar os dados para um banco de dados mais adequado para seu tráfego de leitura ou gravação.

Embora neste capítulo estejamos nos concentrando principalmente na escala no contexto de nosso sistema de software, a decomposição funcional também facilita o dimensionamento nossa organização também, um tópico ao qual voltaremos no Capítulo 15.

Limitações

Conforme exploramos em detalhes no Capítulo 3, a funcionalidade de separação pode ser uma atividade complexa e é improvável que ofereça benefícios no curto prazo. De todos as formas de dimensionamento que analisamos, esta é a que provavelmente terá o maior impacto no código do seu aplicativo, tanto no front-end quanto no back-end. Também pode exigir uma quantidade significativa de trabalho na camada de dados se você também opta por migrar para microsserviços.

Você acabará aumentando o número de microsserviços que está executando, o que aumentará a complexidade geral do sistema, potencialmente levando a mais coisas que precisam ser mantidas, tornadas robustas e dimensionadas. Em geral, quando se trata de escalar um sistema, tento esgotar as outras possibilidades antes de considerar a decomposição funcional. Minha visão sobre isso pode mudar se a mudança para microsserviços potencialmente traz consigo uma série de outras coisas que a organização está procurando. No caso da FoodCo, por exemplo, seu impulso para aumentar sua equipe de desenvolvimento para apoiar mais países e oferecer mais recursos são essenciais, portanto, uma migração para microsserviços oferece o a empresa tem a chance de resolver não apenas alguns de seus problemas de escalabilidade do sistema, mas também seus problemas de escala organizacional também.

Combinando modelos

Um dos principais fatores por trás do Scale Cube original foi nos impedir de pensando de forma restrita em termos de um tipo de escala e para nos ajudar a entender que geralmente faz sentido escalar nosso aplicativo em vários eixos, dependendo da nossa necessidade. Vamos voltar ao exemplo descrito em Figura 13-6. Extraímos nossa funcionalidade de pedido para que agora ela possa ser executada em seu infraestrutura própria. Como próxima etapa, poderíamos escalar o microsserviço Order em isolamento por ter várias cópias dele, como vemos na Figura 13-7.

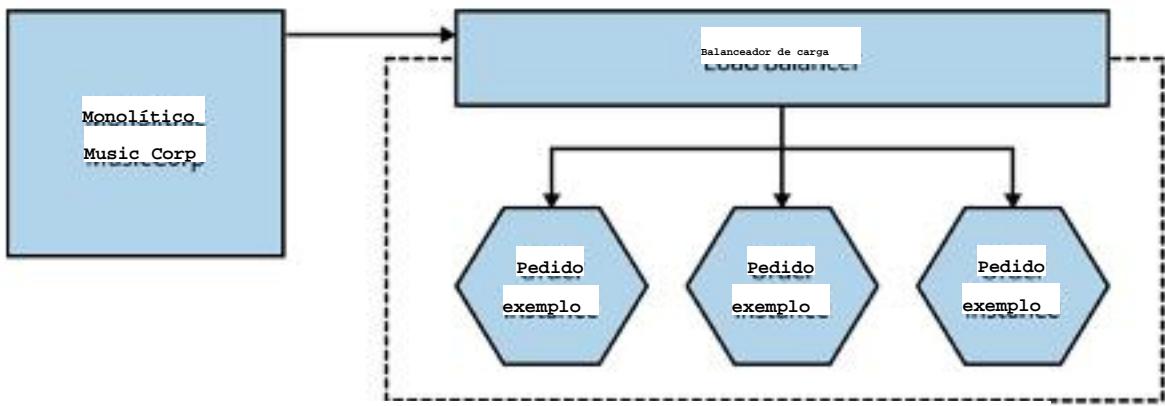


Figura 13-7. O microserviço Order extraído agora está duplicado em escala.

Em seguida, poderíamos decidir executar diferentes fragmentos do nosso microserviço de pedidos em diferentes localidades geográficas, como na Figura 13-8. Duplicação horizontal se aplica dentro de cada limite geográfico.

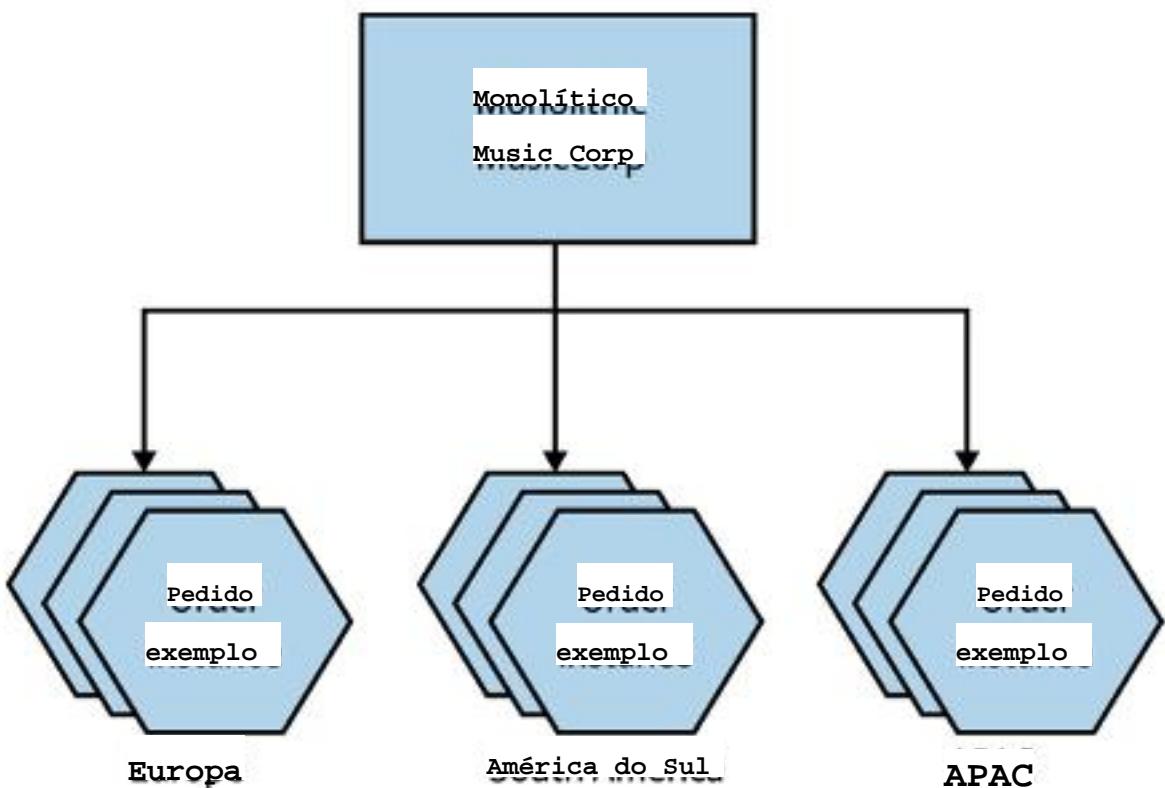


Figura 13-8. O microserviço Order da MusicCorp agora está dividido em toda a geografia, com duplicação em cada grupo.

É importante notar que, ao escalar ao longo de um eixo, outros eixos podem ser mais fáceis de fazer uso de. Por exemplo, a decomposição funcional da Ordem nos permite para, em seguida, criar várias duplicatas do microserviço Order, e também para

particione a carga no processamento do pedido. Sem esse funcional inicial de decomposição, estariamos limitados a aplicar essas técnicas no monólito como um todo.

O objetivo ao escalar não é necessariamente escalar em todos os eixos, mas nós devemos estar cientes de que temos esses diferentes mecanismos à nossa disposição.

Dada essa escolha, é importante que entendamos os prós e os contras de cada mecanismo para descobrir quais fazem mais sentido.

Comece pequeno

Na arte da programação de computadores (Addison-Wesley), Donald Knuth famosamente disse:

O verdadeiro problema é que os programadores passaram muito tempo preocupar-se com a eficiência nos lugares errados e nos momentos errados; a otimização prematura é a raiz de todo mal (ou pelo menos da maior parte dele) em programação.

Otimizar nosso sistema para resolver problemas que não temos é uma ótima maneira de perder tempo que poderia ser melhor gasto em outras atividades e também para garantir que temos um sistema que é desnecessariamente mais complexo. Qualquer forma de a otimização deve ser impulsionada pela necessidade real. Como falamos em "Robustez", adicionando nova complexidade ao nosso sistema, pode introduzir novas fontes de fragilidade também. Ao escalar uma parte do nosso aplicativo, criamos uma fraqueza em outro lugar. Nossa microserviço de pedidos agora pode estar sendo executado em sua infraestrutura própria, nos ajudando a lidar melhor com a carga do sistema, mas temos temos mais um microserviço que precisamos garantir que esteja disponível se queremos que nosso sistema funcione, e ainda mais infraestrutura precisa ser gerenciado e tornado robusto.

Mesmo que você ache que identificou um gargalo, um processo de a experimentação é essencial para ter certeza de que você está certo e que está mais longe o trabalho é justificado. É incrível para mim quantas pessoas ficariam felizes descrevem-se como cientistas da computação parecem não ter nem mesmo um básico compreensão do método científico.⁵ Se você identificou o que acha que é understanding of the scientific method. If you've noticed what you think is

um problema, tente identificar uma pequena quantidade de trabalho que pode ser feito para confirmar se a solução proposta funcionará ou não. No contexto do dimensionamento sistemas para lidar com carga, com um conjunto de testes de carga automatizados, por exemplo, pode ser incrivelmente útil. Execute os testes para obter uma linha de base e recriar o gargalo que você está enfrentando, faça uma mudança e observe a diferença. Isso não é ciência espacial, mas está, mesmo que de uma forma muito pequena, tentando ser pelo menos vagamente científico.

TERCEIRIZAÇÃO DE CARROS E EVENTOS

O padrão Command Query Responsibility Segregation (CQRS) refere-se para um modelo alternativo para armazenar e consultar informações. Em vez de nossa existência de um único modelo de como manipulamos e recuperamos dados, como é comum, as responsabilidades pelas leituras e gravações são, em vez disso, tratadas por modelos separados. Esses modelos separados de leitura e gravação, implementados em código, podem ser implantados como unidades separadas, o que nos dá a capacidade de escala lê e grava de forma independente. O CQRS é frequentemente, embora nem sempre, usado em conjunto com o fornecimento de eventos, onde, em vez de armazenar o estado atual de uma entidade como um único registro - em vez disso, projetamos o estado de uma entidade examinando o histórico de eventos relacionados a essa entidade.

Provavelmente, o CQRS está fazendo algo muito semelhante em nosso nível de aplicativo ao que as réplicas de leitura podem fazer na camada de dados, embora devido à grande variedade de maneiras diferentes de implementar o CQRS, esta é uma simplificação.

Pessoalmente, embora eu veja valor no padrão CQRS em algumas situações, é um padrão complexo para ser bem executado. Eu falei com pessoas muito inteligentes que enfrentaram problemas não insignificantes para fazer o CQRS funcionar. Assim sendo, se você está considerando o CQRS como uma forma de ajudar a escalar sua aplicação, considere isso como uma das formas mais difíceis de escalar que você precisaria implementar, e talvez tente algumas das coisas mais fáceis primeiro. Por exemplo, se você é simplesmente com restrição de leitura, uma réplica de leitura pode muito bem ser significativamente menor. Uma abordagem arriscada e mais rápida para começar. Minhas preocupações com a complexidade da implementação se estende ao fornecimento de eventos - existem algumas situações em que se encaixa muito bem, mas vem com uma série de dores de cabeça que precisam ser acomodado. Ambos os padrões exigem uma grande mudança de pensamento para desenvolvedores, o que sempre torna as coisas mais desafiadoras. Se você decidir para usar qualquer um desses padrões, apenas certifique-se de que esse aumento cognitivo vale a pena carregar seus desenvolvedores.

Uma nota final sobre CQRS e fornecimento de eventos: do ponto de vista de um arquitetura de microserviços, a decisão de usar ou não essas técnicas é um detalhe interno da implementação de um microserviço. Se você decidiu

implementar um microserviço dividindo a responsabilidade pelas leituras e escritas em diferentes processos e modelos, por exemplo, isso deve ser invisível para os consumidores do microserviço. Se as solicitações de entrada precisarem ser redirecionado para o modelo apropriado com base na solicitação feito, faça disso a responsabilidade da implementação do microserviço CARROS. Mantendo esses detalhes de implementação ocultos dos consumidores oferece muita flexibilidade para mudar de ideia mais tarde ou mudar a forma como você está usando esses padrões.

Armazenamento em cache

O armazenamento em cache é uma otimização de desempenho comumente usada, em que a anterior resultado de alguma operação é armazenado para que solicitações subsequentes possam usar isso valor armazenado em vez de gastar tempo e recursos recalcular o valor.

Como exemplo, considere um microserviço de recomendação que precisa verificar os níveis de estoque antes de recomendar um item - não faz sentido recomendando algo que não temos em estoque! Mas decidimos manter uma cópia local dos níveis de estoque na Recomendação (uma forma de cliente)

armazenamento em cache) para melhorar a latência de nossas operações - evitamos a necessidade de verificar os níveis de estoque sempre que precisarmos recomendar algo. A fonte

A verdade para os níveis de estoque é o microserviço de inventário, que é considerado para ser a origem do cache do cliente no microserviço de recomendação.

Quando a Recomendação precisa consultar um nível de estoque, ela pode primeiro examinar seu cache local. Se a entrada necessária for encontrada, isso será considerado um problema de cache. Se o os dados não foram encontrados, é uma falha de cache, o que resulta na necessidade de buscar

informações do microserviço downstream Inventory. Como os dados em a origem pode, é claro, mudar, precisamos de alguma forma de invalidar as entradas em

Cache de recomendação para que saibamos quando os dados armazenados em cache localmente estão tão esgotados da data em que não pode mais ser usado.

Os caches podem armazenar os resultados de pesquisas simples, como neste exemplo, mas na verdade eles podem armazenar qualquer dado, como o resultado de um cálculo complexo.

Podemos armazenar em cache para ajudar a melhorar o desempenho do nosso sistema como parte do ajudando a reduzir a latência, escalar nosso aplicativo e, em alguns casos, até mesmo melhore a robustez do nosso sistema. Tomado em conjunto com o fato de que são vários mecanismos de invalidação que poderíamos usar e vários lugares onde podemos armazenar em cache, isso significa que temos muitos aspectos a discutir quando se trata de armazenamento em cache em uma arquitetura de microserviços Vamos começar falando sobre com quais tipos de problemas os caches podem ajudar.

Para desempenho

Com os microserviços, muitas vezes nos preocupamos com o impacto adverso da latência de rede e sobre o custo de precisar interagir com várias microserviços para obter alguns dados. Obter dados de um cache pode ajudar muito aqui, pois evitamos a necessidade de fazer chamadas de rede, que também tem o impacto da redução da carga em microserviços downstream, além de evitar saltos de rede, isso reduz a necessidade de criar os dados em cada solicitação. Considere uma situação em que estamos solicitando a lista dos mais populares itens por gênero. Isso pode envolver uma consulta de junção cara no banco de dados nível. Poderíamos armazenar em cache os resultados dessa consulta, o que significa que precisaremos regenerar os resultados somente quando os dados em cache forem invalidados.

Para escala

Se você puder desviar as leituras para caches, poderá evitar contendas em partes do seu sistema para permitir uma melhor escala. Um exemplo disso que já temos abordado neste capítulo é o uso de réplicas de leitura de banco de dados. O tráfego de leitura é servido pelas réplicas de leitura, reduzindo a carga no banco de dados principal nó e permitindo que as leituras sejam dimensionadas de forma eficaz. As leituras em uma réplica são feito com base em dados que podem estar obsoletos. A réplica lida acabará sendo atualizado pela replicação do nó primário para o nó de réplica - essa forma de cache a invalidação é tratada automaticamente pela tecnologia do banco de dados.

De forma mais ampla, o armazenamento em cache para escala é útil em qualquer situação na qual a origem é um ponto de discordia. Colocar caches entre os clientes e a origem pode reduzir a carga na origem, permitindo melhor a escalabilidade.

Para robustez

Se você tiver um conjunto inteiro de dados disponível em um cache local, você tem o potencial de operar mesmo que a origem não esteja disponível - isso, por sua vez, poderia melhorar a robustez do seu sistema. Há algumas coisas a serem observadas sobre armazenamento em cache para maior robustez. O principal é que você provavelmente precisaria configurar seu mecanismo de invalidação de cache para não remover automaticamente dados obsoletos e para manter os dados no cache até que possam ser atualizados. Caso contrário, conforme os dados chegam invalidados, ele será removido do cache, resultando em perda de cache e uma falha na obtenção de dados, pois a origem não está disponível. Isso significa que você precisa estar preparado para ler dados que podem estar bastante obsoletos se a origem estiver offline. Em algumas situações isso pode ser bom, enquanto em outras pode ser muito problemático.

Fundamentalmente, usar um cache local para permitir a robustez em uma situação em cuja origem não está disponível significa que você está favorecendo a disponibilidade em vez de consistência.

Uma técnica que vi usada no Guardian e, posteriormente, em outros lugares, foi rastrear o site "ativo" existente periodicamente para gerar uma versão estática do site que poderia ser veiculado em caso de interrupção. Embora isso tenha rastejado a versão não era tão nova quanto o conteúdo em cache veiculado pelo sistema ativo, em um piscar de olhos poderia garantir que uma versão do site fosse exibida.

Onde armazenar em cache

Como já abordamos várias vezes, os microserviços oferecem opções. E esse é absolutamente o caso do cache. Temos muitos lugares diferentes onde poderíamos armazenar. Os diferentes locais de cache que descreverei aqui têm diferentes vantagens e desvantagens, e que tipo de otimização você está tentando fazer será provavelmente direcionará você para o local de cache que faz mais sentido para você.

Para explorar nossas opções de armazenamento em cache, vamos revisitar uma situação que analisamos de volta em "Preocupações com a decomposição de dados", onde estávamos extraiendo informações sobre vendas na MusicCorp. Na Figura 13-9, as vendas o microservice mantém um registro dos itens que foram vendidos. Ele rastreia apenas o ID do item vendido e o registro de data e hora da venda. Ocionalmente, queremos

para solicitar ao microserviço de vendas uma lista dos dez mais vendidos do mundo sete dias anteriores.

O problema é que o microserviço de vendas não sabe os nomes dos registros, apenas os IDs. Não adianta muito dizer: "O best-seller desta semana teve o ID 366548 e vendemos 35.345 cópias!" Queremos saber o nome do CD com o ID 366548 também. O microserviço Catalog armazena que informação... Isso significa, como mostra a Figura 13-9., que ao responder ao uma solicitação para os dez mais vendidos, o microserviço de vendas precisa solicitar os nomes dos dez principais IDs. Vamos ver como o armazenamento em cache pode nos ajudar e em que tipos de caches poderíamos usar.

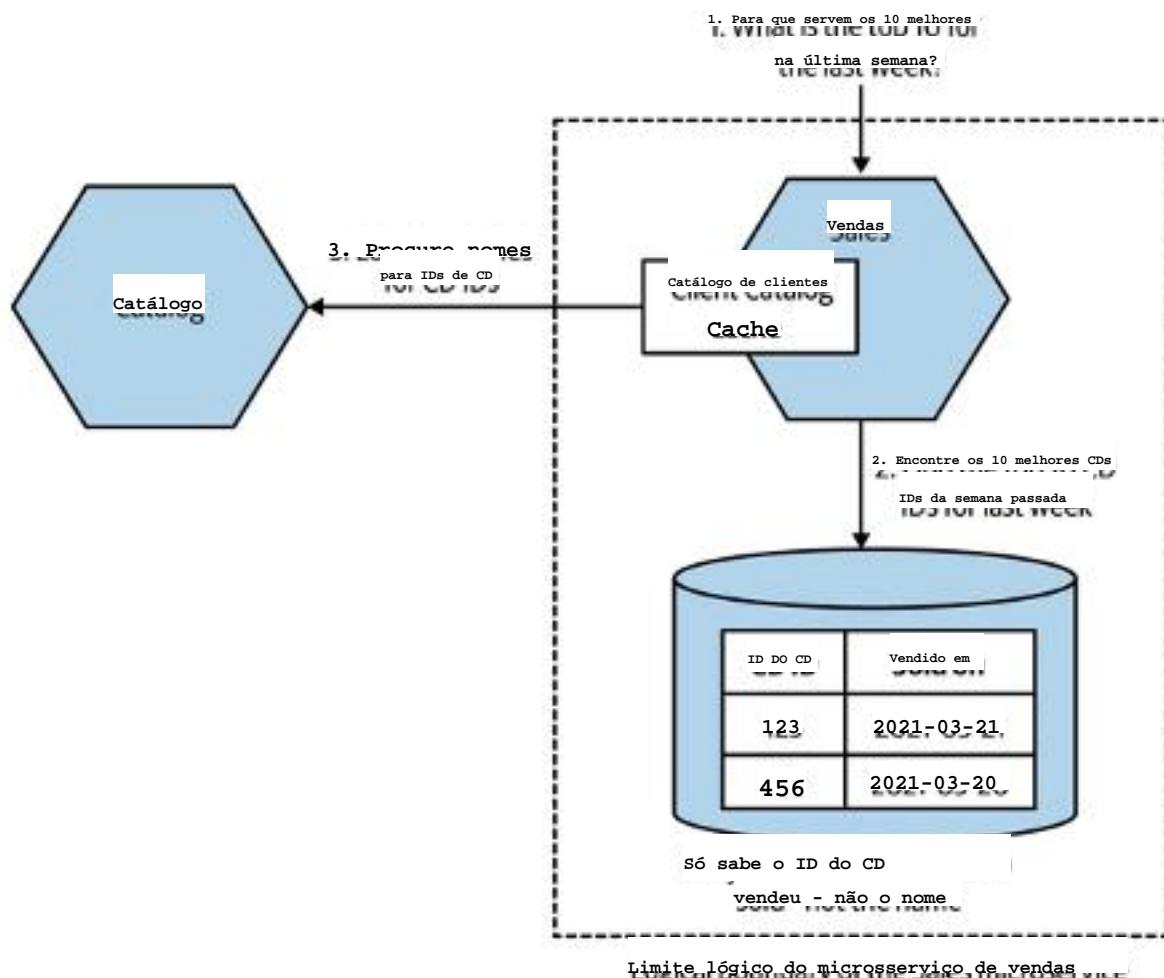


Figura 13-9. Uma visão geral de como a MusicCorp elabora os mais vendidos

Com o cache do lado do cliente, os dados são armazenados em cache fora do escopo da origem. Em nosso exemplo, isso poderia ser feito tão simplesmente quanto manter uma tabela de hash na memória com um mapeamento entre ID e nome do álbum dentro do Sales em execução processo, como na Figura 13-10. Isso significa que gerar nosso top ten exige qualquer interação com o Catalog forá do escopo, supondo que obtenhamos um hit de cache para cada pesquisa que precisamos fazer. É importante observar que o cache do nosso cliente pode decidir armazenar em cache apenas algumas das informações que obtemos do microserviço. Por exemplo, podemos receber muitas informações sobre um CD quando solicitamos para obter informações sobre isso, mas se tudo o que importa é o nome do álbum, isso é tudo o que temos para armazenar em nosso cache local.

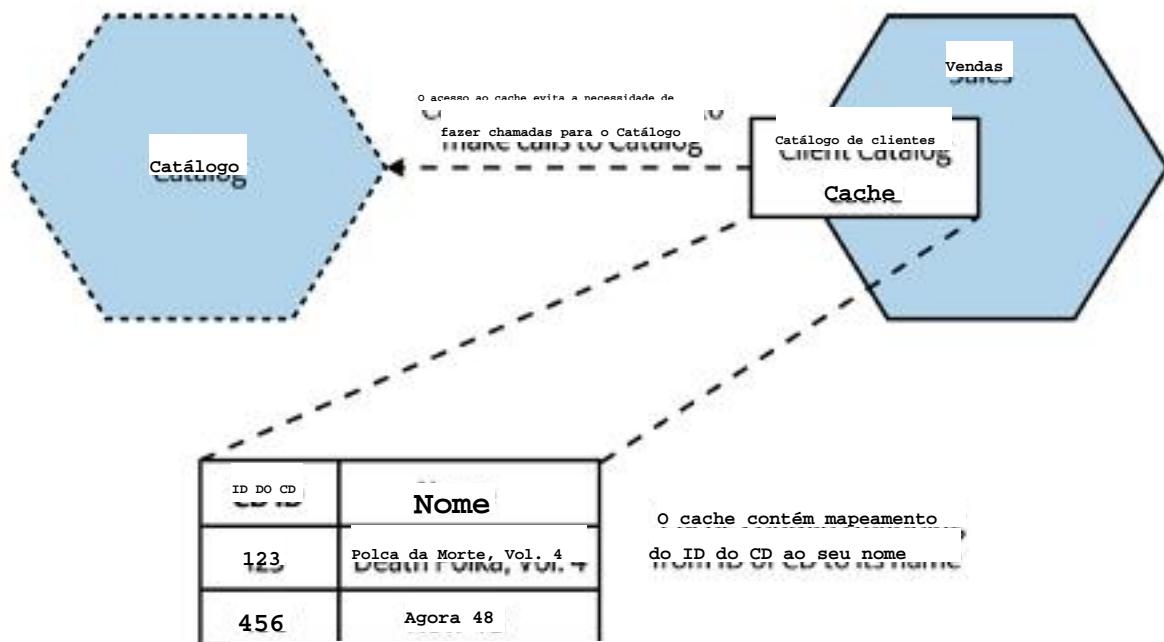


Figura 13-10. O departamento de vendas mantém uma cópia local dos dados do catálogo

Em geral, os caches do lado do cliente tendem a ser bastante eficazes, pois evitam o chamada de rede para o microserviço downstream. Isso os torna adequados não somente para armazenamento em cache para melhorar a latência, mas também para armazenamento em cache para maior robustez. No entanto, o cache do lado do cliente tem algumas desvantagens. Em primeiro lugar, você tende a ser mais restrito em suas opções sobre mecanismos de invalidação - algo que faremos explore mais em breve. Em segundo lugar, quando há muito cache do lado do cliente Continuando, você pode ver um grau de inconsistência entre os clientes. Considere um situação em que microserviços de vendas, recomendações e promoções

todos têm um cache de dados do Catalog no lado do cliente. Quando os dados estão no catálogo mudanças, qualquer mecanismo de invalidação que provavelmente usaremos será incapaz de garantir que os dados sejam atualizados exatamente no mesmo momento de tempo em cada um desses três clientes. Isso significa que você pode ver um diferente visualização dos dados em cache em cada um desses clientes ao mesmo tempo. Quanto mais clientes que você tem, mais problemático isso provavelmente será. Técnicas como, pois a invalidação baseada em notificações, que veremos em breve, pode ajudar reduzirão esse problema, mas não o eliminarão.

Outra mitigação para isso é ter um cache compartilhado do lado do cliente, talvez fazendo uso de uma ferramenta de cache dedicada, como Redis ou memcached, como vemos na Figura 13-11. Aqui, evitamos o problema de inconsistência entre clientes diferentes. Isso também pode ser mais eficiente em termos de uso de recursos, pois estamos reduzindo o número de cópias desses dados que precisamos gerenciar (caches) muitas vezes acabam ficando na memória, e a memória costuma ser uma das maiores restrições de infraestrutura). O outro lado é que nossos clientes agora precisam fazer uma viagem de ida e volta até o cache compartilhado.

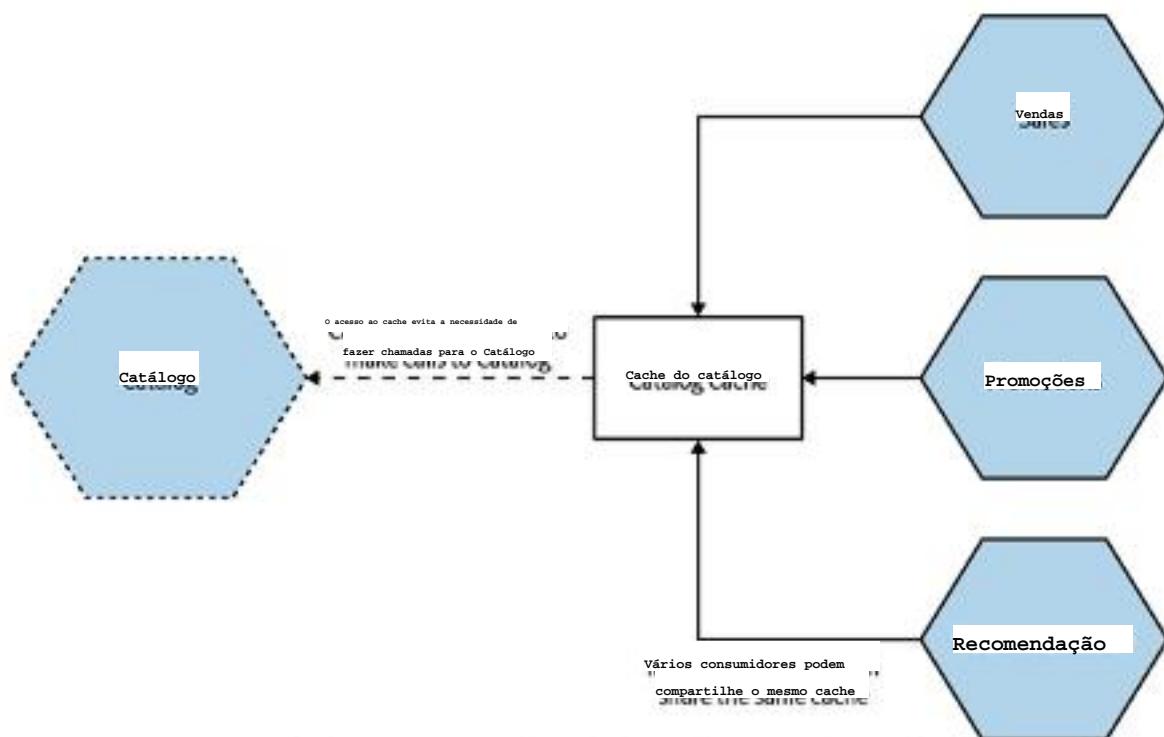


Figura 13-11. Vários consumidores do Catalog usando um único cache compartilhado

Outra coisa a considerar aqui é quem é responsável por esse cache compartilhado.

Dependendo de quem o possui e de como ele é implementado, um cache compartilhado como isso pode confundir as linhas entre o cache do lado do cliente e o cache do lado do servidor, que exploraremos a seguir.

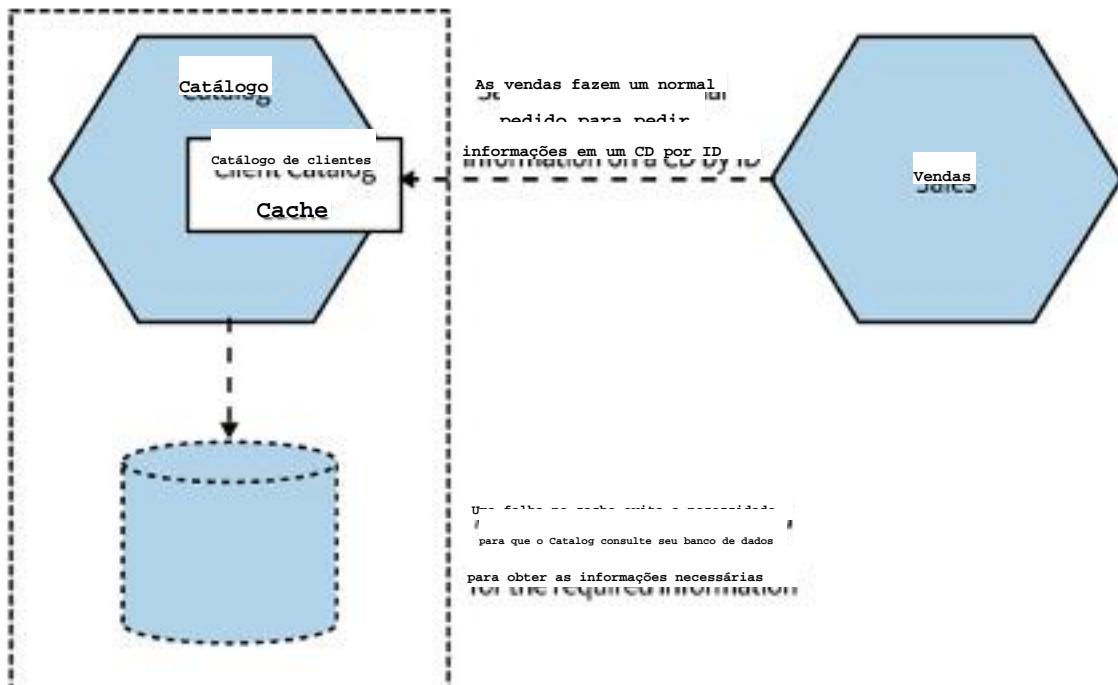
Do lado do servidor

Na Figura 13-12, vemos nossos dez principais exemplos de vendas usando o cache em do lado do servidor. Aqui, o próprio microsserviço Catalog mantém um cache em nome de seus consumidores. Quando o microsserviço de vendas faz sua solicitação de os nomes dos CDs, essas informações são fornecidas de forma transparente por um cache.

Aqui, o microsserviço Catalog tem total responsabilidade pelo gerenciamento do esconderijo. Devido à natureza de como esses caches são normalmente implementados como uma estrutura de dados na memória ou um nó de cache local dedicado - é mais fácil de implementar mecanismos de invalidação de cache mais sofisticados.

Caches de gravação, por exemplo (que veremos em breve), seriam muito mais simples de implementar nessa situação. Ter um cache do lado do servidor também torna mais fácil evitar o problema de consumidores diferentes verem coisas diferentes valores em cache que podem ocorrer com o armazenamento em cache do lado do cliente.

Vale ressaltar que, embora, do ponto de vista do consumidor, esse armazenamento em cache é invisível (é uma preocupação interna de implementação), isso não significa que temos que implementar isso armazenando em cache o código em uma instância de microsserviço. poderia, por exemplo, manter um proxy reverso dentro do nosso microsserviço limite lógico, use um nó oculto do Redis ou desvie as consultas de leitura para ler réplicas de um banco de dados.



Limite lógico do microserviço Catalog

Figura 13-12. O registro Cata implementa o armazenamento em cache internamente, tornando-o invisível para os consumidores

O principal problema com essa forma de armazenamento em cache é que ela reduziu o escopo de otimizar a latência, já que uma viagem de ida e volta dos consumidores ao microserviço é ainda necessário. Ao armazenar em cache no perímetro de um microserviço ou próximo a ele, o cache pode garantir que não precisemos realizar outras operações caras (como consultas ao banco de dados), mas a chamada precisa ser feita. Isso também reduz a eficácia dessa forma de armazenamento em cache para qualquer forma de robustez.

Isso pode fazer com que essa forma de armazenamento em cache pareça menos útil, mas há um grande valor para melhorar o desempenho de forma transparente para todos os consumidores de um microserviço apenas tomando a decisão de implementar o cache internamente. Um microserviço que é amplamente utilizado em uma organização pode se beneficiar enormemente por implementando alguma forma de cache interno, talvez ajudando a melhorar tempos de resposta para vários consumidores, além de permitir o microserviço para escalar com mais eficiência.

No caso do nosso cenário dos dez melhores, teríamos que considerar se isso é ou não a forma de armazenamento em cache pode ajudar. Nossa decisão se resumiria ao que é nossa principal preocupação é. Se for sobre a latência de ponta a ponta da operação, quanto tempo

um cache do lado do servidor salvaria? O cache do lado do cliente provavelmente nos daria uma melhor benefício de desempenho.

Cache de solicitações

Com um cache de solicitações, armazenamos uma resposta em cache para a solicitação original. Então na Figura 13-13, por exemplo, armazenamos as dez principais entradas reais. Subseqüente as solicitações dos dez mais vendidos resultam na devolução do resultado em cache. Sem necessidade de pesquisas nos dados de vendas, sem viagens de ida e volta ao catálogo - isso é longe e remova o cache mais eficaz em termos de otimização de velocidade.

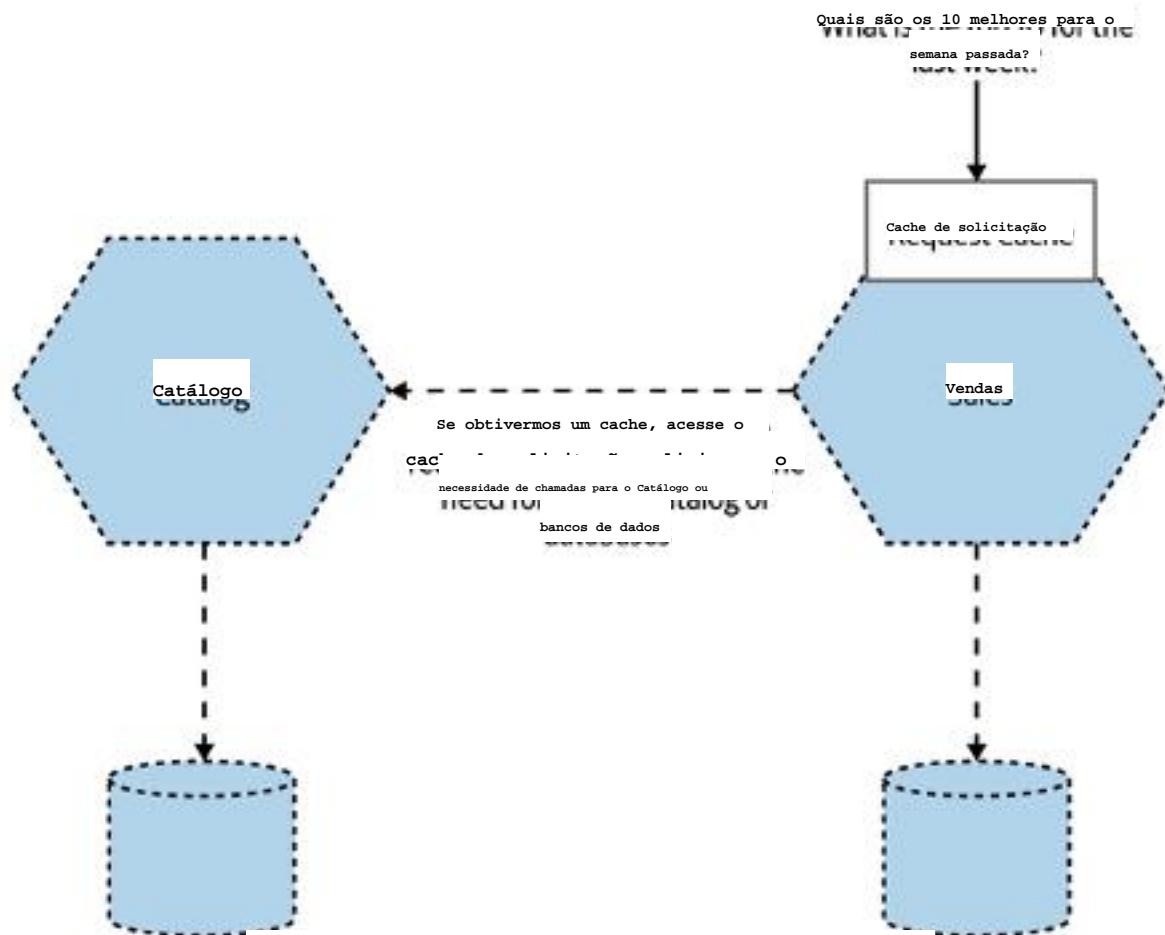


Figura 13-13. Armazenando em cache os resultados das 10 principais solicitações

Os benefícios aqui são óbvios. Isso é supereficiente, por exemplo.

No entanto, precisamos reconhecer que essa forma de armazenamento em cache é altamente específica.

Armazenamos em cache apenas o resultado dessa solicitação específica. Isso significa que outros

as operações que atingem Vendas ou Catálogo não chegarão ao cache, e, portanto, não beneficiar-se-á de alguma forma dessa forma de otimização.

Invalidação

Existem apenas duas coisas difíceis na Ciência da Computação: invalidação de cache e dando nomes às coisas.

-Phil Karlton

A invalidação é o processo pelo qual removemos dados do nosso cache. É uma ideia que é simples no conceito, mas complexa na execução, mesmo que não seja por outro motivo do que há uma grande variedade de opções em termos de como implementá-la, e inúmeras vantagens a serem consideradas em termos de uso de dados que podem ser desatualizado. Fundamentalmente, porém, tudo se resume a decidir em quais situações em que um dado em cache deve ser removido do seu cache.

Às vezes, isso acontece porque nos dizem uma nova versão de um dado está disponível; outras vezes, pode ser necessário presumir que nossa cópia em cache é obsoleto e obtenha uma nova cópia da origem.

Dadas as opções de invalidação, acho que é uma boa ideia analisar umas algumas das opções que você pode usar em uma arquitetura de microserviços. Por favor, não considere isso uma visão geral exaustiva de todas as opções, apesar disso!

Hora de viver (TTL)

Esse é um dos mecanismos mais simples de usar para invalidação de cache. Cada entrada no cache é considerada válida apenas por um determinado período de tempo. Depois desse tempo, os dados são invalidados e buscamos uma nova cópia. Podemos especificar a duração da validade usando um tempo de vida simples (TTL). Duração - portanto, um TTL de cinco minutos significa que nosso cache ficaria feliz em fornecer os dados em cache por até cinco minutos, após os quais a entrada em cache é considerado invalidado e uma nova cópia é necessária. Variações sobre isso o tema pode incluir o uso de um carimbo de data e hora para expiração, que em algumas situações pode ser mais eficaz, especialmente se você estiver lendo vários níveis do cache.

O HTTP suporta tanto um TTL (por meio do cabeçalho Cache-Control) quanto a habilidade para definir um carimbo de data e hora para expiração por meio do cabeçalho Expires nas respostas, o que pode ser incrivelmente útil. Isso significa que a própria origem é capaz de dizer clientes posteriores por quanto tempo eles devem presumir que os dados estão atualizados. Chegando voltando ao nosso microsserviço de inventário, poderíamos imaginar uma situação em que o microsserviço de inventário oferece um TTL mais curto para níveis de estoque rápidos vendendo itens ou para itens para os quais estamos quase sem estoque. Para itens que não vendemos muito, poderia fornecer um TTL mais longo. Isso representa um uso um pouco avançado de controles de cache HTTP e ajustes de controles de cache por resposta, como isso, é algo que eu faria apenas ao ajustar a eficácia de um cache. Um TTL simples e único para qualquer um o tipo de recurso é um ponto de partida sensato.

Mesmo que você não esteja usando HTTP, a ideia da origem dá dicas para o cliente saber como (e se) os dados devem ser armazenados em cache é um conceito muito poderoso. Isso significa que você não precisa adivinhar essas coisas do lado do cliente; você pode na verdade, faça uma escolha informada sobre como lidar com um dado.

O HTTP tem recursos de cache mais avançados do que isso, e vamos veja os GETs condicionais como um exemplo disso em um momento.

Um dos desafios da invalidação baseada em TTL é que, embora seja simples para implementar, é um instrumento bastante contundente. Se solicitarmos uma nova cópia do dados que têm um TTL de cinco minutos e, um segundo depois, os dados na origem mudanças, então nosso cache estará operando com dados desatualizados para o restantes quatro minutos e 59 segundos. Então, a simplicidade da implementação precisa ser equilibrado com a tolerância que você tem em relação à operação em dados desatualizados.

GETs condicionais

Vale a pena mencionar, pois isso é esquecido, é a capacidade de emitir condições de solicitações GET com HTTP. Como acabamos de mencionar, o HTTP fornece a capacidade de especificar cabeçalhos Cache-Control e Expires nas respostas a possibilite um armazenamento em cache mais inteligente do lado do cliente. Mas se estivermos trabalhando diretamente com HTTP, temos outra opção em nosso arsenal de guloseimas HTTP: tags de entidade ou

eTags. Uma ETag é usada para determinar se o valor de um recurso tem alterado. Se eu atualizar um registro de cliente, o URI do recurso será o mesmo, mas o valor é diferente, então eu esperaria que a ETag mudasse. Isso se torna poderoso quando usamos o que é chamado de GET condicional. Ao fazer uma solicitação GET, podemos especificar cabeçalhos adicionais, informando serviço para nos enviar o recurso somente se determinados critérios forem atendidos.

Por exemplo, vamos imaginar que buscamos o registro de um cliente e sua ETag vem de volta como o5t6fk2sa. Mais tarde, talvez por causa de uma diretiva Cache-Control nos disse que o recurso deve ser considerado obsoleto, queremos ter certeza de que obtenha a versão mais recente. Ao emitir a solicitação GET subsequente, podemos passar em um If-None-Match: o5t6fk2sa. Isso diz ao servidor que queremos o recurso no URI especificado, a menos que já corresponda a esse valor de ETag. Se já temos a versão atualizada, o serviço nos envia um 304 Não Resposta modificada, informando que temos a versão mais recente. Se houver um novo versão disponível, obtemos 200 OK com o recurso alterado e um novo ETag para o recurso.

Obviamente, com um GET condicional, ainda fazemos a solicitação do cliente para servidor. Se você estiver armazenando em cache para reduzir as viagens de ida e volta da rede, isso pode não ajudar você muito. Onde é útil é evitar o custo de desnecessariamente regenerando recursos. Com a invalidação baseada em TTL, o cliente solicita uma nova cópia do recurso, mesmo que o recurso não tenha sido alterado - o microserviço que recebe essa solicitação precisa regenerar esse recurso, até mesmo se acabar sendo exatamente igual ao que o cliente já tem. Se o custo de criar a resposta é alta, talvez exigindo um conjunto caro de consultas de banco de dados e, em seguida, solicitações GET condicionais podem ser eficazes mecanismo

Baseado em notificações

Com a invalidação baseada em notificações, usamos eventos para ajudar os assinantes a saber se suas entradas de cache local precisarem ser invalidadas. Na minha opinião, este é o mecanismo mais elegante de invalidação, embora seja equilibrado por seu complexidade relativa em relação à invalidação baseada em TTL.

Na Figura 13-14, nosso microsserviço de recomendação está mantendo um cliente cache lateral. As entradas nesse cache são invalidadas quando o inventário o microsserviço aciona um evento de mudança de estoque, permitindo que a Recomendação (ou qualquer outros assinantes deste evento) sabem que o nível de estoque aumentou ou diminuiu para um determinado item.

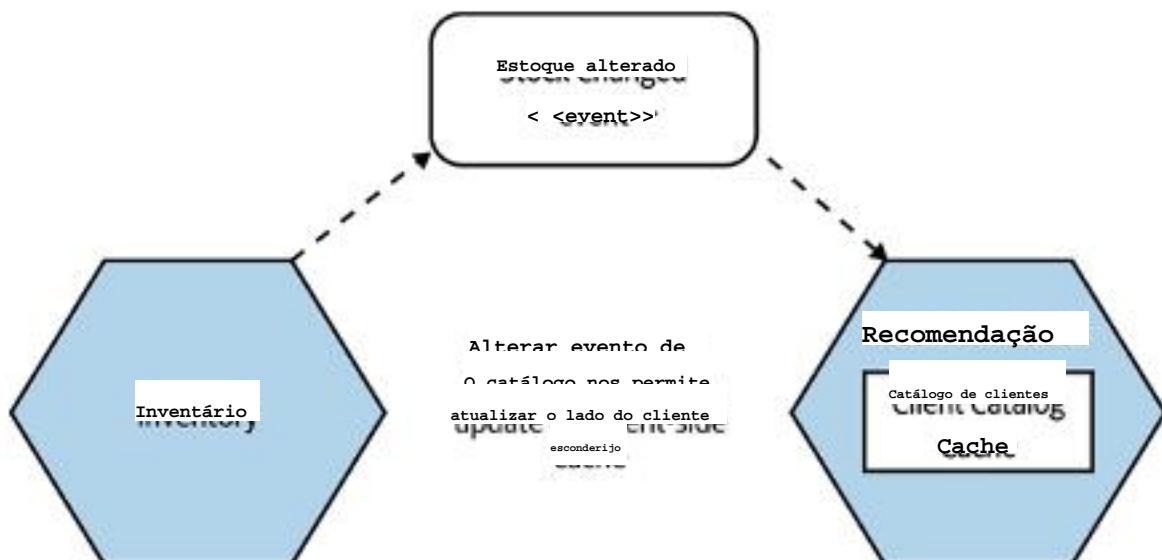


Figura 13-14. O inventário dispara eventos de mudança de estoque, que o Recomendat pode usar para atualize seu cache local

O principal benefício desse mecanismo é que ele reduz a janela potencial em que o cache está servindo dados obsoletos. A janela na qual um cache pode agora, o fornecimento de dados obsoletos está limitado ao tempo necessário para que a notificação seja enviada e processada. Dependendo do mecanismo que você usa para enviar a notificação, isso pode ser bem rápido.

A desvantagem aqui é a complexidade da implementação. Precisamos da origem para poder emitir notificações, e precisamos que as partes interessadas possam responder a essas notificações. Agora, este é um lugar natural para usar algo como um corretor de mensagens, pois esse modelo se encaixa perfeitamente no típico interações no estilo pub/substyle que muitos corretores oferecem. O adicional garantias que o corretor possa nos dar também podem ser úteis. Isso disse que, como já discutimos em "Message Brokers", há uma sobrecarga ao gerenciamento de middleware de mensagens, e poderia ser um exagero se você fosse usando-o apenas para esse propósito. Se você estivesse usando corretores para outras formas de

comunicação entre microsserviços, no entanto, faria sentido fazer uso da tecnologia que você já tinha em mãos.

Um problema a ser observado ao usar a invalidação baseada em notificações é que talvez você queira saber se o mecanismo de notificação é realmente trabalhando ou não. Considere uma situação em que não recebemos nenhum estoque Alterou os eventos do Inventário por um tempo. Isso significa que não temos vendeu itens ou teve itens reabastecidos durante esse período? Talvez. Também poderia significar que nosso mecanismo de notificação está inativo e que não estamos sendo enviados atualizações. Se isso for uma preocupação, então poderíamos enviar um evento de pulsação através do mesmo mecanismo de notificação - recomendação em nosso caso - para deixar os assinantes sabem que as notificações ainda estão chegando, mas nada realmente chegou alterado. Se um evento de pulsação não for recebido, o cliente pode presumir um problema e poderia fazer o que fosse mais apropriado, talvez informando ao usuário que eles estão vendendo dados obsoletos ou talvez apenas desativando a funcionalidade.

Você também precisa considerar o que a notificação contém. Se a notificação apenas diz "essa coisa mudou" sem dizer qual é a mudança, então ao receber a notificação, o consumidor precisaria ir até a origem e busque os novos dados. Por outro lado, se a notificação contiver o atual estado dos dados, para que os consumidores possam carregá-los diretamente no cache local. Ter uma notificação contendo mais dados pode causar problemas de tamanho e também corre o risco de potencialmente expor dados confidenciais de forma muito ampla. Nós explorou anteriormente essa compensação com mais profundidade ao analisar o evento comunicação direcionada em "O que há em um evento?".

Escrita

Com um cache de gravação, o cache é atualizado ao mesmo tempo que o estado na origem. "Ao mesmo tempo" é onde os caches de gravação se tornam complicados, ou curso. A implementação de um mecanismo de gravação em um cache do lado do servidor é um pouco simples, pois você pode atualizar um banco de dados e um arquivo na memória armazene na mesma transação sem muita dificuldade. Se o cache for em outros lugares, é mais difícil raciocinar sobre o que "ao mesmo tempo" significa em termos de essas entradas serem atualizadas.

Devido a essa dificuldade, você normalmente vê o cache de gravação sendo usado em uma arquitetura de microserviços no lado do servidor. Os benefícios são bem claros — a janela na qual um cliente pode ver dados obsoletos pode ser praticamente eliminado. Isso é equilibrado com o fato de que os caches do lado do servidor podem muito bem ser menos útil em geral, limitando as circunstâncias em que uma redação o cache seria eficaz em microserviços.

Escreva para trás

Com um cache de gravação por trás, o cache em si é atualizado primeiro e, em seguida, a origem está atualizada. Conceitualmente, você pode pensar no cache como um buffer. Escrever no cache é mais rápido do que atualizar a origem. Então, escrevemos o resultado no cache, permitindo leituras subsequentes mais rápidas, e confie que a origem será atualizada posteriormente.

A principal preocupação com os caches de gravação reversa será o potencial de perda de dados. Se o cache em si não for durável, poderemos perder os dados antes do os dados são gravados na origem. Além disso, agora estamos em um lugar interessante — qual é a origem nesse contexto? Esperaríamos que a origem fosse a microserviço de onde esses dados são originados, mas se atualizarmos o cache primeiro, essa é realmente a origem? Qual é a nossa fonte de verdade? Ao fazer uso do armazenamento em cache, é importante separar quais dados são armazenados em cache (e potencialmente desatualizado) e quais dados podem realmente ser considerados atualizados. Escreva por trás de caches no contexto de microserviços torna isso muito menos claro.

Embora os caches de gravação por trás sejam frequentemente usados para otimização em processo, eu tenho visto muito mais raramente usados para arquiteturas de microserviços, em parte devido ao fato de que outras formas mais simples de armazenamento em cache são boas o suficiente, mas em grande parte devido à complexidade de lidar com a perda de caches não escritos dados.

A regra de ouro do armazenamento em cache

Tenha cuidado com o armazenamento em cache em muitos lugares! Quanto mais caches entre vocês e a fonte de dados novos, quanto mais obsoletos os dados podem ser, e mais difíceis eles podem ser para determinar a atualização dos dados que um cliente eventualmente vê. É

também pode ser mais difícil raciocinar sobre onde os dados precisam estar invalidado. A desvantagem da atualização dos dados em cache e balanceamento contra a otimização do seu sistema para carga ou latência - é uma questão delicada, e se você não consegue raciocinar facilmente sobre o quanto novos (ou não) os dados podem ser, isso se torna difícil.

Considere uma situação em que o microserviço de inventário está armazenando estoque em cache níveis. Solicitações de inventário para níveis de estoque podem ser atendidas a partir disso. cache do lado do servidor, acelerando a solicitação adequadamente. Vamos também agora suponha que definimos um TTL para esse cache interno em um minuto, ou seja, nosso o cache do lado do servidor pode estar até um minuto atrás do nível real de estoque. Agora, acontece que também estamos armazenando em cache no lado do cliente Recomendação, em que também usamos um TTL de um minuto. Quando um a entrada no cache do lado do cliente expira, fazemos uma solicitação de Recomendação de inventário para obter um nível de estoque atualizado, mas sem que saibamos, nossa solicitação atinge o cache do lado do servidor, que neste momento também pode ter até um minuto de idade. Então, poderíamos acabar armazenando um registro em nosso cache do lado do cliente que já tem até um minuto desde o início. Isso significa que os níveis de estoque que a Recomendação está usando podem potencialmente aumentar até dois minutos desatualizado, mesmo que do ponto de vista de Recomendação, achamos que eles poderiam estar desatualizados em apenas um minuto.

Há várias maneiras de evitar problemas como esse. Usando um carimbo de data/hora- a expiração baseada para começar seria melhor do que TTLs, mas também é uma exemplo do que acontece quando o cache é efetivamente aninhado. Se você armazenar em cache o resultado de uma operação que, por sua vez, é baseada em entradas em cache, quanto claro pode você saber o quanto atualizado está o resultado final?

Voltando à famosa citação anterior de Knuth, otimização prematura pode causar problemas. O armazenamento em cache aumenta a complexidade e queremos adicionar o mínimo complexidade quanto possível. O número ideal de lugares para armazenar em cache é zero. Qualquer outra coisa deve ser uma otimização que você precisa fazer, mas esteja ciente a complexidade que isso pode trazer.

Trate o cache principalmente como uma otimização de desempenho. Cache no menor número possível de lugares para facilitar o raciocínio sobre a atualização dos dados.

Frescura versus otimização

Voltando ao nosso exemplo de invalidação baseada em TTL, expliquei anteriormente que se solicitarmos uma nova cópia dos dados que tenha um TTL de cinco minutos e um segundo depois, os dados na origem mudam, então nosso cache estará operando em dados desatualizados dos quatro minutos e 59 segundos restantes. Se isso é inaceitável, uma solução seria reduzir o TTL, reduzindo assim a duração em que poderíamos operar com dados obsoletos. Então, talvez reduzamos o TTL para um minuto. Isso significa que nossa janela de obsolescência é reduzida a um quinto do que era, mas fizemos cinco vezes mais chamadas para o origem, portanto, temos que considerar a latência associada e o impacto da carga.

Equilibrar essas forças vai se resumir a entender o requisitos do usuário final e do sistema mais amplo. Obviamente, os usuários irão sempre querer operar com os dados mais recentes, mas não se isso significar o sistema cai sob carga. Da mesma forma, às vezes a coisa mais segura a fazer é virar desative os recursos se um cache falhar, a fim de evitar uma sobrecarga na origem causando problemas mais sérios. Quando se trata de ajustar o que, onde e como armazenar em cache, muitas vezes você terá que se equilibrar entre vários eixos. Este é apenas mais um motivo para tentar manter as coisas o mais simples possível - quanto menos caches, mais fácil será raciocinar sobre o sistema.

Envenenamento por cache: um conto de advertência

Com o armazenamento em cache, muitas vezes pensamos que, se errarmos, a pior coisa possível acontece é que servimos dados obsoletos por um tempo. Mas o que acontece se você acabar servindo dados obsoletos para sempre? No capítulo 12, apresentei a AdvertCorp, onde eu estava trabalhando para ajudar a migrar vários legados existentes aplicações em uma nova plataforma usando o padrão strangler fig. Isso envolveu a interceptação de chamadas para vários aplicativos legados e, onde

esses aplicativos foram movidos para a nova plataforma, desviando as chamadas.
Nosso novo aplicativo operou de forma eficaz como um proxy. Tráfego para idosos
aplicativos legados que ainda não havíamos migrado foram roteados por meio de nosso novo
aplicação aos aplicativos legados downstream. Para os apelos ao legado
aplicativos, fizemos algumas tarefas domésticas; por exemplo, garantimos que
que os resultados do aplicativo legado tinham cabeçalhos de cache HTTP adequados
aplicado.

Um dia, logo após uma liberação normal de rotina, algo estranho começou
acontecendo. Foi introduzido um bug em que um pequeno subconjunto de páginas era
passando por uma condição lógica em nosso código de inserção de cabeçalho de cache, resultando
em nós não alteramos o cabeçalho de forma alguma. Infelizmente, isso a jusante
o aplicativo também havia sido alterado algum tempo antes para incluir um
Expira: nunca é cabeçalho HTTP. Isso não tinha surtido nenhum efeito antes, pois nós
estavam substituindo esse cabeçalho. Agora não estávamos.

Nosso aplicativo fez uso intenso do Squid para armazenar tráfego HTTP, e nós
percebi o problema rapidamente, pois estávamos vendo mais solicitações
contornando o próprio Squid para atingir nossos servidores de aplicativos. Corrigimos o cache
código de cabeçalho e lançamos uma versão, e também limpamos manualmente o
região relevante do cache do Squid. No entanto, isso não foi suficiente.

Como acabamos de discutir, você pode armazenar em vários lugares, mas às vezes
ter muitos caches torna sua vida mais difícil, não mais fácil. Quando se trata de
servindo conteúdo para usuários de um aplicativo web voltado para o público, você poderia
tenha vários caches entre você e seu cliente. Você não só pode ser
liderando seu site com algo como uma rede de entrega de conteúdo, mas
alguns ISPs usam o cache. Você pode controlar esses caches? E mesmo que
você poderia, há um cache sobre o qual você tem pouco controle: o cache em um
navegador do usuário.

Essas páginas com expirações: nunca ficam presas nos caches de muitos de nossos usuários
e nunca seria invalidado até que o cache ficasse cheio ou o usuário
limpe-os manualmente. Claramente, não pudemos fazer nenhuma das coisas acontecer;
nossa única opção era alterar os URLs dessas páginas para que fossem
reformulado.

O armazenamento em cache pode ser realmente muito poderoso, mas você precisa entender a íntegra caminho de dados que é armazenado em cache da origem ao destino para realmente apreciar seu complexidades e o que pode dar errado.

Escalonamento automático

Se você tiver a sorte de ter um provisionamento totalmente automatizável de dispositivos virtuais hospeda e pode automatizar totalmente a implantação de suas instâncias de microsserviços, então você tem os blocos de construção para permitir que você escale automaticamente seu microsserviços.

Por exemplo, você pode fazer com que a escala seja acionada por tendências conhecidas. Talvez você saiba que o pico de carga do seu sistema ocorre entre 9h e 17h, então você abre instâncias adicionais às 8h45 e as desativa às 5h15 p.m. Se você estiver usando algo como o AWS (que tem um suporte muito bom para escalonamento automático (incorporado), desativar instâncias que você não precisa mais ajudará economize dinheiro. Você precisará de dados para entender como sua carga muda hora. de dia para dia e de semana para semana. Alcumas empresas têm ciclos sazonais óbvios também, então você pode precisar de dados que retornem de forma justa até faça julgamentos adequados.

Por outro lado, você pode ser reativo, trazendo instâncias adicionais quando você vê um aumento na carga ou uma falha na instância e remove instâncias em que você não precisa mais delas. Sabendo com que rapidez você pode escalar uma vez que você identifica uma tendência de alta é fundamental. Se você souber, receberá apenas alguns minutos de antecedência sobre um aumento na carga, mas o aumento de escala exigirá pelo menos 10 minutos, então você sabe que precisará manter a capacidade extra em torno de preencha essa lacuna. Ter um bom conjunto de testes de carga é quase essencial aqui. Você pode usá-los para testar suas regras de escalonamento automático. Se você não tem testes que pode reproduzir cargas diferentes que acionarão o dimensionamento, então você só vai para descobrir na produção se você errou nas regras. E as consequências de fracassar não é ótimo!

Um site de notícias é um ótimo exemplo de um tipo de negócio no qual você pode querer um combinação de escalonamento preditivo e reativo. No último site de notícias em que trabalhei, nós viu tendências diárias muito claras, com vistas subindo da manhã até

hora do almoço e depois começando a declinar. Esse padrão foi repetido dia após dia dia de folga, com tráfego geralmente mais baixo no fim de semana. Isso nos deu uma boa tendência clara que pode impulsionar o dimensionamento proativo de recursos, seja para cima ou para baixo. Para outro lado, uma grande notícia causaria um aumento inesperado, exigindo mais capacidade e, muitas vezes, em curto prazo.

Na verdade, vejo que o escalonamento automático é usado muito mais para lidar com falhas de instâncias do que para reagir às condições de carga. A AWS permite que você especifique regras como "Lá deve haver pelo menos cinco instâncias nesse grupo", de modo que, se uma instância for para baixo, um novo é lançado automaticamente. Eu vi essa abordagem levar a um divertido jogo de whack-a-mole quando alguém se esquece de desligar a regra e depois tenta remover as instâncias para manutenção, apenas para vê-las continuar girando!

Tanto a escala reativa quanto a preditiva são muito úteis e podem ajudar você a ser muito mais econômico se você estiver usando uma plataforma que permite pagar somente por os recursos de computação que você usa. Mas eles também exigem cuidado observação dos dados disponíveis para você. Eu sugiro usar o escalonamento automático para as condições de falha só as primeiras enquanto você coleta os dados. Uma vez que você queira começar escalonamento automático para carga, certifique-se de ter muito cuidado ao reduzir a escala também rapidamente. Na maioria das situações, ter mais poder de computação disponível do que você precisar é muito melhor do que não ter o suficiente!

Começando de novo

A arquitetura que faz você começar pode não ser a arquitetura que mantém você funciona quando seu sistema precisa lidar com volumes de carga muito diferentes. Como já vimos, existem algumas formas de dimensionamento que podem ser extremamente impacto limitado na arquitetura do seu sistema: escalabilidade vertical e duplicação horizontal, por exemplo. Em certos pontos, porém, você precisa fazer algo bastante radical para mudar a arquitetura do seu sistema para suportar o próximo nível de crescimento.

Relembre a história de Gilt, que abordamos em "Execução Isolada". UMA A aplicação monolítica simples do Rails funcionou bem para a Gilt por dois anos. É os negócios se tornaram cada vez mais bem-sucedidos, o que significou mais clientes e

mais carga. Em um certo ponto de inflexão, a empresa teve que redesenhar o aplicativo para lidar com a carga que estava vendo.

Um redesenho pode significar a separação de um monólito existente, como aconteceu com Gilt. Ou isso pode significar escolher novos armazenamentos de dados que possam lidar melhor com a carga. É também pode significar a adoção de novas técnicas, como sair do síncrono solicitação-resposta a sistemas baseados em eventos, adotando uma nova implantação plataformas, mudando pilhas de tecnologia inteiras ou tudo mais.

Existe o perigo de que as pessoas vejam a necessidade de rearquitetar quando tiverem certeza limites de escala são alcançados como uma razão para construir em grande escala a partir de o começo. Isso pode ser desastroso. No início de um novo projeto, muitas vezes não sabemos exatamente o que queremos construir, nem sabemos se será bem sucedido. Precisamos ser capazes de experimentar e entender rapidamente o que capacidades que precisamos desenvolver. Se tentássemos construir em grande escala desde o início, acabariamos carregando antecipadamente uma grande quantidade de trabalho para nos preparar para carregar isso pode nunca acontecer, ao mesmo tempo em que desvia o esforço de atividades mais importantes, gosto de entender se alguém realmente vai querer usar nosso produto. Eric Ries conta a história de passar seis meses construindo um produto que ninguém nunca baixado. Ele refletiu que poderia ter colocado um link em uma página da web que 404 quando as pessoas clicaram nele para ver se havia alguma demanda, gastaram seis meses na praia e aprendi o mesmo!

A necessidade de mudar nossos sistemas para lidar com a escala não é um sinal de falha. É um sinal de sucesso.

Resumo

Como podemos ver, qualquer que seja o tipo de escalabilidade que você esteja procurando, microserviços oferecem muitas opções diferentes em termos de como você aborda o problema.

Os eixos de escala podem ser um modelo útil para usar ao considerar quais tipos de escalabilidade estão disponíveis para você:

Escala vertical

Em poucas palavras, isso significa adquirir uma máquina maior.

Duplicação horizontal

Ter várias coisas capazes de fazer o mesmo trabalho.

Particionamento de dados

Dividindo o trabalho com base em algum atributo dos dados, por exemplo, grupo de clientes.

Decomposição funcional

Separação do trabalho com base no tipo, por exemplo, decomposição de microserviços

A chave para muito disso é entender o que você quer - técnicas que são

eficaz no escalonamento para melhorar a latência pode não ser tão eficaz para ajudar

escala por volume.

Espero também ter mostrado, porém, que muitas das formas de dimensionamento que temos

discutidos resultam em maior complexidade em seu sistema. Então, ser alvo de

termos do que você está tentando mudar e evitando os perigos de ser prematuro

a otimização é importante.

Em seguida, passamos da análise do que acontece nos bastidores para a

partes mais visíveis do nosso sistema - a interface do usuário.

1 Martin L. Abbott e Michael T. Fisher, *A arte da escalabilidade: arquitetura web escalável, Processos e organizações para a empresa moderna*, 2^a ed. (Nova York: Addison-Wesley, 2015).

2 Como antes, eu anonimizei a empresa - Foodco não é seu nome verdadeiro!

3 Gregor Hohpe e Bobby Woolf, *Padrões de integração empresarial* (Boston: Addison-Wesley, 2003).

4 Pramod J. Sadalage e Martin Fowler, *NoSQL Distilled: um breve guia para os emergentes* Mundo da persistência poliglota (Upper Saddle River, NJ: Addison-Wesley, 2012).

5 Não me fale sobre pessoas que começam a falar sobre hipóteses e depois falam sobre crenja - escolhendo informações para confirmar suas crenças já mantidas.

Parte III. Pessoas

Capítulo 14. Interfaces de usuário

Até agora, ainda não abordamos o mundo da interface do usuário. Alguns dos você pode estar apenas fornecendo uma API clínica fria e rígida para seus clientes, mas muitos de nós queremos criar coisas bonitas, interfaces de usuário funcionais que encantarão nossos clientes. A interface do usuário, afinal de contas, é onde reuniremos todos esses microsserviços algo que faz sentido para nossos clientes.

Quando comecei a computar, falávamos principalmente sobre clientes grandes e gordos que rodava em nossos desktops. Passei muitas horas com o Motif e depois com o Swing tentando para tornar meu software o mais agradável de usar possível. Freqüentemente, esses sistemas eram apenas para a criação e manipulação de arquivos locais, mas muitos deles tinham um componente do lado do servidor. Meu primeiro emprego na Thoughtworks envolveu a criação de um Sistema eletrônico de ponto de venda baseado em swing que era apenas um de um grande número de partes móveis, a maioria das quais estavam no servidor.

Depois veio a web. Em vez disso, começamos a pensar em nossas interfaces de usuário como sendo "finas" com mais lógica no lado do servidor. No início, nosso lado do servidor programas renderizaram a página inteira e a enviaram para o navegador do cliente, o que fez muito pouco. Todas as interações foram tratadas no lado do servidor por meio de GETs e POSTs acionados pelo usuário clicando em links ou preenchendo formulários. Com o tempo, o JavaScript se tornou uma opção mais popular para adicionar comportamento dinâmico ao interface de usuário baseada em navegador, e alguns aplicativos podem ser tão "gordos" quanto os clientes de desktop antigos. Posteriormente, tivemos o surgimento do aplicativo móvel, e hoje temos um cenário variado para fornecer gráficos ao usuário. interfaces para nossos usuários - plataformas diferentes e tecnologias diferentes para essas plataformas. Essa variedade de tecnologias nos oferece uma série de opções de como podemos criar interfaces de usuário eficazes apoiadas por microsserviços. Nós seremos explorando tudo isso e muito mais neste capítulo.

Nos últimos dois anos, as organizações começaram a se afastar de pensar que a web ou o celular devem ser tratados de forma diferente; em vez disso, eles são pensando no digital de forma mais holística. Qual é a melhor maneira para o nosso clientes para usar os serviços que oferecemos? E o que isso faz com o nosso sistema? arquitetura? A compreensão de que não podemos prever exatamente como o cliente pode acabar interagindo com nossos produtos e impulsionou a adoção de APIs mais granulares, como as fornecidas por microserviços. Ao combinar o capacidades que nossos microserviços expõem de maneiras diferentes, podemos organizar experiências diferentes para nossos clientes em seu aplicativo de desktop, celular dispositivo e dispositivo vestível e até mesmo em forma física, se visitarem nosso loja física.

Então, pense nas interfaces de usuário como os lugares onde unimos o várias vertentes dos recursos que queremos oferecer aos nossos usuários. Com isso em Veja bem, como juntamos todos esses fios? Precisamos ver isso problema de dois lados: o quem e o como. Em primeiro lugar, consideraremos o aspectos organizacionais - quem possui quais responsabilidades quando se trata de fornecendo interfaces de usuário? Em segundo lugar, veremos um conjunto de padrões que podem ser usado para implementar essas interfaces.

Modelos de propriedade

Como discutimos no Capítulo 1, a arquitetura tradicional em camadas pode causar problemas quando se trata de fornecer software de forma eficaz. Em Figura 14-1, vemos um exemplo em que a responsabilidade pela interface do usuário. A camada é de propriedade de uma única equipe de front-end, com os serviços de back-end funcionando feito por outro. Neste exemplo, adicionar um controle simples envolve trabalho sendo feito por três equipes diferentes. Esses tipos de organização hierárquica estruturas podem impactar significativamente nossa velocidade de entrega devido à necessidade de coordene constantemente as mudanças e distribua o trabalho entre as equipes.

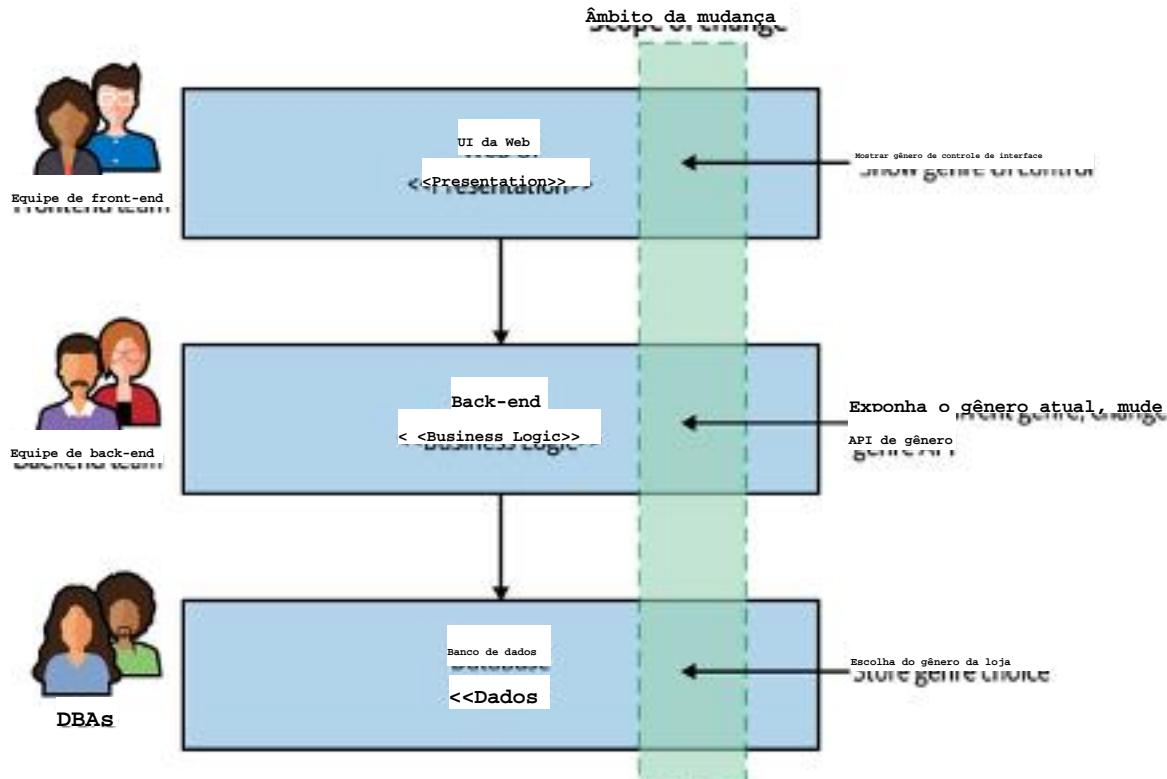


Figura 14-1. Fazer uma mudança em todos os três níveis é mais complicado.

O modelo que eu prefiro e acho que está mais bem alinhado para atingir a meta de implantabilidade independente é ter a interface do usuário dividida e gerenciada por um equipe que também gerencia os componentes do lado do servidor, como vemos na Figura 14-2. Aqui, uma única equipe acaba sendo responsável por todas as mudanças que precisamos para fazer com que adicionemos nosso novo controle.

Equipes com propriedade total da funcionalidade de ponta a ponta são capazes de criar muda mais rapidamente. Ter a propriedade total incentiva cada equipe a ter um ponto de contato direto com um usuário final do software. Com equipes de back-end, é fácil perder a noção de quem é o usuário final.

Apesar das desvantagens, eu (infelizmente) ainda vejo a equipe dedicada de front-end como sendo o padrão organizacional mais comum entre empresas que faça uso de microserviços. Por que isso?

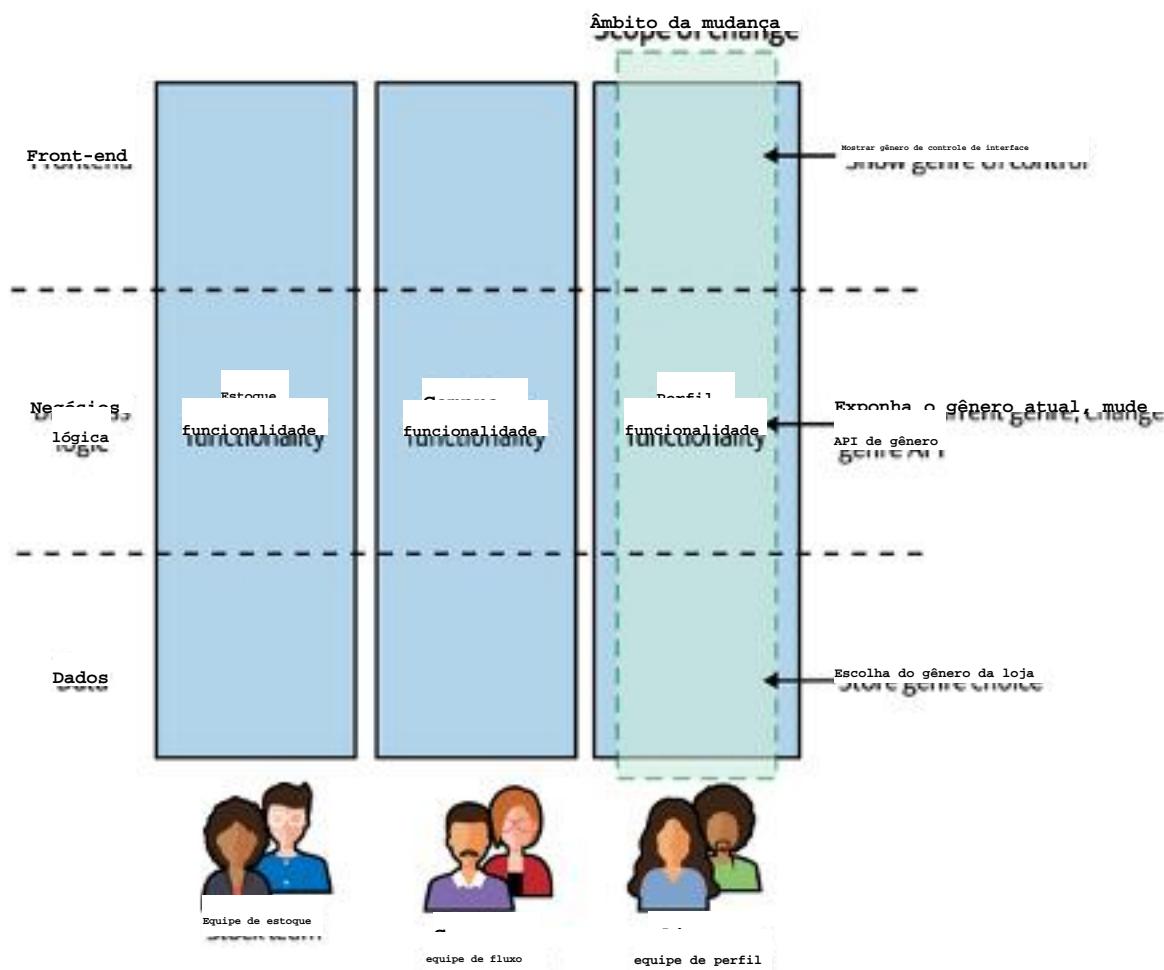


Figura 14-2. A interface do usuário está dividida e pertence a uma equipe que também gerencia o lado do servidor funcionalidade que suporta a interface do usuário

Drivers para equipes dedicadas de front-end

O desejo por equipes de front-end dedicadas parece se resumir a três pontos principais fatores: escassez de especialistas, busca de consistência e técnicas desafios.

Em primeiro lugar, fornecer uma interface de usuário requer um certo grau de habilidades especializadas.

Existem os aspectos de interação e design gráfico, e depois há o conhecimento técnico necessário para oferecer um ótimo aplicativo web ou nativo experiência. Especialistas com essas habilidades podem ser difíceis de encontrar, e como estes pessoas são uma mercadoria tão rara que a tentação é tentá-las todas para que você possa ter certeza de que eles estão se concentrando apenas em sua especialidade.

O segundo fator para uma equipe separada de front-end é a consistência. Se você tiver uma equipe responsável por fornecer sua interface de usuário voltada para o cliente, você pode garantir que sua interface tenha uma aparência consistente. Você usa um conjunto consistente de controles para resolver problemas semelhantes para que a interface do usuário tenha a aparência e o toque como uma entidade única e coesa.

Finalmente, pode ser difícil trabalhar com algumas tecnologias de interface de usuário em um modo não monolítico. Aqui estou pensando especificamente em uma única página aplicativos (SPAs), que historicamente pelo menos não foram fáceis de quebrar separado. Tradicionalmente, uma interface de usuário da web consistiria em várias redes páginas, e você navegaria de uma página para outra. Com SPAs, o Em vez disso, o aplicativo inteiro é exibido em uma única página da web. Estruturas como Angular, React e Vue, teoricamente permitem a criação de mais interfaces de usuário sofisticadas do que sites "antiquados". Estaremos procurando em um conjunto de padrões que podem oferecer diferentes opções de como decompor um interface de usuário mais adiante neste capítulo, e em termos de SPAs, mostrarei como o O conceito de micro front-end pode permitir que você use estruturas de SPA enquanto ainda está evitando a necessidade de uma interface de usuário monolítica.

Rumo a equipes alinhadas ao fluxo

Fu acho que ter uma equipe de front-end dedicada é, em geral, um erro se você for tentando otimizar para um bom rendimento, isso cria novos pontos de transferência em seu organização, desacelerando as coisas. Idealmente, nossas equipes estão alinhadas fatias de funcionalidade de ponta a ponta, permitindo que cada equipe ofereça novos recursos para seus clientes e, ao mesmo tempo, reduz a quantidade de coordenação necessária. Meu o modelo preferido é uma equipe que possui a entrega de funcionalidade de ponta a ponta em uma parte específica do domínio. Isso combina com o que Matthew Skelton e Manuel Pais descreve como equipes alinhadas ao fluxo em seu livro *Equipe Topologias*. Como eles descrevem:

Uma equipe alinhada ao fluxo é uma equipe alinhada a um único e valioso fluxo de trabalho [A] equipe tem o poder de criar e entregar um cliente ou usuário valorize da forma mais rápida, segura e independente possível, sem exigindo transferências para outras equipes para realizar partes do trabalho.

desenvolvedores). 2. Uma equipe com responsabilidade de ponta a ponta pela entrega de dados ao usuário

a funcionalidade de enfrentamento também terá uma conexão direta mais óbvia também para o usuário final. Com muita frequência, já vi equipes de "back-end" com uma nebulosa ideia do que o software faz ou do que os usuários precisam, o que pode causar todos tipos de mal-entendidos quando se trata de implementar novas funcionalidades. Por outro lado, as equipes de ponta a ponta acharão muito mais fácil criar um conexão direta com as pessoas que usam o software que elas criam - elas podem ser mais focados em garantir que as pessoas a quem estão servindo recebam o que eles precisam.

Como exemplo concreto, passei algum tempo trabalhando com a FinanceCo, uma empresa Fintech bem-sucedida e em crescimento com sede na Europa. Na FinanceCo, praticamente todas as equipes trabalham em um software que afeta diretamente o cliente experiência e tenha indicadores-chave de desempenho (KPIs) orientados para o cliente - o sucesso de uma determinada equipe é impulsionado menos pela quantidade de recursos que ela possui enviado e muito mais, ajudando a melhorar a experiência das pessoas que usam o software. Fica muito claro como uma mudança pode impactar os clientes. Isso só é possível devido ao fato de que a maioria das equipes tem, diretamente, responsabilidades voltadas para o cliente em termos do software que eles fornecem. Quando você está mais distante do usuário final, fica mais difícil de entender se suas contribuições são bem-sucedidas e você pode acabar se concentrando em metas que estão muito distantes das coisas que as pessoas usam seu software preocupe-se com.

Vamos revisitar as razões pelas quais existem equipes de front-end dedicadas: especialistas, consistência e desafios técnicos - e agora vamos ver como esses os problemas podem ser resolvidos.

Especialistas em compartilhamento

Pode ser difícil encontrar bons desenvolvedores, e encontrá-los é ainda mais complicado quando você precisa de desenvolvedores com uma determinada especialidade. Na área do usuário interfaces, por exemplo, se você estiver fornecendo dispositivos móveis nativos e web interfaces, você pode precisar de pessoas com experiência em iOS e Android, bem como com desenvolvimento web moderno. Isso é bastante

além do fato de que você pode querer designers de interação dedicados, designers gráficos, especialistas em acessibilidade e afins. Pessoas com o direito a profundidade de habilidades para esses campos mais "estreitos" pode ser escassa, e eles podem sempre ter mais trabalho do que tempo para.

Como mencionamos anteriormente, a abordagem tradicional da organização estruturas fariam com que você colocasse todas as pessoas que possuem a mesma habilidade colocados na mesma equipe, permitindo que você controle rigorosamente em que eles trabalham. Mas como também discutimos, isso leva a organizações isoladas.

Colocar pessoas com habilidades especializadas em sua própria equipe dedicada também priva você de oportunidades para que outros desenvolvedores as aprendam em habilidades de demanda. Você não precisa que todo desenvolvedor aprenda a se tornar um especialista Desenvolvedor iOS, por exemplo, mas ainda pode ser útil para alguns de seus desenvolvedores para aprender habilidades suficientes nessa área para ajudar com as coisas mais fáceis, deixando seus especialistas livres para enfrentar as tarefas realmente difíceis. O compartilhamento de habilidades pode também seja ajudado pelo estabelecimento de comunidades de prática - você pode considerar ter uma comunidade de interface de usuário que abrange suas equipes, permitindo que as pessoas compartilhem ideias e desafios com seus colegas.

Lembro-me de quando todas as alterações no banco de dados precisavam ser feitas por um pool central de administradores de banco de dados (DBAs). Os desenvolvedores tinham pouca consciência de como os bancos de dados funcionavam como resultado e criavam com mais frequência software que usou mal o banco de dados. Além disso, grande parte do trabalho dos experientes Os DBAs que estavam sendo solicitados a fazer consistiam em mudanças triviais. Quanto mais banco de dados o trabalho foi direcionado para as equipes de entrega, os desenvolvedores em geral melhoraram apreciação por bancos de dados e poderiam começar a fazer o trabalho trivial sozinhos, liberando os valiosos DBAs para se concentrarem em problemas mais complexos de banco de dados, que fizeram melhor uso de suas habilidades e experiências mais profundas. Uma mudança semelhante aconteceu no espaço de operações e testadores, com mais deste trabalho sendo colocado em equipes.

Portanto, ao contrário, retirar especialistas de equipes dedicadas não inibirá a capacidade dos especialistas de fazer seu trabalho; na verdade, provavelmente aumentará o largura de banda: eles precisam se concentrar nos problemas difíceis que realmente precisam de seus atenção.

O truque é encontrar uma maneira mais eficaz de contratar seus especialistas. Idealmente, eles seriam incorporados às equipes. Às vezes, porém, pode não haver trabalho suficiente para justificar sua presença em tempo integral em uma determinada equipe, caso em que eles podem muito bem dividir seu tempo entre várias equipes. Outro modelo é tenha uma equipe dedicada com essas habilidades, cujo trabalho explícito é capacitar outras equipes. Em Team Topologies, Skelton e Pais descrevem essas equipes como capacitando equipes. O trabalho deles é sair e ajudar outras equipes que estão focadas sobre o fornecimento de novos recursos para realizar seu trabalho. Você pode pensar nessas equipes como um pouco mais como uma consultoria interna - eles podem entrar e gastar de forma direcionada tempo com uma equipe alinhada ao fluxo, ajudando-a a se tornar mais autossuficiente em um área específica ou então fornecendo algum tempo dedicado para ajudar a implantar um trabalho particularmente difícil.

Então, se seus especialistas estão incorporados em tempo integral em uma determinada equipe ou trabalham para permitir que outras pessoas façam o mesmo trabalho, você pode remover os silos organizacionais e também ajuda a capacitar seus colegas ao mesmo tempo.

Garantindo a consistência

O segundo problema frequentemente citado como motivo para equipes de front-end dedicadas é que de consistência. Ao ter uma única equipe responsável por uma interface de usuário, você garanta que a interface do usuário tenha uma aparência consistente. Isso pode se estender de fácil coisas como usar as mesmas cores e fontes para resolver a mesma interface problemas da mesma forma - usando um design e uma interação consistentes linguagem que ajuda os usuários a interagir com o sistema. Essa consistência não apenas ajude a comunicar um certo grau de polimento em relação ao produto em si, mas também garante que os usuários achem mais fácil usar a nova funcionalidade quando ela estiver entregue.

No entanto, existem maneiras de ajudar a garantir um certo grau de consistência entre as equipes. Se você estiver usando o modelo de equipe capacitadora, com especialistas passando tempo com várias equipes, elas podem ajudar a garantir que o trabalho realizado por cada equipe seja consistente. A criação de recursos compartilhados, como um guia de estilo CSS vivo ou componentes de interface de usuário compartilhados também podem ajudar.

Como um exemplo concreto de uma equipe capacitadora sendo usada para ajudar com consistência, a equipe do Financial Times Origami cria componentes da web em colaboração com a equipe de design, que encapsula a identidade da marca-garantindo uma aparência consistente em todas as equipes alinhadas ao fluxo. Esse tipo de capacitar a equipe fornece duas formas de ajuda: em primeiro lugar, ela compartilha sua experiência em fornecendo componentes já construídos e, em segundo lugar, ajuda a garantir que o

As interfaces de usuário oferecem uma experiência de usuário consistente.

É importante notar, no entanto, que o fator para a consistência não deve ser considerado universalmente correto. Algumas organizações fazem com que pareça consciente decisões de não exigir consistência em suas interfaces de usuário, porque sentem que é preferível permitir uma maior autonomia das equipes. A Amazon é uma dessas organizações. As versões anteriores de seu principal site de compras tinham grandes graus de inconsistência, com widgets usando estilos de controles bem diferentes.

Isso é mostrado em um grau ainda maior quando você analisa o controle da web. painel para Amazon Web Services (AWS). Diferentes produtos na AWS têm modelos de interação extremamente diferentes, tornando a interface do usuário bastante desconcertante. Isso, no entanto, parece ser uma extensão lógica da Amazon esforço para reduzir a coordenação interna entre as equipes.

A maior autonomia das equipes de produto na AWS parece se manifestar também de outras formas, não apenas em termos de uma experiência de usuário muitas vezes desarticulada. Freqüentemente, existem várias maneiras diferentes de realizar a mesma tarefa (por executando uma carga de trabalho de container, por exemplo), com diferentes equipes de produto dentro da AWS, muitas vezes se sobrepondo umas às outras com, mas soluções incompatíveis. Você pode criticar o resultado final, mas a AWS tem mostrou que, ao ter essas equipes altamente autônomas orientadas a produtos, tem criou uma empresa que tem uma clara liderança de mercado. A velocidade de entrega supera um consistência da experiência do usuário, pelo menos no que diz respeito à AWS.

Superando desafios técnicos

Passamos por algumas evoluções interessantes quando se trata do desenvolvimento de interfaces de usuário - a partir de texto baseado em terminal de tela verde interfaces de usuário para aplicativos de desktop avançados, a web e, agora, dispositivos móveis nativos

experiências. De muitas maneiras, fechamos um círculo e, mais um pouco, nosso os aplicativos cliente agora são criados com tanta complexidade e sofisticação que eles rivalizam absolutamente com a complexidade dos ricos aplicativos de desktop que foram a base do desenvolvimento da interface de usuário, até a primeira década do século XXI.

Até certo ponto, quanto mais as coisas mudam, mais elas permanecem as mesmas. Nós muitas vezes ainda funcionam com os mesmos controles de interface de usuário de 20 anos atrás - botões, caixas de seleção, formulários, caixas de combinação e similares. Nós adicionamos mais alguns componentes desse espaço, mas muito menos do que você imagina. O que tem mudou a tecnologia que usamos para criar essas interfaces gráficas de usuário em o primeiro lugar.

Algumas tecnologias mais recentes neste espaço, especificamente aplicativos de página única, nos causam problemas quando se trata de decompor uma interface de usuário. Além disso, a maior variedade de dispositivos nos quais esperamos que a mesma interface de usuário ser entregue causa outros problemas que precisam ser resolvidos.

Fundamentalmente, nossos usuários querem interagir com nosso software da mesma forma que da maneira mais possível. Seja por meio de um navegador em um desktop ou por meio de um nativo ou aplicativo móvel na web, o resultado é que os mesmos usuários interagem com nosso software através de um único painel de vidro. Eles não devem se importar com a interface do usuário é construído de forma modular ou monolítica. Então, temos que procurar maneiras de quebrar separar nossa funcionalidade de interface de usuário e reúna tudo novamente, tudo ao mesmo tempo em que resolve os desafios causados por aplicativos de página única, móveis dispositivos e muito mais. Essas questões nos ocuparão pelo resto deste capítulo.

Padrão: Frontend monolítico

O padrão monolítico de front-end descreve uma arquitetura na qual toda a interface do usuário, o estado e o comportamento são definidos na própria interface do usuário, com chamadas feitas para o suporte microserviços para obter os dados necessários ou realizar as operações necessárias. A Figura 14-3 mostra um exemplo disso. Nossa tela quer exibir informações sobre um álbum e sua lista de faixas, então a interface faz uma solicitação para extraia esses dados do microserviço Album. Também exibimos informações

sobre as ofertas especiais mais recentes, solicitando informações do ...
 Microsserviço de promoções. Neste exemplo, nossos microsserviços retornam JSON que a interface do usuário usa para atualizar as informações exibidas.

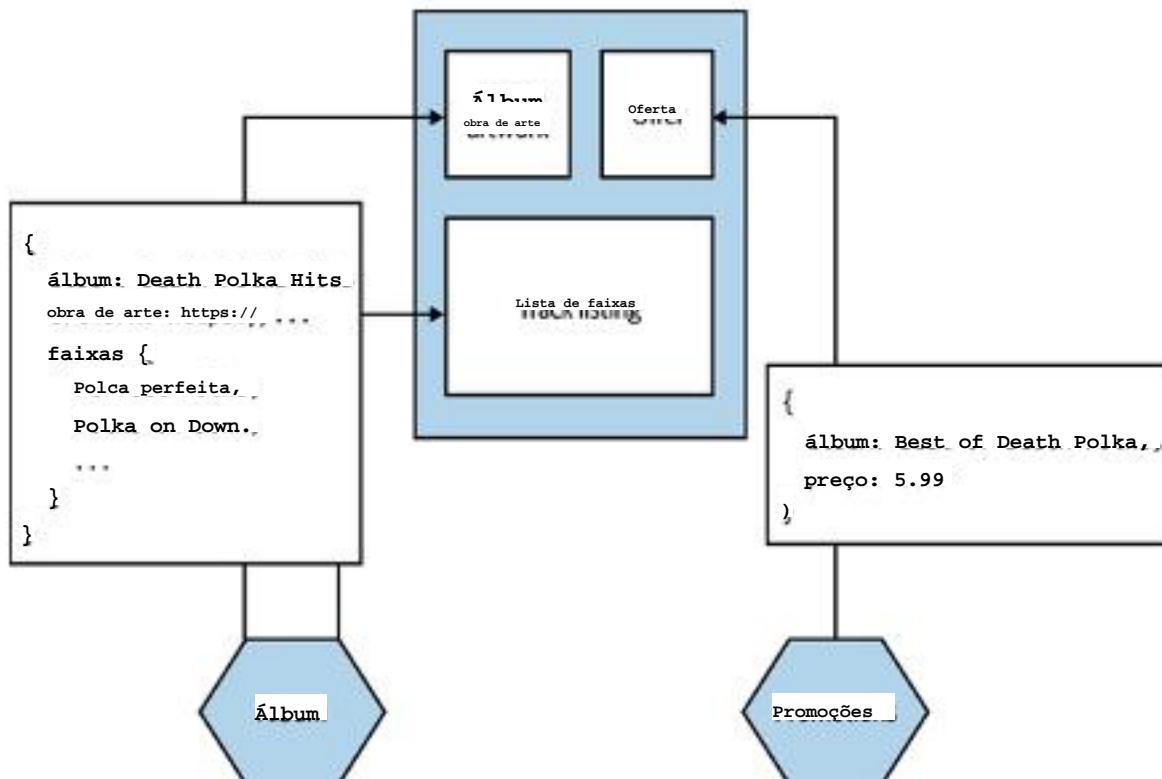


Figura 14-3. Nossa tela de detalhes do álbum extrai informações de microsserviços posteriores para renderizar

Este modelo é o mais comum para pessoas que constroem monolíticos individuais aplicativos de página, geralmente com uma equipe de front-end dedicada. Os requisitos do nossos microsserviços são bastante simples - eles só precisam compartilhar informações em um formato que pode ser facilmente interpretado pela interface do usuário. No caso de um interface de usuário baseada na web, isso significa que nossos microsserviços provavelmente precisariam fornecer dados em formato textual, sendo o JSON a escolha mais provável. A interface do usuário seria em seguida, precisa criar os vários componentes que compõem a interface, lidar com a sincronização de estados e similares com o back-end. Usando um protocolo binário para comunicação de serviço a serviço seria mais difícil para clientes baseados na web, mas pode ser bom para dispositivos móveis nativos ou aplicativos de desktop "grossos".

Quando usá-lo

Existem algumas desvantagens nessa abordagem. Em primeiro lugar, por sua natureza como entidade monolítica, ela pode se tornar um driver para (ou ser causada por) uma entidade dedicada à equipe de front-end. Ter várias equipes compartilhando a responsabilidade por esse monolítico o front-end pode ser um desafio devido às várias fontes de contêncio.

Em segundo lugar, temos pouca capacidade de adaptar as respostas para diferentes tipos de dispositivos. Se estiver usando a tecnologia da web, podemos alterar o layout de uma tela para acomoda diferentes restrições de dispositivos, mas isso não necessariamente se estende à alteração das chamadas que estão sendo feitas para os microserviços de suporte. Meu celular o cliente pode exibir apenas 10 campos de um pedido, mas se o microservice retira todos os cem campos do pedido e acabamos recuperando dados desnecessários. Uma solução para essa abordagem é para o usuário interface para especificar quais campos recuperar ao fazer uma solicitação, mas isso pressupõe que cada microserviço de suporte suporte essa forma de interação.

Em "GraphQL", veremos como usar tanto o backend quanto o front-end pattern e GraphQL podem ajudar nesse caso.

Realmente, esse padrão funciona melhor quando você deseja toda a implementação e comportamento da sua interface de usuário em uma unidade implantável. Para uma única equipe em desenvolvimento tanto o front-end quanto todos os microserviços de suporte, isso pode ser bom. Pessoalmente, se você tem mais de uma equipe trabalhando em seu software, eu acho você deve lutar contra esse desejo, pois isso pode resultar em você cair em um arquitetura em camadas com os silos organizacionais associados. Se, no entanto, você não consegue evitar uma arquitetura em camadas e uma organização correspondente estrutura, esse é provavelmente o padrão que você acabará usando.

Padrão: Micro frontends

A abordagem de micro front-end é um padrão organizacional em que diferentes partes de um frontend podem ser trabalhadas e implantadas de forma independente. Para citar de um artigo altamente recomendado de Cam Jackson sobre o assunto,³ podemos definir micro frontends da seguinte forma: "Um estilo arquitetônico onde aplicativos de front-end que podem ser entregues de forma independente são compostos em um maior inteiro."

Ele se torna um padrão essencial para equipes alinhadas ao fluxo que desejam possuir fornecimento de microserviços de back-end e da interface de usuário de suporte. Onde os microserviços oferecem capacidade de implantação independente para o back-end, os micro front-ends oferecem capacidade de implantação independente para o front-end.

O conceito de micro front-end ganhou popularidade devido aos desafios criados por interfaces de usuário web monolíticas e cheias de JavaScript, conforme tipificado por uma única página aplicativos. Com um micro frontend, diferentes equipes podem trabalhar e criar mudanças em diferentes partes do frontend. Voltando à Figura 14-2, a equipe de estoque, a equipe de fluxo de compras e a equipe de perfil do cliente são capazes de alterar a funcionalidade de front-end associada ao seu fluxo de trabalho independentemente das outras equipes.

Implantação

Para front-ends baseados na web, podemos considerar duas chaves decomposicionais técnicas que podem auxiliar na implementação do padrão micro frontend. Widgetizada a decomposição envolve a junção de diferentes partes de um front-end em uma única tela. A decomposição baseada em páginas, por outro lado, tem o front-end dividido em páginas web independentes. Ambas as abordagens valem a pena de uma exploração mais aprofundada, que abordaremos em breve.

Quando usá-lo

O padrão micro frontend é essencial se você quiser adotar de ponta a ponta equipes alinhadas ao fluxo, nas quais você está tentando se afastar de uma camada arquitetural. Eu também poderia imaginá-la sendo útil em uma situação em que você deseja manter uma arquitetura em camadas, mas a funcionalidade do front-end é agora tão grande que várias equipes de front-end dedicadas são necessárias.

Há um problema importante com essa abordagem que não tenho certeza se pode ser resolvido. Às vezes, os recursos oferecidos por um microserviço não se encaixam perfeitamente em um widget ou uma página. Claro, talvez eu queira colocar as recomendações em uma caixa no uma página em nosso site, mas e se eu quiser tecer de forma dinâmica recomendações em outro lugar? Quando eu pesquiso, quero que o tipo esteja à frente de

aciona automaticamente novas recomendações, por exemplo. Quanto mais cruzado-
cortar uma forma de interação é, quanto menor a probabilidade de esse modelo se encaixar, e o
é mais provável que voltemos a fazer apenas chamadas de API.

SISTEMAS INDEPENDENTES

Um sistema autônomo (SCS) é um estilo de arquitetura que surgiu indiscutivelmente, devido à falta de foco nas questões de interface do usuário durante os primeiros anos de desenvolvimento. Um SCS pode consistir em várias partes móveis (potencialmente microserviços) que, quando tomados em conjunto, constituem um único SCS.

Conforme definido, um sistema independente deve estar em conformidade com alguns requisitos específicos critérios que podemos ver se sobreponem a algumas das mesmas coisas que somos tentando alcançar o sucesso com microserviços. Você pode encontrar mais informações sobre sistemas independentes no site muito claro da SCS, mas aqui estão alguns destaque:

- Cada SCS é um aplicativo web autônomo sem interface compartilhada.
- Cada SCS é de propriedade de uma equipe.
- A comunicação assíncrona deve ser usada sempre que possível.
- Nenhum código comercial pode ser compartilhado entre SCSs.

A abordagem SCS não se popularizou na mesma medida que os microserviços, e não é um conceito que eu encontre muito, apesar do fato de eu concordar com muitos dos princípios que ele descreve. Gosto especialmente de chamar isso de eu... o sistema contido deve ser de propriedade de uma equipe. Eu me pergunto se essa falta de uso mais amplo explica por que alguns aspectos da abordagem SCS parecem excessivamente restrito e prescritivo. Por exemplo, a insistência em cada SCS ser um "aplicativo web autônomo" implica que muitos tipos de usuário a interface nunca poderia ser considerada um SCS. Isso significa que o nativo iOS aplicativo que eu criei que usa gRPC pode fazer parte de um SCS ou não?

Então, a abordagem do SCS está em conflito com os microserviços? Na verdade, não. Eu tenho trabalhado em muitos microserviços que, quando considerados isoladamente, seriam se encaixam na definição de um SCS por si só. Existem algumas interessantes ideias na abordagem SCS com as quais eu concordo, e muitas delas temos já abordado neste livro. Eu só acho que a abordagem é exagerada

prescritivo, na medida em que alguém interessado em SCS possa achar adotar a abordagem é incrivelmente desafiadora, pois pode exigir mudanças generalizadas em muitos aspectos de sua entrega de software.

Eu me preocupo que manifestos como o conceito SCS possam nos guiar até o caminho de se concentrar demais na atividade, em vez de nos princípios e resultados. Você pode seguir todas as características do SCS e ainda potencialmente não entender o assunto. Refletindo, acho que a abordagem SCS é uma abordagem focada em tecnologia para promover um conceito organizacional. Como tal, Prefiro me concentrar na importância de equipes alinhadas ao fluxo com redução de coordenação e deixe a tecnologia e a arquitetura fluírem a partir disso.

Padrão: Decomposição baseada em páginas

Na decomposição baseada em páginas, nossa interface do usuário é decomposta em várias páginas. Diferentes conjuntos de páginas podem ser servidos a partir de diferentes microserviços. Em Figura 14-4, vemos um exemplo desse padrão para a MusicCorp. Solicitações de as páginas em /albums/ são roteadas diretamente para o microserviço Albums, que lida com a veiculação dessas páginas, e fazemos algo semelhante com /artistas/. Uma navegação comum é usada para ajudar a unir essas páginas. Esses microserviços, por sua vez, podem buscar as informações necessárias para construir essas páginas - por exemplo, obtendo níveis de estoque do Microserviço de inventário para mostrar na interface quais itens estão em estoque.

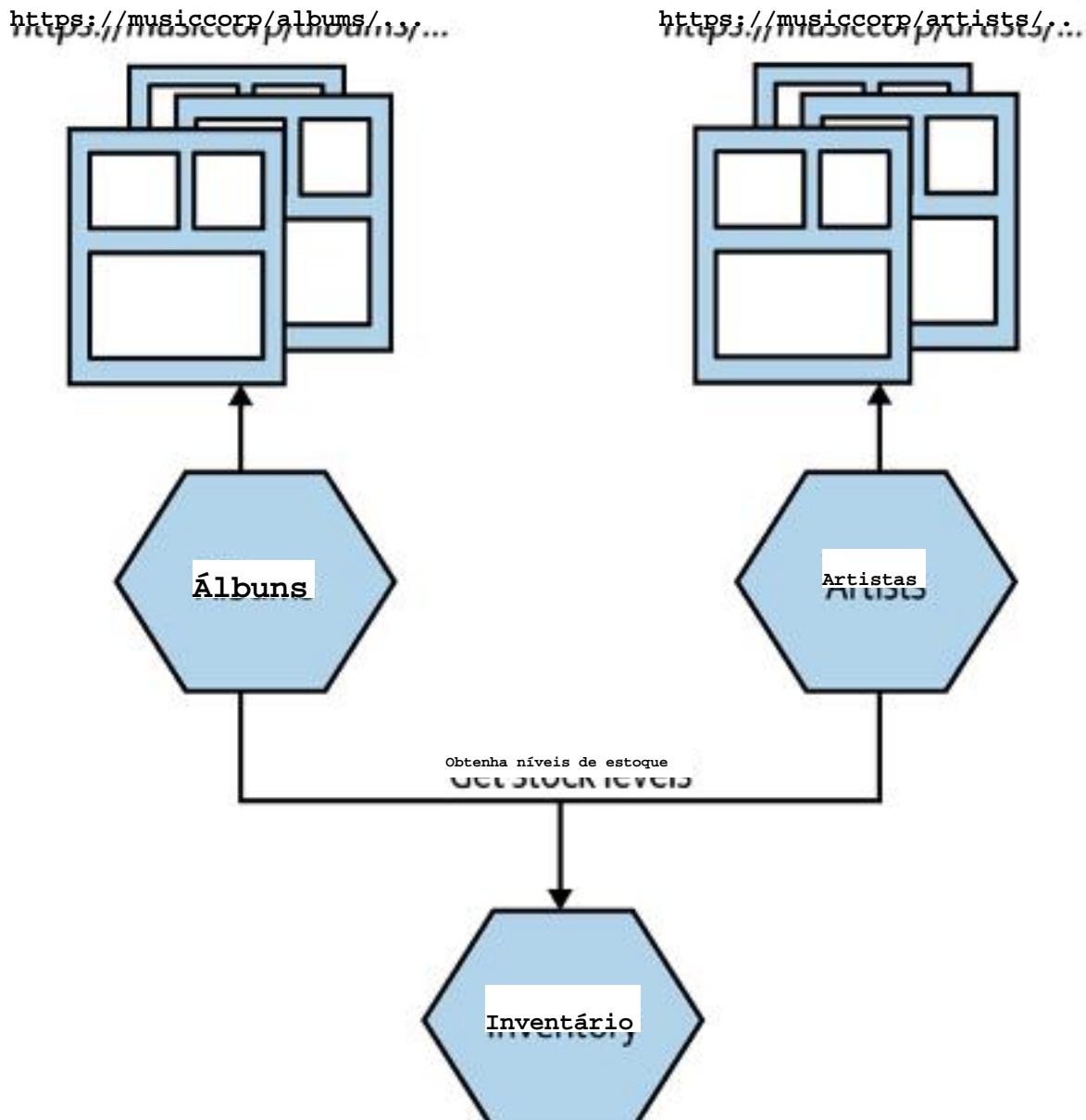


Figura 14-4 A interface do usuário consiste em várias páginas, com diferentes grupos de páginas servidas a partir de diferentes microserviços

Com esse modelo, uma equipe proprietária do microserviço Albums seria capaz de renderizar uma interface de usuário completa de ponta a ponta, facilitando para a equipe entender como as mudanças afetariam o usuário...

THE WEB A WEB

Antes dos aplicativos de página única, tínhamos a web. Nossa interação com a web foi baseado em visitar URLs e clicar em links que geraram novas páginas para ser carregado em nosso navegador. Nossos navegadores foram criados para permitir navegação por essas páginas, com marcadores para marcar páginas de interesse e controles para trás e para frente para revisitar páginas acessadas anteriormente. Vocês todos podem estar revirando os olhos e pensando: "Claro que eu sei como a web funciona!" ; no entanto, é um estilo de interface de usuário que parece cairam em desgraça. Sua simplicidade é algo que sinto falta quando vejo nossas implementações atuais de interface de usuário baseadas na web - perdemos muito por assumindo automaticamente que uma interface de usuário baseada na web significa aplicativos de página única.

Em termos de lidar com diferentes tipos de clientes, não há nada que impeça o página adaptando o que ela mostra com base na natureza do dispositivo que solicita o página. Os conceitos de aprimoramento progressivo (ou degradação graciosa) já deve estar bem compreendido.

A simplicidade da decomposição baseada em páginas a partir de uma implementação técnica O ponto de vista é um verdadeiro apelo aqui. Você não precisa de nenhum JavaScript sofisticado rodando no navegador, nem você precisa usar iFrames problemáticos. O usuário clica em um link e uma nova página é solicitada.

Onde usá-lo

Útil para uma abordagem de front-end monolítico ou microfront-end, page-decomposição baseada seria minha escolha padrão para interface de usuário decomposição se minha interface de usuário fosse um site. A página da web como uma unidade de decomposição é um conceito tão central da web como um todo que se torna uma técnica simples e óbvia para decompor um grande usuário baseado na web interface.

Acho que o problema é que, na pressa de usar um aplicativo de página única tecnologia, essas interfaces de usuário estão se tornando cada vez mais raras, na medida em que que experiências de usuário que, na minha opinião, se encaixariam melhor em um site

a implementação acaba sendo transformada em um aplicativo de uma única página. 4 Você pode, é claro, combinar a decomposição baseada em páginas com algumas das outras padrões que abordamos. Eu poderia ter uma página que contenha widgets, para exemplo - algo que veremos a seguir.

Padrão: Decomposição baseada em widgets

Com a decomposição baseada em widgets, uma tela em uma interface gráfica contém widgets que podem ser alterados de forma independente. Na Figura 14-5, vemos um exemplo do frontend da MusicCorp, com dois widgets fornecendo a interface funcionalidade para o carrinho de compras e as recomendações.

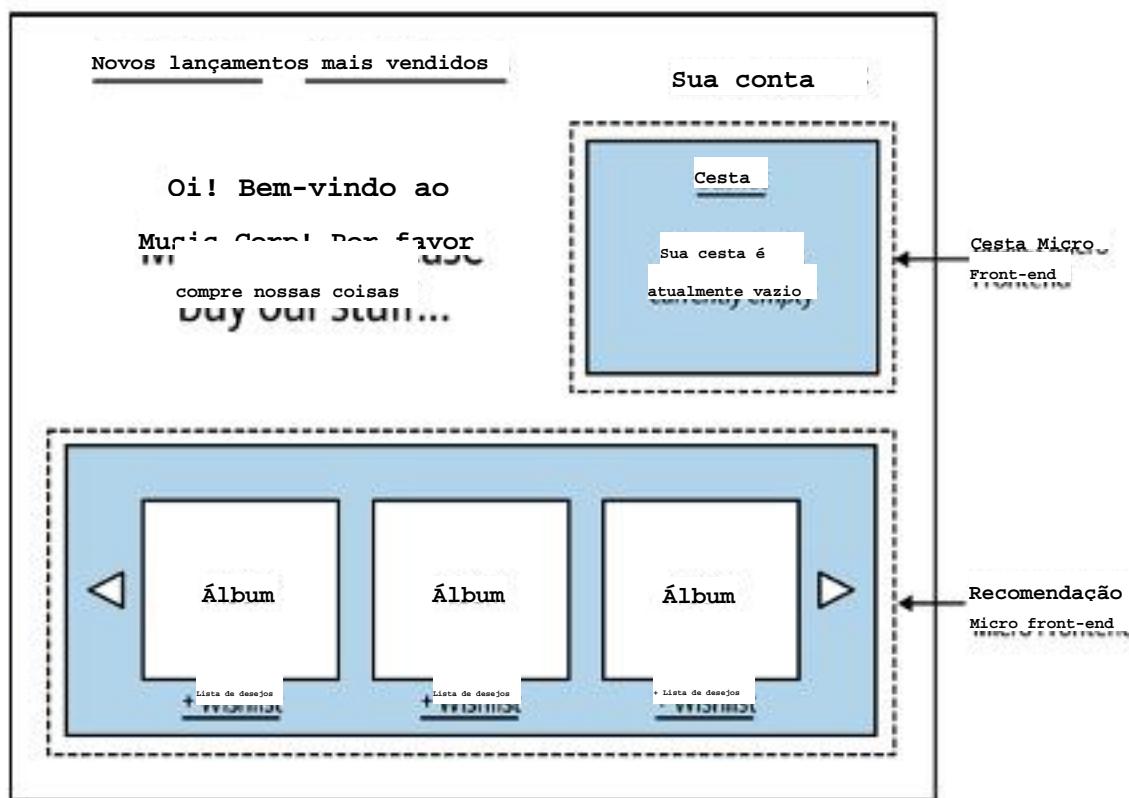


Figura 14-5. Widgets de cesta e recomendação usados na MusicCorp

O widget de recomendação da MusicCorp retira um carrossel de recomendações que podem ser alternadas e filtradas. Como vemos em Figura 14-6, quando o usuário interage com o widget de recomendação - passando para o próximo conjunto de recomendações, por exemplo, ou adicionando itens em sua lista de desejos - isso pode resultar na realização de chamadas para o suporte

microsserviços, talvez neste caso as Recomendações e a Lista de Desejos _
microsserviços. Isso poderia se alinhar bem com uma equipe que possui os dois
apoianto microsserviços e o próprio componente.

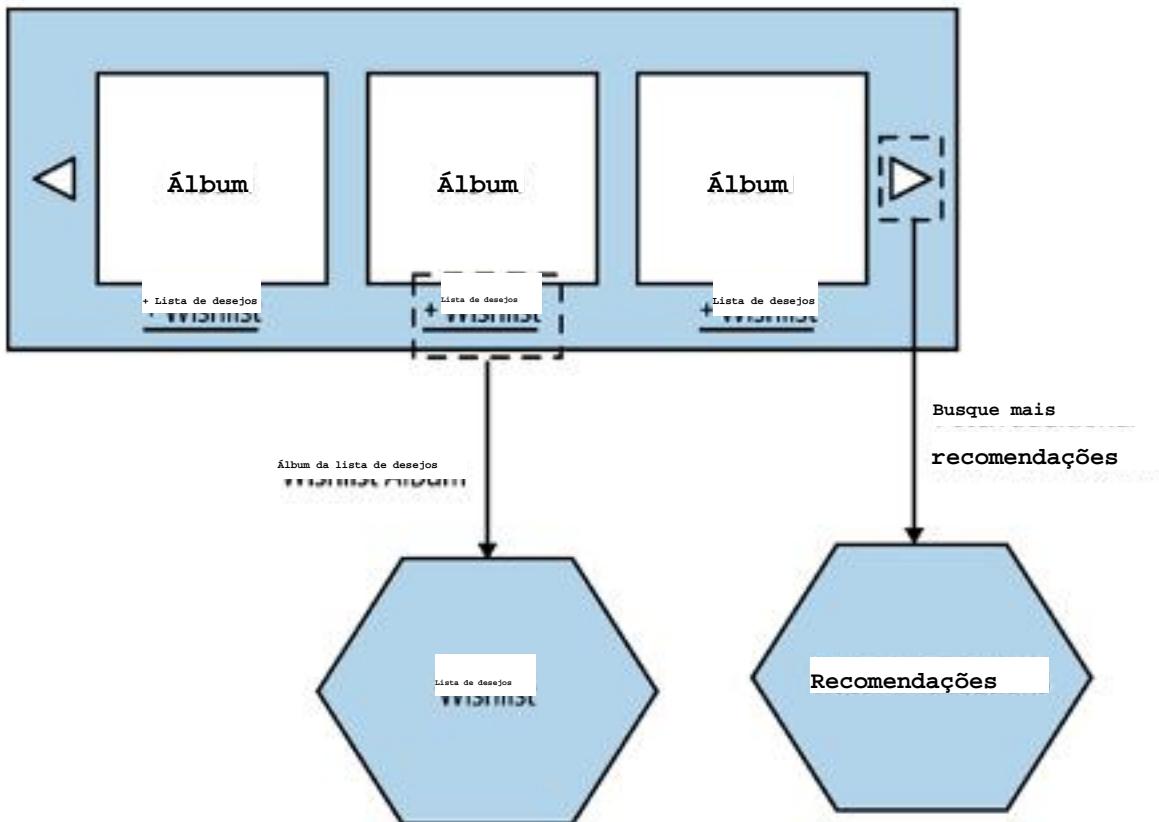


Figura 14-6. Interações entre o microfrontend de recomendação e o suporte
microsserviços

De um modo geral, você precisará de um aplicativo de "contêiner" que defina as coisas como a navegação principal da interface e quais widgets precisam ser incluído. Se estivéssemos pensando em termos de equipes orientadas ao fluxo de ponta a ponta, poderíamos imaginar uma única equipe fornecendo o widget de recomendação e também sendo responsável por um microsserviço de apoio ao Recommendations.

Esse padrão é muito encontrado no mundo real. A interface de usuário do Spotify, para exemplo, faz uso intenso desse padrão. Um widget pode conter uma playlist, outro pode conter informações sobre um artista e um terceiro widget pode conter informações sobre os artistas e outros usuários do Spotify que você segue. Esses widgets são combinados de maneiras diferentes em diferentes situações.

Você ainda precisa de algum tipo de camada de montagem para unir essas peças. Isso pode ser tão simples quanto usar modelos do lado do servidor ou do lado do cliente, embora...

Implantação

Como unir um widget em sua interface de usuário dependerá muito de como ela é criado. Com um site simples, incluindo widgets como fragmentos de HTML usando a modelagem do lado do cliente ou do lado do servidor pode ser bastante simples, embora você possa ter problemas se os widgets tiverem um comportamento mais complexo. Por exemplo, se nosso widget de recomendação contiver muito JavaScript funcionalidade, como garantiríamos que isso não colida com o comportamento carregado no resto da página da web? Idealmente, todo o widget poderia ser empacotado de forma que não quebrasse outros aspectos da interface do usuário.

A questão de como fornecer funcionalidade independente em uma interface de usuário sem quebrar outras funcionalidades tem sido historicamente especialmente problemático com aplicativos de página única, em parte porque o conceito de modularização não parece ter sido a principal preocupação de como o SPA de apoio estruturas foram criadas. Vale a pena explorar esses desafios em mais detalhes.

profundidade.

Dependências

Embora os iFrames tenham sido uma técnica muito usada no passado, tendemos a evite usá-los para unir diferentes widgets em uma única página da web. iFrames têm uma série de desafios em relação ao dimensionamento e em termos de dificultá-lo para se comunicar entre as diferentes partes do frontend. Em vez disso, widgets normalmente são emendados na interface do usuário usando modelos do lado do servidor ou então inserido dinamicamente no navegador no lado do cliente. Em ambos os casos, o desafio é que o widget está sendo executado na mesma página do navegador com outras partes do frontend, o que significa que você precisa ter cuidado com as diferentes os widgets não entram em conflito uns com os outros.

Por exemplo, nosso widget de recomendação pode usar o React v16, enquanto o widget da cesta ainda está usando o React v15. Isso pode ser uma bênção, de

claro, pois pode nos ajudar a experimentar tecnologias diferentes (poderíamos usar diferentes Estruturas de SPA para diferentes widgets), mas também pode ajudar quando se trata de atualizando as versões dos frameworks que estão sendo usados. Eu falei com vários equipes que tiveram desafios para se deslocar entre Angular ou React versões, em grande parte devido às diferenças nas convenções usadas nas mais recentes versões do framework. Atualizar uma interface monolítica inteira pode ser assustador, mas se você puder fazer isso de forma incremental, atualizando partes de sua peça de front-end por peça, você pode interromper o trabalho e também mitigar o risco da atualização introduzindo novos problemas.

A desvantagem é que você pode acabar com muita duplicação entre dependências, o que, por sua vez, pode levar a um grande inchaço em termos de carregamento da página tamanho. Eu poderia acabar incluindo várias versões diferentes do React estrutura e suas dependências transitivas associadas, por exemplo. É Não é de surpreender que muitos sites agora tenham um tamanho de carregamento de página várias vezes maior que o tamanho de alguns sistemas operacionais. Como um rápido estudo não científico, Eu verifiquei o carregamento da página do site da CNN no momento em que este artigo foi escrito, e foi 7,9 MB, que é muito maior do que os 5 MB do Alpine Linux para exemplo. 7,9 MB estão, na verdade, na extremidade menor de alguns dos tamanhos de carregamento da página I ver.

Comunicação entre widgets na página

Embora nossos widgets possam ser criados e implantados de forma independente, ainda queremos para que eles possam interagir uns com os outros. Como exemplo da MusicCorp, quando um usuário seleciona um dos álbuns na parada mais vendida, queremos outras partes da interface do usuário a serem atualizadas com base na seleção, conforme mostrado na Figura 14-7.

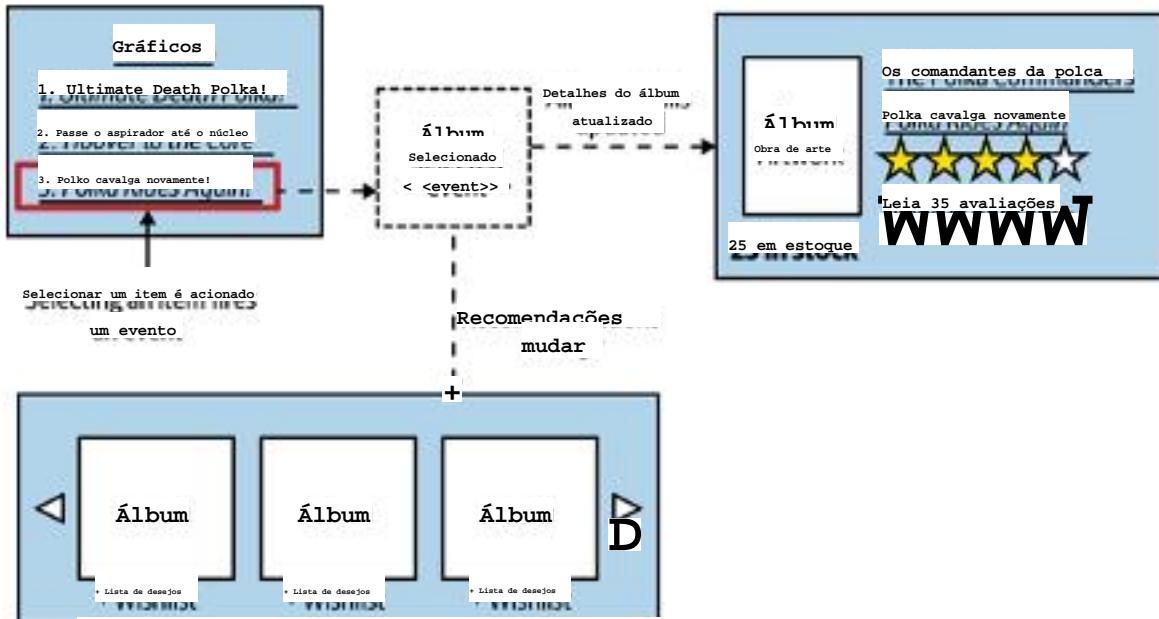


Figura 14-7. O widget de gráficos pode emitir eventos que outras partes da interface do usuário ouvem

A maneira de conseguirmos isso seria fazer com que o widget de gráficos emitisse uma mensagem personalizada evento. Os navegadores já suportam vários eventos padrão que podemos usar para acionar o comportamento. Esses eventos nos permitem reagir ao pressionamento de botões, o mouse sendo rolado e coisas do gênero, e você provavelmente já fez uso intenso desse tratamento de eventos se você passou algum tempo construindo JavaScript front-ends. É uma etapa simples para criar seus próprios eventos personalizados.

Então, no nosso caso, quando um item é selecionado no gráfico, esse widget gera um evento personalizado `Album Selected`. A recomendação e os detalhes do álbum os widgets se inscrevem no evento e reagem de acordo, com o atualização de recomendações com base na seleção e nos detalhes do álbum sendo carregado. Essa interação, é claro, já deve ser familiar para nós, pois imita a interação orientada por eventos entre microsserviços que nós discutido em "Padrão: comunicação orientada por eventos". O único real A diferença é que essas interações de eventos acontecem dentro do navegador.

COMPONENTES DA WEB

À primeira vista, o Web Component Standard deve ser uma forma óbvia para implementar esses widgets. O Web Component Standard descreve como você pode criar um componente de interface do usuário que pode colocar em sandbox seu HTML, CSS, e aspectos de JavaScript. Infelizmente, o padrão de componentes da web parece que levou muito tempo para se estabelecer e mais tempo para ser suportado adequadamente pelos navegadores. Grande parte do trabalho inicial em torno deles parece ter parado, o que obviamente afetou a adoção. Eu ainda tenho para conhecer uma organização que usa componentes da web fornecidos por microserviços, por exemplo.

Dado que o Web Component Standard agora é bastante bem suportado, seu é possível que os vejamos emergir como uma forma comum de implementar widgets em sandbox ou micro front-ends maiores no futuro, mas depois de anos De esperar que isso aconteça, não estou prendendo a respiração.

Quando usá-lo

Esse padrão facilita a contribuição de várias equipes alinhadas ao fluxo para a mesma interface do usuário. Ele permite mais flexibilidade do que o baseado em páginas decomposição, já que os widgets fornecidos por diferentes equipes podem coexistir em a interface do usuário ao mesmo tempo. Também cria a oportunidade de permitir que as equipes forneça widgets reutilizáveis que podem ser usados por equipes alinhadas ao fluxo - um exemplo do qual compartilhei anteriormente ao mencionar o papel do setor financeiro Equipe Times Origami.

O padrão de decomposição do widget é incrivelmente útil se você estiver construindo um rica interface de usuário baseada na web, e eu sugiro fortemente o uso de widgets em qualquer situação em que você esteja usando uma estrutura de SPA e querem dividir as responsabilidades do front-end, avançando em direção a um abordagem de micro front-end. As técnicas e a tecnologia de suporte disponíveis neste conceito melhorou significativamente nos últimos anos, na medida em que, ao criar uma interface web baseada em SPA, dividindo minha interface do usuário em micro frontends seriam minha abordagem padrão.

Minhas principais preocupações com a decomposição de widgets no contexto de SPAs têm relacionado ao trabalho necessário para configurar o agrupamento separado de componentes e os problemas relacionados ao tamanho da carga útil. O primeiro problema provavelmente é um custo único e envolve apenas descobrir qual estilo de embalagem melhor se adapta ao seu existente cadeia de ferramentas. A última questão é mais problemática. Uma pequena mudança simples no dependências de um widget podem resultar em uma série de novas dependências sendo incluídas no aplicativo, inflando drasticamente a página tamanho. Se você estiver criando uma interface de usuário em que o peso da página seja uma preocupação, eu sugiro colocar algumas verificações automatizadas para alertá-lo sobre a página se o peso ultrapassa um certo limite aceitável.

Por outro lado, se os widgets forem de natureza mais simples e em grande parte estáticos componentes, a capacidade de incluí-los usando algo tão simples quanto o cliente. A modelagem lateral ou do lado do servidor é muito simples em comparação.

Restrições

Antes de prosseguir com a discussão de nosso próximo padrão, quero abordar o tópico de restrições. Cada vez mais, os usuários do nosso software interagem com ele a partir de uma variedade de dispositivos diferentes. Cada um desses dispositivos impõe diferentes restrições que nosso software deve acomodar em uma web para desktop aplicativo, por exemplo, consideramos restrições como qual navegador os visitantes estão usando, ou sua resolução de tela. Pessoas com deficiência visual podem fazer uso do nosso software por meio de leitores de tela e pessoas com restrições pode ser mais provável que a mobilidade use entradas no estilo de teclado para navegar na tela.

Portanto, embora nossos principais serviços - nossa oferta principal - possam ser os mesmos, nós precisam de uma maneira de adaptá-los às diferentes restrições que existem para cada tipo da interface e para as diferentes necessidades de nossos usuários. Isso pode ser conduzido puramente do ponto de vista financeiro, se você quiser clientes mais satisfeitos significa mais dinheiro. Mas também há uma consideração humana e ética que chega a o primeiro: quando ignoramos clientes que têm necessidades específicas, nós os negamos a chance de usar nossos serviços. Em alguns contextos, impossibilitando o fato de as pessoas navearem em uma interface de usuário devido a decisões de design levou a ações legais e

multas - por exemplo, o Reino Unido, com vários outros países, tem razão de legislação em vigor para garantir o acesso a sites para pessoas com deficiências.⁵

Os dispositivos móveis trouxeram uma série de novas restrições. Do jeito que nossos aplicativos móveis que se comunicam com um servidor podem ter um impacto. Não é quase todas as preocupações com a largura de banda, onde as limitações da mobilidade as redes podem desempenhar um papel. Diferentes tipos de interações podem esgotar a vida útil da bateria, causando alguns clientes irritados.

A natureza das interações também muda de acordo com o dispositivo. Eu não posso facilmente clique com o botão direito em um tablet. Em um telefone celular, talvez eu queira criar minha interface para uso principalmente com uma mão, com a maioria das operações sendo controladas por um polegar. Em outros lugares, posso permitir que as pessoas interajam com os serviços via SMS em lugares onde a largura de banda é premium - o uso de SMS como interface é enorme no sul global, por exemplo.

Uma discussão mais ampla sobre a acessibilidade das interfaces de usuário está fora do escopo deste livro, mas podemos pelo menos explorar os desafios específicos causados por diferentes tipos de clientes, como dispositivos móveis. Uma solução comum para lidar com as diferentes necessidades dos dispositivos clientes é realizar algum tipo de filtragem e agregação de chamadas no lado do cliente. Dados que não são necessários podem ser removido e não precisa ser enviado para o dispositivo, e várias chamadas podem ser combinado em uma única chamada.

A seguir, veremos dois padrões que podem ser úteis nesse espaço: o central agregando o gateway e o back-end para o padrão de front-end. Também veremos como o GraphQL está sendo usado para ajudar a personalizar respostas para diferentes tipos de interfaces.

Padrão: Gateway de agregação central

Um gateway de agregação de propósito central fica entre interfaces de usuário externas e microserviços downstream e executa filtragem e agregação de chamadas para todas as interfaces de usuário. Sem agregação, uma interface de usuário pode precisar

faça várias chamadas para buscar as informações necessárias, muitas vezes jogando fora os dados que foi recuperado, mas não foi necessário.

Na Figura 14-8, vemos essa situação. Queremos exibir uma tela com informações sobre pedidos recentes de um cliente. A tela precisa ser exibida, algumas informações gerais sobre um cliente e, em seguida, liste algumas de suas pedidos em ordem de data, junto com informações resumidas, mostrando a data e status de cada pedido, bem como o preço.

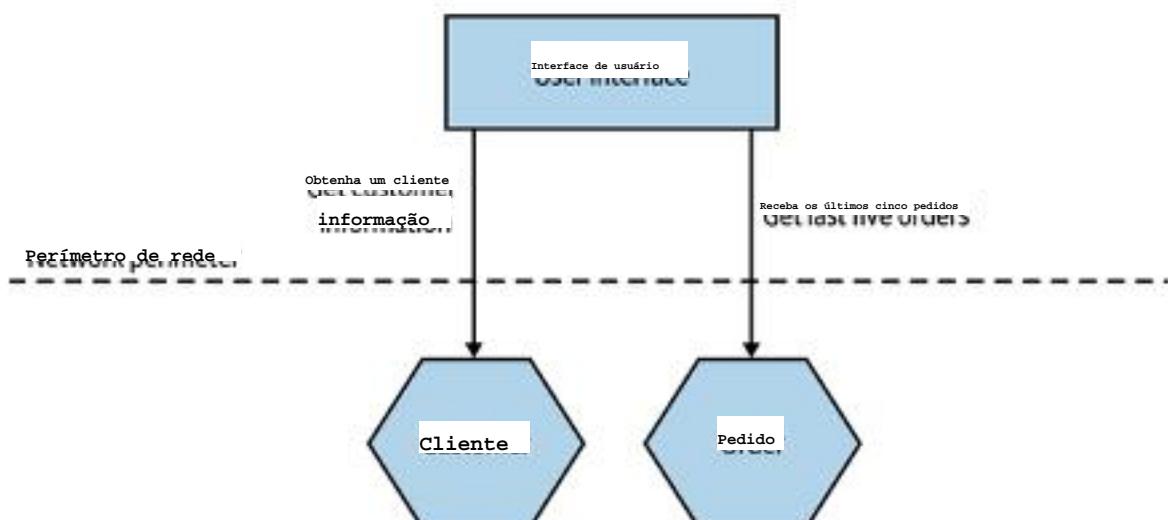


Figura 14-8 Fazendo várias chamadas para buscar informações para uma única tela.

Fazemos uma ligação direta para o microsserviço do Cliente, retirando a informação completa informações sobre o cliente, mesmo que precisemos de apenas alguns campos. Nós em seguida, busque os detalhes do pedido no microsserviço Pedido. Nós poderíamos melhorar um pouco a situação, talvez alterando o Cliente ou o Pedido microsserviço para retornar dados que melhor atendam aos nossos requisitos neste campo específico caso, mas isso ainda exigiria que duas chamadas fossem feitas.

Com um gateway agregador, podemos, em vez disso, emitir uma única chamada do usuário interface para o gateway. O gateway de agregação então executa todas as chamadas necessárias, combina os resultados em uma única resposta e joga fora quaisquer dados que a interface do usuário não exija (Figura 14-9).

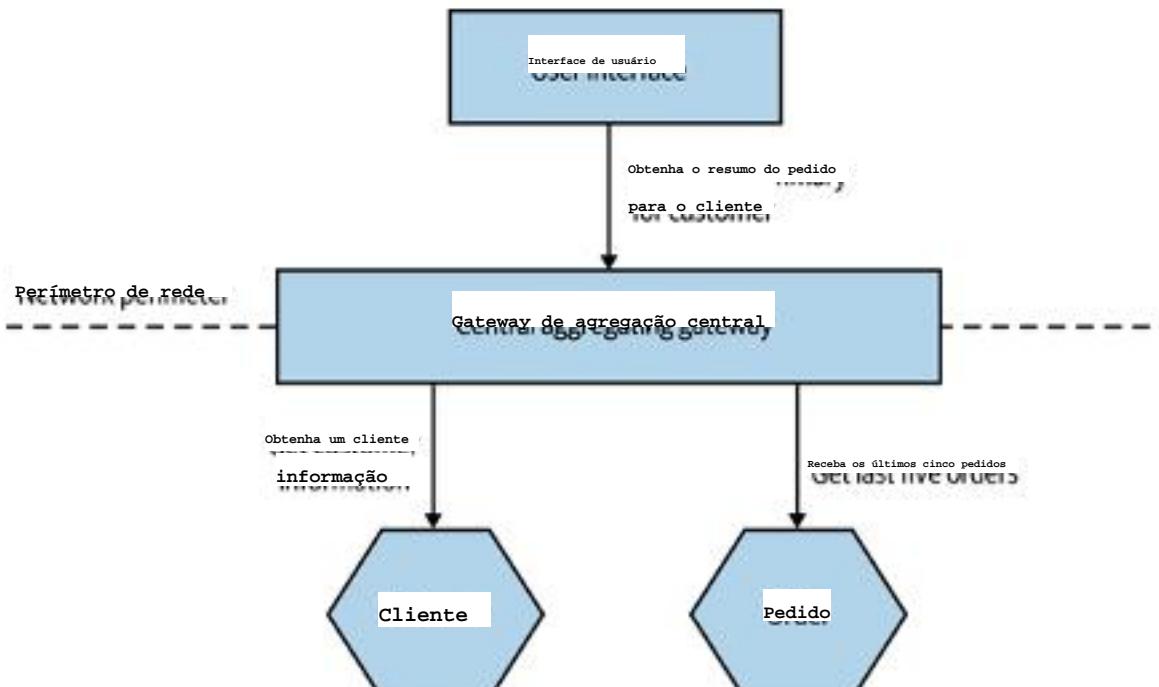


Figura 14-9 O gateway central do lado do servidor gerencia a filtragem e a agregação de chamadas para microserviços downstream

Esse gateway também pode ajudar nas chamadas em lote. Por exemplo, em vez de precisando procurar 10 IDs de pedidos por meio de chamadas separadas, eu poderia enviar um lote solicitação para o gateway de agregação, e ele poderia lidar com o resto.

Fundamentalmente, ter algum tipo de gateway agregador pode reduzir o número de chamadas que o cliente externo precisa fazer e reduzir a quantidade de dados que precisam ser enviados de volta. Isso pode levar a benefícios significativos em termos de redução do uso da largura de banda e melhoria da latência da aplicação.

Propriedade

Quanto mais interfaces de usuário usarem o gateway central e mais os microserviços precisam de agregação de chamadas e lógica de filtragem para esses usuários interfaces, o gateway se torna uma fonte potencial de contenção. Quem possui a porta de entrada? É de propriedade das pessoas que criam as interfaces de usuário ou da pessoas que possuem os microserviços? Muitas vezes eu acho que a agregação central gateway faz tanto que acaba sendo de propriedade de uma equipe dedicada. Olá, arquitetura em camadas isoladas!

Fundamentalmente, a natureza da agregação e filtragem de chamadas é em grande parte impulsionado pelos requisitos das interfaces de usuário externas. Como tal, seria faz sentido natural que o gateway seja de propriedade da (s) equipe (s) que criou o UI. Infelizmente, especialmente em uma organização na qual você tem um equipe de front-end dedicada, essa equipe pode não ter as habilidades para criar tal componente vital de back-end.

Independentemente de quem acabe possuindo o gateway central, ele tem o potencial de torne-se um gargalo para a entrega. Se várias equipes precisarem fazer alterações no o gateway, o desenvolvimento nele exigirá coordenação entre aqueles equipes, desacelerando as coisas. Se uma equipe a possuir, essa equipe pode se tornar uma gargalo quando se trata de entrega. Veremos como funciona o back-end para front-end o padrão pode ajudar a resolver esses problemas em breve.

Diferentes tipos de interfaces de usuário

Se os desafios relacionados à propriedade puderem ser gerenciados, uma agregação central o gateway ainda pode funcionar bem, até considerarmos a questão de diferentes dispositivos e suas diferentes necessidades. Como discutimos, as possibilidades de um dispositivos móveis são muito diferentes. Temos menos espaço na tela, o que significa que podemos exibir menos dados. Abrindo muitas conexões com o lado do servidor os recursos podem esgotar a vida útil da bateria e os planos de dados limitados. Além disso, o a natureza das interações que queremos fornecer em um dispositivo móvel pode ser diferente drasticamente. Pense em um típico varejista físico. Em um aplicativo para desktop, eu pode permitir que você veja os itens à venda e os encomende on-line ou reserve eles em uma loja. No entanto, no dispositivo móvel, talvez eu queira permitir que você escaneie códigos de barras para fazer comparações de preços ou oferecer ofertas baseadas em contexto enquanto estiver na loja. À medida que criamos mais e mais aplicativos móveis, percebem que as pessoas os usam de forma muito diferente e, portanto, o a funcionalidade que precisamos expor também será diferente.

Então, na prática, nossos dispositivos móveis vão querer fazer diferentes e menos liga e desejará exibir dados diferentes (e provavelmente menos), do que seus contrapartes de desktop. Isso significa que precisamos adicionar funcionalidades adicionais ao nosso back-end de API para oferecer suporte a diferentes tipos de interfaces de usuário. Na Figura 14-10, vemos a interface web e a interface móvel da MusicCorp usando o

mesmo gateway para a tela de resumo do cliente, mas cada cliente quer um conjunto diferente de informações. A interface da web deseja obter mais informações sobre o cliente e também tem um breve resumo dos itens em cada pedido. Isso nos leva a implementar duas chamadas diferentes de agregação e filtragem em nosso gateway de back-end.

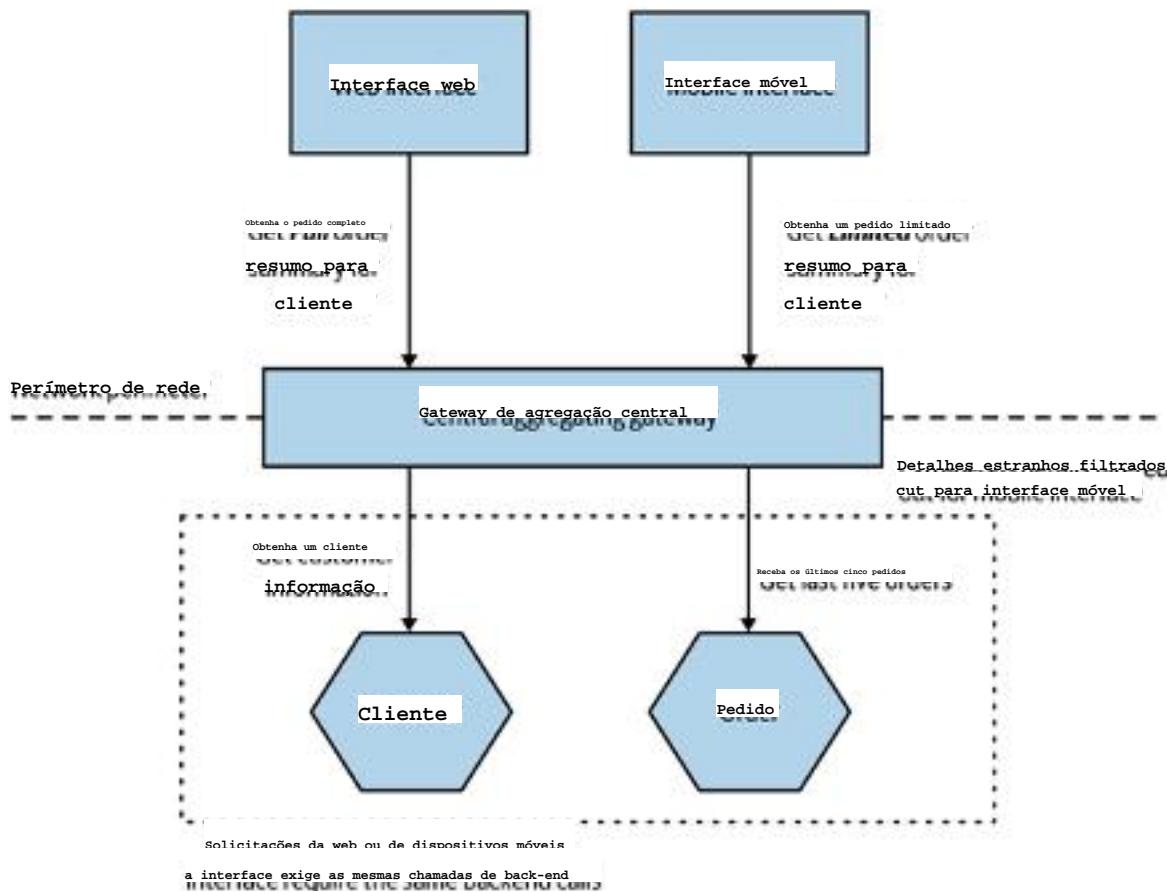


Figura 14-10. Suporte a diferentes chamadas agregadas para diferentes dispositivos

Isso pode levar a um grande inchaço no gateway, especialmente se considerarmos diferentes aplicativos móveis nativos, sites voltados para o cliente, internas, interfaces de administração e similares. Nós também temos o problema, é claro, que, embora essas diferentes interfaces de usuário possam pertencer a equipes diferentes, o gateway é uma única unidade - temos os mesmos velhos problemas de várias equipes precisarem trabalhar na mesma unidade implantada. Nosso único back-end de agregação pode se tornar um gargalo, pois muitas mudanças estão tentando ser feitas no mesmo artefato implantável.

Várias preocupações

Há uma série de preocupações que podem precisar ser abordadas no lado do servidor quando se trata de lidar com chamadas de API. Além da ligação, agregação e filtragem, podemos pensar em questões mais genéricas, como chave de API, gerenciamento, autenticação de usuários ou roteamento de chamadas. Muitas vezes, esses genéricos as preocupações podem ser tratadas por produtos de gateway de API, que estão disponíveis em muitos tamanhos e por muitas faixas de preço diferentes (algumas das quais são óculares) muito alto!). Dependendo da sofisticação que você precisa, ele pode criar um faz muito sentido comprar um produto (ou licenciar um serviço) para lidar com alguns dos essas preocupações para você. Você realmente deseja gerenciar a emissão de chaves de API, rastreamento, limitação de taxas e assim por diante? De qualquer forma, veja os produtos em este espaço para resolver essas preocupações genéricas, mas tenha cuidado ao tentar também use esses produtos para fazer sua agregação e filtragem de chamadas, mesmo que eles afirmam que podem.

Ao personalizar um produto criado por outra pessoa, muitas vezes você precisa trabalhar em seu mundo. Seu conjunto de ferramentas é restrito porque talvez você não consiga usar sua linguagem de programação e suas práticas de desenvolvimento. Em vez de escrevendo código Java, você está configurando regras de roteamento em algum produto estranho DSL específico (provavelmente usando JSON). Pode ser uma experiência frustrante... e você está transformando parte da inteligência do seu sistema em um produto de terceiros. Isso pode reduzir sua capacidade de mudar esse comportamento posteriormente. É comum perceba que um padrão de agregação de chamadas realmente está relacionado a algum domínio funcionalidade que poderia justificar um microsserviço por si só (algo exploraremos mais em breve quando falarmos sobre melhores amigos). Se esse comportamento estiver em um configuração específica do fornecedor, mover essa funcionalidade pode ser mais problemático, pois você provavelmente teria que reinventá-lo.

A situação pode se tornar ainda pior se o gateway de agregação se tornar complexo o suficiente para exigir uma equipe dedicada para possuí-lo e gerenciá-lo. Na pior das hipóteses, adotar uma propriedade de equipe mais horizontal pode levar a uma situação na qual implemente algumas novas funcionalidades que você precisa criar por uma equipe de front-end mudanças, a equipe agregadora do gateway para fazer mudanças e a(s) equipe(s) que possui o microsserviço para também fazer suas alterações. De repente, tudo começa indo muito mais devagar.

Então, se você quiser usar um gateway de API dedicado, vá em frente, mas considere seriamente ter sua lógica de filtragem e agregação em outro lugar.

Quando usá-lo

Para uma solução de propriedade de uma única equipe, em que uma equipe desenvolve o usuário interface e microserviços de back-end, eu ficaria bem em ter um único gateway de agregação central. Dito isso, parece que essa equipe está fazendo um Muito trabalho - nessas situações, costumo ver um grande grau de conformidade em todas as interfaces de usuário, o que geralmente elimina a necessidade desses pontos de agregação em primeiro lugar.

Se você decidir adotar um único gateway de agregação central, por favor, seja tenha cuidado para limitar a funcionalidade que você coloca dentro dela. Eu seria extremamente cauteloso com incorporando essa funcionalidade em um produto de gateway de API mais genérico, para exemplo, pelos motivos descritos anteriormente.

O conceito de fazer alguma forma de filtragem e agregação de chamadas no No entanto, o back-end pode ser muito importante em termos de otimização do usuário experiência de nossas interfaces de usuário. O problema é que, em uma organização de entrega com várias equipes, um gateway central pode gerar requisitos para muitas coordenação entre essas equipes.

Então, se ainda quisermos agregar e filtrar no back-end, mas quisermos remover os problemas associados ao modelo de propriedade de uma central gateway, o que podemos fazer? É aí que está o back-end para o padrão de front-end entra.

Padrão: Backend para front-end (BFF)

A principal distinção entre um BFF e um gateway de agregação central é que um BFF é um propósito único na natureza - é desenvolvido para um usuário específico interface. Esse padrão provou ser muito bem-sucedido em ajudar a lidar com o preocupações diferentes com as interfaces de usuário, e já vi que funcionam bem em várias de organizações, incluindo SoundCloud e REA. Como vemos na Figura 14-

11, onde revisitamos a MusicCorp, a web e as interfaces de compras móveis.....
agora têm seus próprios back-ends de agregação.

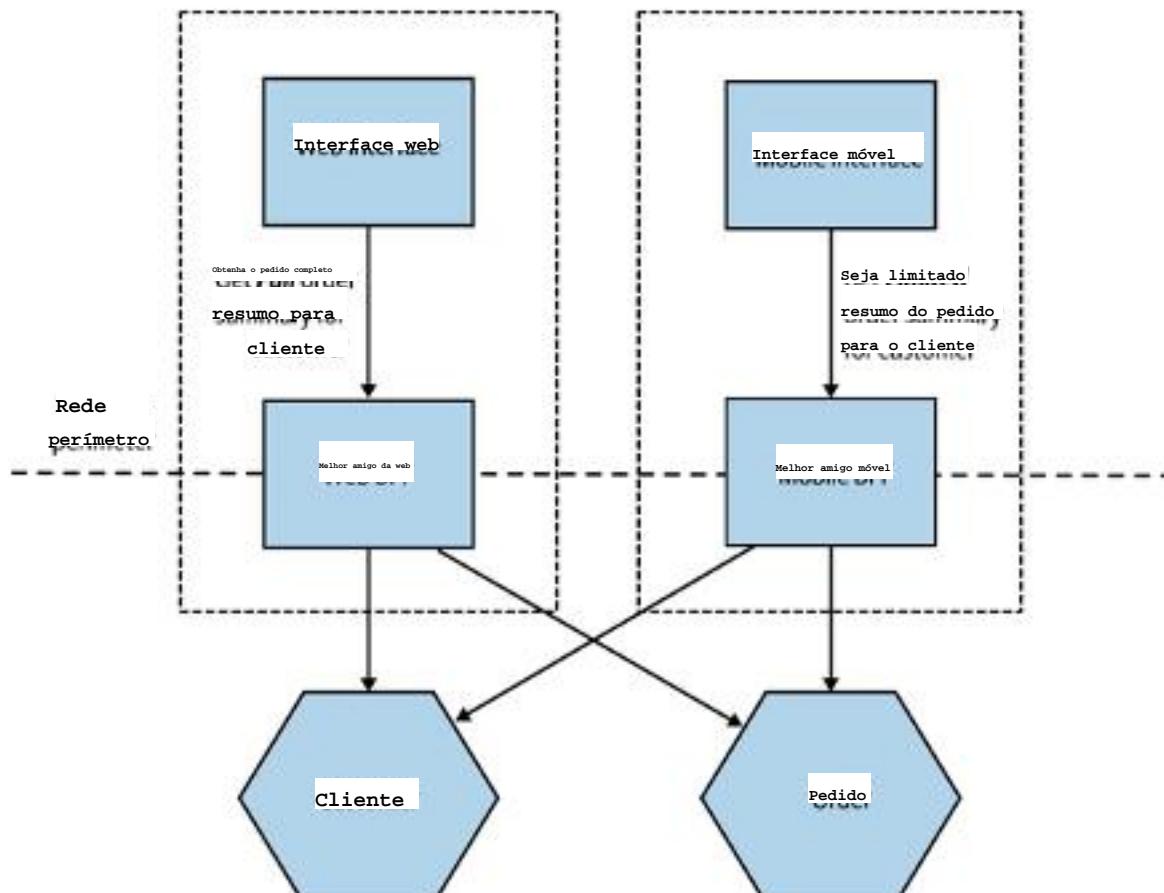


Figura 14-11. Cada interface de usuário tem seu próprio BFF

Devido à sua natureza específica, o BFF evita algumas das preocupações em torno do gateway de agregação central. Como não estamos tentando ser tudo para todos pessoas, um melhor amigo evita se tornar um gargalo para o desenvolvimento, com vários todas as equipes estão tentando compartilhar a propriedade. Também estamos menos preocupados com o acoplamento com a interface do usuário, pois o acoplamento é muito mais aceitável. Um determinado melhor amigo é para uma interface de usuário específica, supondo que eles pertençam à mesma equipe, então, o acoplamento inherent é muito mais fácil de gerenciar. Costumo descrever o uso de um BFF com uma interface de usuário, como se a interface estivesse realmente dividida em duas partes. Uma parte fica no dispositivo cliente (a interface web ou o celular nativo). aplicativo), com a segunda parte, o BFF, sendo incorporado no servidor lado.

O BFF está intimamente associado a uma experiência específica do usuário e, normalmente, ser mantido pela mesma equipe da interface do usuário, facilitando assim para definir e adaptar a API conforme a UI exige, ao mesmo tempo em que simplifica o processo de alinhamento da liberação dos componentes cliente e servidor.

Quantos melhores amigos?

Quando se trata de oferecer a mesma experiência de usuário (ou similar) no plataformas diferentes, eu vi duas abordagens diferentes. O modelo que eu prefiro (e o modelo que vejo com mais frequência) é ter estritamente um único melhor amigo para cada tipo diferente de cliente - este é um modelo que vi usado na REA, conforme descrito em

Figura 14-12. Os aplicativos para Android e iOS, ao mesmo tempo que abrangem aplicativos semelhantes funcionalidade, cada um tinha seu próprio melhor amigo.

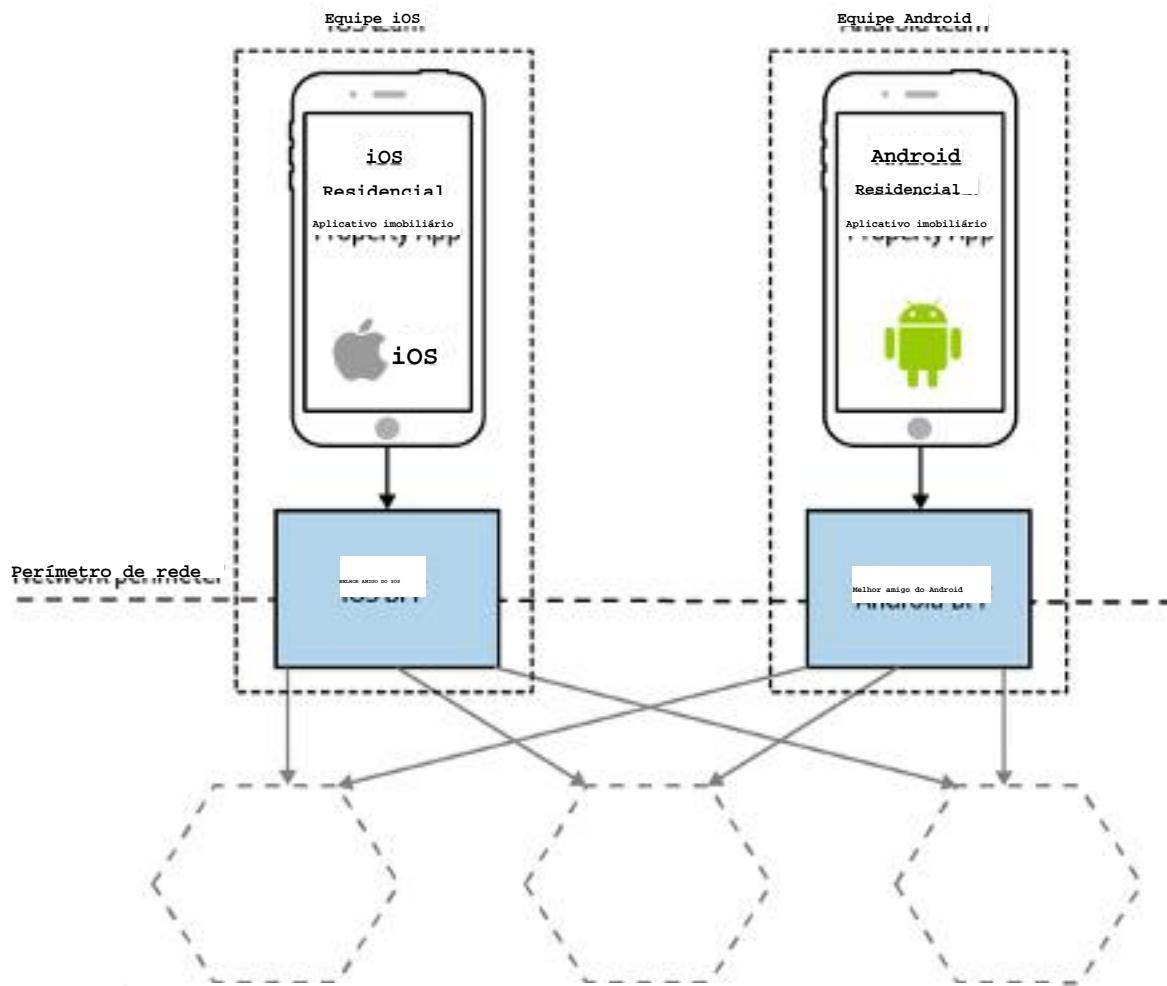


Figura 14-12. Os aplicativos REA para iOS e Android têm diferentes melhores amigos

Uma variação é procurar oportunidades de usar o mesmo melhor amigo por mais de um tipo de cliente, embora para o mesmo tipo de interface de usuário. SoundClouds O aplicativo Listener permite que as pessoas ouçam conteúdo em seu Android ou Dispositivos iOS. O SoundCloud usa um único melhor amigo para Android e iOS, como mostrado na Figura 14-13.

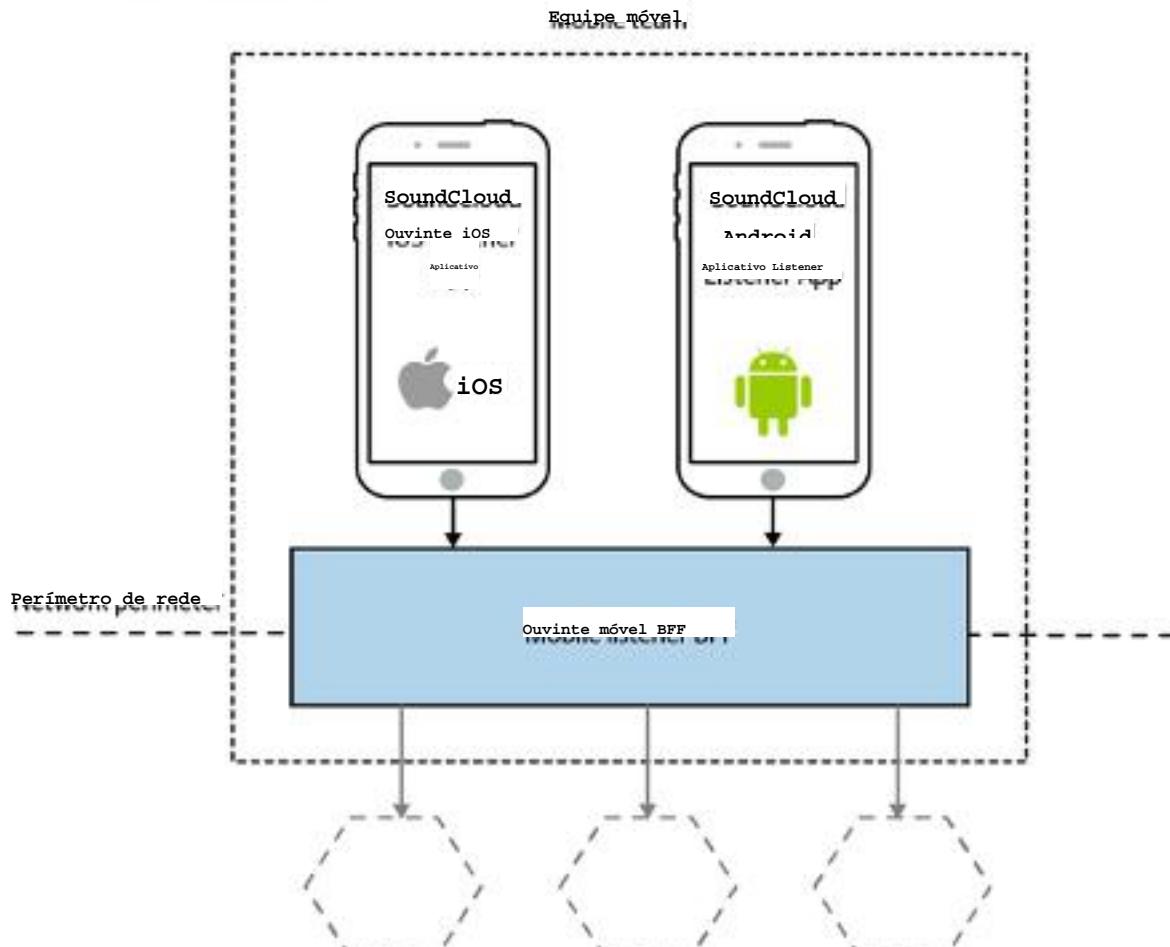


Figura 14-13. SoundCloud compartilhando um melhor amigo entre aplicativos iOS e Android

Minha principal preocupação com esse segundo modelo é que quanto mais tipos de clientes você se estiver usando uma única melhor amiga, maior será a tentação de a melhor amiga de ficar inchado ao lidar com várias preocupações. A principal coisa a entender aqui, porém, é que mesmo quando os clientes compartilham uma melhor amiga, ela é para a mesma classe de interface de usuário - então, enquanto os aplicativos Listener nativos do SoundCloud para iOS e o Android usam o mesmo melhor amigo, outros aplicativos nativos usariam diferentes BFFs. Estou mais tranquilo em usar esse modelo se a mesma equipe possuir os dois aplicativos Android e iOS e também possuir o BFF - se estes

os aplicativos são mantidos por equipes diferentes, estou mais inclinado a recomendar o modelo mais rigoroso. Para que você possa ver sua organização estrutura como sendo um dos principais impulsionadores para decidir qual modelo faz o mais sensato (a lei de Conway vence novamente).

Stewart Gleadow, da REA, sugeriu a diretriz "uma experiência, um melhor amigo." Então, se as experiências de iOS e Android são muito semelhantes, então é mais fácil justificar ter um único melhor amigo. Se, no entanto, eles divergirem muito, então ter melhores amigos separados faz mais sentido. No caso da REA, embora haja houve uma sobreposição entre as duas experiências, equipes diferentes as possuíam e lançou recursos semelhantes de maneiras diferentes. Às vezes, o mesmo recurso pode ser implantado de forma diferente em um dispositivo móvel diferente - o que equivale a uma experiência nativa para um aplicativo Android pode precisar ser reformulada para sinta-se nativo no iOS.

Outra lição da história da REA (e que abordamos várias vezes) já é que o software geralmente funciona melhor quando alinhado com a equipe. Limites e melhores amigos não são exceção. SoundCloud com um único celular. A equipe faz com que ter um único melhor amigo pareça sensato à primeira vista, assim como o REA ter dois melhores amigos diferentes para as duas equipes separadas. Vale a pena notar que os engenheiros do SoundCloud com quem falei sugeriram que ter um melhor amigo para o aplicativos de ouvintes para Android e iOS eram algo que eles poderiam reconsiderar - eles tinham uma única equipe móvel, mas na realidade eram uma mistura de Android e especialistas em iOS, e eles se viram trabalhando principalmente em um dos outras aplicações, o que implica que na verdade eram duas equipes.

Freqüentemente, o fator que leva a ter um número menor de melhores amigos é o desejo de reutilizar funcionalidade do lado do servidor para evitar muita duplicação, mas existem outras maneiras de lidar com isso, que abordaremos a seguir.

Reutilização e melhores amigos

Uma das preocupações de ter um único melhor amigo por interface de usuário é que você pode acabar com muita duplicação entre os melhores amigos. Por exemplo, eles podem acabar executando os mesmos tipos de agregação, ter o mesmo código ou similar para interface com serviços downstream e assim por diante. Se você está procurando

extraia uma funcionalidade comum e, muitas vezes, um dos desafios é encontrá-la. Essa duplicação pode ocorrer nos próprios BFFs, mas também pode acabar sendo incorporado aos diferentes clientes. Devido ao fato de que esses clientes usam pilhas de tecnologia muito diferentes, identificando o fato de que essa duplicação é ocorrer pode ser difícil. Com organizações que tendem a ter um comum pilha de tecnologia para componentes do lado do servidor, com vários melhores amigos com a duplicação pode ser mais fácil de identificar e considerar.

Algumas pessoas reagem a isso querendo mesclar os melhores amigos novamente, e eles acabam com um gateway agregador de uso geral. Minha preocupação com regredir para um único gateway de agregação é que podemos acabar perdendo mais do que ganhamos, especialmente porque pode haver outras maneiras de abordar isso duplicação.

Como eu disse antes, estou bastante relaxado com relação ao código duplicado em microserviços. Ou seja, enquanto estiver em um único limite de microserviços. Normalmente, farei o possível para refatorar a duplicação para torná-la adequada abstrações, não tenho a mesma reação quando confrontado com a duplicação em microserviços. Isso ocorre principalmente porque muitas vezes estou mais preocupado com o potencial de extração de código compartilhado para levar a um acoplamento estreito entre serviços (um tópico que exploramos em "DRY and the Perils of Code"). Reutilização em um mundo de microserviços". Dito isso, certamente há casos em que isso é garantido.

Quando chegar a hora de extraír código comum para permitir a reutilização entre BFFs, existem duas opções óbvias. O primeiro, que geralmente é mais barato, mas mais complicado, é extraír algum tipo de biblioteca compartilhada. A razão pela qual isso pode ser problemático é que as bibliotecas compartilhadas são a principal fonte de acoplamento, especialmente quando usado para gerar bibliotecas de clientes para chamadas downstream serviços. No entanto, há situações em que isso parece certo-especialmente quando o código que está sendo abstraido é puramente uma preocupação dentro do serviço.

A outra opção é extraír a funcionalidade compartilhada em uma nova microserviço. Isso pode funcionar bem se a funcionalidade estiver sendo extraída representar a funcionalidade do domínio comercial. Uma variação dessa abordagem pode

seja levar as responsabilidades de agregação aos microserviços ainda mais adiante. Vamos considerar uma situação em que queremos exibir uma lista dos itens em um lista de desejos do cliente, juntamente com informações sobre se esses itens ou não estão em estoque e no preço atual, conforme mostrado na Tabela 14-1.

Tabela 14-1. Exibindo a lista de desejos de um cliente de Music Corp

The Brakes, Give Blood	Em estoque!	\$5,99
Blue Juice, retrospectivo	Fora de estoque	\$7,50
Hot Chip, por que fazer sentido? Indo rápido! (2 restantes) \$9,99		

O microserviço do cliente armazena informações sobre a lista de desejos e o ID de cada item. O microserviço Catalog armazena o nome e o preço de cada item, e os níveis de estoque são armazenados em nosso microserviço de inventário. Para exiba esse mesmo controle nos aplicativos iOS e Android, cada um A melhor amiga precisaria fazer as mesmas três chamadas para o suporte microserviços, conforme mostrado na Figura 14-14.

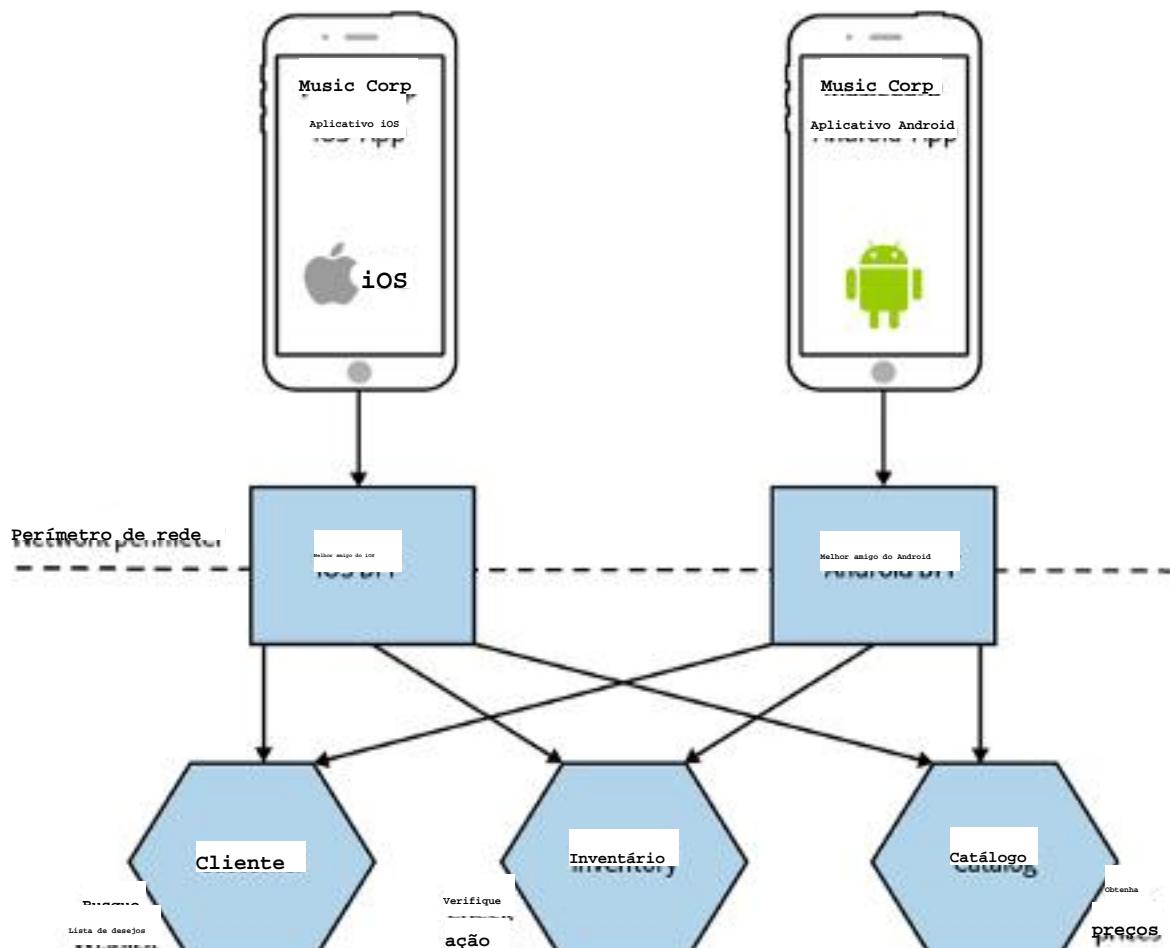


Figura 14-14 Ambos os melhores amigos estão realizando as mesmas operações para exibir uma lista de desejos

Uma forma de reduzir a duplicação na funcionalidade aqui seria extrair isso comportamento comum em um novo microsserviço. Na Figura 14-15, vemos nosso novo microsserviço de lista de desejos dedicado que nossos aplicativos Android e iOS podem ambos fazem uso de.

Devo dizer que o mesmo código usado em dois lugares não seria automaticamente faz com que eu queira extrair um serviço dessa forma, mas eu certamente estamos considerando o custo da transação de criar um novo serviço era baixo o suficiente, ou se eu estivesse usando o código em mais de alguns lugares- nesta situação específica, se também estivéssemos mostrando listas de desejos em nossa web interface, por exemplo, um microsserviço dedicado começaria a parecer uniforme mais atraente. Acho que o velho ditado de criar uma abstração quando você está prestes a implementar algo pela terceira vez ainda parece uma boa regra de polegar, mesmo no nível de serviço.

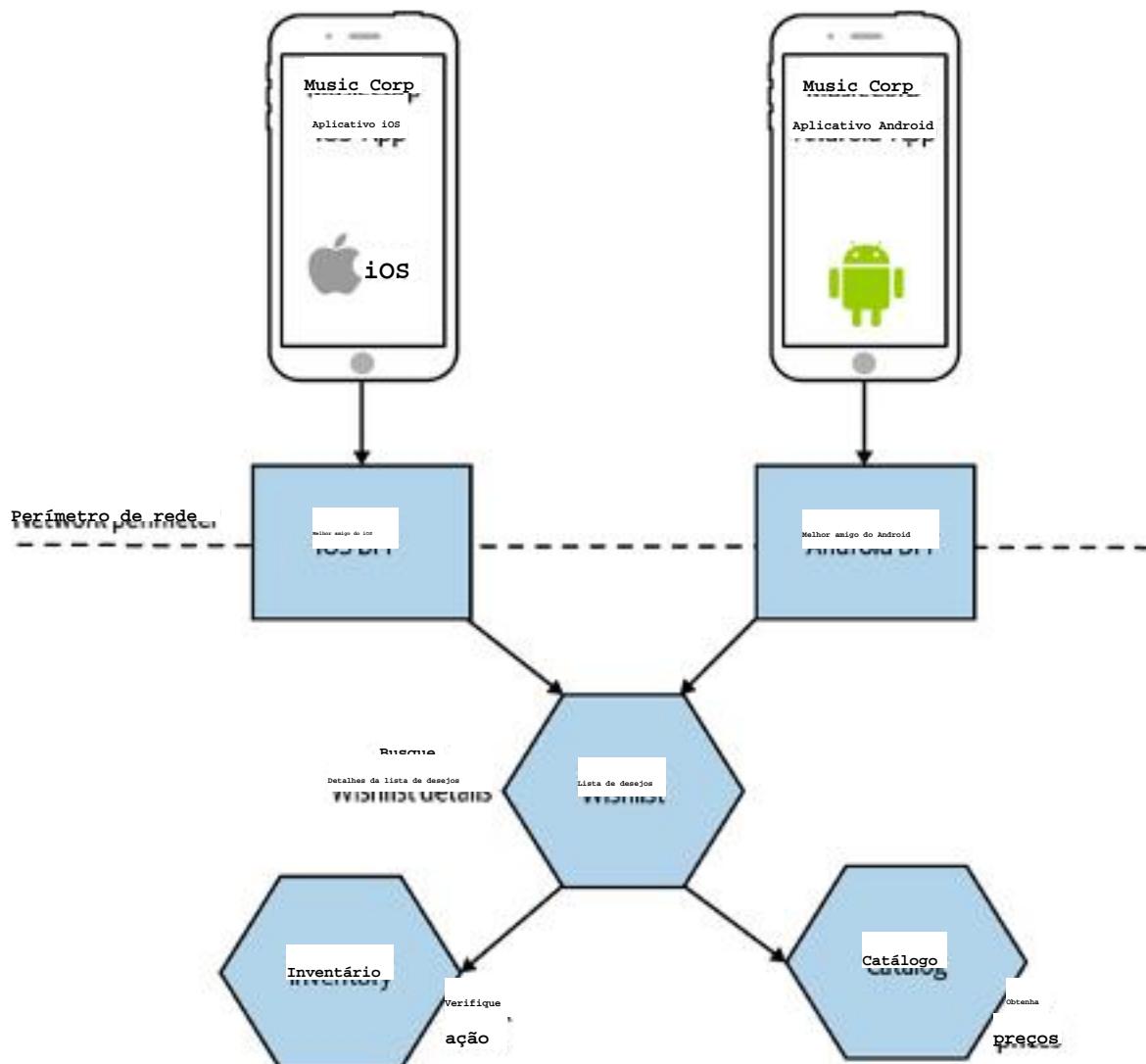


Figura 14-15. A funcionalidade comum é extraída em um microsserviço da Lista de desejos, permitindo

reutilização entre melhores amigos

Melhores amigos para desktop, web e muito mais

Você pode pensar que os melhores amigos estão apenas ajudando a resolver as restrições de dispositivos móveis. A experiência web para desktop geralmente é oferecida em mais dispositivos poderosos com melhor conectividade, onde o custo de criar vários as chamadas downstream são gerenciáveis. Isso pode permitir que seu aplicativo web faça várias chamadas diretamente para serviços downstream sem a necessidade de um MELHOR AMIGO.

No entanto, já vi situações em que o uso de um melhor amigo para a web também pode seja útil. Quando você está gerando uma parte maior da interface de usuário da web no

do lado do servidor (por exemplo, usando modelos do lado do servidor), um BFF é o lugar óbvio onde isso pode ser feito. Essa abordagem também pode simplificar um pouco o armazenamento em cache, já que você pode colocar um proxy reverso na frente do BFF, permitindo que você armazene os resultados das chamadas agregadas.

Já vi pelo menos uma organização usar BFFs para outras partes externas que preciso fazer chamadas. Voltando ao meu exemplo verene de MusicCorp, eu pode expor um melhor amigo para permitir que terceiros extraiam o pagamento de royalties informações ou para permitir o streaming para uma variedade de dispositivos set-top box, como nós veja na Figura 14-16. Esses não são os melhores amigos, pois os externos as partes não estão apresentando uma "interface de usuário", mas este é um exemplo da mesma padrão sendo usado em um contexto diferente, então achei que valia a pena compartilhar.

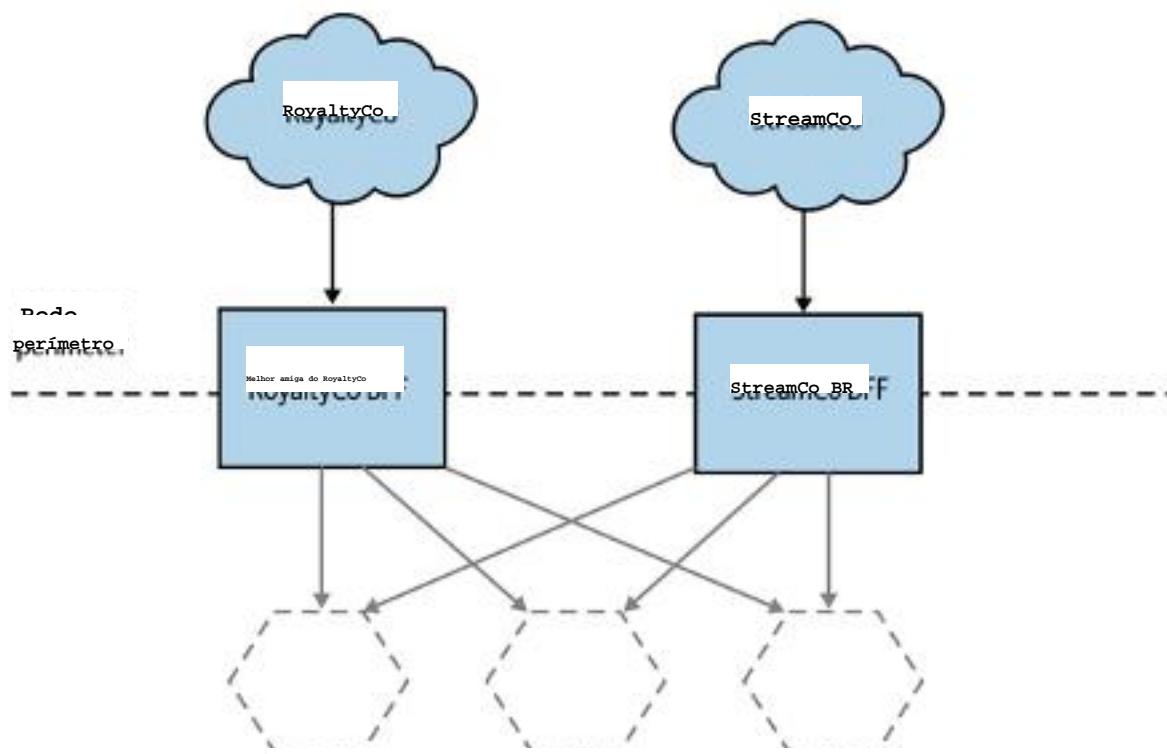


Figura 14-16 Usando BFFs para gerenciar APIs externas

Essa abordagem pode ser especialmente eficaz, pois terceiros geralmente limitam sem a capacidade (ou desejo) de usar ou alterar as chamadas de API que eles fazem. Com um back-end central da API, talvez seja necessário manter as versões antigas da API apenas para satisfazer um pequeno subconjunto de partes externas incapazes de fazer uma mudança; com um melhor amigo, esse problema é substancialmente reduzido. Também limita o impacto de mudanças significativas. Você pode alterar a API do Facebook de uma forma que

quebraria a compatibilidade com outras partes, mas como elas usam uma melhor amiga diferente, eles não são afetados por essa mudança.

Ouando usar

Para um aplicativo que fornece apenas uma interface de usuário da web, suspeito que um melhor amigo fornecerá só faz sentido se e quando você tiver uma quantidade significativa de agregação exigido no lado do servidor. Caso contrário, acho que algumas das outras interfaces as técnicas de composição que já abordamos podem funcionar da mesma forma sem exigir um componente adicional do lado do servidor.

No momento em que você precisa fornecer uma funcionalidade específica para uma interface de usuário móvel ou terceirizado, porém, eu consideraria fortemente o uso de BFFs para cada cliente da o início. Eu poderia reconsiderar se o custo da implantação de serviços adicionais é alto, mas a separação de preocupações que um melhor amigo pode trazer o torna um proposta convincente na maioria dos casos. Eu estaria ainda mais inclinado a usar um BFF se houver uma separação significativa entre as pessoas que constroem a interface do usuário e serviços downstream, por motivos que descrevi.

Em seguida, chegamos à questão de como implementar o BFF—vamos dar uma olhada o GraphQL e o papel que ele pode desempenhar.

GraphQL

O GraphQL é uma linguagem de consulta que permite que os clientes emitam consultas para acessar ou mutar dados. Assim como o SQL, o GraphQL permite que essas consultas sejam alteradas dinamicamente, permitindo que o cliente defina exatamente quais informações ele deseja de volta. Com uma chamada padrão de REST sobre HTTP, por exemplo, ao emitir um solicitação GET para um recurso de pedido, você recuperaria todos os campos desse recurso pedido. Mas e se, nessa situação específica, você quisesse apenas o total valor do pedido? Você poderia simplesmente ignorar os outros campos, é claro, ou então forneça um recurso alternativo (talvez um resumo do pedido) que contenha apenas as informações de que você precisa. Com o GraphQL, você pode emitir uma solicitação solicitando somente os campos de que você precisa, como vemos no Exemplo 14-1.

Exemplo 14-1. Um exemplo de consulta GraphQL sendo usada para buscar o pedido
informação

```
{  
  pedido 123 {  
    data  
    total  
    status  
    entrega {  
      empresa  
      driver  
      data de vencimento  
    }  
  }  
}
```

Nesta consulta, solicitamos o pedido 123 e solicitamos o preço total e o status do pedido. Fomos mais longe e pedimos informações sobre a entrega deste pedido, para que possamos obter informações sobre o nome do motorista que entregará nosso pacote, a empresa para a qual trabalha e quando espera-se que o pacote chegue. Com uma API REST normal, a menos que as informações de entrega estavam contidas no recurso do pedido, podemos ter para fazer uma chamada adicional para buscar essas informações. Então, o GraphQL não é apenas nos ajudando a pedir exatamente os campos que queremos, mas também pode reduzir a rodada viagens. Uma consulta como essa exige que definamos os vários tipos de dados que somos acessar tipos de definição explícita é uma parte fundamental do GraphQL.

Para implementar o GraphQL, precisamos de um resolvedor para lidar com as consultas. UMA O resolvedor GraphQL fica no lado do servidor e mapeia as consultas do GraphQL em chamadas para realmente buscar as informações. Então, no caso de um microsserviço arquitetura, precisaríamos de um resolvedor que fosse capaz de mapear a solicitação do pedido com ID 123 em uma chamada equivalente para um microsserviço.

Dessa forma, podemos usar o GraphQL para implementar um gateway de agregação ou até mesmo um melhor amigo. A vantagem do GraphQL é que podemos facilmente alterar o agregação e filtragem que queremos simplesmente alterando a consulta do cliente; nenhuma alteração é necessária no lado do servidor GraphQL, desde que os tipos GraphQL suportam a consulta que queremos fazer. Se não mais queria ver o nome do driver na consulta de exemplo, poderíamos simplesmente omitir isso da própria consulta e ela não seria mais enviada. Por outro lado, se nós

queria ver o número de pontos que recebemos por este pedido, supondo que essas informações estejam disponíveis no tipo de pedido, poderíamos simplesmente adicionar isso para a consulta e as informações seriam retornadas. Isso é significativo vantagem sobre implementações de BFF que exigem mudanças na agregação lógica a ser aplicada também ao próprio BFF.

A flexibilidade que o GraphQL oferece ao dispositivo cliente de forma dinâmica alterar as consultas que estão sendo feitas sem alterações no lado do servidor significa que há há menos chances de seu servidor GraphQL se tornar um servidor compartilhado e competitivo recurso, conforme discutimos com o gateway de agregação de uso geral. Dito isso, mudanças no lado do servidor ainda serão necessárias se você precisar expor novas digitas ou adiciona campos aos tipos existentes. Como tal, você ainda pode querer vários Back-ends de servidor GraphQL para alinhar os limites da equipe, então GraphQL torna-se uma forma de implementar um BFF.

Eu tenho preocupações com o GraphQL, que descrevi em detalhes no Capítulo 5. Dito isso, é uma solução bacana que permite consultas dinâmicas para atender às necessidades de diferentes tipos de interfaces de usuário.

Uma abordagem híbrida

Muitas das opções mencionadas acima não precisam ser iguais para todos. EU pude ver uma organização adotando a abordagem baseada em widgets decomposição para criar um site, mas usando uma abordagem de back-end para front-end quando se trata de seu aplicativo móvel. O ponto chave é que precisamos manter a coesão dos recursos subjacentes que oferecemos aos nossos usuários. Nós precisa garantir a lógica associada ao pedido de música ou à mudança os detalhes do cliente estão dentro dos serviços que lidam com essas operações e não fica espalhado por todo o nosso sistema. Evitando a armadilha de colocar também muito comportamento em qualquer camada intermediária é um ato de equilíbrio complicado.

Resumo

Como espero ter mostrado, a decomposição da funcionalidade não precisa parar no lado do servidor, e ter equipes de front-end dedicadas não é inevitável. Eu tenho

compartilhou várias maneiras diferentes de construir uma interface de usuário que pode fazer uso de microsserviços de suporte e, ao mesmo tempo, possibilita uma entrega focada de ponta a ponta.

Em nosso próximo capítulo, passaremos do lado técnico para o lado pessoal, quando exploramos com mais detalhes a interação de microsserviços e estruturas organizacionais.

1 Matthew Skelton e Manuel Pais, Team Topologies (Portland, OR: IT Revolution, 2019).

2 Como diz Charity Majors, "Você não é um desenvolvedor completo, a menos que construa os chips".

3 Cam Jackson, "Micro Frontends", martinfowler.com, 19 de junho de 2019, <https://oreil.ly/U3K40>.

4 Estou olhando para você, Sydney Morning Herald!

5 Espero que nem seja preciso dizer que não sou advogado, mas se você quiser examinar a legislação que cobre isso no Reino Unido, é a Lei da Igualdade de 2010, especificamente a seção 20. O W3C também tem uma boa visão geral das diretrizes de acessibilidade.

6 Veja o artigo "BFF @ SoundCloud" de Lukasz Plotnicki para uma ótima visão geral de como o SoundCloud usa o padrão BFF.

7 Stewart, por sua vez, creditou Phil Calcado e Mustafa Sezgin por essa recomendação.

Capítulo 15. Organizacional

Estruturas

Embora grande parte do livro até agora tenha se concentrado nos desafios técnicos em avançando em direção a uma arquitetura refinada, também analisamos o interação entre nossa arquitetura de microserviços e como organizamos nossa equipes. Em "Toward Stream-Aligned Teams", analisamos o conceito de equipes alinhadas ao fluxo, que têm responsabilidade de ponta a ponta pela entrega de funcionalidade voltada para o usuário e como os microserviços podem ajudar a formar essa equipe estrutura uma realidade.

Agora precisamos desenvolver essas ideias e analisar outras organizações considerações. Como veremos, se você quiser aproveitar ao máximo microserviços, você ignora o organograma da sua empresa por sua conta e risco!

Organizações fracamente acopladas

Ao longo do livro, defendi uma arquitetura pouco acoplada e argumentou que o alinhamento com um fluxo mais autônomo e fracamente acoplado equipes alinhadas provavelmente fornecerão os melhores resultados. Uma mudança para a arquitetura de microserviços sem uma mudança na estrutura organizacional será atenuar a utilidade dos microserviços - você pode acabar pagando o custo (considerável) da mudança arquitetônica, sem obter o retorno seu investimento. Eu escrevi geralmente sobre a necessidade de reduzir a coordenação entre equipes para ajudar a acelerar a entrega, o que, por sua vez, permite que as equipes tomam mais decisões por si mesmos. Essas são ideias que vamos explorar mais totalmente neste capítulo, e detalharemos algumas dessas questões organizacionais e mudanças comportamentais que são necessárias, mas antes disso, acho importante compartilhar minha visão sobre a aparência de uma organização pouco unida.

Em seu livro *Accelerate*, Nicole Forsgren, Jez Humble e Gene Kim

analisou as características de equipes autônomas e fracamente acopladas para melhorar

entender quais comportamentos são mais importantes para alcançar o melhor desempenho. De acordo com os autores, o fundamental é se as equipes podem:

- Faça alterações em grande escala no design de seu sistema sem a permissão de alguém de fora da equipe
- Faça alterações em grande escala no design de seu sistema sem dependendo de outras equipes para fazer mudanças em seus sistemas ou criando um trabalho significativo para outras equipes
- Conclua seu trabalho sem se comunicar e coordenar com pessoas fora de sua equipe
- Implemente e lance seu produto ou serviço sob demanda, independentemente de outros serviços dos quais depende
- Faça a maioria dos testes sob demanda, sem exigir um ambiente de teste integrado
- Execute implantações durante o horário comercial normal com tempo de inatividade insignificante

A equipe alinhada ao fluxo, um conceito que encontramos pela primeira vez no Capítulo 1, se alinha com essa visão de uma organização pouco unida. Se você está tentando se mover em direção a uma estrutura de equipe alinhada ao fluxo, essas características criam uma lista de verificação fantástica para garantir que você esteja indo na direção certa.

Algumas dessas características parecem de natureza mais técnica - por exemplo, a capacidade de implantação durante o horário comercial normal pode ser ativada por meio de uma arquitetura que oferece suporte a implantações sem tempo de inatividade. Mas tudo isso em fatos exigem uma mudança comportamental. Para permitir que as equipes tenham uma noção mais completa de propriedade de seus sistemas, um afastamento do controle centralizado é obrigatório, incluindo como a tomada de decisão arquitetônica é feita (algo que nós explore no Capítulo 16). Fundamentalmente, alcançando um acoplamento frágil as estruturas organizacionais exigem que o poder e a responsabilidade sejam descentralizados.

A maior parte deste capítulo discutirá como fazemos tudo isso funcionar, procurando no tamanho da equipe, nos tipos de modelos de propriedade, no papel da plataforma e muito mais.

Há uma série de mudanças que você pode considerar fazer para mover sua organização na direção certa.

Antes de tudo isso, porém, vamos explorar a interação entre organização e arquitetura um pouco mais.

Lei de Conway

Nossa indústria é jovem e parece estar se reinventando constantemente. E ainda assim um poucas "leis" importantes resistiram ao teste do tempo. A lei de Moore, por exemplo, afirma que a densidade dos transistores em circuitos integrados dobra a cada dois anos, e provou ser estranhamente preciso (embora essa tendência tenha sido desacelerando). Uma lei que descobri ser quase universalmente verdadeira e que muito mais útil no meu trabalho diário, é a lei de Conway.

Artigo de Melvin Conway, "Como os comitês inventam?", publicado em A revista Datamation, em abril de 1968, observou que:

Qualquer organização que projeta um sistema (definido de forma mais ampla aqui do que apenas sistemas de informação) inevitavelmente produzirá um design cujo estrutura é uma cópia da estrutura de comunicação da organização.

Essa afirmação é frequentemente citada, em várias formas, como a lei de Conway. Eric S. Raymond resumiu esse fenômeno no The New Hacker's Dictionary (MIT Press) afirmando: "Se você tem quatro grupos trabalhando em um compilador, você receberá um compilador de 4 passagens."

A lei de Conway nos mostra que uma organização fracamente acoplada resulta em uma arquitetura fracamente acoplada (e vice-versa), reforçando a ideia de que esperando obter os benefícios de uma arquitetura de microserviços fracamente acoplada será problemático sem também considerar a organização que está construindo o software.

Evidências

A história conta que, quando Melvin Conway enviou seu artigo sobre esse assunto para a Harvard Business Review, a revista a rejeitou, alegando que ele não tinha

provou sua tese. Eu vi sua teoria ser confirmada de muitas maneiras diferentes. Em muitas situações que eu aceitei como verdadeiras. Mas você não precisa acreditar na minha palavra para isso: desde a apresentação original de Conway, muito trabalho foi feito em... esta área. Vários estudos foram realizados para explorar o inter-relação da estrutura organizacional e dos sistemas que elas criam.

Organizações fracas e estreitamente acopladas

Em "Explorando a dualidade entre produto e organização Arquiteturas,"² os autores analisam vários sistemas de software diferentes, vagamente categorizado como sendo criado por "acoplado frouxamente organizações" ou por "organizações fortemente acopladas". Para acoplamento estreito organizações, pense em empresas de produtos comerciais que normalmente são colocadas com visões e metas fortemente alinhadas, enquanto organizações fracamente acopladas são bem representados por comunidades distribuídas de código aberto.

Em seu estudo, no qual eles combinaram pares de produtos similares de cada tipo de organização, os autores descobriram que quanto mais fracamente acopladas as organizações na verdade, criou sistemas mais modulares e menos acoplados, enquanto que quanto mais o software da organização fortemente acoplado era menos modularizado.

Windows Vista

A Microsoft realizou um estudo empírico³ no qual analisou como seus próprios a estrutura organizacional impactou a qualidade de um produto de software específico, Windows Vista. Fornecido, os pesquisadores analisaram vários fatores para determinar o quanto propenso a erros um componente no sistema seria.⁴ Depois analisando várias métricas, incluindo a qualidade de software comumente usada métricas como complexidade do código, eles descobriram que as métricas associadas a estruturas organizacionais (como o número de engenheiros que trabalharam em um pedaço de código) provaram ser as medidas mais estatisticamente relevantes.

Então, aqui temos outro exemplo da estrutura de uma organização impactando a natureza do sistema que a organização cria.

Netflix e Amazon

Provavelmente, as duas crianças-propaganda da ideia de que organizações e a arquitetura deve estar alinhada com a Amazon e a Netflix. Logo no início, a Amazon começou a entender os benefícios de as equipes possuírem todo o ciclo de vida dos sistemas que eles gerenciavam. Ela queria que as equipes possuíssem e operassem os sistemas que elas cuidada, gerenciando todo o ciclo de vida. Mas a Amazon também sabia que era pequeno equipes podem trabalhar mais rápido do que equipes grandes. Isso levou à sua infame pizza dupla. equipes, onde nenhuma equipe deveria ser tão grande que não pudesse ser alimentada por duas pizzas. Obviamente, essa não é uma métrica totalmente útil - nunca descobrimos se somos comer pizza no almoco ou jantar (ou café da manhã!), nem o tamanho das pizzas-mas, novamente, a ideia geral é que o tamanho ideal da equipe é de 8 a 10 pessoas, e que essa equipe deve estar voltada para o cliente. Este driver para pequenas equipes que possuem todo o ciclo de vida de seus serviços é a principal razão pela qual a Amazon desenvolveu a Amazon Web Services. Era necessário criar as ferramentas para permitir suas equipes para serem autossuficientes.

A Netflix aprendeu com esse exemplo e garantiu que, desde o início, estruturou-se em torno de equipes pequenas e independentes, para que os serviços criados também seriam independentes uns dos outros. Isso garantiu que a arquitetura do sistema foi otimizada para acelerar a mudança. Efetivamente, A Netflix projetou a estrutura organizacional para a arquitetura do sistema queria. Também ouvi dizer que isso se estendeu aos planos de assentos para equipes da Netflix. As equipes da Netflix cujos serviços conversariam entre si se sentariam próximas juntas. A ideia é que você gostaria de ter mais frequência comunicação com equipes que usam seus serviços ou cujos serviços você use você mesmo.

Tamanho da equipe

Pergunte a qualquer desenvolvedor qual deve ser o tamanho de uma equipe e enquanto você poderá variar respostas, haverá um consenso geral de que, até certo ponto, quanto menor, melhor. Se você os pressiona a colocar um número no tamanho "ideal" da equipe, você obterá em grande parte respostas na faixa de 5 a 10 pessoas.

Eu fiz algumas pesquisas sobre o melhor tamanho de equipe para desenvolvimento de software. Eu achei muitos estudos, mas muitos deles são falhos de tal forma que desenhar

conclusões para o mundo mais amplo do desenvolvimento de software são muito difíceis. O melhor estudo que encontrei, "Descobertas empíricas sobre o tamanho e a produtividade da equipe em "Desenvolvimento de software", foi pelo menos capaz de se basear em um grande corpus de dados, embora não sejam necessariamente representativos do software desenvolvimento como um todo. As descobertas dos pesquisadores indicaram que, "como esperado pela literatura, a produtividade é pior para aqueles projetos com (um tamanho médio da equipe) maior ou igual a 9 pessoas." Esta pesquisa, pelo menos parece apoiar minha própria experiência anedótica.

Gostamos de trabalhar em equipes pequenas, e não é difícil entender o porquê. Com um pequeno grupo de pessoas todas focadas nos mesmos resultados, é mais fácil ficar alinhado, e mais fácil de coordenar no trabalho. Eu tenho uma hipótese (não testada) de que estar geograficamente disperso ou ter grandes diferenças de fuso horário entre os membros da equipe causarão desafios que podem limitar ainda mais o ideal tamanho da equipe, mas esse pensamento provavelmente é mais bem explorado por alguém que não eu.

Então, equipes pequenas são boas, equipes grandes são ruins. Isso parece muito simples. Agora, se você pode fazer todo o trabalho que precisa fazer com uma única equipe, ótimo! Seu mundo é simples e você provavelmente poderia pular grande parte do resto do neste capítulo. Mas e se você tiver mais trabalho do que tempo para? Um A reação óbvia a isso é adicionar pessoas. Mas, como sabemos, adicionar pessoas pode não necessariamente ajudá-lo a fazer mais.

Entendendo a Lei de Conway

Evidências anedóticas e empíricas sugerem que nossa estrutura organizacional tem uma forte influência na natureza (e na qualidade) dos sistemas que criamos. Nós também sei que também queremos equipes menores. Então, como esse entendimento nos ajuda? Bem, fundamentalmente, se quisermos uma arquitetura fracamente acoplada que permita Para que as mudanças sejam feitas com mais facilidade, também queremos uma organização pouco unida. Dito de outra forma, a razão pela qual muitas vezes queremos um acoplamento mais frrouxo organização é que queremos que diferentes partes da organização sejam capazes de decidir e aja com mais rapidez e eficiência, e sistemas fracamente acoplados a arquitetura ajuda muito nisso.

Em Accelerate, os autores encontraram uma correlação significativa entre aqueles organizações que tinham uma arquitetura fracamente acoplada e sua capacidade de fazer mais faça uso eficaz de equipes de entrega maiores:

Se obtivermos uma arquitetura fracamente acoplada e bem encapsulada com um estrutura organizacional à altura, duas coisas importantes acontecem. Primeiro, podemos obter um melhor desempenho de entrega, aumentando o tempo e estabilidade e, ao mesmo tempo, reduz o desgaste e a dor da implantação. Em segundo lugar, podemos aumentar substancialmente o tamanho de nossa organização de engenharia e aumentamos a produtividade linearmente, ou melhor do que linearmente, à medida que fazemos isso.

Organizacionalmente, a mudança que vem ocorrendo há algum tempo, especialmente para organizações que operam em grande escala, é um afastamento da centralização modelos de comando e controle. Com a tomada de decisão centralizada, a velocidade em o fato de nossa organização poder reagir é significativamente enfraquecido. Isso é composto à medida que a organização cresce, quanto maior ela se torna, mais centralizada a natureza reduz a eficiência da tomada de decisão e a velocidade da ação.

As organizações reconhecem cada vez mais que, se você quiser escalar seu organização, mas ainda quer se mover rapidamente, você precisa distribuir responsabilidade de forma mais eficaz, dividindo a tomada de decisão central e empurrando decisões para partes da organização que podem operar com maior autonomia.

O truque, então, é criar grandes organizações a partir de organizações menores e autônomas equipes.

Equipes pequenas, grande organização

Adicionar mão de obra a um projeto de software atrasado o torna mais tarde.

-Fred Brooks (Lei de Brooks)

Em seu famoso ensaio, "The Mythical Man-Month", o autor Fred Brooks tenta explicar por que usar "um homem-mês" como técnica de estimativa é problemático, devido ao fato de que nos faz cair na armadilha de pensar que podemos faça com que mais pessoas resolvam o problema para ir mais rápido. A teoria é a seguinte: se um

um trabalho levará seis meses para um desenvolvedor, então, se adicionarmos um segundo desenvolvedor, levará apenas três meses. Se adicionarmos cinco desenvolvedores, então Agora temos seis no total, o trabalho deve ser feito em apenas um mês! É claro, o software não funciona assim.

Para que você consiga colocar mais pessoas (ou equipes) em um problema para ir mais rápido, o trabalho precisa ser dividido em tarefas que possam ser trabalhadas, até certo ponto, em paralelo. Se um desenvolvedor está fazendo algum trabalho que outro desenvolvedor está esperando, o trabalho não pode ser feito em paralelo, ele deve ser feito sequencialmente. Mesmo que o trabalho possa ser feito em paralelo, muitas vezes é necessário coordenar entre as pessoas que realizam os diferentes fluxos de trabalho, resultando em sobrecarga adicional. Quanto mais entrelacado o trabalho, menos eficaz ele é para adicionar mais pessoas.

Se você não conseguir dividir o trabalho em subtarefas que possam ser trabalhadas de forma independente, você não pode simplesmente confundir as pessoas com o problema. Pior, fazer isso provavelmente fará com que você demore a adicionar novas pessoas ou a criar novas equipes têm um custo. É necessário tempo para ajudar essas pessoas a serem plenamente produtivo e, muitas vezes, os desenvolvedores que têm muito trabalho a fazer são os mesmos desenvolvedores que precisarão gastar tempo ajudando as pessoas a velocidade.

O maior custo para trabalhar com eficiência em grande escala na entrega de software é o necessidade de coordenação. Quanto maior a coordenação entre as equipes que trabalham tarefas diferentes, mais lento você será. A Amazon, como empresa, tem reconheceu isso e se estrutura de forma a reduzir a necessidade de coordenação entre suas pequenas equipes de duas pizzas. Na verdade, houve um desejo consciente de limitar a quantidade de coordenação entre as equipes para isso muito raciocinar, e restringir essa coordenação, sempre que possível, às áreas onde era absolutamente necessário - entre equipes que compartilham uma fronteira entre microserviços. De Think Like Amazon, do ex-executivo da Amazon John Rossman diz:

A equipe Two-Pizza é autônoma. A interação com outras equipes é limitado e, quando ocorre, está bem documentado e interage estando claramente definidos. Ela possui e é responsável por todos os aspectos de sua sistemas. Um dos principais objetivos é diminuir as comunicações sobre carga nas organizações, incluindo o número de reuniões, pontos de coordenação, planejamento, testes ou lançamentos. Equipes que são mais movimento independente mais rápido.

Descobrir como uma equipe se encaixa em uma organização maior é essencial. Equipe

As topologias definem o conceito de uma API de equipe, que define amplamente como essa equipe interage com a rede da organização. Isso inclui as interfaces de microserviços, mas também em termos de práticas de trabalho:⁸

A API da equipe deve considerar explicitamente a usabilidade por outras equipes. Will outras equipes acham fácil e simples interagir conosco, ou irão é difícil e confuso? Será fácil para uma nova equipe conseguir concorda com nosso código e práticas de trabalho? Como respondemos a pull requests e sugestões de outras equipes? A lista de pendências da nossa equipe? e o roteiro do produto facilmente visível e compreensível por outras equipes?

Sobre autonomia

Seja qual for o setor em que você opera, tudo gira em torno de seu pessoal e captando-os fazendo as coisas certas e fornecendo-lhes o confiança, motivação, liberdade e desejo de alcançar sua verdadeira potencial...

-John Timpson

Ter muitas equipes pequenas por si só não vai ajudar se essas equipes simplesmente se transformam em mais silos que ainda dependem de outras equipes para realizar as coisas. Precisamos garantir que cada uma dessas pequenas equipes tenha autonomia para fazer o trabalho pelo qual é responsável. Isso significa que precisamos dar mais poder às equipes, para tomar decisões e as ferramentas para garantir que elas possam fazer o máximo possível possível sem a necessidade de coordenar constantemente o trabalho com outras equipes. Então possibilitar a autonomia é fundamental.

Muitas organizações demonstraram os benefícios de criar equipes autônomas. Manter os grupos organizacionais pequenos, permitindo que eles construam laços estreitos e trabalhar de forma eficaz em conjunto sem trazer muita burocracia, tem ajudou muitas organizações a crescer e escalar com mais eficiência do que algumas seus pares. W. L. Gore and Associates obteve grande sucesso ao fazer com certeza, nenhuma de suas unidades de negócios chega a mais de 150 pessoas, para fazer certeza de que todos se conhecem. Para essas unidades de negócios menores, trabalhar, eles precisam ter o poder e a responsabilidade de trabalhar como autônomos unidades.

Muitas dessas organizações parecem ter se inspirado no trabalho realizado pela antropólogo Robin Dunbar, que analisou a capacidade dos humanos de se formarem sociais agrupamentos. Sua teoria era que nossa capacidade cognitiva impõe limites sobre como efetivamente, podemos manter diferentes formas de relações sociais. Ele estimou em 150 pessoas o tamanho que um grupo poderia atingir antes de precisar se separaria ou então desmoronaria sob seu próprio peso.

A Timpson, uma varejista britânica de grande sucesso, alcançou grande escala por capacitando sua força de trabalho, reduzindo a necessidade de funções centrais e permitindo que as lojas locais tomem decisões por si mesmas, como quanto reembolsar clientes insatisfeitos. Agora presidente da empresa, John Timpson é famoso por descartar regras internas e substituí-las por apenas duas:

- Olha o papel.
- Coloque dinheiro na caixa.

A autonomia também funciona em menor escala, e a maioria das empresas modernas I trabalham com estão procurando criar equipes mais autônomas dentro de seus organizações, muitas vezes tentando copiar modelos de outras organizações, como O modelo de duas equipes de pizza da Amazon, ou o "modelo Spotify" que popularizou o conceito de guildas e capítulos. Devo emitir uma nota de cautela aqui, de curso - por todos os meios, aprenda com o que outras organizações fazem, mas entendam que copiar o que outra pessoa faz e esperar os mesmos resultados, sem realmente entender por que a outra organização faz as coisas que faz, pode não resultar no resultado que você deseja.

Se bem feita, a autonomia da equipe pode capacitar as pessoas, ajudá-las a se destacarem e cresça e faça o trabalho com mais rapidez. Quando as equipes possuem microsserviços e têm controle total sobre esses microsserviços, eles podem ter maior autonomia dentro de uma organização maior.

O conceito de autonomia começa a mudar nossa compreensão da propriedade em um arquitetura de microsserviços. Vamos explorar isso com mais detalhes.

Propriedade forte versus propriedade coletiva

Em "Definindo propriedade", discutimos diferentes tipos de propriedade e explorou as implicações desses estilos de propriedade no contexto de fazendo alterações no código. Como uma breve recapitulação, as duas formas principais de código a propriedade que descrevemos é:

Forte propriedade

Um microsserviço pertence a uma equipe, e essa equipe decide o que muda para fazer com esse microsserviço. Se uma equipe externa quiser fazer alterações, ou precisa pedir à equipe proprietária que faça a mudança em seu nome ou caso contrário, pode ser necessário enviar uma solicitação de pull - seria então totalmente cabe à equipe proprietária decidir sob quais circunstâncias a atração o modelo de solicitação é aceito. Uma equipe pode possuir mais de um microsserviço.

Propriedade coletiva

Qualquer equipe pode mudar qualquer microsserviço. É necessária uma coordenação cuidadosa para garantir que as equipes não atrapalhem umas às outras.

Vamos explorar ainda mais as implicações desses modelos de propriedade e como eles podem ajudar (ou atrapalhar) a busca por uma maior autonomia da equipe.

Forte propriedade

Com uma forte propriedade, a equipe proprietária do microsserviço dá as ordens. Em nível mais básico, ele tem controle total sobre quais alterações de código são feitas.

Além disso, a equipe pode decidir sobre os padrões de codificação, expressões idiomáticas de programação, quando implantar o software, qual tecnologia é usada para criar o microsserviço, a plataforma de implantação e muito mais. Ao ter mais responsabilidade pelas mudanças que acontecem no software, equipes com uma propriedade forte terá um maior grau de autonomia, com todos os benefícios isso implica.

Em última análise, uma forte propriedade tem tudo a ver com otimizar a autonomia dessa equipe. Voltando à maneira de fazer as coisas da Amazon com Think Like Amazon:

Quando se trata das famosas equipes de duas pizzas da Amazon, a maioria das pessoas sente falta o ponto. Não se trata do tamanho da equipe. É sobre a autonomia da equipe, responsabilidade e mentalidade empreendedora. A equipe Two-Pizza é sobre equipar uma pequena equipe dentro de uma organização para operar de forma independente e com agilidade.

Modelos de propriedade fortes podem permitir uma maior variação local. Você pode ser relaxado, por exemplo, sobre uma equipe decidir criar seu microsserviço em um estilo funcional de Java, pois essa é uma decisão que só deve impactá-los. Declaro, essa variação precisa ser moderada um pouco, pois algumas decisões garantem um certo grau de consistência em torno deles. Por exemplo, se todos os outros faz uso de APIs baseadas em REST-over-HTTP para seus endpoints de microsserviço mas você decide usar o GRPC, pode estar causando alguns problemas para outras pessoas que querem usar seu microsserviço. Por outro lado, se isso O endpoint GRPC foi usado apenas internamente em sua equipe, isso pode não ser problemático. Então, ao tomar decisões localmente que tenham impacto sobre outras pessoas equipes, a coordenação ainda pode ser necessária. Descobrir quando e como engajar uma organização mais ampla é algo que exploraremos em breve quando veja como equilibrar a otimização local versus a global.

Fundamentalmente, quanto mais forte o modelo de propriedade que uma equipe pode adotar, menos a coordenação é necessária e, portanto, mais produtiva a equipe pode ser.

Até onde vai uma forte propriedade?

Até esse estágio, falamos principalmente sobre aspectos como criar código mudanças ou escolha de tecnologia. Mas o conceito de propriedade pode ser muito importante

mais profundo. Algumas organizações adotam um modelo que eu descrevo como ciclo de vida completo de propriedade. Com a propriedade do ciclo de vida completo, uma única equipe cria o projeto, faz as alterações, implanta o microsserviço, gerencia-o em produção e, por fim, descomissiona o microsserviço quando ele não está exigido por mais tempo.

Esse modelo de propriedade do ciclo de vida completo aumenta ainda mais a autonomia da equipe, tem, à medida que os requisitos de coordenação externa são reduzidos. Os ingressos não são criado com equipes de operações para implantar as coisas, sem assinatura de terceiros desligada das mudanças, e a equipe decide quais mudanças devem ser feitas e quando para enviar.

Para muitos de vocês, esse modelo pode ser fantasioso, pois você já tem um número dos procedimentos existentes em vigor sobre como as coisas devem ser feitas. Você pode também não ter as habilidades certas na equipe para assumir a propriedade total, ou você pode exigir novas ferramentas (por exemplo, mecanismos de implantação de autoatendimento). É vale a pena notar, é claro, que você não obtém a propriedade total do ciclo de vida da noite para o dia, mesmo se você considerar que é uma meta ambiciosa. Não seja surpreso se essa mudança pode levar anos para ser totalmente adotada, especialmente com uma organização maior. Muitos aspectos da propriedade do ciclo de vida completo podem exigir mudanças culturais e ajustes significativos em termos das expectativas de você ter de alguns de seus funcionários, como a necessidade de apoiar seu software fora do horário de expediente. Mas, à medida que você pode assumir mais responsabilidade por aspectos de seu microsserviço em sua equipe, você aumentará ainda mais a autonomia de ter.

Não quero sugerir que esse modelo seja de alguma forma essencial para o uso de Microserviços - acredito firmemente na forte propriedade de várias equipes. As organizações são o modelo mais sensato para tirar o máximo proveito dos microsserviços. Um modelo de propriedade sólido em torno de mudanças de código é um bom lugar para começar a trabalhar para alcançar a propriedade total do ciclo de vida ao longo do tempo.

Propriedade coletiva

Com um modelo de propriedade coletiva, um microsserviço pode ser alterado por qualquer uma das várias equipes. Um dos principais benefícios da propriedade coletiva é

que você pode mover as pessoas para onde elas são necessárias. Isso pode ser útil se o gargalo em termos de entrega é causado pela falta de pessoas. Por exemplo, várias mudanças exigem que algumas atualizações sejam feitas no pagamento microsserviço para permitir o faturamento mensal automatizado - você pode simplesmente atribuir pessoas extras para implementar essa mudança. Claro, jogando pessoas em um O problema nem sempre faz você ir mais rápido, mas com propriedade coletiva você certamente tem mais flexibilidade nesse sentido.

Com equipes e pessoas migrando com mais frequência do microsserviço para microsserviço, exigimos um maior grau de consistência sobre como as coisas estão prontos. Você não pode pagar uma ampla variedade de opções de tecnologia ou diferentes tipos de modelos de implantação se você espera que um desenvolvedor trabalhe em um diferente microsserviço a cada semana. Para obter qualquer grau de eficácia de um modelo de propriedade coletiva com microsserviços, você acabará precisando garantir que trabalhar em um microsserviço seja praticamente o mesmo que trabalhar em qualquer outro outros.

Inerentemente, isso pode minar um dos principais benefícios dos microsserviços. Voltando a uma citação que usamos no início do livro, James Lewis tem disse que "os microsserviços compram opções para você". Com uma propriedade mais coletiva modelo, você provavelmente precisará reduzir as opções para introduzir um maior grau de consistência entre o que as equipes fazem e como os microsserviços são implementados.

A propriedade coletiva requer um alto grau de coordenação entre indivíduos e entre as equipes em que esses indivíduos estão. Este grau superior de coordenação resulta em um maior grau de acoplamento em um nível organizacional. Voltando ao artigo de MacCormack et al., mencionada no início deste capítulo, temos a seguinte observação:

Em organizações fortemente acopladas, mesmo que não sejam gerenciais explícitas escolha, o design naturalmente se torna mais estreitamente acoplado.

Mais coordenação pode levar a um maior acoplamento organizacional, o que, por sua vez leva a projetos de sistemas mais acoplados. Os microsserviços funcionam melhor quando pode adotar totalmente o conceito de implantação independente - e de forma rigorosa a arquitetura acoplada será o oposto do que você deseja.

Se você tiver um pequeno número de desenvolvedores e talvez apenas uma única equipe, o modelo de propriedade coletiva pode ser totalmente bom. Mas como o número de os desenvolvedores aumentam, a coordenação refinada necessária para tornar o coletivo propriedade, o trabalho acabará se tornando um fator negativo significativo em termos de obter os benefícios da adoção de arquiteturas de microserviços.

No nível da equipe versus no nível organizacional

Os conceitos de propriedade forte e coletiva podem ser aplicados em diferentes níveis de uma organização. Dentro de uma equipe, você quer que as pessoas estejam na mesma página, capaz de colaborar eficientemente entre si, e você deseja garantir um alto grau de propriedade coletiva como resultado. Por exemplo, isso seria se manifesta em termos de todos os membros de uma equipe serem capazes de fazer diretamente alterações na base de código. Uma equipe polivalente focada em ponta a ponta a entrega de software voltado para o cliente precisará ser muito boa propriedade coletiva. No nível organizacional, se você quiser que as equipes tenham um alto grau de autonomia, então é importante que eles também tenham um forte modelo de propriedade.

Modelos de balanceamento

Em última análise, quanto mais você tende à propriedade coletiva, mais é importante ter consistência na forma como as coisas são feitas. Quanto mais sua organização tende a ter uma forte propriedade, quanto mais você puder permitir otimização local, como vemos na Figura 15-1. Esse equilíbrio não precisa ser fixo - você provavelmente mudará isso em momentos diferentes e em diferentes ocasiões fatores. Você pode, por exemplo, dar às equipes total liberdade na escolha de uma linguagem de programação, mas ainda exige que eles sejam implantados na mesma nuvem plataforma, por exemplo.



Figura 15-1. O equilíbrio entre consistência global e otimização local.

Fundamentalmente, porém, com um modelo de propriedade coletiva, você quase sempre seja forçado em direção à extremidade esquerda desse espectro, ou seja, em direção a exigindo um maior grau de consistência global. Na minha experiência, organizações que tiram o máximo proveito dos microsserviços estão constantemente tentando encontrar maneiras de mudar a balança mais para a direita. Dito isso, para o melhor organizações, isso não é algo gravado em pedra, mas é mais alguma coisa que está sendo constantemente avaliado.

A realidade é que você não pode realizar nenhum balanceamento a menos que esteja ciente de em certa medida, do que está acontecendo em sua organização. Mesmo se você empurrar um muita responsabilidade para com as próprias equipes, ainda pode haver valor em ter uma função em sua organização de entrega que possa realizar isso ato de equilíbrio.

Capacitando equipes

Analisamos pela última vez a capacitação de equipes em "Especialistas em compartilhamento" no contexto de interfaces de usuário, mas elas têm uma aplicabilidade mais ampla do que isso. Conforme descrito em Topologias de equipe, essas são as equipes que trabalham para apoiar nosso alinhamento ao fluxo equipes. Onde quer que nosso stream seja proprietário de microsserviços e focado de ponta a ponta equipes alinhadas estão se concentrando em oferecer a funcionalidade voltada para o usuário, de que precisam ajuda de outras pessoas para fazer seu trabalho. Ao discutir a interface do usuário, conversamos sobre a ideia de ter uma equipe capacitadora que possa ajudar a apoiar outras equipes em termos de criação de experiências de usuário eficazes e consistentes. Conforme mostrado em Figura 15-2, podemos imaginar isso como capacitar equipes trabalhando para apoiar várias equipes alinhadas ao fluxo em algum aspecto transversal.

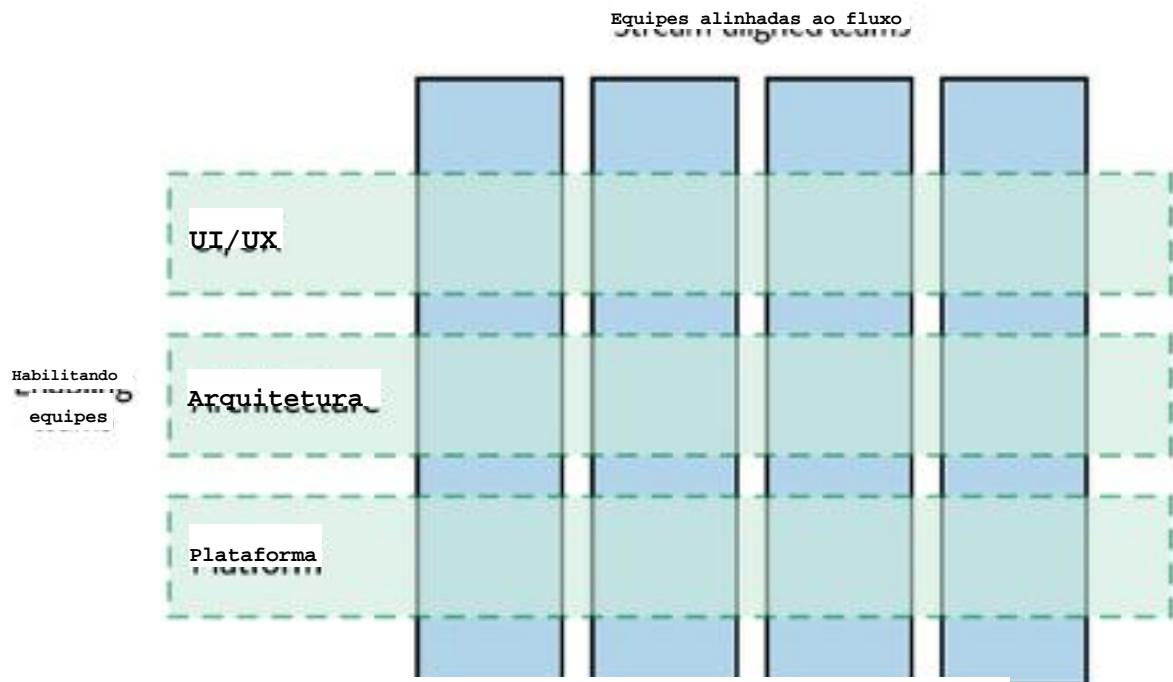


Figura 15-2. Permitindo que as equipes suportem várias equipes alinhadas ao fluxo

Mas as equipes capacitadoras podem ter diferentes formas e tamanhos.

Considerando que cada equipe decidiu escolher um diferente tipo de linguagem de programação. Quando analisadas isoladamente, cada uma dessas decisões parece fazer sentido - cada equipe escolheu a linguagem de programação que usa e está mais feliz com. Mas e a organização como um todo? Você quer que sua organização suporte várias linguagens de programação diferentes em seu dia-a-dia? Como isso complica a rotação entre equipes e como isso afeta a contratação, aliás?

Você pode decidir que, na verdade, não quer tanta otimização local mas você precisa estar ciente dessas diferentes escolhas que estão sendo feitas e ter alguma capacidade de discutir essas mudanças se você quiser algum grau de controle. Normalmente é aqui que veio um grupo de apoio muito pequeno trabalhando em suas equipes para ajudar a conectar as pessoas para que essas discussões possam acontecer adequadamente.

É nesses grupos de apoio transversais que eu posso ver o lugar óbvio para que arquitetos se baseiem, pelo menos em parte de seu tempo. Um antiquado O arquiteto diria às pessoas o que fazer. No novo, moderno e descentralizado organismo, arquitetos estão pesquisando a paisagem, identificando tendências, ajudando a conectar pessoas e atuar como uma caixa de ressonância para ajudar outras equipes a conseguir coisas

feito. Eles não são uma unidade de controle neste mundo; eles são mais uma capacitação função (geralmente com um novo nome) - já vi termos como engenheiro principal usado para pessoas que desempenham o papel do que eu consideraria um arquiteto). Exploraremos mais o papel dos arquitetos no Capítulo 16.

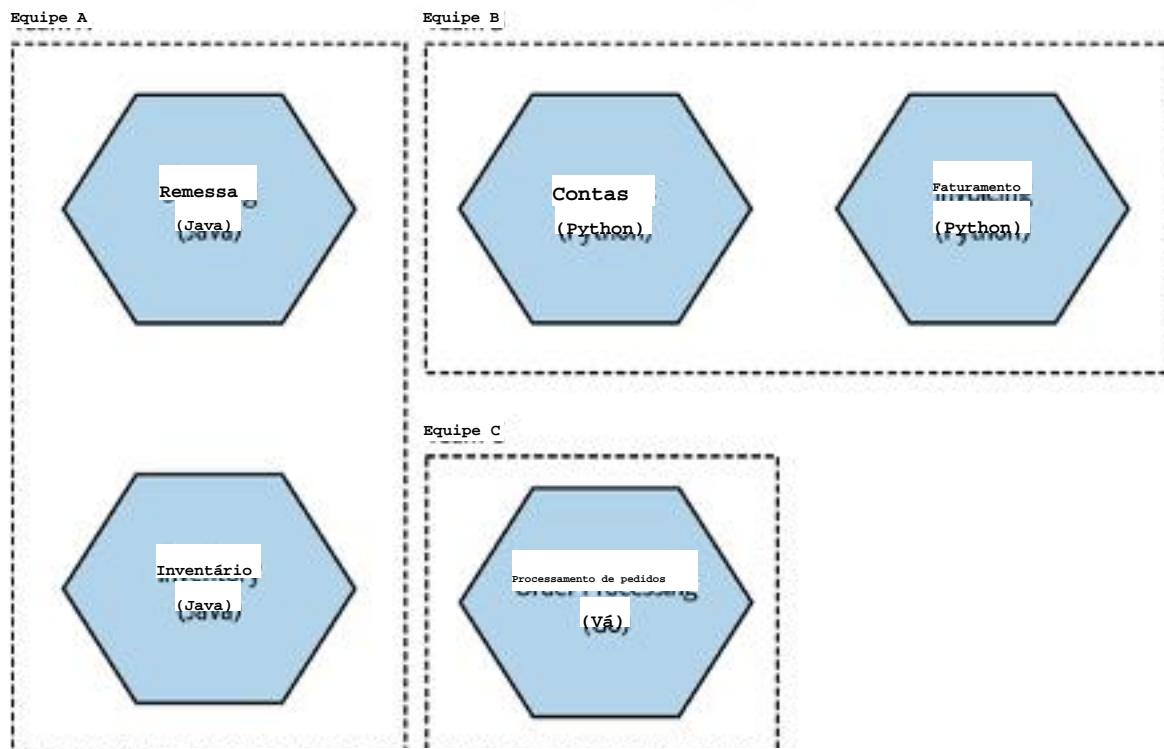


Figura 15-3. Cada equipe escolheu uma linguagem de programação diferente

Uma equipe de capacitação pode ajudar a identificar problemas que seriam melhor resolvidos fora das equipes também. Considere um cenário em que cada equipe estava achando difícil criar bancos de dados com dados de teste. Cada equipe tinha resolvido esse problema de maneiras diferentes, mas o problema nunca foi importante o suficiente para qualquer equipe consertá-lo adequadamente. Mas depois analisamos várias equipes e descobrem que muitos deles podem se beneficiar de uma solução adequada para o problema, e de repente fica óbvio que ele precisa ser resolvido.

Comunidades de prática

Uma comunidade de prática (CoP) é um grupo transversal que promove o compartilhamento e aprendizado entre colegas. Quando bem feitas, as comunidades de prática são maneira fantástica de criar uma organização na qual as pessoas possam

aprenda e cresça continuamente. Em seu excelente livro sobre o assunto, Construindo Comunidades de prática bem-sucedidas, 11 Emily Webber escreve:

Comunidades de prática criam o ambiente certo para o social aprendizagem, aprendizagem experiencial e um currículo completo, levando a aprendizagem acelerada para membros. Pode incentivar uma cultura de aprendizagem onde as pessoas buscam maneiras melhores de fazer as coisas, em vez de apenas usar modelos existentes.

Agora, acredito que, em alguns casos, o mesmo grupo pode ser tanto um CoP quanto uma equipe capacitadora, mas na minha própria experiência isso é raro. Há um definitivo sobrepõe-se, no entanto. Tanto as equipes capacitadoras quanto as comunidades de prática oferecem insights sobre o que está acontecendo em diferentes equipes em sua organização. Essa visão pode ajudá-lo a entender se você precisa reequilibrar sua otimização global versus local, ou ajudar você a identificar a necessidade de mais ajuda central, mas a divisão aqui está nas responsabilidades e capacidades de o grupo.

Os membros de uma equipe capacitadora geralmente trabalham em tempo integral como parte da equipe, ou então eles têm uma quantidade significativa de seu tempo reservada para tal propósito. Dessa forma, eles têm mais largura de banda para colocar as mudanças em ação realmente trabalhe com outras equipes e as ajude. As comunidades de prática são focado mais em possibilitar o aprendizado - os indivíduos do grupo geralmente são participar de um fórum por algumas horas por semana, no máximo, e a associação de tal grupo geralmente é fluido.

Os COPs e as equipes capacitadoras podem trabalhar juntos de forma muito eficaz, é claro. Muitas vezes, um CoP é capaz de fornecer informações valiosas que podem ajudar a capacitar a equipe entende melhor o que é necessário. Considere um compartilhamento Kubernetes CoP sua experiência de como foi doloroso trabalhar no desenvolvimento de sua empresa cluster com a equipe da plataforma que gerencia o cluster. Sobre o assunto de equipes de plataforma, esse é um tópico que vale a pena examinar com mais detalhes.

A plataforma

Para retornar às nossas equipes fracamente acopladas e alinhadas ao fluxo, esperamos que elas o façam seus próprios testes em ambientes isolados, gerencie implantações em tal

maneira de fazer isso durante o dia e fazer alterações em seu sistema arquitetura quando necessário. Tudo isso parece estar pressionando cada vez mais responsabilidade e trabalho nessas equipes. Capacitando equipes como general o conceito pode ajudar aqui, mas, em última análise, equipes alinhadas ao fluxo precisam de uma equipe autônoma conjunto de ferramentas de serviços que lhes permite fazer seu trabalho - essa é a plataforma.

Sem uma plataforma, na verdade, você pode achar difícil mudar a organização. Em seu artigo "Convergence to Kubernetes", o CTO da RVU, Paul Ingles, compartilha a experiência do site de comparação de preços Uswitch moving longe do uso direto de serviços de baixo nível da AWS em direção a uma mais alta plataforma abstrata baseada em Kubernetes. A ideia era que essa plataforma permitiu que as equipes alinhadas ao fluxo da RVU se concentrassem mais na entrega de novos recursos e gaste menos tempo gerenciando a infraestrutura. Como diz Paul:

Não mudamos nossa organização porque queríamos usar Kubernetes; usamos o Kubernetes porque queríamos mudar nosso organização.

Uma plataforma que pode implementar funcionalidades comuns, como a capacidade de lidar com o gerenciamento do estado desejado para microsserviços, agregação de registros e autorização e autenticação entre microsserviços, podem oferecer grandes quantidades de melhorias na produtividade e permitem que as equipes assumam mais responsabilidade sem também ter que aumentar drasticamente a quantidade de trabalho que eles fazem. Na verdade, uma plataforma deve dar às equipes mais largura de banda para se concentrarem fornecendo recursos.

A equipe da plataforma

Uma plataforma precisa de alguém para executá-la e gerenciá-la. Essas pilhas de tecnologia podem ser complexas ou suficientes para garantir alguma experiência específica. Minha preocupação, porém, é que às vezes pode ser muito fácil para as equipes da plataforma perderem a noção do porquê elas existem.

Uma equipe de plataforma tem usuários, da mesma forma que qualquer outra equipe tem usuários. Os usuários da equipe da plataforma são outros desenvolvedores - seu trabalho, se você estiver em um equipe da plataforma, é facilitar suas vidas (isso, é claro, é o trabalho de qualquer pessoa) (equipe capacitadora). Isso significa que a plataforma que você cria precisa atender às necessidades

das equipes que o usam. Isso também significa que você precisa trabalhar com as equipes usando sua plataforma não apenas para ajudá-los a usá-la bem, mas também para incorporar seus feedback e requisitos para melhorar a plataforma que você oferece.

No passado, eu preferia chamar essa equipe de algo como "entrega". "serviços" ou "suporte de entrega" para melhor articular seu objetivo. Realmente, o trabalho de uma equipe de plataforma é criar uma plataforma; é fazer com que o desenvolvimento e funcionalidade de envio fácil. Construir uma plataforma é apenas uma forma de membros de uma equipe de plataforma podem conseguir isso. Eu me preocupo com isso ligando eles mesmos são uma equipe de plataforma, eles verão todos os problemas como coisas que podem e deve ser resolvido pela plataforma, em vez de pensar mais amplamente sobre outras formas de facilitar a vida dos desenvolvedores.

Como qualquer boa equipe de capacitação, uma equipe de plataforma precisa operar quase como uma consultoria interna até certo ponto. Se você faz parte de uma equipe de plataforma, você precisa saia para descobrir quais problemas as pessoas estão enfrentando e com os quais estão trabalhando eles para ajudá-los a resolver esses problemas. Mas como você também acaba construindo a plataforma também, você precisa ter uma grande dose de desenvolvimento de produtos trabalhe lá também. Na verdade, adotando uma abordagem de desenvolvimento de produtos para saber como você criar sua plataforma é uma ótima ideia e pode ser um ótimo lugar para ajudar desenvolva novos proprietários de produtos.

A estrada pavimentada

Um conceito que se tornou popular no desenvolvimento de software é o de "o estrada pavimentada." A ideia é que você comunique claramente como quer as coisas a ser feito e, em seguida, fornecer mecanismos pelos quais essas coisas possam ser feitas facilmente. Por exemplo, talvez você queira garantir que todos os microserviços comunique-se via TLS mútuo. Você pode então fazer backup disso fornecendo um estrutura comum ou plataforma de implantação que forneceria automaticamente TLS mútuo para os microserviços executados nele. Uma plataforma pode ser ótima maneira de entregar naquela estrada pavimentada.

O conceito-chave por trás de uma estrada pavimentada é que usar a estrada pavimentada não é Enforced-it apenas fornece uma maneira mais fácil de chegar ao seu destino. Então, se uma equipe queria garantir que seus microserviços se comunicassem via TLS mútuo sem usar a estrutura comum, teria que encontrar outra maneira

de fazer isso, mas ainda assim seria permitido. A analogia aqui é que, enquanto nós pode querer que todos cheguem ao mesmo destino, as pessoas são livres para encontrar seus próprios caminhos lá - a esperança é que a estrada pavimentada seja a maneira mais fácil de chegando onde você precisa.

O conceito de estrada pavimentada visa facilitar os casos comuns, ao sair espaço para exceções quando necessário

Se pensarmos na plataforma como uma estrada pavimentada, ao torná-la opcional, você incentive a equipe da plataforma a tornar a plataforma fácil de usar. Aqui está Paul Ingles novamente, ao medir a eficácia da equipe da plataforma:

13

Definiríamos [objetivos e resultados-chave (OKRs)] em torno do número de equipes que gostaríamos de adotar: a plataforma, o número de aplicativos que usamos o serviço de escalonamento automático da plataforma, a proporção de aplicativos mudou para o serviço de credenciais dinâmicas da plataforma e assim por diante. Alguns dos aqueles que rastreávamos por longos períodos de tempo, e outros foram úteis para guiaremos o progresso por um trimestre e depois os abandonaríamos em favor de outra coisa.

Nunca exigimos o uso da plataforma, portanto, definimos os principais resultados para o número de equipes integradas nos forçou a nos concentrar na solução de problemas que impulsionaria a adoção. Também buscamos medidas naturais de progresso: a proporção de tráfego atendido pela plataforma e a proporção de a receita atendida por meio de serviços de plataforma é um bom exemplo de isso.

Quando você coloca barreiras no caminho das pessoas que parecem arbitrárias e caprichosas, essas pessoas encontrarão maneiras de contornar as barreiras para realizar um trabalho. Então, em geral, acho muito mais eficaz explicar por que as coisas deveriam ser feito de uma certa maneira e, em seguida, torne mais fácil fazer as coisas dessa maneira, em vez disso do que tentar fazer com que as coisas que você não gosta sejam impossíveis.

A mudança para equipes mais autônomas e alinhadas ao fluxo não elimina a necessidade de ter uma visão técnica clara ou de ser claro sobre certos coisas que todas as equipes precisam fazer. Se houver restrições concretas (o precisa ser independente do fornecedor de nuvem, por exemplo) ou de requisitos específicos todas as equipes precisam obedecer (todas as PII precisam ser criptografadas em repouso usando dados específicos

algoritmos), então eles ainda precisam ser comunicados com clareza e os motivos para eles ficou claro. A plataforma pode então desempenhar um papel na criação dessas coisas. Fácil de fazer. Ao usar a plataforma, você está na estrada pavimentada - você acabará fazendo muitas coisas certas sem ter que gastar muito esforço.

Por outro lado, vejo algumas organizações tentando governar por meio da plataforma. Em vez de articular claramente o que precisa ser feito e por que, em vez disso, eles simplesmente diga "você deve usar esta plataforma". O problema com isso é que, se o a plataforma não é fácil de usar ou não se adequa a um caso de uso específico, as pessoas o farão encontrar maneiras de contornar a própria plataforma. Quando as equipes trabalham fora do plataforma, eles não têm uma noção clara de quais restrições são importantes para a organização e se verão fazendo a coisa "errada" sem percebendo isso.

Microsserviços compartilhados

Como já discuti, sou um grande defensor de modelos de propriedade fortes para microsserviços. Em geral, um microsserviço deve pertencer a um único equipe. Apesar disso, ainda acho comum que os microsserviços sejam de propriedade de várias equipes. Por que isso? E o que você pode (ou deve) fazer sobre isso? Os drivers que fazem com que as pessoas tenham microsserviços compartilhados por várias equipes são importante entender, especialmente porque talvez possamos encontrar alguns modelos alternativos convincentes que podem abordar o subjacente das pessoas preocupações.

Muito difícil de dividir

Obviamente, uma das razões pelas quais você pode se deparar com um microsserviço de propriedade de mais de uma equipe é que o custo de dividir o microsserviço em peças que podem pertencer a equipes diferentes é muito alto, ou talvez seu a organização pode não ver o objetivo disso. Essa é uma ocorrência comum com sistemas grandes e monolíticos. Se este é o principal desafio que você enfrenta, espero alguns dos conselhos dados no Capítulo 3 serão úteis. Você também pode considere a fusão de equipes para se alinhar mais de perto com a própria arquitetura.

FinanceCo, uma empresa FinTech que conhecemos anteriormente em "Toward Stream-
"Equipes alinhadas", operam em grande parte um modelo de propriedade forte com uma alta
grau de autonomia da equipe. No entanto, ele ainda tem um sistema monolítico existente
que estava lentamente sendo dividido. Essa aplicação monolítica foi para todos
intenções e propósitos compartilhados por várias equipes e o aumento do custo de
trabalhar nessa base de código compartilhada era óbvio.

Mudanças transversais

Muito do que discutimos até agora neste livro sobre a interação de
a estrutura organizacional e a arquitetura visam reduzir a necessidade
para coordenação entre equipes. A maior parte disso é tentar reduzir o cruzamento
cortando as mudanças o máximo que pudermos. No entanto, temos que reconhecer que
algumas mudanças transversais podem ser inevitáveis.

O FinanceCo enfrentou um desses problemas. Quando começou originalmente, uma conta era
vinculado a um usuário. À medida que a empresa crescia e conquistava mais usuários corporativos
(sendo anteriormente mais focado nos consumidores), isso se tornou uma limitação. O
a empresa queria mudar para um modelo em que uma única conta com
O FinanceCo pode acomodar vários usuários. Isso foi fundamental
mudança, pois até aquele momento a suposição em todo o sistema era uma
conta = um usuário.

Uma única equipe foi formada para fazer essa mudança acontecer. O problema
foi que uma grande parte do trabalho envolveu fazer mudanças em
microsserviços já pertencentes a outras equipes. Isso significava que a equipe
o trabalho consistia, em parte, em fazer alterações e enviar pull requests, ou
pedindo a outras equipes que façam as mudanças acontecerem. Coordenando essas mudanças
foi muito doloroso, pois um número significativo de microsserviços precisava ser
modificado para suportar a nova funcionalidade.

Ao reestruturar nossas equipes e arquitetura para eliminar um conjunto de cruzamentos
reduzindo as mudanças, podemos, de fato, nos expor a um conjunto diferente de cruzamentos
cortando mudanças que possam ter um impacto mais significativo. Esse foi o caso
com FinanceCo - o tipo de reorganização que seria necessária para reduzir
o custo da funcionalidade multiusuário teria aumentado o custo de

fazendo outras mudanças mais comuns. FinanceCo entendeu que isso...
Uma mudança específica seria muito dolorosa, mas foi uma
tipo excepcional de mudança em que essa dor era aceitável....

Gargalos de entrega

Uma das principais razões pelas quais as pessoas adotam a propriedade coletiva, com
o compartilhamento de microserviços entre equipes é para evitar gargalos na entrega.
E se houver um grande acúmulo de mudanças que precisam ser feitas em uma única
serviço? Vamos voltar à MusicCorp e imaginar que estamos lançando
a capacidade de um cliente ver o gênero de uma faixa em nossos produtos, como
além de adicionar um novo tipo de estoque: toques musicais virtuais para o
telefone celular. A equipe do site precisa fazer uma mudança para revelar o gênero
informações, com a equipe de aplicativos móveis trabalhando para permitir que os usuários naveguem,
pré-visualize e compre os toques. Ambas as mudanças precisam ser feitas no
Microsserviço de catálogo, mas infelizmente metade da equipe não consegue diagnosticar um
falha na produção, com a outra metade saindo com intoxicação alimentar após uma recente
equipe saindo para um caminhão de comida pop-up sendo expulso de um beco.

Temos algumas opções que podemos considerar para evitar a necessidade de
equipes de sites e dispositivos móveis para compartilhar o microsserviço Catalog. A primeira é
apenas espere. As equipes de sites e aplicativos móveis avançam para algo
outra coisa. Dependendo da importância do recurso ou da duração do atraso
Provavelmente, isso pode ser bom ou pode ser um grande problema.

Em vez disso, você pode adicionar pessoas à equipe do catálogo para ajudá-las a avançar
seu trabalho é mais rápido. Quanto mais padronizada a pilha de tecnologia e
expressões idiomáticas de programação em uso em seu sistema, mais fácil é para outros
pessoas para fazer mudanças em seus serviços. O outro lado, é claro, como nós
discutido anteriormente, é que a padronização tende a reduzir a capacidade de uma equipe de
adote a solução certa para o trabalho e pode levar a diferentes tipos de
ineficiências.

Outra opção que evitaria a necessidade de um microsserviço de catálogo compartilhado
poderia ser dividir o catálogo em um catálogo geral de músicas separado e um
catálogo de toques. Se a alteração que está sendo feita para suportar toques for justa

pequeno, e a probabilidade de esta ser uma área na qual nos desenvolveremos pesadamente no futuro também é bastante baixo, isso pode muito bem ser prematuro. Sobre o por outro lado, se houver 10 semanas de recursos relacionados ao toque acumulados, dividir o serviço pode fazer sentido, com a equipe móvel aceitando propriedade.

No entanto, existem alguns outros modelos que podemos considerar. Em um Em seguida, veremos o que pode ser feito em termos de tornar o compartilhado microsserviço mais "conectável", permitindo que outras equipes contribuam seu código por meio de bibliotecas ou então se estende a partir de uma estrutura comum. Primeiro, no entanto, devemos explorar o potencial de trazer algumas ideias do mundo do desenvolvimento de código aberto em nossa empresa.

Código aberto interno

Muitas organizações decidiram implementar alguma forma de abertura interna fonte, tanto para ajudar a gerenciar o problema de bases de código compartilhadas quanto para torná-lo é mais fácil para pessoas de fora de uma equipe contribuirem com mudanças em um microsserviço pode estar fazendo uso de.

Com o código aberto normal, um pequeno grupo de pessoas é considerado essencial cometedor. Eles são os guardiões do código. Se você quiser mudar para um projeto de código aberto, ou você pede a um dos comitês que faça a mudança para você ou você mesmo faz a alteração e envia uma solicitação de pull. Os principais committers ainda são responsáveis pela base de código; eles são os proprietários.

Dentro de uma organização, esse padrão também pode funcionar bem. Talvez as pessoas que trabalharam no serviço originalmente não estão mais juntos em uma equipe; talvez eles estejam agora espalhados por toda a organização. Se eles ainda têm comprometer direitos, você pode encontrá-los e pedir a ajuda deles, talvez formando pares com eles; ou se você tiver as ferramentas certas, poderá enviar a eles uma solicitação de pull.

Papel dos comitês principais

Ainda queremos que nossos serviços sejam sensatos. Queremos que o código seja decente qualidade e o próprio microsserviço para exibir algum tipo de consistência na forma como

está montado. Também queremos ter certeza de que as mudanças estão sendo feitas agora não torne as mudanças planejadas para o futuro muito mais difíceis do que elas precisam ser. Isso significa que precisamos adotar os mesmos padrões usados no código aberto normal internamente também, o que significa separar os comprometidos confiáveis (a equipe principal) de autores não confiáveis (pessoas de fora da equipe que estão enviando mudanças).

A equipe principal de propriedade precisa ter alguma forma de examinar e aprovar mudanças. Ele precisa garantir que as mudanças sejam idiomaticamente consistentes, ou seja, que eles sigam as diretrizes gerais de codificação do resto da base de código. O as pessoas que fazem a verificação, portanto, terão que passar um tempo trabalhando com os remetentes para garantir que cada alteração seja de qualidade suficiente.

Bons guardiões se esforçam muito nisso, comunicando-se claramente com remetentes e incentivando o bom comportamento. Guardiões ruins podem usar isso como uma desculpa para exercer poder sobre os outros ou ter guerras religiosas arbitrárias decisões técnicas. Tendo visto os dois conjuntos de comportamento, posso te dizer um A coisa está clara: de qualquer forma, leva tempo. Ao considerar permitir que não seja confiável para enviar alterações em sua base de código, você deve decidir se o a sobrecarga de ser um guardião vale a pena: será que a equipe principal poderia fazendo coisas melhores com o tempo que passa examinando os patches?

Maturidade

Quanto menos estável ou maduro for um serviço, mais difícil será permitir que pessoas fora da equipe principal para enviar patches. Antes que a coluna vertebral de um serviço seja no local, a equipe pode não saber o que é "bom" e, portanto, pode tenho dificuldade em saber como é uma boa apresentação. Durante esse estágio, o o serviço em si está passando por um alto grau de mudança.

A maioria dos projetos de código aberto tende a não receber submissões de um grupo maior de confirmadores não confiáveis até que o núcleo da primeira versão seja concluído. Seguindo um um modelo semelhante para sua própria organização faz sentido. Se um serviço é bonito maduro e raramente é alterado - por exemplo, nosso serviço de carrinho - então talvez esse é o momento de abri-lo para outras contribuições.

Ferramentas

Para oferecer melhor suporte a um modelo interno de código aberto, você precisará de algumas ferramentas em lugar. O uso de uma ferramenta de controle de versão distribuída com a capacidade de as pessoas enviarem solicitações de pull (ou algo semelhante) é importante. Dependendo do tamanho da organização, você também pode precisar de ferramentas para permitir para uma discussão e evolução das solicitações de patch; isso pode ou não significar um sistema completo de revisão de código, mas com a capacidade de comentar em linha sobre patches é muito útil. Finalmente, você precisará tornar muito fácil para um comprometido crie e implante seu software e disponibilize-o para outras pessoas. Tipicamente, isso envolve ter pipelines de construção e implantação bem definidos e repositórios de artefatos centralizados. Quanto mais padronizada for sua tecnologia stack is, mais fácil será para as pessoas de outras equipes fazerem edições e fornecer patches para um microsserviço.

Microsserviços modulares e conectáveis

Na FinanceCo, vi um desafio interessante associado a um determinado microsserviço que estava se tornando um gargalo para muitas equipes. Para cada país, a FinanceCo tinha equipes dedicadas com foco em funcionalidades específicas para esse país. Isso fazia muito sentido, pois cada país tinha características específicas requisitos e desafios. Mas isso causou problemas para esse serviço central, que precisava de uma funcionalidade específica para ser atualizada para cada país. A equipe possuir o microsserviço central foi sobrecarregado por solicitações de pull enviado para ele. A equipe estava fazendo um excelente trabalho ao processar esses puxões. solicita rapidamente e, de fato, se concentrou em que isso fosse uma parte essencial de sua responsabilidades, mas estruturalmente a situação não era realmente sustentável.

Este é um exemplo de como uma equipe tem um grande número de pull requests pode ser um sinal de muitos problemas potenciais diferentes. São pull requests de outros equipes sendo levadas a sério? Ou esses pull requests são um sinal de que o o microsserviço deve potencialmente mudar de propriedade?

ADVERTÊNCIA

Se uma equipe tem muitas solicitações de pull de entrada, isso pode ser um sinal de que você realmente tem uma microserviço que está sendo compartilhado por várias equipes.

Alterando a propriedade

Às vezes, a coisa certa a fazer é mudar quem é o proprietário de um microserviço.

Considere um exemplo na MusicCorp. A equipe de engajamento do cliente é

ter que enviar um grande número de solicitações de pull relacionadas ao

Microserviço de recomendação para a equipe de Marketing e Promocões. Isso

é porque várias mudanças estão acontecendo no que diz respeito à forma como o cliente

as informações são gerenciadas, e também porque precisamos revelá-las

recomendações de maneiras diferentes.

Nessa situação, pode fazer sentido que a equipe de engajamento do cliente apenas

assuma a propriedade do microserviço Recommendation. No exemplo de

FinanceCo, no entanto, essa opção não existia. O problema era que as fontes

das pull requests vinham de várias equipes diferentes. Então, o que mais?

poderia ser feito?

Execute várias variações

Uma opção que exploramos foi que cada equipe do país dirigisse sua própria equipe.

variação do microserviço compartilhado. Então, a equipe dos EUA executaria sua versão,

a equipe de Cingapura, sua própria variação, e assim por diante.

Obviamente, o problema com essa abordagem é a duplicação de código. O compartilhado

o microserviço implementou um conjunto de comportamentos padrão e regras comuns, mas

também queria que algumas dessas funcionalidades mudassem para cada país. Nós não

deseja duplicar a funcionalidade comum. A ideia era para a equipe que

atualmente gerenciava o microserviço compartilhado para, em vez disso, fornecer uma estrutura,

que na verdade consistia apenas no microserviço existente com apenas o

funcionalidade comum. Cada equipe específica de cada país poderia lançar a sua própria

instância do microserviço básico, conectando seu próprio microserviço personalizado

funcionalidade, conforme mostrado na Figura 15-4.

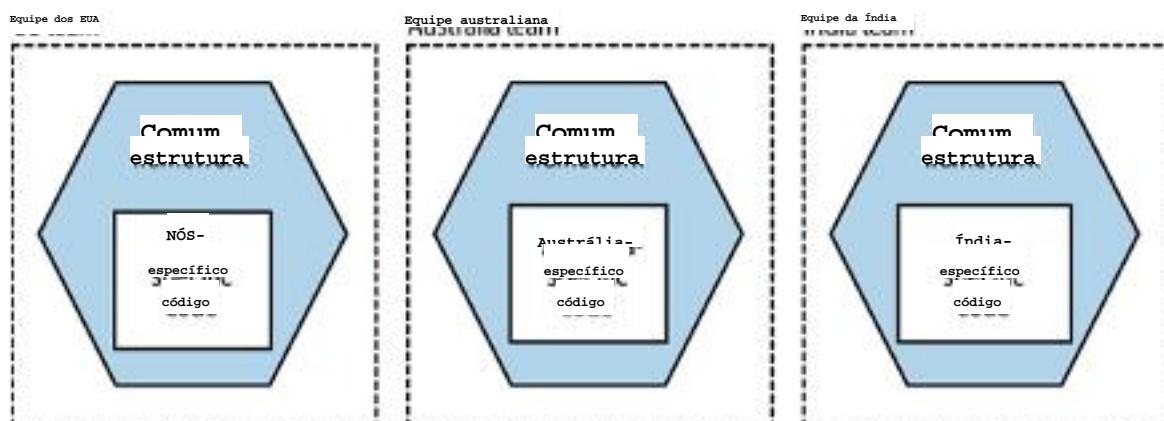


Figura 15-4. Uma estrutura comum pode permitir que várias variações do mesmo microsserviço sejam operado por equipes diferentes

O importante a observar aqui é que, embora possamos compartilhar o comum funcionalidade neste exemplo em cada variação específica do país do microsserviço, essa funcionalidade comum não pode ser atualizada em todas as variações do microsserviço ao mesmo tempo sem exigir um bloqueio em grande escala. liberar. A equipe principal que gerencia a estrutura pode criar uma nova versão disponível, mas caberia a cada equipe obter a versão mais recente do código comum e reimplante-o. Nesta situação específica, a FinanceCo estaria bem com essa restrição.

Vale ressaltar que essa situação específica era bastante rara, e algo que eu havia encontrado apenas uma ou duas vezes antes. Meu foco inicial foi sobre encontrar maneiras de dividir as responsabilidades desse compartilhamento central microsserviço ou então reatribua a propriedade. Minha preocupação era que a criação estruturas internas podem ser uma atividade complicada. É muito fácil para o estrutura para ficar excessivamente inchada ou restringir o desenvolvimento do equipes que o usam. Esse é o tipo de problema que não se manifesta no dia um. Ao criar uma estrutura interna, tudo começa com o melhor intenções. Embora, na situação da FinanceCo, eu achasse que era o caminho certo.

Para frente, eu alertaria contra a adoção dessa abordagem com muita facilidade, a menos que você tenha esgotou suas outras opções.

Contribuição externa por meio de bibliotecas

Uma variação dessa abordagem seria, em vez disso, ter cada país específico a equipe contribui com uma biblioteca com sua funcionalidade específica de cada país dentro dela, e

em seguida, tenha essas bibliotecas empacotadas em um único compartilhamento microsserviço, conforme mostrado na Figura 15-5.

Equipe de serviço principal

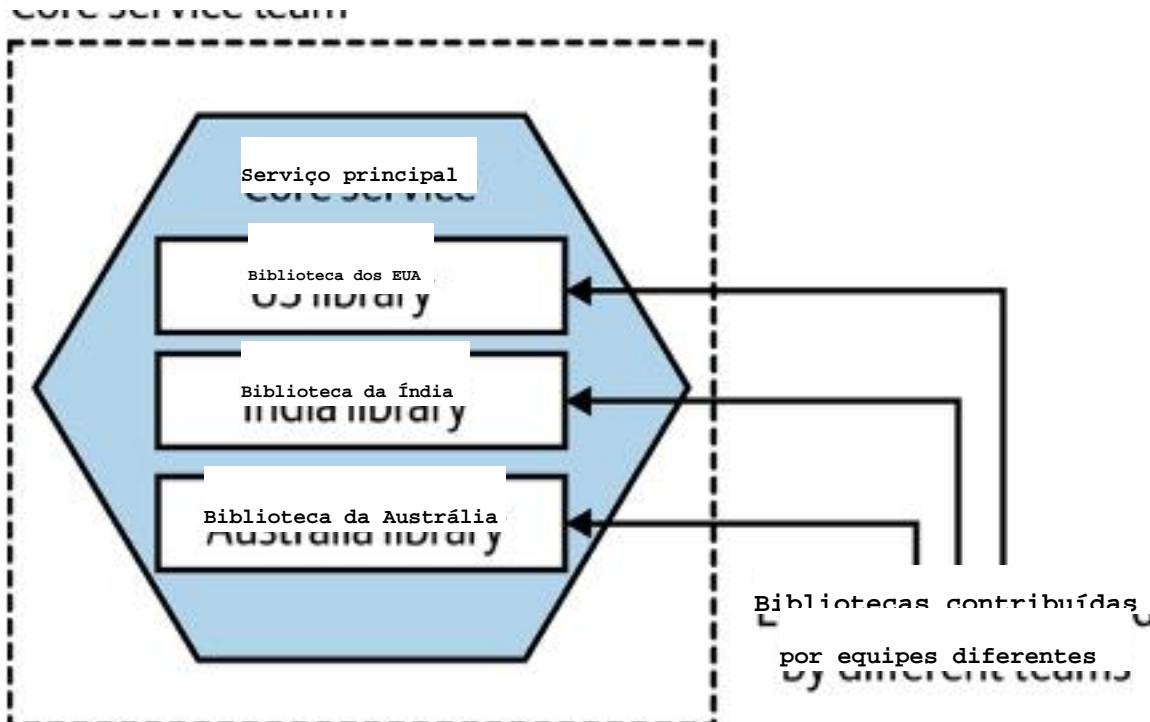


Figura 15-5. As equipes contribuem com bibliotecas com seu comportamento personalizado para um microsserviço central.

A ideia aqui é que, se a equipe dos EUA precisar implementar uma lógica específica dos EUA, faz uma alteração em uma biblioteca que é então incluída como parte da construção do microsserviço central.

Essa abordagem reduz a necessidade de executar microsserviços adicionais. Nós não precisamos executar um serviço por país – podemos executar um microsserviço central que lida com a funcionalidade personalizada de cada país. O desafio aqui é que as equipes nacionais não estão encarregadas de decidir quando são seus costumes... a funcionalidade entra em operação. Eles podem fazer a alteração e solicitar que esta nova mudança seja implantada, mas a equipe central teria que agendar isso... implantação.

Além disso, é possível que haja um bug em uma dessas bibliotecas específicas do país pode causar um problema de produção que a equipe central pode então ser responsável para resolver. Isso pode tornar a solução de problemas de produção mais complexa, pois um resultado.

No entanto, vale a pena considerar essa opção se ela ajudar você a se mover longe de um microsserviço central estar sob propriedade coletiva, especialmente quando você não pode justificar a execução de várias variações do mesmo microsserviço.

Alterar avaliações

Ao adotar uma abordagem interna de código aberto, o conceito da revisão é um princípio fundamental - a mudança deve ser revisada antes que a mudança possa ser aceito. Mesmo quando trabalha dentro de uma equipe em uma base de código onde você tenha permissões de confirmação direta, ainda vale a pena ter suas alterações revisada

Sou um grande fã de ter minhas alterações revisadas. Sempre achei que meu código se beneficiou de um segundo par de olhos. De longe, minha forma preferida de revisão é o tipo de avaliação imediata que você recebe como parte da programação em pares. Você e outro desenvolvedor escrevem o código juntos e discutem as mudanças um com o outro. Ele é revisado antes de você fazer o check-in.

Você não precisa acreditar apenas na minha palavra. Voltando ao Accelerate, um livro já mencionamos várias vezes:

Descobrimos que a aprovação somente para mudanças de alto risco não estava correlacionada com desempenho de entrega de software. Equipes que não relataram aprovação o processo ou a revisão por pares usados obtiveram maior entrega de software desempenho. Finalmente, equipes que precisavam da aprovação de um órgão externo alcançou menor desempenho.

Aqui vemos uma distinção entre revisão por pares e mudança externa revisão. Uma revisão de mudança por pares é feita por alguém que provavelmente está em a mesma equipe que você e que trabalha na mesma base de código que você. Eles são obviamente, estão em melhor posição para avaliar o que contribui para uma boa mudança e também estão provavelmente realizará a revisão mais rapidamente (mais sobre isso em breve). Um a revisão externa, no entanto, é sempre mais complicada. Como o indivíduo é externo para sua equipe, eles provavelmente avaliarão a mudança com base em uma lista de critérios isso pode ou não fazer sentido, e como eles estão em uma equipe separada, eles pode não fazer suas alterações por um tempo. Como observam os autores do Accelerate:

Quais são as chances de um corpo externo, não intimamente familiarizado com a parte interna de um sistema, pode revisar dezenas de milhares de linhas de código altere potencialmente centenas de engenheiros e determine com precisão o impacto em um sistema de produção complexo?

Então, em geral, queremos fazer uso de avaliações de mudança por pares e evitar a necessidade de revisões externas de código.

Revisões de código síncronas versus assíncronas

Com a programação em pares, a revisão do código acontece em linha no momento em que o código é executado está sendo escrito. Na verdade, é mais do que isso. Ao emparelhar, você tem o driver (a pessoa no teclado) e o navegador (que está atuando como um segundo par dos olhos). Ambos os participantes estão em um diálogo constante sobre as mudanças que eles estão fazendo - o ato de revisar e o ato de fazer as mudanças são acontecendo ao mesmo tempo. A revisão se torna implícita e contínua aspecto da relação de emparelhamento. Isso significa que, à medida que as coisas são vistas, elas são imediatamente corrigidos.

Se você não estiver emparelhando, o ideal seria que a revisão acontecesse muito rapidamente depois de escrever o código. Então você gostaria que a avaliação em si fosse como o mais síncrono possível. Você quer poder discutir diretamente com o revisor quaisquer problemas que eles tenham, concorde em um caminho a seguir, faça a muda e segue em frente.

Quanto mais rápido você receber feedback sobre uma alteração de código, mais rápido você poderá analisar o feedback, avalie-o, peça esclarecimentos, discuta o assunto mais detalhadamente, se necessário, e, finalmente, faça as alterações necessárias. Quanto mais tempo demora entre enviar a alteração do código para revisão e a revisão realmente acontecendo, o as coisas se tornam mais longas e difíceis.

Se você enviar uma alteração de código para análise e não receber feedback sobre isso mude até vários dias depois, você provavelmente passou para outro trabalho. Para processar o feedback, você precisará mudar de contexto e se envolver novamente com o trabalho que você fez anteriormente. Você pode concordar com as alterações do revisor (se obrigatório), caso em que você pode fazê-las e reenviar a alteração para aprovação. Na pior das hipóteses, talvez seja necessário ter uma discussão mais aprofundada sobre o

pontos sendo levantados. Essa ida e volta assíncrona entre o remetente e o revisor pode adicionar dias ao processo de fazer alterações.

FAÇA REVISÕES DE CÓDIGO IMEDIATAMENTE!

Se você quiser fazer revisões de código e não estiver programando em pares, faça a revisão tão rápido quanto possível após o envio da alteração e passar pelo feedback de forma síncrona a da maneira mais possível, de preferência cara a cara com o revisor.

Programação em conjunto

A programação em conjunto (também conhecida como programação mob 14) às vezes é discutida como uma forma de fazer uma revisão de código em linha. Com programação em conjunto, um maior grupo de pessoas (talvez toda a equipe) trabalha em conjunto em uma mudança. É principalmente sobre trabalhar coletivamente em um problema e receber contribuições de um grande número de indivíduos.

Das equipes com quem falei que fazem uso da programação em conjunto, a maioria use-o apenas ocasionalmente para problemas específicos e complicados ou mudanças importantes, mas muito desenvolvimento também é feito fora do exercício conjunto. Como tal, embora o exercício de programação em conjunto provavelmente fornecesse revisão suficiente das mudanças que estão sendo feitas durante o próprio conjunto, e em de uma forma muito síncrona, você ainda precisaria de uma maneira de garantir uma revisão das mudanças feitas fora da multidão.

Alguns argumentam que você só precisa fazer uma revisão das mudanças de alto risco, e portanto, fazer avaliações apenas como parte de um conjunto é suficiente. Vale a pena observando que os autores de Accelerate surpreendentemente não encontraram nenhuma correlação entre o desempenho da entrega de software e a revisão somente de alto risco mudanças, em comparação com uma correlação positiva quando todas as mudanças são iguais revisado. Então, se você quiser programar em conjunto, vá em frente! Mas você pode também querer considerar a revisão de outras mudanças feitas fora do conjunto.

Pessoalmente, tenho algumas reservas profundas sobre alguns aspectos do conjunto programação. Você descobrirá que sua equipe é, na verdade, um grupo neurodiverso, e desequilíbrios de poder no conjunto podem minar ainda mais o objetivo de

resolução coletiva de problemas. Nem todo mundo se sente confortável trabalhando em grupo, e um conjunto é definitivamente isso. Algumas pessoas prosperarão em tal meio ambiente, enquanto outros se sentirão totalmente incapazes de contribuir. Quando eu tenho levantei essa questão com alguns proponentes da programação em conjunto, recebi uma variedade de respostas, muitas das quais se resumem à crença de que se você crie o ambiente de conjunto certo, qualquer pessoa poderá "sair de sua casca e sua contribuição." Digamos que depois desse conjunto específico de conversando, revirei tanto os olhos que quase fiquei cega. Para ser justo, essas mesmas preocupações também podem ser levantadas sobre a programação em pares!

Embora eu não tenha dúvidas de que muitos proponentes da programação em conjunto não serão por mais inconsciente ou surdo que isso, é importante lembrar que criar um espaço de trabalho inclusivo trata, em parte, de entender como criar um ambiente em que todos os membros da equipe possam contribuir plenamente uma forma segura e confortável para eles. E não se engane com isso porque você tem todo mundo em uma sala, todo mundo está realmente contribuindo. Se você gostaria de algumas dicas concretas sobre programação de conjuntos, então eu sugiro lendo o conjunto conciso e autopublicado de Maaret Pyhäjärvi.

Guia de programação. 15

O Serviço Órfão

E quanto aos serviços que não estão mais sendo mantidos ativamente? Como nós avance em direção a arquiteturas mais refinadas, os próprios microsserviços fique menor. Uma das vantagens dos microsserviços menores, pois temos discutido, é o fato de serem mais simples. Microsserviços mais simples com menos a funcionalidade pode não precisar ser alterada por um tempo. Considere os humildes Microsserviço Shopping Basket, que fornece alguns serviços bastante modestos capacidades: Adicionar à cesta, remover da cesta e assim por diante. É bastante é concebível que esse microsserviço não precise mudar por meses depois sendo escrito pela primeira vez, mesmo que o desenvolvimento ativo ainda esteja em andamento. O que acontece aqui? Quem é o dono desse microsserviço?

Se as estruturas da sua equipe estiverem alinhadas ao longo dos contextos limitados de seu organização, então, mesmo os serviços que não são alterados com frequência ainda têm um

proprietário de fato. Imagine uma equipe alinhada com as vendas online para consumidores contexto. Pode lidar com a interface de usuário baseada na web e com o Shopping Microsserviços Basket e Recommendat. Mesmo que o serviço de carrinho não tenha sido alterado em meses, naturalmente caberia a essa equipe fazer mudanças se necessário. Um dos benefícios dos microsserviços, é claro, é que, se a equipe precisa alterar o microsserviço para adicionar um novo recurso e não encontra o recurso ao seu gosto, reescrevê-lo não deve demorar muito.

Dito isso, se você adotou uma abordagem verdadeiramente poliglota e está fazendo uso de várias pilhas de tecnologia e, em seguida, os desafios de fazer alterações em um o serviço órfão pode ser agravado se sua equipe não conhece a tecnologia empilhe por mais tempo.

Estudo de caso: realestate.com.au

Na primeira edição deste livro, passei algum tempo conversando com realestate.com.au (REA) sobre o uso de microsserviços e muito do que eu aprendi ajudou muito em termos de compartilhamento de exemplos reais de microsserviços em ação. Também descobri que a interação organizacional da REA a estrutura e a arquitetura devem ser especialmente fascinantes. Esta visão geral de seu a estrutura organizacional é baseada em nossas discussões em 2014.

Tenho certeza de que a aparência do REA hoje é bem diferente. Esta visão geral representa um instantâneo, um ponto no tempo. Não estou sugerindo que isso seja o melhor maneira de estruturar uma organização - só que foi a que funcionou melhor para a REA em a hora. Aprender com outras organizações é sensato; copiar o que elas fazem sem entender por que eles fazem isso é tolice

Como acontece hoje, o negócio principal de imóveis da REA abrangeu diferentes facetas. Em 2014, a REA foi dividida em linhas de negócios independentes (LOBs). Por exemplo, uma linha de negócios lidava com propriedades residenciais em Austrália, outra com propriedade comercial, e outra linha administrava uma das Negócios da REA no exterior. Essas linhas de negócios tinham equipes de entrega de TI (ou "esquadrões") associados a eles; apenas alguns tinham um único esquadrão, enquanto a maior linha tinha quatro esquadrões. Então, para propriedades residenciais, havia vários

equipes envolvidas na criação do site e na listagem de serviços para permitir pessoas para procurar propriedades. As pessoas alternavam entre essas equipes de vez em quando. e depois, mas tendia a permanecer nessa linha de negócios por longos períodos, garantindo que os membros da equipe possam desenvolver uma forte consciência dessa parte do domínio. Isso, por sua vez, ajudou na comunicação entre os vários partes interessadas da empresa e a equipe que fornece recursos para elas.

Esperava-se que cada esquadrão dentro de uma linha de negócios fosse dono de toda a vida ciclo de cada serviço criado, incluindo construção, teste e lançamento, apoio e até mesmo desmantelamento. A equipe principal de serviços de entrega tinha o trabalho de fornecer aconselhamento, orientação e ferramentas aos esquadrões dos LOBs, ajudando esses esquadrões a trabalharem com mais eficiência. Usando nossa terminologia mais recente, a equipe principal de serviços de entrega estava desempenhando o papel de uma equipe capacitadora. UMA uma forte cultura de automação foi fundamental, e a REA fez uso intenso da AWS como parte importante para permitir que as equipes sejam mais autônomas. Figura 15-6 ilustra como tudo isso funcionou.

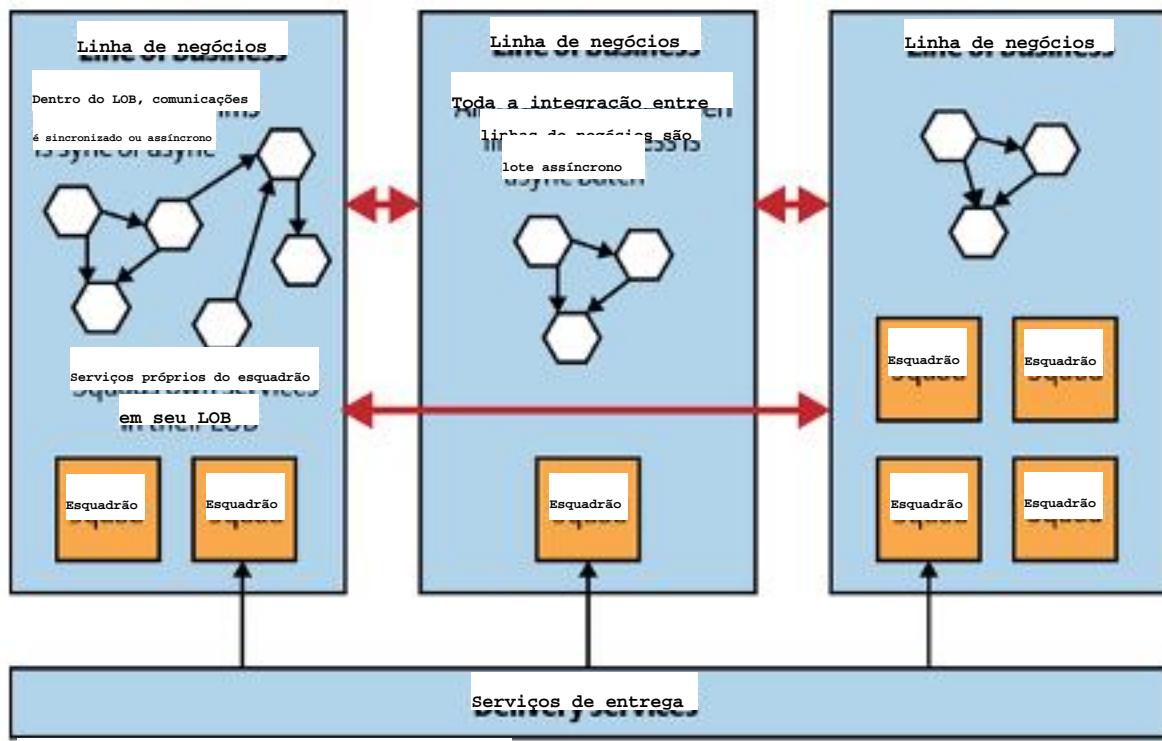


Figura 15-6. Uma visão geral da estrutura e alinhamento organizacional e da equipe de realestate.com.au com arquitetura

Não foi apenas a organização de entrega que estava alinhada à forma como a empresa estava alinhada. operado. Esse modelo também se estendeu à arquitetura. Um exemplo disso foram os métodos de integração. Dentro de um LOB, todos os serviços eram gratuitos para conversar uns aos outros da maneira que achassem melhor, conforme decidido pelos esquadrões que atuam como seus guardiões. Mas a comunicação entre LOBs foi obrigada a ser lote assíncrono, uma das poucas regras de ferro fundido dos muito pequenos equipes de arquitetura. Essa comunicação grosseira correspondia àquele também existia entre as diferentes partes do negócio. Ao insistir nisso sendo em lote, cada LOB tinha muita liberdade em como atuava e como gerenciou a si mesmo. Podia se dar ao luxo de desativar seus serviços sempre que quisesse, sabendo que, desde que pudesse satisfazer a integração do lote com outras peças da empresa e de suas próprias partes interessadas, ninguém se importaria.

Essa estrutura permitiu uma autonomia significativa, não apenas entre as equipes, mas também entre as diferentes partes do negócio, e a capacidade de entregar a mudança ajudou a empresa a alcançar um sucesso significativo no mercado local. Essa estrutura mais autônoma também ajudou a empresa a crescer a partir de uma punhado de serviços em 2010 para centenas até 2014, facilitando a capacidade de oferecer mudanças mais rapidamente.

As organizações que são adaptáveis o suficiente para mudar não apenas seu sistema de arquitetura, mas também sua estrutura organizacional, podem colher enormes benefícios em termos de maior autonomia das equipes e menor tempo de comercialização de novos recursos e funcionalidades. A REA é apenas uma das várias organizações que perceberam que a arquitetura do sistema não existe no vácuo.

Distribuição geográfica

As equipes colocadas descobrirão que a comunicação síncrona é muito simples, especialmente porque eles estão normalmente no mesmo lugar ao mesmo tempo. Se o seu time é distribuído, a comunicação síncrona pode ser mais difícil, mas ainda é possível se os membros da equipe estiverem no mesmo fuso horário ou em fusos horários semelhantes. Ao se comunicar com pessoas em vários fusos horários diferentes, o custo de coordenação pode aumentar drasticamente. Em uma das minhas funções anteriores, trabalhei como arquiteto, ajudando equipes de suporte baseadas na Índia, no Reino Unido, no Brasil e na

EUA, enquanto eu mesmo morava na Austrália. Marcando uma reunião entre... Eu e a liderança das várias equipes fomos incrivelmente difíceis. Isso significava que realizávamos essas reuniões com pouca frequência (normalmente mensalmente) e também tínhamos para se certificar de discutir apenas as questões mais importantes durante essas sessões, já que, frequentemente, mais da metade dos participantes trabalhava fora de sua sede horas de trabalho.

Fora dessas sessões, teríamos comunicação assíncrona, principalmente por e-mail, sobre outras questões menos urgentes. Mas comigo dentro Austrália, o atraso nessa forma de comunicação foi significativo. Eu gostaria acorde na segunda-feira de manhã para ter um início de semana bem tranquilo, pois a maior parte do mundo não tinha acordado - isso me daria tempo para processar os e-mails que recebi de equipes no Reino Unido, Brasil e EUA em seus Sexta-feira à tarde, que foi minha manhã de sábado.

Lembro-me de um projeto de cliente em que trabalhei, no qual a propriedade de um único microserviço foi compartilhado entre duas localizações geográficas. Eventualmente, cada site começou a se especializar no trabalho que realizava. Isso permitiu que ele realizasse propriedade de parte da base de código, dentro da qual poderia ter um custo mais fácil de mudança. As equipes então tiveram uma comunicação mais grosseira sobre como as duas partes se inter-relacionaram; efetivamente, os caminhos de comunicação tornado possível dentro da estrutura organizacional, correspondeu à granulação grosseira API que formou o limite entre as duas metades da base de código.

Então, onde isso nos deixa quando pensamos em desenvolver nosso próprio serviço? design? Bem, eu sugeriria que os limites geográficos entre as pessoas envolvidas com o desenvolvimento deve ser uma forte consideração quando procurando definir os limites da equipe e do software. É muito mais fácil formar uma única equipe quando seus membros estão localizados. Se a colocação não é possível, e você está procurando formar uma equipe distribuída, então garantir que os membros da equipe estejam no mesmo fuso horário ou em fusos horários muito semelhantes será auxiliará na comunicação dentro dessa equipe, pois isso reduzirá a necessidade de comunicação assíncrona

Talvez sua organização decida que deseja aumentar o número de pessoas que trabalham em seu projeto abrindo um escritório em outro país.

Neste ponto, você deve pensar ativamente sobre quais partes do seu sistema podem ser movido. Talvez seja isso que orienta suas decisões sobre o que funcionalidade a ser dividida em seguida.

Também é importante notar neste momento que, pelo menos com base nas observações dos autores de "Explorando a dualidade entre produto e Arquiteturas organizacionais", relatório que mencionei anteriormente, se a organização a construção do sistema é mais frouxamente acoplada (por exemplo, consiste em equipes distribuídas geograficamente), os sistemas que estão sendo construídos tenderão a ser mais modular e, portanto, esperançosamente menos acoplado. A tendência de um solteiro Uma equipe que possui muitos serviços para se inclinar para uma integração mais estreita é muito difícil para manter em uma organização mais distribuída.

Lei de Conway ao contrário

Até agora, falamos sobre como a organização afeta o design do sistema. Mas e o contrário? Ou seja, um design de sistema pode alterar a organização? Embora eu não tenha conseguido encontrar a mesma qualidade de evidência para apoiar a ideia de que a lei de Conway funciona ao contrário, eu vi anedoticamente.

Provavelmente, o melhor exemplo foi um cliente com quem trabalhei há muitos anos. Voltar na época em que a web era bastante incipiente e a internet era vista como algo que chegou em um disquete da AOL pela porta, esta empresa era uma grande gráfica com um site pequeno e modesto. Tinha um site porque essa era a coisa a fazer, mas no grande esquema das coisas, o site era pouco importante para a forma como a empresa operava. Quando o sistema original foi criada, uma decisão técnica bastante arbitrária foi tomada sobre como o sistema funcionaria.

O conteúdo desse sistema foi obtido de várias maneiras, mas a maior parte veio de terceiros que estavam colocando anúncios para visualização pelo público em geral. Havia um sistema de entrada que permitia que o conteúdo fosse criado pelo pagador terceiros, um sistema central que pegou esses dados e os enriqueceu em vários formas, e um sistema de saída que criou o site final que o general público poderia navegar.

Se as decisões de design originais estavam corretas no momento é uma conversa para historiadores, mas muitos anos na empresa mudaram bastante, e eu e muitos dos meus colegas estavam começando a se perguntar se o design do sistema era adequado ao estado atual da empresa. Seu negócio de impressão física havia diminuído significativamente, e as receitas e, portanto, as operações comerciais da organização agora era dominada por sua presença on-line.

O que vimos naquela época foi uma organização fortemente alinhada a essas três sistemas de peças. Três canais ou divisões no lado de TI da empresa alinhado com cada uma das partes de entrada, núcleo e saída do negócio. Dentro nesses canais, havia equipes de entrega separadas. O que eu não percebi em a época era que essas estruturas organizacionais não eram anteriores ao sistema design, mas na verdade cresceu em torno dele. Como o lado impresso da empresa diminuiu e o lado digital do negócio cresceu, o design do sistema inadvertidamente, traçou o caminho para o crescimento da organização.

No final, percebemos que, quaisquer que sejam as deficiências do design do sistema se, teríamos que fazer mudanças na estrutura organizacional para fazer uma mudança. A empresa agora mudou muito, mas isso aconteceu durante um período de muitos anos.

Pessoas

Não importa como pareça à primeira vista, é sempre um problema de pessoas.

-Gerry Weinberg, A Segunda Lei da Consultoria

Temos que aceitar que, em um ambiente de microserviços, é mais difícil para um desenvolvedor para pensar em escrever código em seu próprio mundinho. Eles têm que esteja mais ciente das implicações de coisas como chamadas em toda a rede, limites ou implicações do fracasso. Também falamos sobre a habilidade de microserviços para facilitar o teste de novas tecnologias, a partir de dados armazena em dois idiomas. Mas se você está se movendo de um mundo em que você tem um sistema monolítico, onde a maioria de seus desenvolvedores teve que usar apenas um idioma e permaneceram completamente alheios ao operacional.

preocupações, lançá-las no mundo dos microserviços pode ser rude. ...
despertar para eles.

Da mesma forma, impulsionar as equipes de desenvolvimento para aumentar a autonomia pode seja tenso. Pessoas que no passado jogaram o trabalho por cima da parede para outra pessoa está acostumada a ter outra pessoa para culpar e talvez não sinta-se confortável em ser totalmente responsável por seu trabalho. Você pode até encontrar barreiras contratuais para que seus desenvolvedores carreguem pagers para os sistemas eles apoiam! Essas mudanças podem ser feitas de forma gradual e, inicialmente, fará sentido procurar mudar as responsabilidades das pessoas que são mais disposto e capaz de fazer a mudança.

Embora este livro tenha sido principalmente sobre tecnologia, as pessoas não são apenas consideração secundária; eles são as pessoas que construíram o que você tem agora e construirá o que acontecerá a seguir. Criando uma visão de como as coisas deveriam seja feito sem considerar como sua equipe atual se sentirá sobre isso e sem considerar quais capacidades eles têm, é provável que leve a um problema lugar.

Cada organização tem seu próprio conjunto de dinâmicas em torno desse tópico. Compreendo o apetite de sua equipe por mudanças. Não os empurre muito rápido! Talvez você possa faça com que uma equipe separada cuide do suporte ou da implantação na linha de frente por um curto período período, dando aos desenvolvedores tempo para se adaptarem às novas práticas. Você pode, no entanto, tem que aceitar que você precisa de diferentes tipos de pessoas em seu organização para fazer tudo isso funcionar. Você provavelmente precisará mudar a forma como você contrate - na verdade, pode ser mais fácil mostrar o que é possível trazendo novos pessoas de fora que já têm experiência em trabalhar da maneira que você quero. As novas contratações certas podem muito bem ajudar muito a mostrar aos outros o que é possível.

Seja qual for a sua abordagem, entenda que você precisa ser claro ao articular as responsabilidades de seu pessoal em um mundo de microserviços, e também ser esclareça por que essas responsabilidades são importantes para você. Isso pode ajudar você vê quais podem ser suas lacunas de habilidades e pensa em como fechá-las. Para muitos, esta será uma jornada muito assustadora. Basta lembrar que sem pessoas,

a bordo, qualquer mudança que você queira fazer pode ser condenada pelo
começar.

Resumo

A lei de Conway destaca os perigos de tentar impor um projeto de sistema que não corresponde à organização. Isso nos indica, pelo menos para microserviços, rumo a um modelo em que a forte propriedade de microserviços seja a norma. Compartilhando microserviços ou tentando praticar a propriedade coletiva em a escala geralmente pode fazer com que prejudiquemos os benefícios dos microserviços.

Quando a organização e a arquitetura não estão alinhadas, ficamos tensos pontos, conforme descrito ao longo deste capítulo. Ao reconhecer a ligação entre os dois, vamos garantir que o sistema que estamos tentando construir faça sentido a organização para a qual estamos construindo.

Se você quiser explorar este tópico mais detalhadamente, além do anterior mencionei Team Topologies, eu também posso recomendar completamente a palestra "Scale, Microservices and Flow", de James Lewis, 16 anos, do qual ganhei muitos insights que ajudaram a moldar este capítulo. Vale a pena assistir se você está interessado em se aprofundar em algumas das ideias que abordei neste capítulo.

Em nosso próximo capítulo, exploraremos com mais profundidade um tópico que eu já abordei abordou o papel do arquiteto.

1 Nicole Forsgren, Jez Humble e Gene Kim, *Accelerate: a ciência da construção e Dimensionando organizações de tecnologia de alto desempenho* (Portland, OR: IT Revolution, 2018).

2 Alan MacCormack, Carliss Baldwin e John Rusnak, "Explorando a dualidade entre produtos e Arquiteturas organizacionais: um teste da hipótese de espelhamento", *Research Policy* 41, no. 8 (outubro de 2012): 1309-24.

3 Nachiappan Nagappan, Brendan Murphy e Victor Basili, "A influência da organização Estrutura sobre qualidade de software: um estudo de caso empírico", *ICSE '08: Anais do 30º Conferência Internacional sobre Engenharia de Software* (Nova York: ACM, 2008).

4 E todos sabemos que o Windows Vista era bastante propenso a erros!

- 5 Daniel Rodriguez et al., "Descobertas empíricas sobre o tamanho da equipe e a produtividade em software Desenvolvimento", *Journal of Systems and Software*, 85, nº 3 (2012).
doi.org/10.1016/j.jss.2011.09.009.
- 6 Frederick P. Brooks Jr., *O mítico homem-mês: ensaios sobre engenharia de software*, Ed. de aniversário. (Boston: Addison-Wesley, 1995).
- 7 John Rossman, *Pense como a Amazon: 50 1/2 ideias para se tornar um líder digital* (Nova York: McGraw-Hill, 2019).
- 8 Matthew Skelton e Manuel Pais, *Team Topologies* (Portland OR: IT Revolution, 2019).
- 9 Famosa pelo desenvolvimento do material impermeável Gore-Tex
- 10 Que, notoriamente, nem o Spotify usa mais.
- 11 Emily Webber, *Construindo comunidades de prática bem-sucedidas* (San Francisco: Blurb, 2016).
- 12 Paul Ingles, "Convergence to Kubernetes", Medium, 18 de junho de 2018, <https://oreil.ly/Ho7kY>.
- 13 Equipe "Evolução organizacional para acelerar a entrega de serviços de computação na Uswitch" Topologias, 24 de junho de 2020, <https://oreil.ly/zoyvv>.
- 14 Embora o termo programação de multidões seja mais difundido, não gosto da imagem que isso pinta, preferindo, em vez disso, o termo conjunto, pois deixa claro que temos uma coleção de pessoas trabalhando juntas, em vez de uma coleção de pessoas jogando coquetéis molotov e quebrando janelas. Não tenho 100% de certeza de quem inventou essa mudança de nome, mas acho que foi Maaret Pyhäjärvi
- 15 Maaret Pyhäjärvi, *Guia de programação de conjuntos* (autopublicação, 2015-2020).
<https://ensembleprogramming.xyz>
- 16 James Lewis, "Escala, microsserviços e fluxo". COMO! Conferências, 10 de fevereiro de 2020, Vídeo do YouTube, 51:03, <https://oreil.ly/ON81J>.

Capítulo 16. O evolucionário Arquiteto

Como vimos até agora, os microserviços nos oferecem muitas opções e, consequentemente, muitas decisões a serem tomadas. Por exemplo, quantos diferentes tecnologias que devemos usar, devemos permitir que equipes diferentes usem diferentes expressões idiomáticas de programação, e devemos dividir ou mesclar um microserviço? Como fazer isso sem comprometer a consistência? Vamos tomar essas decisões? Com o ritmo mais rápido da mudança e o ambiente mais fluido que essas arquiteturas permitem, o papel do arquiteto também tem que mudar. Neste capítulo, você terá uma visão bastante opinativa do que o papel de um arquiteto é, e espero, lançar um ataque final ao marfim torre.

O que há em um nome?

Você continua usando essa palavra. Eu não acho que isso signifique o que você pensa significa.

--Inigo Montoya, de A Princesa Noiva

Os arquitetos têm um trabalho importante. Eles são responsáveis por garantir que o sistema tem uma visão técnica conjunta, que deve ajudar a fornecer o software que os clientes precisam. Em alguns lugares, eles podem ter que trabalhar com apenas uma equipe, caso em que o papel do arquiteto e o do técnico os chumbo geralmente são a mesma coisa. Em outros lugares, eles podem estar definindo a visão de todo um programa de trabalho, coordenando com várias equipes em todo o mundo, ou talvez até mesmo em uma organização inteira. Em qualquer nível, arquitetos operam, seu papel é difícil de definir e, apesar disso, muitas vezes sendo a progressão óbvia na carreira para desenvolvedores corporativos. Organizações, também é uma função que recebe mais críticas do que praticamente qualquer outra em nosso campo. Mais do que qualquer outra função, os arquitetos podem ter um impacto direto sobre a qualidade dos sistemas construídos, com base nas condições de trabalho de seus colegas,

e sobre a capacidade de sua organização de responder às mudanças, e ainda sobre seu papel parecer muito mal compreendido. Por que isso?

Nossa indústria é jovem. Parece que às vezes esquecemos que fomos criando programas que são executados no que reconhecemos como computadores por apenas 75% anos ou mais. Nossa profissão não se encaixa em uma caixa bonita que a sociedade, como todo entende. Não somos como eletricistas, encanadores, médicos, ou engenheiros. Quantas vezes você disse a alguém o que você faz em uma festa, só para a conversa parar? O mundo como um todo luta para entender o desenvolvimento de software, conforme descrevi várias vezes ao longo deste livro, muitas vezes parece que nós mesmos não o entendemos. Então, pegamos emprestado de outras profissões. Nós nos chamamos de software "engenheiros" ou "arquitetos". Mas não somos arquitetos ou engenheiros no caminho que a sociedade comprehenda essas profissões. Arquitetos e engenheiros têm um rigor e disciplina com os quais só poderíamos sonhar, e sua importância na sociedade é bem compreendido. Lembro-me de conversar com um amigo meu um dia antes dele tornou-se arquiteto qualificado. "Amanhã", disse ele, "se eu lhe der um conselho no bar sobre como construir algo e isso está errado, eu me pergunto conta. Eu poderia ser processado, pois, aos olhos da lei, agora sou qualificado arquiteto e eu devemos ser responsabilizados se eu errar." A importância desses empregos para a sociedade significa que são necessárias qualificações: pessoas. Tenho que conhecer. No Reino Unido, por exemplo, um mínimo de sete anos de estudo é necessário antes que você possa ser chamado de arquiteto. Mas esses empregos também são baseados em um corpo de conhecimento que remonta a milhares de anos. E software arquitetos? Não é bem assim. Em parte, é por isso que vejo muitas formas de TI. A certificação é inútil, pois sabemos muito pouco sobre a aparência de "bom". Não digo isso para menosprezar o termo engenharia de software, cunhado na Década de 1960, de Margaret Hamilton, mas era tão ambiciosa quanto sobre a realidade atual. O termo surgiu como um chamado para melhorar a qualidade do software sendo criado, e em reconhecimento ao fato de que projetos de software muitas vezes falharam e, no entanto, estavam sendo cada vez mais usados em missões vitais- e campos críticos de segurança. Muito trabalho foi feito para melhorar a situação. desde então, mas minha opinião depois de 20 anos no setor é que ainda Tenho muito a aprender sobre como fazer um bom (ou pelo menos um melhor) trabalho.

Parte de nós quer reconhecimento, então pegamos emprestados nomes de outras profissões que já têm o reconhecimento que desejamos. Mas isso pode ser problemático, se emprestamos práticas de trabalho dessas profissões sem entender a mentalidade por trás deles ou levando em consideração como é o desenvolvimento de software diferente, digamos, da engenharia civil. Nada disso deve ser considerado um argumento de que não devemos ter como objetivo ter mais rigor em nosso trabalho, apenas que não pode simplesmente pegar ideias de outros lugares e presumir que elas funcionarão para nós. Nosso setor é muito jovem e o desafio é que temos muito menos absolutos em torno dos quais concordamos como indústria.

Talvez o termo arquiteto, ou pelo menos o entendimento comum do que os arquitetos fazem, causaram o maior dano dessa maneira: a ideia de alguém que elabora planos detalhados para que outros interpretem e esperam que sejam executados fora; o equilíbrio entre parte artista, parte engenheiro, supervisionando a criação do que geralmente é uma visão singular, com todos os outros pontos de vista sendo subservientes, exceto pela objeção ocasional do engenheiro estrutural em relação às leis da física. Em nossa indústria, essa visão do arquiteto leva a algumas coisas terríveis práticas, com arquitetos criando diagrama após diagrama, página após página de documentação, com o objetivo de informar a construção do sistema perfeito, sem levar em conta o futuro fundamentalmente desconhecido, e totalmente desprovidos de qualquer compreensão de quão difíceis serão seus planos implementar, ou se eles realmente funcionarão ou não, muito menos ter alguma capacidade de mudar à medida que aprendemos mais.

Mas os arquitetos do ambiente construído estão operando em um reino diferente do a dos arquitetos de software. Suas restrições são diferentes, o produto final diferente. O custo da mudança é muito maior na construção do que na desenvolvimento de software. Você não pode desenterrar concreto, mas você pode mudar o código, e até mesmo a infraestrutura em que executamos nosso código é muito mais maleável do que antes, gracias à virtualização. Os edifícios são bastante fixos uma vez construídos - eles pode ser alterado, expandido ou reduzido, mas os custos associados são muito alto. Mas esperamos que nosso software mude continuamente para atender às nossas necessidades. Então, se a arquitetura de software é diferente da arquitetura construída ambiente, talvez devêssemos ser um pouco mais claros em termos de qual software arquitetura, na verdade, é.

O que é arquitetura de software?

Uma das definições mais famosas de arquitetura de software vem por meio de um e-mail do Ralph Johnson. "Arquitetura é sobre coisas importantes. Seja o que for. "2 Então, isso significa que qualquer coisa importante é feita pelo arquiteto? Isso significa que todos os outros trabalhos que estão sendo feitos não são importantes? O problema com essa declaração frequentemente citada é que ela é frequentemente usada isoladamente, sem qualquer compreensão da resposta mais ampla com a qual Ralph a compartilhou. Em primeiro lugar, está claro que ele está falando da perspectiva de um software desenvolvedor. Ele continua dizendo:

Então, uma definição melhor seria "Na maioria dos projetos de software bem-sucedidos, os desenvolvedores especialistas que trabalham nesse projeto compartilharam um entendimento compartilhado de como o sistema é dividido em componentes e como os componentes interagem por meio de interfaces. Esse entendimento inclui somente os componentes e interfaces que são compreendidos por todos os desenvolvedores."

Essa seria uma definição melhor porque deixa claro que arquitetura é uma construção social (bem, software também é, mas arquitetura é ainda mais) porque não depende apenas do software, mas de qual parte do software é considerada importante pelo consenso do grupo.

Aqui, Ralph está usando o termo componentes em seu sentido mais geral. No contexto deste livro, podemos pensar nos componentes como nossos microsserviços, e talvez os módulos dentro desses microsserviços.

A arquitetura de software é a forma do sistema. A arquitetura acontece, por projeto ou acidente. Tomamos uma série de decisões ad hoc e acabamos com os resultados - sem pensar nas coisas em termos de arquitetura, nós Mesmo assim, acabo com a arquitetura. A arquitetura às vezes pode ser o que acontece enquanto estamos ocupados fazendo outros planos.

Um arquiteto dedicado é alguém que deve ver e entender esse todo sistema, entenda as forças que atuam sobre ele. Eles precisam garantir que haja um

visão da arquitetura que é adequada ao propósito e é claramente compreendida. uma visão arquitetônica que satisfaça as necessidades do sistema e de seus usuários, como bem como as das pessoas que trabalham no próprio sistema. Olhando apenas uma faceta - por exemplo, lógica, mas não física, forma, mas não experiência do desenvolvedor - limita a eficácia de um arquiteto. Se você aceitar que a arquitetura é sobre compreender o sistema e, em seguida, limitar o escopo daquilo que lhe interessa limita sua capacidade de raciocinar e fazer mudanças.

A arquitetura pode ser invisível para as pessoas que vivem com ela. Pode ser tão leve para não estar realmente lá. Pode ser algo que orienta e ajuda a alcançar o resultado certo. Pode ser sufocante e autoritário. Pode deliciar-se sem você perceber que é mesmo uma coisa, e esmaga o espírito de você sem nada malícia sendo intencional. Então, se a arquitetura é ou não "sobre o importante" coisas", certamente é importante.

Outra citação concisa que é frequentemente usada para definir a arquitetura de software vem do mesmo artigo em que Martin compartilha as opiniões de Ralph: "Então você pode acabar definindo arquitetura como coisas que as pessoas consideram difíceis de mudar." A ideia de Martin de que a arquitetura é algo difícil de mudar faz sentido em algum nível e nos traz de volta ao conceito de arquitetura no construído ambiente. Onde as coisas são mais difíceis de mudar, elas precisam de um pouco mais de atualização pensamento frontal para realmente garantir que estamos indo na direção certa. Mas aí é um problema em pegar uma definição simples de uma ideia complexa e executar com isso como uma definição funcional - se essa afirmação fosse inteiramente como você. Se pensasse em arquitetura de software, você perderia muita coisa. Sim, muitos arquitetura de software é pensar nas coisas que serão difíceis de fazer mudança, mas também se trata de criar espaço para permitir mudanças no design.

Tornando a mudança possível

Voltando ao mundo dos edifícios em vez dos sistemas de software: arquiteto Mies van der Rohe, sem dúvida, fez mais para ser pioneiro no que agora consideramos o arranha-céu moderno do que qualquer outro arquiteto - seu famoso Seagram Building tornou-se o modelo para grande parte do que se seguiu. Edifício Seagram difere de muito do que veio antes. As paredes externas do edifício são

não estruturais - eles envolvem uma estrutura externa de aço. Os principais serviços de construção-elevadores, 3 escadas, ar condicionado, água e resíduos e sistema elétrico
elevadores, 3 escadas, ar condicionado, água e resíduos e sistema elétrico
-passe por um núcleo central de concreto. Assista a um arranha-céu moderno
construído hoje, e é esse núcleo central de concreto que é construído primeiro, um gigante
quindaste frequentemente visto empoleirado no topo. Cada andar do Edifício Seagram não tem
paredes estruturais interiores - isso significa que você tem total flexibilidade em termos de
como o espaço é usado. Você pode reconfigurar o espaço como achar melhor, roteamento
fiação elétrica e ar condicionado para diferentes partes de cada andar via
tetos suspensos e dutos no próprio piso.

É interessante notar que o Seagram Building foi desenvolvido usando um
processo em que o design do edifício evoluiu durante a construção
foi realizado. Agora, onde vimos essa ideia antes?

A ideia com esse design era entregar o que Mies van der Rohe chamou
"espaço universal" - um grande volume de um único intervalo que pode ser reconfigurado para
atendem a diferentes necessidades. O uso de edifícios muda, então a ideia era entregar
espaço que seja o mais flexível possível em termos de como ele pode ser usado. Neste
De qualquer forma, Mies van der Rohe não teve que se concentrar apenas na estética fundamental
do prédio, encontrar um espaço para os principais serviços que seria difícil se não
impossível mudar mais tarde, mas ele também tinha que garantir que o prédio pudesse
ser usado de maneiras diferentes das originalmente previstas. Em breve, veremos
como permitimos mudanças no espaço de uma arquitetura de microserviços.

Uma visão evolutiva para o arquiteto

Nossos requisitos como arquitetos de software mudam mais rapidamente do que para
pessoas que projetam e constroem edifícios, assim como as ferramentas e técnicas da
nossa disposição. As coisas que criamos não são pontos fixos no tempo. Uma vez
lancado em produção, nosso software continuará a evoluir da mesma forma que
é usado para mudar. Para a maioria das coisas que criamos, temos que aceitar que, uma vez que o
o software chega às mãos de nossos usuários, teremos que reagir e nos adaptar
em vez de esperar um artefato que nunca muda. Portanto, os arquitetos de software precisam
deixe de pensar em criar o produto final e o foco perfeitos

em vez de ajudar a criar uma estrutura na qual os sistemas certos possam surgir e continue crescendo à medida que aprendemos mais.

Embora eu tenha passado grande parte do capítulo desencorajando comparações com outros profissões, há uma analogia que eu gosto quando se trata do papel da TI arquiteto e que eu acho que encapsula bem esse aspecto da função. Erik Doernenburg, da Thoughtworks, primeiro compartilhou comigo a ideia de que deveríamos pensar no papel do arquiteto mais como urbanista do que como arquiteto do construído meio ambiente. O papel do urbanista deve ser familiar para qualquer um de vocês que já jogaram SimCity ou Cities: Skylines antes. O papel de um urbanista é olhar para uma infinidade de fontes de informação e depois tentar otimizar o layout de uma cidade para melhor atender às necessidades dos cidadãos atuais, ao mesmo tempo que leva em consideração o uso futuro. A forma como influenciam a forma como a cidade evolui, no entanto, é interessante. Eles não dizem: "Construa este edifício específico lá"; em vez disso, eles definem zonas que permitem a tomada de decisões locais dentro certas restrições. Então, como no SimCity, você pode designar parte da sua cidade como uma zona industrial e outra parte como zona residencial. Cabe então a outras pessoas para decidir quais edifícios serão criados, mas existem restrições: se você Se quiser construir uma fábrica, ela precisará estar em uma zona industrial. Em vez de se preocupando demais com o que acontece em uma zona, o planejador urbano em vez disso, passe muito mais tempo descobrindo como as pessoas e as empresas de serviços públicos saem de uma zona para outra.

Mais de uma pessoa comparou uma cidade a uma criatura viva. A cidade muda ao longo do tempo. Ela muda e evolui à medida que seus ocupantes o usam de maneiras diferentes, ou como forças externas o moldam. O planejador da cidade faz o possível para antecipá-los muda, mas aceita que tentar exercer controle direto sobre todos os aspectos do que acontece é inútil. Portanto, nossos arquitetos, como planejadores urbanos, precisam definir uma direção em linhas gerais e envolva-se em ser altamente específico sobre a implementação, detalhes somente em casos limitados. Eles precisam garantir que o sistema seja adequado para propósito agora, mas também uma plataforma para o futuro.

A comparação com o software deve ser óbvia. À medida que nossos usuários usam nosso software, precisamos reagir e mudar. Não podemos prever tudo o que acontecerá e, portanto, em vez de planejar cada eventualidade, nós deve planejar para permitir a mudança, evitando o desejo de especificar demais cada

última coisa. Nossa cidade - o sistema - precisa ser um lugar bom e feliz para todo mundo que o usa.

Definindo limites do sistema

Para continuar com a metáfora do arquiteto como urbanista por um momento, quais são nossas zonas? Esses são nossos limites de microserviços, ou talvez grupos grosseiros de microserviços. Como arquitetos, precisamos nos preocupar muito menos sobre o que acontece dentro de uma zona e mais sobre o que acontece entre as zonas. Isso significa que precisamos gastar tempo pensando em como nossos microserviços conversam entre si e garantem que possamos monitorar adequadamente a saúde geral do nosso sistema. De um espaço de arquitetura, é assim que criar nosso próprio espaço universal - definindo alguns limites específicos, nós destacar para nossos colegas que constroem o sistema as áreas em que as mudanças podem ser feito de forma mais livre sem quebrar algum aspecto fundamental do nosso arquitetura

Para ver um exemplo muito simples, na Figura 16-1, vemos o

Recomendações: microserviço acessando informações do

Microserviços de promoções e vendas. Como já abordamos detalhadamente, somos livres para alterar a funcionalidade oculta dentro desses três microserviços sem me preocupar em quebrar o sistema geral - eu posso mude o que eu quiser em vendas ou promoções, desde que eu continue manter as expectativas que o Recommendations tem sobre como será interaja com esses microserviços downstream.

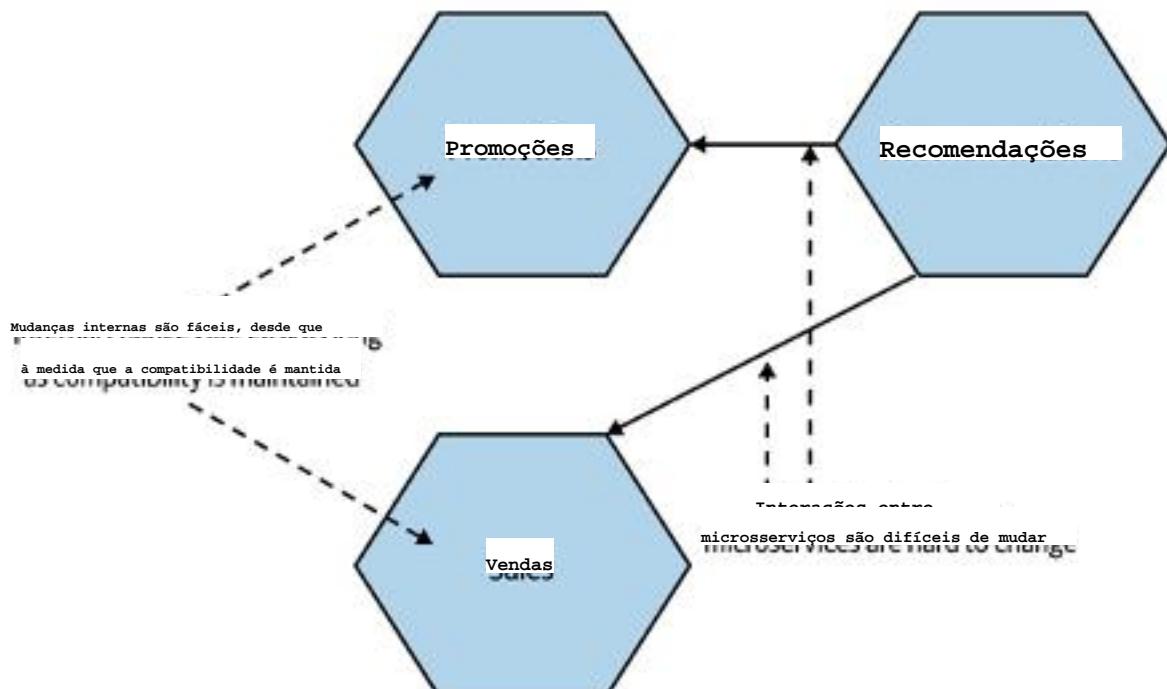


Figura 16-1. Mudanças dentro de um limite de microserviços são fáceis de fazer, desde que o interações entre microserviços desligados, mudanças

Também podemos criar espaço para mudanças em níveis mais amplos. Na Figura 16-2, vemos que os microserviços da Figura 16-1 realmente existem em um marketing zona que mapeia a responsabilidade de uma equipe específica. Definimos um esperado comportamento em termos de como a funcionalidade de marketing interage com a maior sistema. Dentro da zona de marketing, podemos fazer as alterações que quisermos, desde que à medida que a compatibilidade com o sistema maior é mantida. Voltando ao ideia de entender o que é difícil de mudar, organizacional as estruturas geralmente se enquadram nessa categoria e, como tal, as estruturas de equipe existentes pode ajudar a definir essas zonas para você. Coordenando mudanças dentro de uma equipe entre os microserviços pertencentes a essa equipe será mais fácil do que mudar o interações que são expostas a outras equipes.

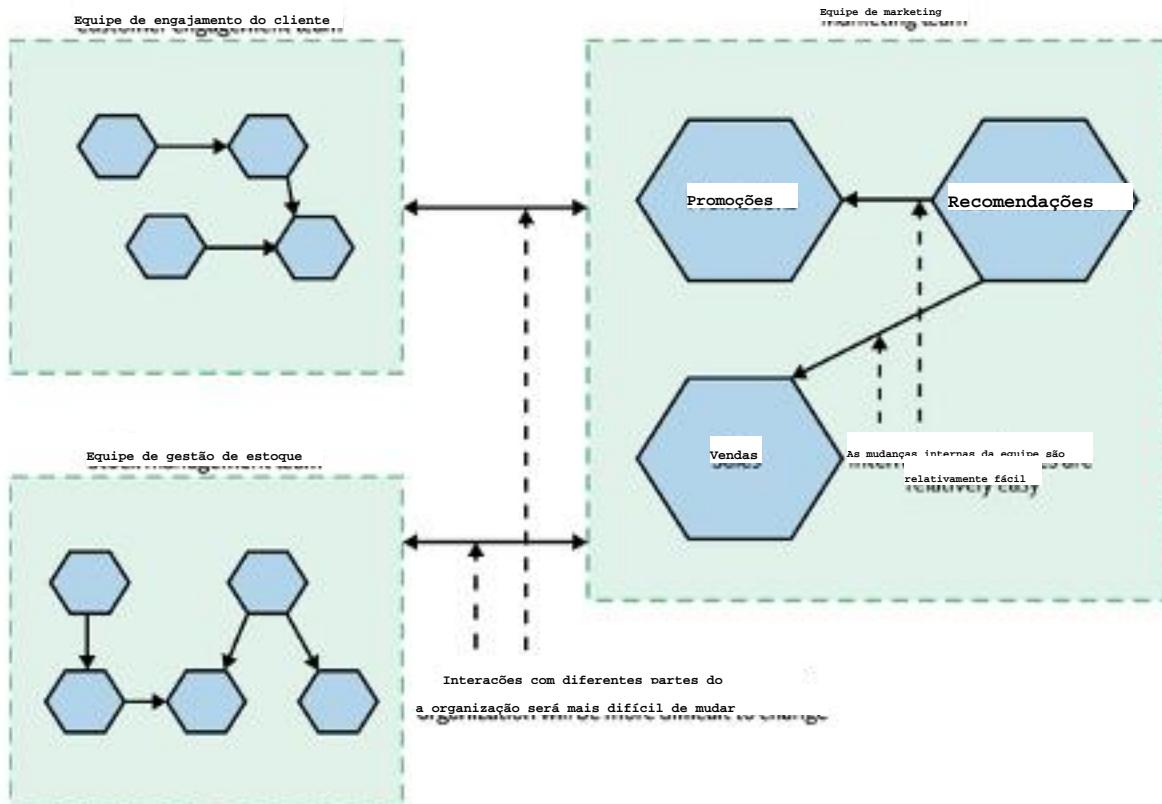


Figura 16-2 Mudanças dentro de uma zona são mais fáceis de fazer do que mudanças entre zonas

Isso se encaixa perfeitamente com o conceito de uma API de equipe, que discutimos em “Equipes pequenas, grande organização”. Um arquiteto pode ajudar a facilitar a criação de uma API de equipe, garantindo os microsserviços e as práticas de trabalho da equipe se encaixam na organização em geral.

Ao definir espaços nos quais essas mudanças podem ser feitas sem comprometendo o sistema como um todo, facilitamos a vida dos desenvolvedores e também concentre nossa atenção em partes do sistema que são mais difíceis de mudar. Lembra do conceito de ocultação de informações que exploramos no Capítulo 2? Conforme exploramos lá, ocultar informações dentro de um limite de microsserviços facilita muito a criação de uma interface estável para os consumidores. Quando nós fazemos alterações no microsserviço, é mais fácil garantir que não tenhamos quebrado a compatibilidade com consumidores externos. Aqui, podemos definir uma arquitetura para fornecer informações escondidas no nível da equipe, em vez de apenas no nível de microsserviço. Isso nos dá outro nível de ocultação de informações e cria um espaço seguro maior no qual uma equipe pode fazer mudanças locais sem quebrando o sistema mais amplo.

Dentro de cada microsserviço ou zona maior, você pode concordar com a equipe que possui essa zona escolhendo uma pilha de tecnologia ou armazenamento de dados diferente. Outros preocupações podem surgir aqui, é claro. Sua inclinação para deixar que as equipes escolham o a ferramenta certa para o trabalho pode ser temperada pelo fato de que se torna mais difícil contrate pessoas ou mova-as entre equipes se você tiver 10 tecnologias diferentes pilhas para apoiar. Da mesma forma, se cada equipe escolher dados completamente diferentes loja, você pode não ter experiência suficiente para administrar qualquer uma delas em escala. A Netflix, por exemplo, padronizou principalmente o Cassandra como um dado tecnologia de loja. Embora possa não ser a melhor opção para todos os casos, A Netflix acredita que o valor obtido com a criação de ferramentas e experiência em Cassandra é mais importante do que ter que apoiar e operar em grande escala várias outras plataformas que podem ser mais adequadas para determinadas tarefas. Netflix é um exemplo extremo, onde a escala é provavelmente o fator predominante mais forte, mas você entendeu a ideia.

No entanto, entre os microsserviços é onde as coisas podem ficar confusas. Se um microservice decide expor REST sobre HTTP, outro faz uso de gRPC e um terceiro usam Java RMI, então a integração pode se tornar um pesadelo, já que os microsserviços consumidores precisam entender e oferecer suporte a vários estilos de intercâmbio. É por isso que tento seguir a diretriz de que devemos "ser" preocupado com o que acontece entre as caixas e seja liberal no que acontece lá dentro."

Portanto, uma arquitetura bem-sucedida consiste tanto em permitir mudanças adequadas à necessidades de nossos usuários como qualquer outra coisa. Mas uma coisa que as pessoas geralmente esquecem é que nosso sistema não acomoda apenas usuários; ele também acomoda as pessoas que na verdade constroem o software sozinhos. Uma arquitetura bem-sucedida também ajuda criar um ambiente agradável para fazer nosso trabalho.

Uma construção social

Nenhum plano sobrevive ao contato com o inimigo.

— Helmuth von Moltke (fortemente parafraseado)

Então você pensou sobre a visão, sobre as restrições e sobre o que você precisa realizar. Você acha que entende o que será difícil de mudar, e os espaços em que você precisa para tornar a mudança possível. Agora o que? Bem, a arquitetura é o que acontece, não o que você acha que deveria acontecer - isso é a diferença entre visão e realidade. Arquitetos do ambiente construído precisa trabalhar com as pessoas que constroem o prédio para ajudá-las entender qual é a visão, mas também mudar o plano quando for realidade desafia essa visão. É possível que o que você acha que seja possível fundamentalmente não é. Se um arquiteto não estiver incorporado até certo ponto ao pessoas criando o sistema, então elas não poderão ajudar a comunicar o visão para as pessoas que fazem o trabalho, nem o arquiteto entenderá onde essa visão não é mais adequada ao propósito. A equipe de construção pode encontrar coisas no terreno que não estavam previstas, ou talvez uma escassez de oferta pode causar uma reformulação em termos de design.

Arquitetura é o que acontece, não o que é planejado. Se, como arquiteto, você retire-se do processo de colocar essa visão em prática, então você não é um arquiteto, você é um sonhador. A arquitetura que surgirá pode ou não ter qualquer relação com o que você deseja. Isso acontecerá com ou sem você. A implementação de uma arquitetura exige o trabalho de muitos pessoas e uma série de decisões, grandes e pequenas.

Como disse Grady Booch: 4

No início, a arquitetura de um sistema intensivo de software é declaração de visão. No final das contas, a arquitetura [a] de cada um desses sistemas é um reflexo dos bilhões e bilhões de pequenos e grandes, intencionais e decisões de design accidentais tomadas ao longo do caminho.

Isso significa que, mesmo que você tenha uma pessoa dedicada que, em última análise, seja responsável pela arquitetura, há muitas pessoas responsáveis por colocando essa visão em prática. Implementar uma arquitetura bem-sucedida é será um esforço de equipe. Voltando à citação anterior de Ralph Johnson, "a arquitetura é uma construção social." Um ótimo exemplo disso vem de fazendo através do uso de uma guilda de arquitetura. 5 Dada sua escala, a Comcast

decidiu aproveitar as experiências de grupos diretores do setor, onde a tomada de decisão coletiva é fundamental:

Na Comcast, percebemos que esse problema era muito semelhante ao caminho aberto. os órgãos de padrões funcionam: fazer com que vários grupos autônomos cheguem a um acordo abordagens técnicas. Projetamos uma Guilda de Arquitetura interna explicitamente inspirado em um órgão de padrões muito bem-sucedido, a Internet Força-Tarefa de Engenharia (IETF) que define muitas informações importantes da Internet protocolos.

-Jon Moore, arquiteto chefe de software da Comcast Cable

A abordagem da Comcast tem um nível de formalidade que algumas organizações podem ter acho oneroso, mas parece funcionar bem para a empresa, dado seu tamanho e distribuição.

Habitabilidade

Mais um conceito que vem do ambiente construído e tem ressonância no campo do desenvolvimento de software está a habitabilidade. Eu soube disso pela primeira vez

termo de Frank Buschmann-ele explicou que um arquiteto tem responsabilidade de garantir que o ambiente que eles criam seja agradável para trabalhar. Se a arquitetura é o enquadramento do sistema, que descreve como o difícil para mudar as coisas se encaixam, então também há momentos em que as restrições podem precisam ser implementados. No entanto, entenda errado e trabalhe no sistema pode se tornar doloroso e propenso a erros.

Como explica Richard Gabriel, autor de Patterns of Software, 6:

A habitabilidade é a característica do código-fonte que permite programadores acessando o código mais tarde em sua vida para entender seu construção e intenções e para alterá-la confortavelmente e confiantemente.

Um ecossistema moderno de desenvolvimento de software consiste em mais do que apenas código, no entanto, vai além disso, às tecnologias que usamos e ao trabalho práticas que adotamos. Com muita frequência, vi desenvolvedores amaldiçoando a tecnologia que eles são instruídos a usar - frequentemente tecnologia selecionada por pessoas que

nunca preciso fazer uso disso. Quanto mais você faz a evolução do seu arquitetura e a seleção das ferramentas e tecnologias que você usa processo colaborativo, mais fácil será para você garantir que o fim O resultado é um ambiente habitável no qual as pessoas que constroem o sistema se sentem felizes e produtivos em seu trabalho.

Se quisermos garantir que os sistemas que criamos sejam habitáveis para nossos desenvolvedores, então nossos arquitetos e outros tomadores de decisão precisam entender o impacto de suas decisões. No mínimo, isso significa passar tempo com a equipe e, idealmente, passar tempo programando com a equipe. Para aqueles de vocês que praticam programação em pares, isso se torna uma questão simples para um arquiteto para se juntar a uma equipe por um curto período como membro de uma dupla. Participar de exercícios de programação em conjunto também pode render resultados significativos benefícios, embora um arquiteto que participe dessa atividade de grupo precise ser ciente de como sua presença pode mudar a dinâmica do conjunto.

Idealmente, você deve trabalhar em tarefas normais para realmente entender o que é "normal" trabalhar é como. Não posso enfatizar o quanto é importante para o arquiteto na verdade, passe algum tempo com as equipes que constroem o sistema! Isso é significativamente mais eficaz do que receber uma chamada ou apenas ver o código. Quanto a como muitas vezes você deve fazer isso, isso depende muito do tamanho da (s) equipe (s) estão trabalhando com. Mas a chave é que deve ser uma atividade rotineira. Se você é trabalhando com quatro equipes, por exemplo, talvez certifique-se de gastar metade de um dia com cada equipe a cada quatro semanas, trabalhando com eles em sua entrega tarefas para garantir que você aumente a conscientização e melhore a comunicação com o equipes com as quais você está trabalhando.

Uma abordagem baseada em princípios

As regras são para a obediência dos tolos e a orientação dos sábios.

-Geralmente atribuído a Douglas Bader

Tomar decisões no design do sistema tem tudo a ver com vantagens e desvantagens e microserviços as arquiteturas nos oferecem muitas vantagens a serem feitas! Ao escolher um armazenamento de dados, faça escolhemos uma plataforma com a qual temos menos experiência, mas que nos oferece melhor

dimensionamento? É normal termos duas pilhas de tecnologia diferentes em nosso sistema? Que tal três? Algumas decisões podem ser tomadas completamente no identifique com as informações disponíveis para nós, e essas são as mais fáceis de criar. Mas e aquelas decisões que talvez precisem ser tomadas de forma incompleta? informação?

O enquadramento pode ajudar aqui, e uma ótima maneira de ajudar a estruturar nossa tomada de decisão é definir um conjunto de princípios e práticas que o orientam, com base em metas que estamos tentando alcançar. Vejamos cada um desses aspectos do enquadramento em virar.

Objetivos estratégicos

O papel do arquiteto já é assustador o suficiente, então, felizmente, geralmente não precisa também definir metas estratégicas! Os objetivos estratégicos devem falar com para onde sua empresa está indo e para onde ela se considera a melhor para fazer sua clientes satisfeitos. Essas serão metas de alto nível e podem não incluir tecnologia em tudo. Eles podem ser definidos no nível da empresa ou em uma divisão nível. Podem ser coisas como "Expandir para o Sudeste Asiático para desbloquear novos mercados" ou "Permita que o cliente alcance o máximo possível usando o autoconhecimento serviço." A chave é que eles definam para onde sua organização está indo, então você precisa ter certeza de que a tecnologia está alinhada a isso.

Se você é a pessoa que define a visão técnica da empresa, isso pode significar você precisará passar mais tempo com as partes não técnicas do seu organização (ou "o negócio", como costumam ser chamados). O que é a condução visão para o negócio? E como isso muda?

Princípios

Princípios são regras que você criou para alinhar o que está fazendo às algum objetivo maior, e às vezes eles mudam. Por exemplo, se um dos seus objetivos estratégicos como organização são diminuir o tempo de comercialização de novos recursos, você pode definir um princípio que diz que as equipes de entrega têm controle total durante todo o ciclo de vida de seu software para ser enviado sempre que estão prontos, independentemente de qualquer outra equipe. Se outro objetivo é que seu

a organização está se movendo para aumentar agressivamente suas ofertas em outros países, você pode decidir implementar um princípio de que todo o sistema deve ser portátil para permitir que seja implantado localmente a fim de respeitar a soberania de dados.

Você provavelmente não quer muitos desses. Menos de 10 princípios é bom. Um número pequeno o suficiente para que as pessoas se lembrem deles ou caibam em um número pequeno de cartazes. Quanto mais princípios você tiver, maior a chance de que eles se sobreponham ou se contradizem.

Os Doze Fatores da Heroku são um conjunto de princípios de design estruturados em torno do objetivo de ajudar você a criar aplicativos que funcionem bem no Heroku. Esses princípios também podem fazer sentido em outros contextos. Alguns dos elas são, na verdade, restrições baseadas nos comportamentos que seu aplicativo precisa expor para trabalhar no Heroku. Uma restrição é realmente algo que é muito difícil (ou virtualmente impossível) de mudar, enquanto os princípios são coisas que decidimos escolher. Você pode decidir mencionar explicitamente aquelas coisas que são princípios versus aqueles que são restrições para ajudar a destacar essas coisas que você realmente não pode mudar. Pessoalmente, acho que pode haver algum valor em mantendo-os na mesma lista, para incentivar restrições desafiadoras de vez em quando e depois veja se eles realmente estão imóveis!

Práticas

Nossas práticas são a forma como garantimos que nossos princípios sejam cumpridos. Eles são um conjunto de diretrizes práticas e detalhadas para a execução de tarefas. Eles vão geralmente são específicos da tecnologia e devem ser de nível baixo o suficiente para qualquer desenvolvedor pode entendê-los. As práticas podem incluir diretrizes de codificação, o fato de que todos os dados de log precisam ser capturados centralmente, ou o fato de que HTTP/REST é o estilo de integração padrão. Devido à sua natureza técnica, as práticas normalmente mudam com mais frequência do que os princípios.

Assim como acontece com os princípios, às vezes as práticas refletem restrições dentro de sua organização. Por exemplo, se você decidiu escolher o Azure como sua nuvem, isso precisará ser refletido em suas práticas.

~~As práticas devem sustentar seus princípios. Um princípio que afirma que a entrega~~
~~as equipes controlam todo o ciclo de vida de seus sistemas, o que pode significar que você tem um~~
~~prática afirmando que todos os microserviços são implantados em uma AWS isolada~~
~~contas, fornecendo gerenciamento de autoatendimento dos recursos e isolamento~~
~~de outras equipes.~~

Combinando princípios e práticas

~~Os princípios de uma pessoa são as práticas de outra. Você pode decidir ligar para o~~
~~uso de HTTP/REST é um princípio em vez de uma prática, por exemplo. E isso~~
~~estaria bem. O ponto chave é que há valor em ter uma abordagem abrangente~~
~~ideias que orientam como o sistema evolui e em ter detalhes suficientes para que~~
~~as pessoas sabem como implementar essas ideias. Para um grupo pequeno o suficiente,~~
~~talvez uma única equipe, combinando princípios e práticas, possa ser boa.~~
No entanto, para organizações maiores, onde a tecnologia e o trabalho
as práticas podem diferir de um lugar para outro, você pode querer um conjunto diferente de
práticas em lugares diferentes, desde que todas elas sejam mapeadas para um conjunto comum de
princípios. Uma equipe do .NET, por exemplo, pode ter um conjunto de práticas, e um
equipe Java, outra. Os princípios, no entanto, podem ser os mesmos para ambos.

Um exemplo do mundo real

~~Um antigo colega meu, Evan Bottcher, desenvolveu o diagrama mostrado em~~
~~Figura 16-3 durante o trabalho com um de seus clientes. A figura~~
~~mostra a interação de metas, princípios e práticas em um formato muito claro.~~
Ao longo de alguns anos, as práticas na extrema direita serão
mudam com bastante regularidade, enquanto os princípios permanecem bastante estáticos. UMA
um diagrama como esse pode ser impresso muito bem em uma única folha de papel e
compartilhada, e cada ideia é simples o suficiente para um desenvolvedor comum
lembrar. É claro que há mais detalhes por trás de cada ponto aqui, mas sendo
ser capaz de articular isso de forma resumida é muito útil.

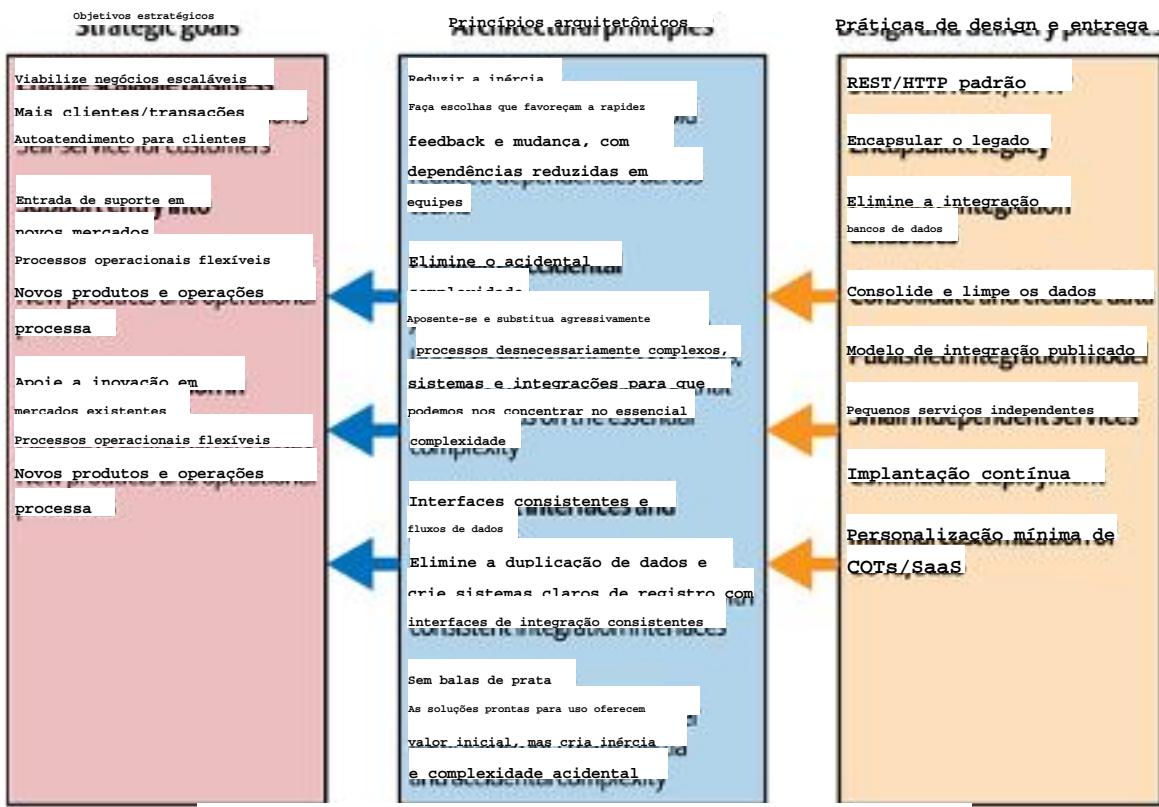


Figura 16-3. Um exemplo real de princípios e práticas

Faz sentido ter documentação que suporte alguns desses itens, e ainda melhor é ter um código funcional que mostre como essas práticas podem ser implementado. Em "A Plataforma", analisamos como é a criação de um comum Um conjunto de ferramentas pode tornar mais fácil para os desenvolvedores fazerem a coisa certa - idealmente, o a plataforma deve tornar a sequência dessas práticas o mais fácil possível, e as práticas mudam, a plataforma deve mudar de acordo.

Orientando uma arquitetura evolutiva

Então, se nossa arquitetura não é estática, mas está em constante mudança e evolução, como garantimos que esteja crescendo e mudando da maneira que queremos, em vez de apenas transformando-se em uma bolha gigante incontrolável de dor, sofrimento e recriminações? Em Construindo Arquiteturas Evolutivas, 7 os autores descrevem funções de condicionamento físico para ajudar a coletar informações sobre a "aptidão" relativa do arquitetura para ajudar os arquitetos a decidir se precisam agir.

Do livro:

A computação evolutiva inclui vários mecanismos que permitem uma solução para emergir gradualmente por meio de pequenas mudanças em cada geração de software. Em cada geração da solução, o engenheiro avalia o estado atual: Está mais perto ou mais longe do objetivo final? Por exemplo, ao usar um algoritmo genético para otimizar o design da asa, a função de aptidão avalia [es] a resistência ao vento, o peso, o fluxo de ar e outras características desejáveis para um bom design de asas. Arquitetos definem um função de condicionamento físico para explicar o que é melhor e para ajudar a medir quando o a meta foi atingida. No software, as funções de condicionamento físico verificam se os desenvolvedores preservam características arquitetônicas importantes.

A ideia de uma função de condicionamento físico é que ela seja usada para entender o estado atual de alguma propriedade importante, de modo que, se essa propriedade mudar fora de alguma limites permitidos, então a mudança precisa ser analisada. Normalmente, condicionamento físico funções serão usadas para garantir que a arquitetura esteja sendo construída para seguir os princípios e restrições que foram estabelecidos.

Para usar um exemplo de Building Evolutionary Architectures, considere a exigência de que a resposta de um determinado serviço seja recebida em 100 ms ou menos. Você pode implementar uma função de condicionamento físico para coletar dados de desempenho desse serviço, talvez em um teste de desempenho ambiente ou de um sistema em execução do mundo real para garantir que o real o comportamento do sistema atende aos requisitos. Edifício evolutivo. Arquiteturas entra em muito mais detalhes sobre esse tópico, e eu detalhadamente recomendo se você quiser explorar mais esse conceito.

As funções de condicionamento físico para arquitetura podem ter várias formas e formatos. O conceito fundamental, porém, é que você colete dados do mundo real para entender se sua arquitetura está ou não alcançando "adequação" a esses critérios. Isso pode estar relacionado ao desempenho do sistema, acoplamento de código, tempo de ciclo ou um host de outros aspectos. Essas funções de condicionamento físico atuam como outra fonte de informação para ajudar um arquiteto a entender onde ele pode precisar se envolver. Por favor observe, no entanto, que, para mim, as funções de condicionamento físico funcionam melhor quando combinadas com estreita colaboração com as pessoas que constroem o sistema. Funções de fitness deve ser uma forma útil de ajudar você a entender se a arquitetura está se movendo a direção certa, mas elas não substituem a necessidade de realmente falar com

pessoas no chão. Na verdade, eu sugiro que definir o condicionamento físico correto as funções exigirão uma colaboração estreita.

Arquitetura em um fluxo alinhado

Organização

No Capítulo 15, analisamos como são as organizações modernas de entrega de software mudando para um modelo mais alinhado ao fluxo, no qual autônomas equipes independentes se concentram na entrega de funcionalidades de ponta a ponta, com suas prioridades são orientadas pelo produto. Também falamos sobre corte transversal equipes que capacitam equipes que apoiam equipes alinhadas ao fluxo. Onde está o arquiteto se encaixou neste mundo? Bem, às vezes o escopo de um fluxo alinhado A equipe é complexa o suficiente para exigir um arquiteto dedicado (aqui, novamente, frequentemente veja uma indefinição das linhas entre o líder técnico tradicional e o arquiteto funções). Em muitos casos, porém, os arquitetos são convidados a trabalhar em vários equipes.

Muitas das responsabilidades do arquiteto podem ser vistas como facilitadoras responsabilidades - comunicando claramente a visão técnica, a compreensão desafios à medida que surgem e ajudando a adaptar a visão técnica em conformidade. O arquiteto ajuda a conectar pessoas, mantendo um olho nas maiores, imagine e ajude as equipes a entender como o que estão fazendo se encaixa no um todo maior. Isso se encaixa perfeitamente na ideia de um arquiteto fazer parte de um capacitar a equipe, como vemos na Figura 16-4. Essa equipe capacitadora poderia consistir de uma mistura de pessoas, talvez pessoas que se dedicam à equipe em tempo integral, e outros que contribuem para ajudar de vez em quando.

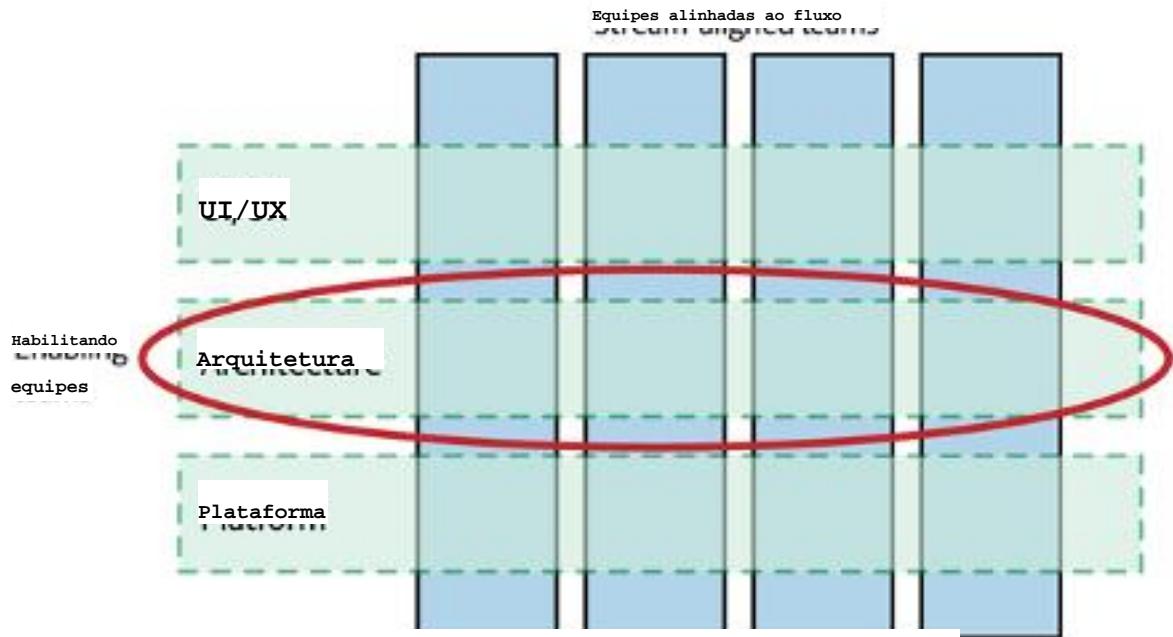


Figura 16-4. Uma arquitetura funciona como uma equipe capacitadora

Um modelo que eu prefiro muito é ter um pequeno número de arquitetos dedicados, em essa equipe (talvez apenas uma ou duas pessoas em muitos casos), mas tendo isso, esse aumentado ao longo do tempo com tecnólogos de cada equipe de entrega - a líderes técnicos de cada equipe, no mínimo. O arquiteto é responsável por certificando-se de que o grupo funcione. Isso distribui o trabalho e garante que é um nível mais alto de adesão. Também garante que as informações fluam livremente de as equipes entram no grupo e, como resultado, a tomada de decisão é muito maior, sensato e informado.

Às vezes, o grupo pode tomar decisões com as quais o arquiteto discorda. Em Neste ponto, o que o arquiteto deve fazer? Tendo estado nessa posição antes, eu posso dizer que essa é uma das situações mais desafiadoras de enfrentar. Muitas vezes eu tomo a abordagem que eu deveria seguir com a decisão do grupo. Eu considero que Eu fiz o meu melhor para convencer as pessoas, mas no final das contas eu não fui convincente suficiente.. O grupo geralmente é muito mais sábio do que o indivíduo, e eu fui provado estar errado mais de uma vez! E imagine o quanto enfraquecedor isso pode ser para que um grupo tenha espaço para tomar uma decisão e depois em última análise, ser ignorado. Mas às vezes eu rejeitei o grupo. Mas por que, e quando? Como você desenha as linhas?

Pense em ensinar uma criança a andar de bicicleta. Você não pode pilotá-lo para eles. Você observe-os balancar, mas se você entrar toda vez, parece que eles podem cair. Se eu sair, eles não nunca aprenderão e, de qualquer forma, cairão muito menos do que você. Acho que eles vão! Mas se você os ver prestes a entrar no trânsito ou em um local próximo ao lago de patos, então você tem que entrar. Claro, eu tenho sido frequentemente provado errado em tais situações - deixei a equipe sair e fazer algo que eu senti estava errado, e o que eles fizeram funcionou! Da mesma forma, como arquiteto, você precisa ter uma noção firme de quando, figurativamente, sua equipe está se transformando em um pato na lagoa. Você também precisa estar ciente de que, mesmo sabendo que está certo e anular a equipe, isso pode minar sua posição e também fazer com que a equipe sentem que eles não têm uma palavra a dizer. Às vezes, a coisa certa é concordar com uma decisão com a qual você não concorda. Saber quando fazer isso e quando não fazer é difícil, mas às vezes vital.

Onde as coisas ficam interessantes, como discutiremos em breve, é quando um arquiteto também precisa se envolver em atividades de governança. Isso pode causar alguns confusões sobre o papel de qualquer equipe de arquitetura transversal. O que acontece quando uma equipe diverge da estratégia técnica? Está tudo bem? Talvez seja uma exceção sensata, mas também pode causar problemas mais fundamentais. Uma decisão de curto prazo tomada em nome da conveniência pode comprometer mudanças maiores que estão tentando ser feitas. Imagine que o grupo de arquitetura está tentando ajudar a organização a deixar de usar dados centralizados devido aos problemas operacionais e de acoplamento que isso causa, mas uma das equipes decide simplesmente colocar alguns novos dados em um banco de dados compartilhado, pois está em pressão para entregar rapidamente. O que acontece então?

Na minha experiência, tudo isso se resume a uma comunicação boa e clara e a uma compreensão das responsabilidades. Se eu visse um proprietário do produto tomando decisões que eu sentia que iriam minar algum tipo de atividade transversal que eu estava trabalhando para isso, eu ia e conversaria com eles. Talvez a resposta seja que a decisão de curto prazo está certa (e, sem dúvida, isso acaba sendo algum tipo de dívida técnica que assumimos conscientemente). Em outros casos, talvez o proprietário do produto seja capaz de mudar o que planeja ajudar a trabalhar com a estratégia geral. Nos piores casos, o problema pode precisar ser escalado.

Na REA, a empresa imobiliária on-line sobre a qual falei algumas vezes antes... capítulos, proprietários de produtos ocasionalmente tomavam decisões para priorizar trabalhar de tal forma que causou o aumento da dívida técnica, levando a problemas subsequentes. O problema era que os proprietários do produto eram principalmente tidos em conta em relação à sua capacidade de fornecer recursos e fazer clientes satisfeitos, embora muitas vezes os problemas relacionados à dívida técnica fossem estabelecidos aos pés dos líderes técnicos. Foi feita uma mudança para fabricar o produto proprietários também responsáveis por aspectos do software que eram técnicos em natureza - isso significava que eles tinham que assumir um papel mais ativo na compreensão os aspectos mais técnicos do sistema (segurança ou desempenho, para exemplo) e trabalhe de forma mais colaborativa com os especialistas técnicos em termos de priorizar o trabalho a ser feito. O ato de fabricar um produto não técnico os proprietários mais responsáveis pela priorização de atividades técnicas são não é trivial, mas vale absolutamente a pena na minha experiência.

Construindo uma equipe

Sendo a pessoa principal para a visão técnica do seu sistema e garantir que você esteja executando essa visão não significa apenas garantir que as decisões tecnológicas corretas são tomadas. São as pessoas com quem você trabalha que o farão. Esteja fazendo o trabalho. É papel de qualquer líder técnico ajudar essas pessoas crescer - para ajudá-los a fazer parte da criação dessa visão - e para garantir que eles também podem ser participantes ativos na formação e implementação da visão.

Ajudar as pessoas ao seu redor com o crescimento de suas próprias carreiras pode exigir muitas coisas. Formas, a maioria das quais está fora do escopo deste livro. Há um aspecto, no entanto, em que uma arquitetura de microsserviços é especialmente relevante. Com sistemas maiores e monolíticos, há menos oportunidades para as pessoas pisarem se levante e "possua" algo. Com microsserviços, por outro lado, temos várias bases de código autônomas que terão sua própria vida independente ciclos. Ajudando as pessoas a progredir fazendo com que elas se apropriem de indivíduos microsserviços antes de aceitar mais responsabilidades podem ser uma ótima maneira de ajudá-los a alcançar seus próprios objetivos de carreira e, ao mesmo tempo, ilumina o sobrecarregue quem está no comando!

Acredito firmemente que um ótimo software vem de ótimas pessoas. Se você se preocupe apenas com o lado tecnológico da equação, você está perdendo o caminho mais da metade da imagem.

O padrão exigido

Quando você está trabalhando em suas práticas e pensando nas vantagens e desvantagens você precisa fazer, um dos saldos mais importantes a serem encontrados é quanto variabilidade para permitir em seu sistema. Uma das principais formas de identificar o que deve ser constante de microsserviço para microsserviço é definir o que é Parece um microsserviço bem comportado e bom. O que é um "bom cidadão" microsserviço em seu sistema? Quais recursos ele precisa ter para garanta que seu sistema seja gerenciável e que um microsserviço ruim não derrubar todo o sistema? Tal como acontece com as pessoas, que "bom cidadão" o microsserviço está em um contexto e não reflete sua aparência em algum lugar outra coisa. No entanto, existem algumas características comuns de pessoas bem-comportadas microsserviços que eu acho que são bastante importantes de observar. Estes são os poucos áreas-chave nas quais permitir muita divergência pode resultar em uma situação bastante horrível tempo. Como diz Ben Christensen, do Facebook, quando você pensa sobre o visão geral, "precisa ser um sistema coeso feito de muitas peças pequenas com ciclos de vida autônomos, mas todos se unindo." Então você precisa encontrar um equilíbrio no qual você otimiza a autonomia de microsserviços individuais sem perder de vista o panorama geral. Definindo atributos claros de cada um o microsserviço deve ter é uma forma de deixar claro onde esse equilíbrio senta. Vamos abordar alguns desses atributos.

Monitoramento

É essencial que sejamos capazes de elaborar visões coerentes e interservicos de a saúde do nosso sistema. Isso tem que ser uma visão de todo o sistema, não um microsserviço visão específica. Conforme discutimos no Capítulo 10, conhecer a saúde de um o microsserviço individual é útil, mas geralmente somente quando você está tentando diagnosticar um problema maior ou entenda uma tendência maior. Para tornar isso tão fácil

quanto possível, eu sugeriria garantir que todos os microserviços emitam saudáveis métricas relacionadas e relacionadas ao monitoramento geral da mesma forma.

Você pode optar por adotar um mecanismo push, em que cada microserviço precisa enviar esses dados para um local central. O que quer que você escolha, tente manter é padronizado. Torne a tecnologia dentro da caixa opaca e não o faça exigem que seus sistemas de monitoramento mudem para dar suporte a eles. Exploração de se enquadra na mesma categoria aqui: precisamos dela em um só lugar.

Interfaces

Escolher um pequeno número de tecnologias de interface definidas ajuda a integrar novas consumidores. Ter um padrão é bom. Dois também não é tão ruim. Tendo vinte estilos diferentes de integração não são bons. Não se trata apenas de escolher a tecnologia e o protocolo. Se você escolher HTTP/REST, por exemplo, você usa verbos ou substantivos? Como você lidará com a paginação de recursos? Como você lidará com o controle de versão dos endpoints?

Segurança arquitetônica

Não podemos nos dar ao luxo de um microserviço mal comportado arruinar a festa todo mundo. Temos que garantir que nossos microserviços se protejam consequentemente, de chamadas insalubres e posteriores. Quanto mais microserviços nós fazermos com que não lidem adequadamente com a possível falha de chamadas downstream, quanto mais frágeis nossos sistemas serão. Isso pode significar, por exemplo, que você querem impor certas práticas em torno da comunicação entre serviços, como como exigindo o uso de disjuntores (um tópico que exploraremos em "Estabilidade Padrões").

Jogar de acordo com as regras também é importante quando se trata de códigos de resposta. Se seus disjuntores dependem de códigos HTTP, e um microserviço decide devolva códigos 2XX em caso de erros ou confunda códigos 4XX com códigos 5XX, então essas medidas de segurança podem desmoronar. Preocupações semelhantes se aplicariam até mesmo se você não estiver usando HTTP; precisamos saber a diferença entre uma solicitação que estava OK e processado corretamente, uma solicitação que estava incorreta e, portanto, impediu que o microserviço fizesse qualquer coisa com ele e uma solicitação de que

pode estar OK, mas não sabemos porque o servidor estava inativo. Saber disso é fundamental para garantir que possamos falhar rapidamente e rastrear problemas. Se nossos microserviços jogue rápido e solto com essas regras, acabamos com uma pessoa mais vulnerável no sistema.

Governança e o caminho pavimentado

Parte do que os arquitetos precisam lidar é com a governança. O que eu quero dizer com governança? Acontece que o COBIT (Control Objectives for Information) da estrutura Technologies () tem uma definição muito boa: 8

A governança garante que os objetivos corporativos sejam alcançados por avaliar as necessidades, condições e opções das partes interessadas; definir a direção por meio de priorização e tomada de decisão; e monitoramento, desempenho, conformidade e progresso em relação à orientação acordada e objetivos.

Em poucas palavras, podemos considerar a governança como um acordo sobre como as coisas devem ser feito, garantindo que as pessoas saibam como as coisas devem ser feitas e garantindo as coisas são feitas dessa maneira. Em alguns ambientes, a governança simplesmente acontece informalmente, como parte das atividades normais de desenvolvimento de software. Em outros ambientes, especialmente em organizações maiores, isso pode precisar ser uma função mais concreta.

A governança pode ser aplicada a várias coisas no fórum de TI. Queremos nos concentrar sobre o aspecto da governança técnica, algo que eu acho que é o trabalho do arquiteto. Se um dos trabalhos do arquiteto é garantir que haja uma visão técnica, então a governança consiste em garantir que o que estamos construindo corresponda a isso visão e evolução da visão, se necessário.

Fundamentalmente, a governança deve ser uma atividade em grupo. Um propriamente um grupo de governança funcional pode trabalhar em conjunto para compartilhar o trabalho e a forma a visão. Pode ser um bate-papo informal com uma equipe pequena ou suficiente ou mais reunião regular estruturada com membros formais do grupo para um escopo maior. É aqui que acho que os princípios que abordamos anteriormente devem ser discutidos e alterado conforme necessário. Se for necessário um grupo formal, esse grupo precisa

consistem predominantemente de pessoas que estão executando o trabalho que está sendo governado. Esse grupo também deve ser responsável pelo rastreamento e gerenciamento técnico de riscos.

Reunir-se e concordar sobre como as coisas podem ser feitas é uma boa ideia. Mas gastar tempo certificando-se de que as pessoas estão seguindo essas diretrizes é menos divertido, assim como sobrecarregar os desenvolvedores a implementação de todas essas coisas padrão. Você espera que cada microserviço funcione. Eu acredito muito em tornar mais fácil fazer a coisa certa e, como discutimos no Capítulo 15, a estrada pavimentada é uma conceito muito útil aqui. O arquiteto tem o papel de articular claramente a visão de onde você está indo e para facilitar o acesso lá. Como tal, eles deve estar envolvido em ajudar a moldar os requisitos de qualquer pavimentação estrada que você constrói. Para muitos, a plataforma será o maior exemplo disso. O arquiteto acaba sendo uma parte interessada importante para a equipe da plataforma.

Já analisamos o papel da plataforma com alguma profundidade, então vamos dar uma olhada em algumas outras técnicas que podemos usar para facilitar ao máximo pessoas para fazer a coisa certa.

Exemplos

A documentação escrita é boa e útil. Eu veio claramente o valor disso: depois de tudo, eu escrevi este livro. Mas os desenvolvedores também gostam de códigos de código que podem ser executados e explore. Se você tem um conjunto de padrões ou melhores práticas que gostaria para incentivar, então, ter exemplos que você possa indicar às pessoas é útil. A ideia é que as pessoas não podem errar muito só imitando algumas das melhores partes do seu sistema.

Idealmente, esses devem ser microserviços do mundo real em execução em seu sistema que faça as coisas da maneira certa, em vez de microserviços isolados que são implementados meramente como "exemplos perfeitos". Ao garantir que seus exemplares estejam realmente sendo usado, você garante que todos os princípios que você tem realmente façam sentido.

Modelo de microserviço personalizado

Não seria ótimo se você pudesse tornar as coisas realmente fáceis para todos os desenvolvedores segue a maioria das diretrizes que você tem com muito pouco trabalho? E se, fora de

na caixa, os desenvolvedores tinham a maior parte do código pronto para implementar o núcleo atributos que cada microserviço precisa?

Existem muitos frameworks para diferentes linguagens de programação que tentam fornecer os elementos básicos do seu próprio modelo de microserviço. Primavera O Boot é provavelmente o exemplo mais bem-sucedido dessa estrutura para o JVM. A estrutura principal do Spring Boot é bastante leve, mas você pode então decidir para reunir um conjunto de bibliotecas para fornecer recursos como verificação de integridade, veiculando HTTP ou expondo métricas. Então, logo que sai da caixa, você tem um microserviço simples "Hello World" que pode ser iniciado a partir do comando linha.

Muitas pessoas então pegam essas estruturas e padronizam essa configuração para seus companhia. Por exemplo, ao criar um novo microserviço, eles podem programar coisas para que elas obtenham um modelo Spring Boot com as bibliotecas principais sua organização usa o que já está conectado; talvez já tenha incluído as bibliotecas para lidar com disjuntores e ser configurado para lidar com a autenticação JWT para chamadas recebidas. Normalmente, essa criação automatizada de modelo criaria um também corresponde ao pipeline de construção.

Cautela garantida

A seleção e a configuração desses modelos personalizados de microserviços são geralmente uma tarefa para a equipe da plataforma. Eles podem, por exemplo, fornecer um modelo para cada idioma suportado, garantindo que, ao usar o modelo os microserviços resultantes funcionam bem com a própria plataforma. Isso pode causar desafios, no entanto.

Eu vi o moral e a produtividade de muitas equipes serem destruídos por ter um estrutura obrigatória imposta a ele. Em um esforço para melhorar a reutilização de código, mais e mais trabalho é colocado em uma estrutura centralizada até que ela se torne uma monstruosidade avassaladora. Se você decidir usar um microserviço personalizado modelo, pense com muito cuidado sobre qual é seu trabalho. Idealmente, seu uso deve ser puramente opcional, mas se você for mais enérgico em sua adoção, você preciso entender que a facilidade de uso para os desenvolvedores deve ser uma prioridade força orientadora. Permitindo que os desenvolvedores que usam o modelo recomendem

e até mesmo contribuir com mudanças na estrutura, talvez como parte de uma estrutura interna. O modelo de código aberto, pode ajudar muito aqui.

Como discutimos em "DRY e os perigos da reutilização de código em um microsserviço Mundo", temos que estar cientes dos perigos do código compartilhado. Em nosso desejo de criar código reutilizável, podemos introduzir fontes de acoplamento entre microsserviços. Pelo menos uma organização com a qual falei está muito preocupada com isso que, na verdade, copia seu código de modelo de microsserviço manualmente em cada microsserviço. Isso significa que uma atualização para o modelo principal de microsserviço leva mais tempo para ser aplicado em todo o sistema, mas isso é menos preocupante para a organização do que o perigo do acoplamento. Outras equipes com quem falei têm simplesmente tratou o modelo de microsserviço como uma dependência binária compartilhada, embora tenham que ser muito diligentes em não deixar a tendência de DRY (não se repita) resulta em um sistema excessivamente acoplado!

A estrada pavimentada em grande escala

O uso interno de modelos e estruturas de microsserviços internos é frequente encontrado em organizações que têm um grande número de microsserviços. Netflix e Monzo são duas dessas organizações. Cada um decidiu padronizar sua pilha de tecnologia até certo ponto (a JVM no caso da Netflix, Go em termos do Monzo), permitindo acelerar a criação de um novo microsserviço com comportamento padrão esperado usando um conjunto comum de ferramentas. Com mais pilha de tecnologia divergente, com um modelo de microsserviço padrão para suas próprias necessidades se tornam mais difíceis.

Se você adotasse várias pilhas de tecnologia diferentes, precisaria de um modelo de microsserviço correspondente para cada um. Essa pode ser uma maneira sutil de você No entanto, restrinja as opções de idioma em suas equipes. Se o interno O modelo de microsserviço suporta apenas a JVM, então as pessoas podem ser desencorajados de escolher pilhas alternativas se precisarem trabalhar muito mais eles mesmos. A Netflix, por exemplo, está especialmente preocupada com aspectos como tolerância a falhas para garantir que a interrupção de uma parte de seu sistema não possa suportar tudo abaixo. Para lidar com isso, uma grande quantidade de trabalho foi feita para garantir que haja bibliotecas de clientes na JVM para fornecer às equipes as ferramentas de que eles precisam para manter seu microsserviço bem comportado. Apresentando um novo

Uma pilha de tecnologia significaria ter que reproduzir todo esse esforço. O principal a preocupação com a Netflix é menos com o esforço duplicado e mais com o fato que é tão fácil errar. O risco de um microsserviço se tornar novo a tolerância a falhas implementada incorreta é alta se puder impactar mais do sistema. A Netflix mitiga isso usando "servicos secundários", que comunique-se localmente com uma JVM que esteja usando as bibliotecas apropriadas.

As malhas de serviços nos deram outra maneira potencial de descarregar o comum comportamento. Algumas funcionalidades que costumavam ser vistas como internas

A responsabilidade do microsserviço agora pode ser transferida para uma malha de microsserviços.

Isso pode garantir mais consistência de comportamento em todos os microsserviços escritos em diferentes linguagens de programação e também reduz as responsabilidades de esses modelos de microsserviços.

Dívida técnica

Muitas vezes somos colocados em situações em que não podemos seguir à risca em nossa visão técnica. Muitas vezes, precisamos fazer uma escolha para cortar alguns cantos para lançar alguns recursos urgentes. Esta é apenas mais uma compensação que encontraremos nós mesmos tendo que fazer. Nossa visão técnica existe por um motivo. Se nós desviando-se desse motivo, pode ter um benefício de curto prazo, mas de longo prazo custo. Um conceito que nos ajuda a entender essa compensação é a dívida técnica. Quando acumulamos dívidas técnicas, assim como a dívida no mundo real, ela tem uma dívida contínua custo e é algo que queremos pagar.

Às vezes, a dívida técnica não é apenas algo que causamos ao usar atalhos. O que acontece se nossa visão do sistema mudar, mas nem todo o nosso sistema fósforos? Nesta situação, também, criamos novas fontes técnicas dívida.

O trabalho do arquiteto é olhar para o panorama geral e entender isso. equilíbrio. Ter alguma visão sobre o nível da dívida e onde se envolver é importante. Dependendo da sua organização, talvez você possa fornecer orientação gentil, mas faça com que as próprias equipes decidam como rastrear e pagar reduza a dívida. Para outras organizações, talvez seja necessário ser mais estruturado, talvez mantendo um registro de dívidas que seja revisado regularmente.

Tratamento de exceções

Portanto, nossos princípios e práticas orientam como nossos sistemas devem ser construídos. Mas o que acontece quando nosso sistema se desvia disso? Às vezes fazemos uma decisão que é apenas uma exceção à regra. Nesses casos, pode valer a pena capturando tal decisão em um registro em algum lugar para referência futura. Se for suficiente exceções são encontradas; eventualmente, pode fazer sentido alterar o aplicável princípio ou prática para refletir uma nova compreensão do mundo. Para exemplo, podemos ter uma prática que afirma que sempre usaremos MySQL para armazenamento de dados. Mas então vemos razões convincentes para usar Cassandra para armazenamento altamente escalável, momento em que mudamos nossa prática para dizer: "Use o MySQL para a maioria dos requisitos de armazenamento, a menos que você espere grandes crescimento em volumes, caso em que use Cassandra."

Vale a pena reiterar, porém, que cada organização é diferente. Eu trabalhei com algumas empresas nas quais as equipes de desenvolvimento têm um alto grau de confiança e autonomia, e os princípios são leves (e a necessidade de transparência). O tratamento de exceções é bastante reduzido, se não eliminado. Em um ambiente mais estruturado organizações nas quais os desenvolvedores têm menos liberdade, rastreando exceções pode ser vital para garantir que as regras em vigor refletem adequadamente os desafios as pessoas estão enfrentando. Com tudo isso dito, sou fã de microserviços como forma de otimizando a autonomia das equipes, dando-lhes o máximo de liberdade possível para resolver o problema em questão. Se você estiver trabalhando em uma organização que impõe muitas restrições sobre como os desenvolvedores podem fazer seu trabalho, então microserviços podem não ser para você.

Resumo

Para resumir este capítulo, aqui estão as principais responsabilidades da arquitetura evolucionária:

Visão

Garanta que haja uma visão técnica claramente comunicada do sistema que o ajudará a atender aos requisitos de seus clientes e organização.

Empatia

Entenda o impacto de suas decisões em seus clientes e colegas.

Colaboração

Interaja com o maior número possível de colegas e colegas para ajudar definir, refinar e executar a visão

Adaptabilidade

Certifique-se de que a visão técnica mude conforme exigido por seus clientes ou organização.

Autonomia

Encontre o equilíbrio certo entre padronizar e viabilizar a autonomia para suas equipes.

Governança

Certifique-se de que o sistema que está sendo implementado se encaixa na visão técnica e certifique-se de que seja fácil para as pessoas fazerem a coisa certa.

O arquiteto evolucionário é aquele que entende que demonstrar esses as principais responsabilidades são um ato de equilíbrio constante. As forças estão sempre pressionando nós de uma forma ou de outra, e entendendo para onde recuar ou para onde ir com o fluxo geralmente é algo que só vem com a experiência. Mas o a pior reação a todas essas forças que nos impulsionam em direção à mudança é tornar-se mais rígido ou fixo em nosso pensamento.

Embora muitos dos conselhos deste capítulo possam ser aplicados a qualquer arquiteto de sistemas, os microsserviços nos dão muito mais decisões a serem tomadas. Portanto, sendo melhor ser capaz de equilibrar todas essas compensações é essencial. Se você quiser explorar isso tópico com mais profundidade, posso recomendar o edifício já citado Arquiteturas evolutivas, bem como The Software, de Gregor Hohpe

Architect Elevator,⁹ que ajuda arquitetos a entender como eles podem fazer pontes
~~ARCHITECT ELEVATOR: WHICH HELPS ARCHITECTS UNDERSTAND HOW THEY CAN BRIDGE
THE GAP BETWEEN STRATEGIC PLANNING AND LOCAL DELIVERY.~~

Estamos quase no final do livro e cobrimos muito terreno. Em
no Posfácio, agora resumiremos o que aprendemos.

1 Por vários motivos, entre os quais o fato de eu ter um diploma em engenharia de software.

2 Isso é de uma troca de e-mails na lista de e-mails da Extreme Programming, que Martin Fowler
então compartilhou em seu artigo "Quem precisa de um arquiteto?"

3 elevadores, para meus leitores norte-americanos.

4 Grady Booch (@Grady_Booch) Twitter, 4 de setembro de 2020, 05h12, <https://oreil.ly/ZgPRZ>.

5 Jon Moore, "Arquitetura com 800 dos meus amigos mais próximos: a evolução da Comcast
Architecture Guild", InfoQ, 14 de maio de 2019, <https://oreil.ly/alvbi>.

6 Richard Gabriel, Padrões de software: histórias da comunidade de software (Nova York:
Imprensa da Universidade de Oxford, 1996).

7 Neal Ford, Rebecca Parsons e Patrick Kua, Construindo arquiteturas evolucionárias
(Sebastopol: O'Reilly, 2017).

8 COBIT 5. Uma estrutura de negócios para a governança e o gerenciamento da TI corporativa
(Rolling Meadows, IL: ISACA, 2012).

9 Gregor Hohpe, O arquiteto de software Elevador (Sebastopol: O'Reilly, 2020).

Posfácio: Reunindo tudo

Este livro abordou muitos tópicos, e eu compartilhei muitos conselhos o caminho. Dada a amplitude da cobertura, achei sensato resumir alguns dos meus principais conselhos sobre arquiteturas de microsserviços. Para aqueles para você que leu o livro inteiro, isso deve ser uma ótima atualização. Para aqueles de vocês que estão impacientes e pularam até o fim, estejam cientes de que existe muitos detalhes por trás desse conselho, e eu recomendo que você leia os detalhes por trás de algumas dessas ideias, em vez de apenas adotá-las cegamente.

Com tudo isso dito, pretendo manter este último capítulo o mais breve possível, então vamos começar.

O que são microsserviços?

Conforme apresentado no Capítulo 1, os microsserviços são um tipo de serviço orientado arquitetura que se concentra na capacidade de implantação independente. Independente implantabilidade significa que você pode fazer uma alteração em um microsserviço, implantar esse microsserviço e libere sua funcionalidade para os usuários finais sem exigindo que outros microsserviços mudem. Tirando o máximo proveito de um arquitetura de microsserviços significa adotar esse conceito. Normalmente, cada o microsserviço é implantado como um processo, com comunicação com outros microsserviços sendo feitos por meio de algum tipo de protocolo de rede. É comum para implantar várias instâncias de um microsserviço, talvez para que você possa fornecer mais escala, ou então melhore a robustez por meio da redundância.

Para oferecer uma implantação independente, precisamos ter certeza de que, ao mudar um microsserviço que não interrompemos as interações com outros microsserviços. Isso exige que nossas interfaces com outros microsserviços sejam estáveis e que as alterações devem ser feitas de forma compatível com versões anteriores. Ocultação de informações, que expandi no Capítulo 2, descrevo uma abordagem na qual, as informações possíveis (código, dados) estão ocultas atrás de uma interface. Você deve expor apenas o mínimo necessário em suas interfaces de serviço para satisfazer seus consumidores. Quanto menos você expõe, mais fácil é garantir as mudanças você fará com que sejam compatíveis com versões anteriores. Informações também estão escondidas

nos permite fazer mudanças tecnológicas dentro de um limite de microsserviços em uma forma que não afetará os consumidores.

Uma das principais formas de implementarmos a capacidade de implantação independente é ocultando o banco de dados. Se um microsserviço precisar armazenar o estado em um banco de dados, isso deve estar totalmente escondido do mundo exterior. Os bancos de dados internos não devem ser diretamente expostos a consumidores externos, pois isso causa muito acoplamento entre os dois, o que prejudica a capacidade de implantação independente. Em geral, evite situações em que vários microsserviços acessem o mesmo banco de dados.

Os microsserviços funcionam muito bem com o design orientado por domínio (DDD). ADICIONOU nos fornece conceitos que nos ajudam a encontrar nossos limites de microsserviços, com o arquitetura resultante sendo alinhada em torno do domínio de negócios. Isso é extremamente útil em situações em que as organizações estão criando mais equipes de TI centradas nos negócios. Com uma equipe focada em uma parte do negócio domínio, agora ele pode se apropriar dos microsserviços que correspondem a esta parte do negócio.

Mudando para microsserviços

Os microsserviços trazem muita complexidade, complexidade suficiente para explicar os motivos pois usá-los precisam ser seriamente considerados. Continuo convencido de que um monólito simples de processo único é um ponto de partida totalmente sensato para um novo sistema. Com o tempo, porém, aprendemos coisas e começamos a ver como cuja arquitetura de sistema atual não é mais adequada para o propósito. Com isso No entanto, procurar mudanças é apropriado.

É importante entender o que você está tentando obter de um microsserviço arquitetura Qual é o objetivo? Que resultado positivo você espera de uma mudança? para levar microsserviços? O resultado que você está almejando será diretamente impacta a forma como você separa seu monólito. Se você está tentando mudar seu arquitetura de sistema para lidar melhor com a escala, você acabará criando uma arquitetura diferente muda do que se seu principal motivador fosse melhorar a autonomia organizacional. EU abordo isso mais no Capítulo 3 e com ainda mais detalhes em meu livro Monolith para microsserviços.

Muitos dos problemas com microserviços são evidentes somente depois que você clica em produção. Portanto, eu recomendo fortemente um incremental, evolutivo decomposição de um monólito existente em vez de uma reescrita do "big bang". Identifique um microserviço que você deseja criar, extraia o apropriado funcionalidade do monólito, implante o novo microserviço em produção e comece a usá-la com raiva. Com base nisso, você verá se está ajudando a atingir seu objetivo, mas você também aprenderá muito que fará com que a próxima extração de microserviços é mais fácil - ou talvez sugira que Afinal, os microserviços podem não ser o caminho a seguir!

Estilos de comunicação

Resumimos as principais formas de comunicação entre microserviços em Capítulo 4, compartilhado novamente na Figura E-1. Isso não é para ser universal modelo, mas se destina apenas a dar uma visão geral dos diferentes tipos de comunicações que são mais comuns.

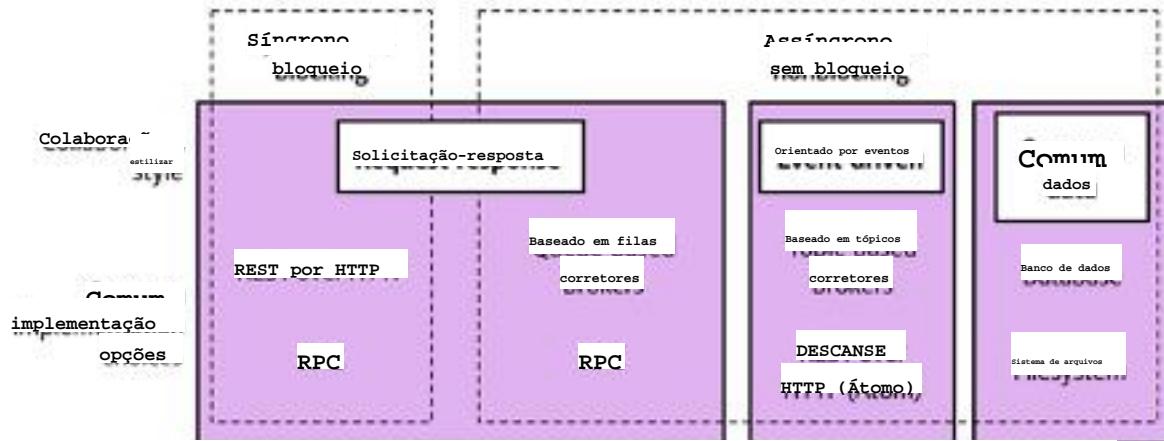


Figura E-1. Diferentes estilos de comunicação entre microserviços, juntamente com exemplos implementando tecnologias

Com a comunicação solicitação-resposta, um microserviço envia uma solicitação para um microserviço downstream e espera uma resposta. Com solicitação-síncrona-resposta, esperaríamos que a resposta voltasse ao microserviço instância que enviou a solicitação. Com solicitação-resposta assíncrona, é possível que a resposta volte para uma instância diferente do upstream microserviços.

Com a comunicação orientada por eventos, um microsserviço emite um evento e outros os microsserviços, se estiverem interessados nesse evento, podem reagir a ele. Os eventos são apenas declarações de informações de fatos que são compartilhadas sobre algo que tem aconteceu. Com a comunicação orientada por eventos, um microsserviço não diz outro microsserviço: o que fazer; ele apenas compartilha eventos. Depende até a jusante microsserviços para fazer um julgamento sobre o que eles fazem com isso informação. A comunicação orientada por eventos é, por definição, assíncrona em natureza.

Um microsserviço pode se comunicar por mais de um protocolo. Para exemplo, na Figura E-2, vemos um microsserviço de envio fornecendo um REST interface para interação solicitação-resposta, que também dispara eventos quando mudanças são feitas.

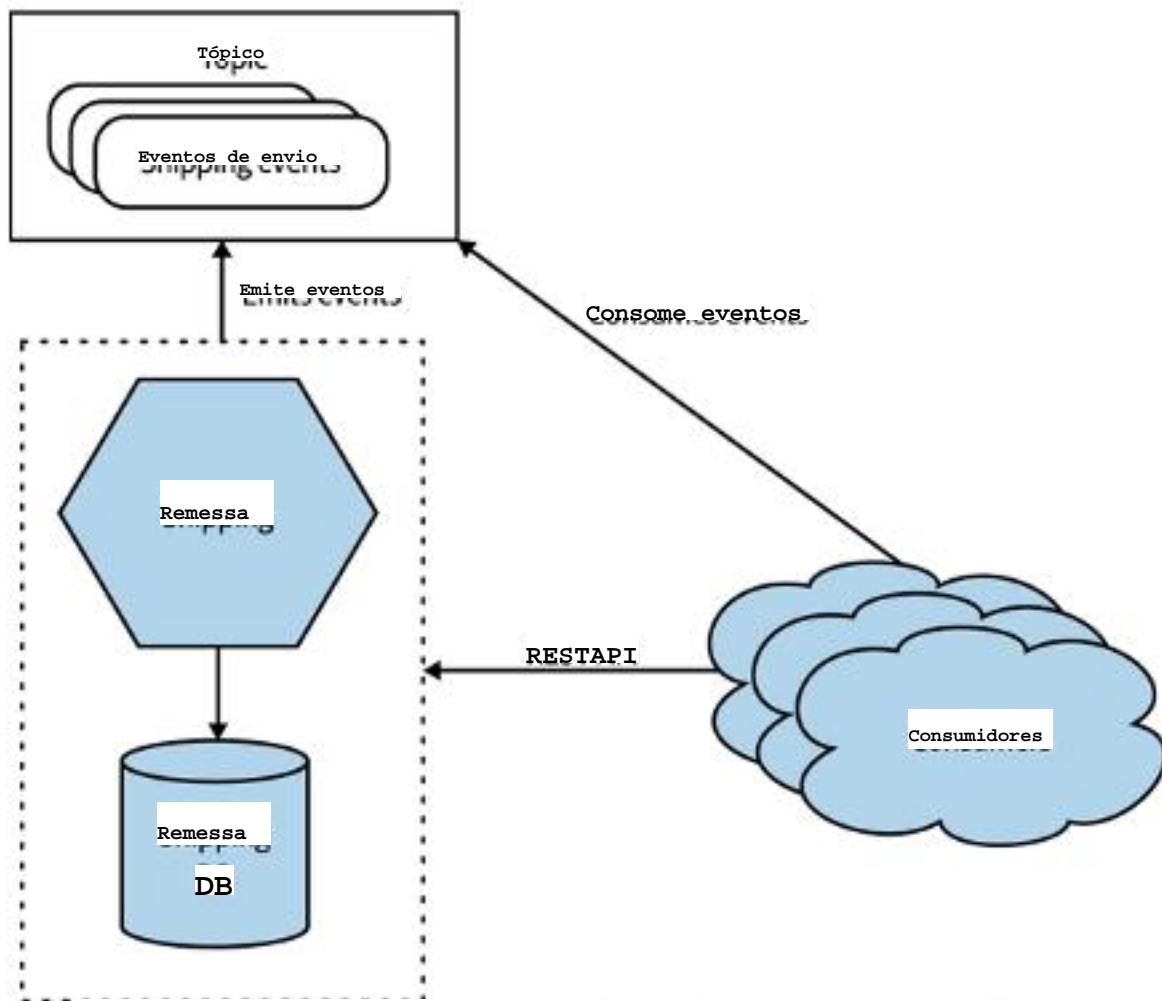


Figura E-2. Um microsserviço expondo sua funcionalidade em uma API REST e um tópico

A colaboração orientada por eventos pode facilitar a criação de uma forma mais frouxa arquiteturas, mas pode exigir mais trabalho para entender como o sistema é se comportando. Esse tipo de comunicação também costuma exigir o uso de tecnologia especializada, como corretores de mensagens, o que pode complicar ainda mais assuntos. Se você puder usar um agente de mensagens totalmente gerenciado, isso pode ajudar a reduzir o custo desses tipos de sistemas.

Ambos os modelos de interação orientada a solicitações e eventos têm seu lugar, e muitas vezes qual deles você usa será uma preferência pessoal. Alguns problemas basta caber em um modelo mais do que em outro, e é comum usar um microsserviço arquitetura para ter uma mistura de estilos.

fluxo de trabalho

Ao procurar que vários microsserviços colaborem para realizar alguns operações abrangentes, procure modelar explicitamente o processo usando sagas, um tópico que exploramos no Capítulo 6.

Em geral, as transações distribuídas devem ser evitadas em situações em que você pode usar uma saga em vez disso. As transações distribuídas adicionam uma complexidade significativa ao sistemas, têm modos de falha problemáticos e geralmente não entregam o que você esperam até mesmo quando trabalham. As sagas são, em praticamente todos os casos, mais adequadas para implementando processos de negócios que abrangem vários microsserviços.

Há dois estilos diferentes de sagas a serem considerados: sagas orquestradas e sagas coreografadas. As sagas orquestradas usam um orquestrador centralizado para coordene com outros microsserviços e garanta que as coisas sejam feitas. Em geral, esta é uma abordagem simples e direta, mas a central orquestrador pode acabar fazendo demais se você não for cuidadoso, e pode se tornam uma fonte de discórdia quando várias equipes trabalham na mesma processo de negócios. Com sagas coreografadas, não há coordenador central; em vez disso, a responsabilidade pelo processo de negócios é distribuída em um número de microsserviços colaborativos. Isso pode ser mais complexo arquitetura para implementar, e requer mais trabalho para garantir que a correta as coisas estão acontecendo, mas, por outro lado, é muito menos propenso ao acoplamento e funciona bem para várias equipes.

Pessoalmente, adoro sagas coreografadas, mas as usei muito e cometeu muitos erros ao implementá-los. Meu conselho geral é que sagas orquestradas funcionam bem quando uma única equipe é responsável pela todo o processo, mas eles se tornam mais problemáticos com várias equipes. As sagas coreografadas podem justificar sua maior complexidade em situações em em que se espera que várias equipes colaborem em um processo.

Construir

Cada microsserviço deve ter sua própria construção, seu próprio pipeline de CI. Quando eu faça uma alteração em um microsserviço, espero ser capaz de criar isso microsserviço por si só. Evite situações em que você tenha que construir todos os seus microsserviços juntos, pois isso dificulta muito a implantação independente. Pelas razões descritas no Capítulo 7. Eu não sou fã de monorepos. Se você realmente quer usar-los, então, por favor, entenda os desafios que eles causam ao redor linhas claras de propriedade e complexidade potencial das construções. Mas definitivamente certifique-se de que, se você usa uma abordagem monorepo ou multirepo, cada o microsserviço tem seu próprio processo de criação de CI que pode ser acionado independentemente de quaisquer outras construções.

Implantação

Os microsserviços normalmente são implantados como um processo. Esse processo pode ser implantado em uma máquina física, uma máquina virtual, um contêiner ou um FaaS plataforma. Idealmente, queremos que os microsserviços sejam tão isolados uns dos outros quanto possível em um ambiente implantado. Não queremos uma situação em que um microsserviço usando muitos recursos de computação pode impactar um diferente microsserviço. Em geral, isso significa que queremos ter cada microsserviço usando seu próprio sistema operacional cercado e conjunto de recursos de computação. Os contêineres são especialmente eficazes em dar a cada instância de microsserviço sua conjunto próprio de recursos cercado, tornando-os uma ótima opção para implantações de microsserviços.

O Kubernetes pode ser muito útil se você quiser executar cargas de trabalho de contêineres em várias máquinas. Não é algo que eu recomendaria para apenas alguns microserviços, pois traz consigo suas próprias fontes de complexidade. Onde possível, use um cluster Kubernetes gerenciado, pois isso permite que você evite alguns dessa complexidade.

O FaaS é um padrão emergente interessante na implantação de código. Em vez de ter para especificar quantas cópias de algo você deseja, basta fornecer seu código para a plataforma FaaS e diga: "Quando isso acontecer, execute esse código". Isso é muito bom do ponto de vista do desenvolvedor, e acho que uma abstração como esse é provavelmente o futuro de uma grande quantidade de desenvolvimento do lado do servidor. No entanto, as implementações atuais não estão isentas de problemas. Em termos de microserviços, implantando um microserviço inteiro como uma única "função" em um A plataforma FaaS é uma ótima maneira de começar.

Uma nota final: separe em sua mente os conceitos de implantação e liberar. Só porque você implantou algo na produção não significa ele tem que ser liberado para seus usuários. Ao separar esses conceitos, você abre a oportunidade de implantar seu software de maneiras diferentes, por exemplo, por usando versões canárias ou execuções paralelas. Tudo isso e muito mais é abordado em profundidade no Capítulo 8.

Testando

Faz muito sentido ter um conjunto de testes funcionais automatizados para fornecer você faz um feedback rápido sobre a qualidade do seu software antes que os usuários o vejam, e isso é absolutamente algo que você deve fazer. Os microserviços oferecem muitas opções em termos dos diferentes tipos de testes que você pode escrever, conforme exploramos no Capítulo 9.

No entanto, quando comparados a outros tipos de arquiteturas, os testes de ponta a ponta podem seja especialmente problemático para arquiteturas de microserviços. Eles podem acabar sendo mais caro escrever e manter para arquiteturas de microserviços do que para arquiteturas não distribuídas mais simples, e os testes em si podem acabam tendo muito mais falhas que não necessariamente apontam para um problema.

com seu código. Testes de ponta a ponta que abrangem várias equipes são particularmente desafiador.

Com o tempo, procure reduzir sua dependência de testes de ponta a ponta; considere substituindo parte do esforço investido nessa forma de teste pelo consumidor-contratos conduzidos, verificação de compatibilidade de esquemas e testes na produção. Essas atividades podem oferecer muito mais eficiência do que testes de ponta a ponta em detectando problemas rapidamente em sistemas mais distribuídos.

Monitoramento e observabilidade

No Capítulo 10, expliquei como o monitoramento é uma atividade, algo que fazemos para um sistema, mas que focar em uma atividade e não em um resultado é problemático, um tópico que percorreu este livro. Em vez disso, devemos nos concentrar sobre a observabilidade de nossos sistemas. A observabilidade é a medida em que pode entender o que um sistema está fazendo examinando as saídas externas. Fazendo um sistema que tenha boa observabilidade requer que construamos esse pensamento em nosso software e garanta que os tipos certos de saídas externas estejam disponíveis.

Os sistemas distribuídos podem falhar de maneiras estranhas, e os microserviços não são diferentes. Não podemos prever todas as causas da falha do sistema, então pode ser difícil saiba de quais informações precisaremos com antecedência para diagnosticar e corrigir problemas.

Usando ferramentas que podem ajudá-lo a interrogar essas saídas externas de forma que você não pode esperar que se torne cada vez mais importante. Eu sugiro que você olhe ferramentas como Lightstep e Honeycomb que foram construídas com esse pensamento em mente.

Finalmente, à medida que seu sistema cresce em escala, torna-se cada vez mais provável que sempre haverá um erro em algum lugar. Mas em um sistema de grande escala, um ter um problema na máquina não é necessariamente motivo para que todos entrem ação, nem isso deve necessariamente resultar em um despertar rude para qualquer pessoa aos 3 anos a.m. Usando técnicas de "teste em produção", como ensaios paralelos e sintéticos as transações podem ser muito mais eficazes para resolver problemas que possam na verdade, estão impactando os usuários finais.

Segurança

Os microserviços nos dão mais oportunidades de defender nosso aplicativo em profundidade, o que, por sua vez, pode levar a sistemas mais seguros. Por outro lado, muitas vezes temos uma área de superfície de ataque maior, o que pode nos deixar mais expostos ao ataque! Esse ato de equilíbrio é o motivo pelo qual é tão importante ter uma compreensão holística de segurança, algo que compartilhei no Capítulo 11.

Com mais informações fluindo pelas redes, torna-se mais importante considere a proteção dos dados em trânsito. O aumento do número de movimentos partes também significa que a automação é uma parte vital da segurança de microserviços.

Gerenciando patches, certificados e segredos usando métodos manuais e propensos a erros os processos podem deixá-lo vulnerável a ataques. Portanto, use ferramentas que permitam facilidade da automação.

Os JWTs podem ser usados para descentralizar a lógica de autorização de uma forma que também evita a necessidade de viagens de ida e volta adicionais. Isso pode ajudar a protegê-lo de questões como o problema confuso do deputado, ao mesmo tempo em que garante seu microserviço pode ser executado de forma mais independente.

Finalmente, um número cada vez maior de pessoas está adotando uma mentalidade de confiança zero. Com confiança zero, você opera como se seu sistema já tivesse sido comprometido e você precisa criar seus microserviços adequadamente. Pode parecer uma postura paranoica, mas sou cada vez mais da opinião de que adotar esse princípio pode realmente simplificar a forma como você vê a segurança de seu sistema.

Resiliência

No Capítulo 12, analisamos a resiliência como um todo e compartilhei com vocês os quatro conceitos-chave que precisam ser considerados ao pensar em resiliência:

Robustez

A capacidade de absorver a perturbação esperada.

Recuperação

A capacidade de se recuperar após um evento traumático.

Extensibilidade elegante

Como lidamos bem com uma situação inesperada

Adaptabilidade sustentada

A capacidade de se adaptar continuamente a ambientes em mudança, partes interessadas, e demandas

Como um todo, as arquiteturas de microserviços podem ajudar com algumas dessas coisas (ou seja, robustez e recuperação), mas, como vemos nesta lista, que por si só não o torna resiliente. Muito de ser resiliente tem a ver com equipe e comportamento e cultura organizacionais.

Fundamentalmente, você precisa fazer coisas explicitamente para criar sua inscrição mais robusto. Robustez não é gratuito: microserviços nos dão a opção de melhorarmos a resiliência de nossos sistemas, mas precisamos fazer essa escolha. Para exemplo, temos que entender que qualquer ligação que fazemos para outra microservice pode falhar, que as máquinas podem morrer e que coisas ruins aconteçam para bons pacotes de rede. Padrões de estabilidade como anteparos, disjuntores, e tempos limite configurados corretamente podem ajudar muito.

Dimensionamento

Os microserviços nos oferecem várias maneiras diferentes de escalar um aplicativo. Em Capítulo 13, exploro os quatro eixos de escala, que compartilho abaixo:

Escala vertical

Em poucas palavras, isso significa adquirir uma máquina maior.

Duplicação horizontal

Ter várias coisas capazes de fazer o mesmo trabalho.

Particionamento de dados

Dividindo o trabalho com base em algum atributo dos dados, por exemplo, grupo de clientes.

Decomposição funcional

Separação do trabalho com base no tipo, por exemplo, decomposição de microsserviços.

Com o dimensionamento, faça as coisas mais fáceis primeiro. Dimensionamento vertical e horizontal a duplicação é rápida e fácil em comparação com os outros dois eixos apresentados aqui. Se funcionarem, ótimo! Caso contrário, você pode examinar os outros mecanismos. É comum também para misturar os diferentes tipos de particionamento de escala de seu tráfego baseado em clientes, por exemplo, e depois em ter cada partição dimensionada horizontalmente.

Interfaces de usuário

Muitas vezes, a interface do usuário é uma reflexão tardia quando se trata de sistema decomposição - separamos nossos microsserviços, mas deixamos um monolítico interface de usuário. Isso, por sua vez, leva aos problemas de ter um front-end separado. e equipes de back-end. Em vez disso, queremos equipes alinhadas ao fluxo, onde uma equipe possui todas as funcionalidades associadas a uma fatia de usuário de ponta a ponta funcionalidade. Para fazer essa mudança acontecer e se livrar do front-end isolado e equipes de back-end, precisamos separar nossas interfaces de usuário.

No Capítulo 14, compartilho como podemos usar micro frontend para oferecer interfaces de usuário compostas usando estruturas de aplicativos de página única, como React.

As interfaces de usuário geralmente enfrentam problemas em termos do número de chamadas necessárias para fazer, ou porque eles precisam realizar a agregação e filtragem de chamadas de acordo com as necessidades dispositivos móveis. O padrão backend for frontend (BFF) pode ajudar a fornecer agregação e filtragem do lado do servidor nessas situações, embora se você estiver capaz de usar o GraphQL, você pode evitar o uso de BFFs.

Organização

No Capítulo 15, analisamos a mudança do alinhamento horizontal e do isolamento equipes em direção a estruturas de equipe que são organizadas em torno de fatias de ponta a ponta de

funcionalidade. Essas equipes alinhadas ao fluxo, como os autores de Team Topologies descreva-os, são apoiados por equipes capacitadoras, como mostra a Figura E-3. As equipes capacitadoras geralmente terão um foco transversal específico, como focando na segurança ou na usabilidade e apoia as equipes alinhadas ao fluxo em esses aspectos.

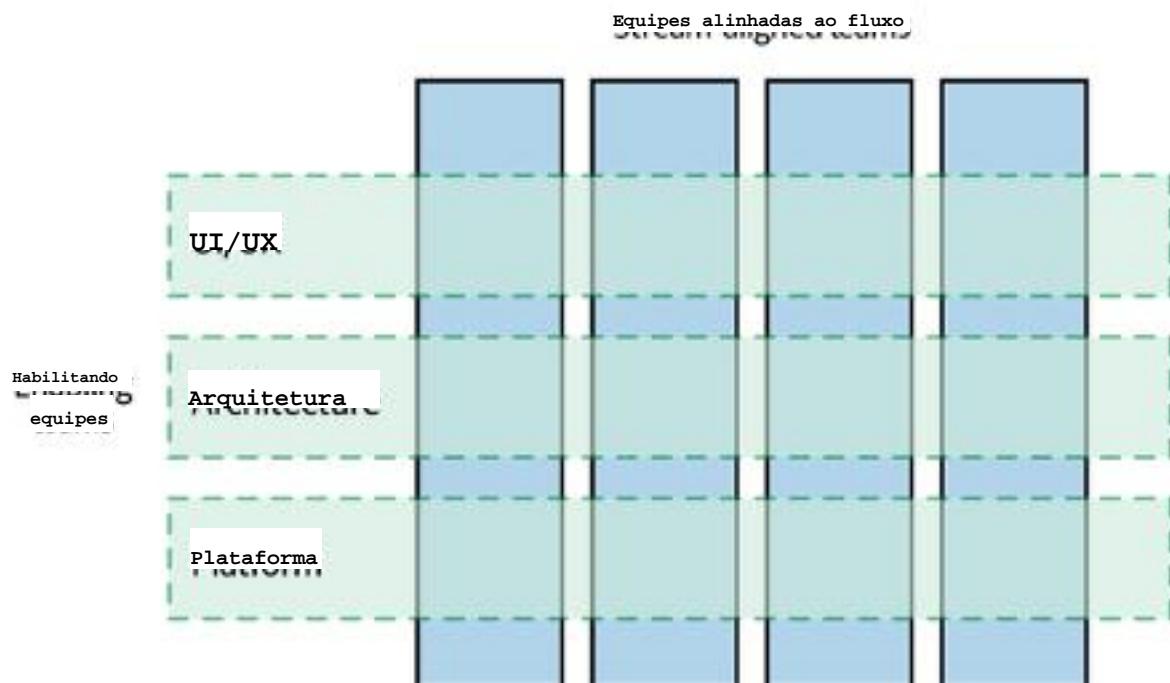


Figura E-3. Permitindo que as equipes suportem várias equipes alinhadas ao fluxo

Tornar essas equipes alinhadas ao fluxo o mais autônomas possível significa que eles precisam de ferramentas de autoatendimento para evitar a necessidade constante de outras equipes faça coisas por eles. Como parte disso, uma plataforma pode ser incrivelmente útil. É importante, porém, que vejamos uma plataforma como um tipo de estrada pavimentada, ou seja, como algo que torna mais fácil fazer a coisa certa, sem exigir que deve ser usado. Tornar uma plataforma opcional garante que tornar a plataforma fácil de usar continua sendo o foco principal da equipe proprietária, ao mesmo tempo em que permite equipes para fazer uma escolha diferente quando necessário.

Arquitetura

É importante que não vejamos a arquitetura do nosso sistema como fixa e imutável. Em vez disso, devemos ver nossa arquitetura de sistema como algo

que devem ser capazes de mudar continuamente conforme as circunstâncias exigirem. Para você tirar o máximo proveito das arquiteturas de microserviços, migrando para uma organização onde mais autonomia é atribuída às equipes significa que a responsabilidade pela a visão técnica precisa se tornar um processo mais colaborativo. O arquiteto sentar em uma torre de marfim será um bloqueador significativo para um microserviço arquitetura ou então se tornará uma irrelevância ignorada.

O papel de guiar a arquitetura de um sistema pode ser inteiramente distribuído nas equipes e, em um determinado nível de escala, isso pode funcionar bem. No entanto, à medida que a organização cresce, há pessoas com tempo dedicado a examinar o sistema como um todo se torna essencial. Chame-os de diretor, engenheiros, proprietários de produtos técnicos ou arquitetos, isso realmente não importa, o papel que eles precisam desempenhar é o mesmo. Como mostrei no Capítulo 16, arquitetos em uma organização de microserviços precisam apoiar equipes e se conectar pessoas, identifique padrões emergentes e passe bastante tempo incorporado às equipes para ver como as coisas gerais se desenrolam na realidade.

Leitura adicional

Ao longo deste livro, fiz referência a muitos artigos, apresentações e livros com os quais aprendi muito e fiz questão de listá-los na Bibliografia. Desde a primeira edição, porém, os dois livros que tiveram o maior impacto em meu pensamento e, como resultado, foram referenciados extensivamente nesta nova edição, vale a pena mencionar aqui como "leituras obrigatórias". O primeiro é Accelerate, de Nicole Forsgren, Jez Humble e Gene Kim. O segundo é Team Topologies, de Matthew Skelton e Manuel Pais. Esses dois livros são, na minha opinião, os dois livros mais úteis sobre software desenvolvimento escrito nos últimos dez anos, e eu os considero essenciais lendo se você gosta de microserviços ou não.

Como complemento deste livro, meu próprio Monolith to Microservices aborda mais detalhes sobre como separar as arquiteturas de sistema existentes.

Olhando para frente

No futuro, suspeito que seja a tecnologia que torna os microservices mais fáceis build and run continuarão melhorando, e estou especialmente interessado em ver o que a aparência dos produtos FaaS de segunda (e terceira) geração. Seja ou não O FaaS decola, o Kubernetes se tornará ainda mais difundido, mesmo que estarão cada vez mais escondidos por trás de abstrações mais amigáveis ao desenvolvedor camadas. O Kubernetes venceu, mas de uma forma que eu acho que a maioria dos aplicativos os desenvolvedores não deveriam se preocupar com isso. Continuo muito interessado em ver como o Wasm muda a forma como pensamos sobre implantações, e eu ainda tenho um suspeita de que os unikernels também possam ter uma segunda vinda.

Desde a primeira edição deste livro, os microservices realmente desapareceram popular de uma forma que me surpreendeu, e isso também me preocupou. É parece que muitas pessoas que adotam microservices estão fazendo isso mais porque todos os outros estão fazendo isso, em vez de os microservices serem adequados para eles. Como então, espero que ouçamos mais histórias de terror sobre microservices fracassados implementações, que vou digerir com prazer para ver o que pode ser aprendido. EU também espero totalmente uma reação mais ampla do setor contra os microservices em alguns ponto em que os estudos de caso de desastres de microservices atingem massa crítica. Aplicando o pensamento crítico para descobrir qual abordagem faz mais sentido em qualquer situação não é muito sexy ou comercializável, e eu não espero que isso aconteça mudança em um mundo onde vender tecnologia é mais lucrativo do que vender ideias.

Não quero parecer pessimista! Como indústria, ainda somos muito jovens, e ainda estamos encontrando nosso lugar no mundo. A quantidade de energia e a engenhosidade que é colocada no desenvolvimento de software continua me mantendo Estou interessado e mal posso esperar para ver o que a próxima década trará.

Palavras finais

As arquiteturas de microservices oferecem mais opções e mais decisões para fazer. Tomar decisões neste mundo é uma atividade muito mais comum do que em sistemas mais simples e monolíticos. Você não vai acertar todas essas decisões, eu posso garantir isso. Então, sabendo que você vai errar algumas coisas, quais são suas opções? Bem, eu sugeriria encontrar maneiras de tomar cada decisão

pequeno em escopo; dessa forma, se você errar, afetará apenas uma pequena parte do seu sistema. Aprenda a abraçar o conceito de arquitetura evolutiva, em que seu sistema dobra, flexiona e muda com o tempo à medida que você aprende algo novo das coisas. Não pense em reescritas de big bang, mas em vez de uma série de mudanças feitas em seu sistema ao longo do tempo para mantê-lo flexível.

Espero já ter compartilhado com você informações e experiências suficientes para ajudá-lo a decidir se os microserviços são para você. Se estiverem, espero que você pense nisso como uma jornada, não como um destino. Vá de forma incremental. Quebre seu sistema separado peça por peça, aprendendo à medida que avança. E se acostume com isso: em muitos de qualquer forma, a disciplina para mudar e evoluir continuamente nossos sistemas está longe da licção mais importante a aprender do que qualquer outra que compartilhei com você por meio deste livro. A mudança é inevitável. Abraça isso.

Bibliografia

Relatório de investigações de violação de dados de 2020. Verizon, 2020.

<https://oreil.ly/ps0Cx>.

Abbott, Martin L. e Michael T. Fisher. *A arte da escalabilidade: escalável Arquitetura, processos e organizações da Web para os modernos Enterprise.* 2ª ed. Boston: Addison-Wesley, 2015.

Allspaw, John. "Pós-morte irrepreensíveis e uma cultura justa". Codifique como artesanato (blog). Etsy, 22 de maio de 2012. <https://oreil.ly/PiBcX>.

Bache, Emily. "Teste automatizado de ponta a ponta em um microserviço Arquitetura." Conferências do NDC. 5 de julho de 2017. Vídeo do YouTube, 56:48.
<https://oreil.ly/DbFdR>.

Bell, Laura, Michael Brunton-Spall, Rich Smith e Jim Bird. *Ágil Segurança de aplicativos.* Sebastopol: O'Reilly, 2017.

Beyer, Betsy, Chris Jones, Jennifer Petoff e Niall Richard Murphy, eds. *Engenharia de confiabilidade do site: como o Google executa sistemas de produção.* Sebastopol: O'Reilly, 2016.

Beyer, Betsy, Niall Richard Murphy, David K. Rensin, Kent Kawahara e Stephen Thorne, eds. *A apostila de confiabilidade do site: maneiras práticas de Implemente o SRE.* Sebastopol: O'Reilly, 2018.

Bird, Christian, Nachi Nagappan, Brendan Murphy, Harald Gall e Premumar Devabu, "Não toque no meu código! Examinando os efeitos do Propriedade da qualidade do software." No ESEC/FSE '11: Anais do 19º Simpósio ACM SIGSOFT e a 13ª Conferência Europeia sobre Fundamentos da engenharia de software, 4-14. Nova York: ACM, 2011.
doi.org/10.1145/2025113.2025119.

Brandolini, Alberto. *Events Storming*. Victoria, BC: Leanpub, em breve.

Brooks, Frederick P., Jr. *O mítico mês do homem: ensaios sobre software Engenharia*, Edição de aniversário. Boston: Addison-Wesley, 1995.

Brown, Alanna, Nicole Forsgren, Jez Humble, Nigel Kersten e Gene Kim.

Relatório sobre o estado do DevOps de 2016. <https://oreil.ly/WJjhA>.

- Bryant, Daniel. "A Apple reconstrói os serviços de back-end da Siri usando o Apache Mesos." InfoQ, 3 de maio de 2015. <https://oreil.ly/NsjEQ>.
- Burns, Brendan, Brian Grant, David Oppenheimer, Eric Brewer e John Wilkes. "Borg, Omega e Kubernetes". acmqueue 14, nº 1 (2016). <https://oreil.ly/2TIYG>.
- Calçado, Phil. "Padrão: usando pseudo-URIs com microserviços." 22 de maio 2017. <https://oreil.ly/uZuto>.
- Cockburn, Alistair. "Arquitetura hexagonal". alistair.cockburn.us, janeiro 4, 2005. <https://oreil.ly/0Jelm>.
- Cohn, Mike. Obtendo sucesso com o Agile. Upper Saddle River, Nova Jersey: Addison-Wesley, 2009.
- Colyer, Adrian. "Aspectos da distribuição de informações da metodologia de design". The Morning Paper (blog), 17 de outubro de 2016. <https://oreil.ly/qxj2m>.
- Crispin, Lisa e Janet Gregory. Teste ágil: um guia prático para Testadores e equipes ágeis. Upper Saddle River, NJ: Addison-Wesley, 2008.
- Evans, Eric. Design orientado por domínio: enfrentando a complexidade no cerne da Software. Boston: Addison-Wesley, 2004.
- Ford, Neal, Rebecca Parsons e Patrick Kua. Edifício evolutivo Arquiteturas Sebastopol: O'Reilly, 2017.
- Forsgren, Nicole, Dustin Smith, Jez Humble e Jessie Frazelle. Accelerate: relatório sobre o estado do DevOps de 2019. <https://oreil.ly/A3zGn>.
- Forsgren, Nicole, Jez Humble e Gene Kim. Accelerate: A ciência da Construindo e escalando organizações de tecnologia de alto desempenho. Portland, OR: Revolução da TI, 2018.
- Fowler, Martin. "CodeOwnership". martinfowler.com, 12 de maio de 2006. <https://oreil.ly/a42c7>.
- Fowler, Martin. "Erradicando o não determinismo em testes." martinfowler.com, 14 de abril de 2011. <https://oreil.ly/sqPOD>.

Fowler, Martin. "StranglerFig Application". martinfowler.com, 29 de junho de 2004. <https://oreil.ly/fotio>.

Freeman, Steve e Nat Pryce. Desenvolvimento de software orientado a objetos, guiado por testes. Upper Saddle River, NJ: Addison-Wesley, 2009.

Friedrichsen, Uwe. "Os limites do padrão de saga". ufried.com (blog). 19 de fevereiro de 2021. <https://oreil.ly/X1B/K>.

Garcia-Molina, Hector, Dieter Gawlick, Johannes Klein e Karl Kleissner.

"Modelando atividades de longa duração como sagas aninhadas". Engenharia de dados 14, no. 1 (março de 1991): 14-18. <https://oreil.ly/RVp7A11>.

Garcia-Molina, Hector e Kenneth Salem. "Sagas". Registro ACM Sigmod 16, nº. 3 (1987); 249-59.

Governador, James. "Rumo à entrega progressiva." James Governor MonkChips (blog). RedMonk, 6 de agosto de 2018. <https://oreil.ly/OlkEY>.

Heinemeier Hansson, David. "O Monólito Maiestoso". Sinal V. Ruído, 29 de fevereiro de 2016. <https://oreil.ly/fN5CR>.

Hodgson, Pete. "Alternâncias de recursos (também conhecidas como sinalizadores de recursos)." martinfowler.com, 9 de outubro de 2017. <https://oreil.ly/pSPrd>.

Espero, Gregor. O elevador do arquiteto de software: redefinindo o arquiteto Papel na empresa digital. Sebastopol: O'Reilly, 2020.

Hope, Gregor e Bobby Woolf. Padrões de integração empresarial. Boston: Addison-Wesley, 2003.

Humble, Jez e David Farley. Entrega contínua: software confiável Lançamentos por meio de automação de criação, teste e implantação. Sela superior River, NJ: Addison-Wesley, 2010.

Cace, Troy. "As senhas evoluíram: orientação de autenticação para os modernos Era." troyhunt.com, 26 de julho de 2017. <https://oreil.ly/r4ava>.

Inglês, Paul. "Convergência para o Kubernetes." Medium, 18 de junho de 2018. <https://oreil.ly/QB2F1>.

Ismael, Jonathan. "Otimizando o Serverless para a BBC Online." Tecnologia e Criatividade na BBC (blog). BBC, 26 de janeiro de 2021.
<https://oreil.ly/mPp2L>.

Jackson, Cam. "Micro Frontends". martinfowler.com 19 de junho de 2019.
<https://oreil.ly/nYu15>.

Kingsbury, Kyle. "Jepsen: Elasticsearch." Aphyr, 15 de junho de 2014.
<https://oreil.ly/612sR>.

Kingsbury, Kyle. "Jepsen: Elasticsearch 1.5.0." Aphyr, 27 de abril de 2015.
<https://oreil.ly/jlu8p>.

Kleppmann, Martin. Projetando aplicativos com uso intensivo de dados. Sebastopol: O'Reilly, 2017.

Krishnan, Kripa. "Weathering the Unexpected". acmqueue 10, nº 9 (2012).
<https://oreil.ly/BN2Ek>.

Kubis, Robert. "Google Cloud Spanner: consistência global em escala por Robert Kubis." Devoxx. 7 de novembro de 2017. Vídeo do YouTube, 33:22.
<https://oreil.ly/fwbdM>.

Lamport, Leslie. "Tempo, relógios e a ordenação de eventos em um sistema distribuído Sistema." Comunicações da ACM. 21, nº 7 (julho de 1978): 558-65.
<https://oreil.ly/Y07gu>.

Lewis, James. "Escala, microservicos e fluxo". COMO! Conferências. 10 de fevereiro de 2020. Vídeo do YouTube, 51:03. <https://oreil.ly/nzXqX>.

Losio, Renato "A Elastic altera as licenças para Elasticsearch e Kibana: A AWS bifurca os dois." InfoQ, 25 de janeiro de 2021. <https://oreil.ly/PCIFv>.

MacCormack, Alan, Carliss Y. Baldwin e John Rusnak. "Explorando o Dualidade entre arquiteturas organizacionais e de produto: um teste da Hipótese de espelhamento." Política de Pesquisa 41, nº 8 (outubro de 2012): 1309-24..

Majors, Charity. "Métricas: não são os dróides de observabilidade que você está procurando Para." Honeycomb (blog), 24 de outubro de 2017. <https://oreil.ly/RpZaz>.

- Majors, Charity, Liz Fong-Jones e George Miranda. *Observabilidade em Engenharia*. Sebastopol: O'Reilly, 2022.
- McAllister, Neil. "Code Spaces fica ativo para sempre depois do atacante NUKES: seus dados hospedados na Amazon." *The Register*, 18 de junho de 2014. <https://oreil.ly/IUOD0>.
- Miles, Russ. *Aprendendo a engenharia do caos*. Sebastopol: O'Reilly, 2019.
- Moore, Jon. "Arquitetura com 800 dos meus amigos mais próximos: a evolução da Guilda de Arquitetura da Comcast." *InfoQ*, 14 de maio de 2019. <https://oreil.ly/dvfhi>.
- Morris, Kiev. *Infraestrutura como código*. 2ª ed. Sebastopol: O'Reilly, 2016.
- Nagappan, Nachiappan, Brendan Murphy e Victor Basili. "A influência da estrutura organizacional sobre qualidade de software: um estudo de caso empírico." *ICSE '08: Anais da 30ª Conferência Internacional sobre Software Engenharia*. Nova York: ACM, 2008.
- Newman, Sam. *Monolith to Microservices*. Sebastopol: O'Reilly, 2019.
- Olá, Saeed. "Histórico de design do sistema de arquivos virtual Git." <https://t.co/mlQR4uzWKS?amp=1>
- Nygard, Michael. *Solte isso!* 2ª ed. Raleigh: Pragmatic Bookshelf, 2018.
- Oberlehner, Markus. "Monorepos na selva." *Medium*, 12 de junho de 2017. <https://oreil.ly/Sk6am>.
- Padmanabhan, Senthil e Pranav Jha. "WebAssembly no eBay: um verdadeiro Caso de uso mundial." *eBay*, 22 de maio de 2019. <https://oreil.ly/r1r7d>.
- Page-Jones, Meilir. *Guia prático de projeto de sistemas estruturados*, 2ª ed. Nova York: Yourdon Press, 1980.
- Palino, Todd, Neha Narkhede e Gwen Shapira. *Kafka: O definitivo Guia*. Sebastopol: O'Reilly, 2017.
- Parnas, David. "Aspectos da distribuição de informações da metodologia de design". Em *Processamento de Informações: Anais do Congresso do IFIP*, 339-44. Vol. 1. Amsterdã: Holanda do Norte, 1972.

Parnas, David. "Sobre os critérios a serem usados na decomposição de sistemas em Módulos." Contribuição do periódico, Universidade Carnegie Mellon, 1971. <https://oreil.ly/nWtQA>.

Plotnicki, Lukasz "Melhor amigo do Soundcloud". ThoughtWorks, 9 de dezembro de 2015. <https://oreil.ly/ZyR01>.

Potvin, Rachel e Josh Levenberg. "Por que o Google armazena bilhões de linhas de código em um único repositório." Comunicações do ACM 59, nº 7 (Julho de 2016): 78-87. <https://oreil.ly/Eupyi>.

Pyhäjärvi, Maaret. Guia de programação do Ensemble. Publicada por conta própria, 2015-2020. <https://ensembleprogramming.xyz>.

Riggins, Jennifer. "A ascensão da entrega progressiva de sistemas Resiliência." The New Stack, 1º de abril de 2019. <https://oreil.ly/merIs>.

Rodriguez, Daniel, M., Ángel Sicilia, Elena García Barriocanal e Rachel Harrison. "Descobertas empíricas sobre o tamanho da equipe e a produtividade em software Desenvolvimento." Journal of Systems and Software 85, nº 3 (2012). doi.org/10.1016/j.jss.2011.09.009.

Rossman, John. Pense como a Amazon: 50 1/2 ideias para se tornar digital Líder, Nova York: McGraw-Hill, 2019.

Ruecker, Bernd. Automação prática de processos, Sebastopol: O'Reilly, 2021. Sadalage, Pramod e Martin Fowler. NoSQL Distilled: um breve guia para o mundo emergente da persistência poliglota. Upper Saddle River, Nova Jersey: Addison-Wesley, 2012.

Schneider, Jonny. Entendendo o Design Thinking, o Lean e o Agile. Sebastopol: O'Reilly, 2017.

Shankland, Stephen. "O Google revela um servidor outrora secreto." CENTAVO, 11 de dezembro de 2009. <https://oreil.ly/hHKvE>.

Shorrock, Steven "Design de alarmes: da energia nuclear ao WebOps". Humanistic Systems (blog), 16 de outubro de 2015. <https://oreil.ly/AiJ5i>.

Shostack, Adam Threat Modeling: projetando para a segurança. Indianápolis: Wiley, 2014.

Sigelman, Ben. "Três pilares com zero respostas - rumo a um novo Scorecard for Observability." Lightstep (postagem do blog), 5 de dezembro de 2018. <https://oreil.ly/qdtSS>.

Skelton, Matthew e Manuel Pais. Topologias de equipe. Portland, OR: TI Revolução, 2019.

Steen, Maarten van e Andrew Tanenbaum Distributed Systems. 3ª ed. Scotts Valley, CA: Plataforma de publicação independente CreateSpace, 2017.

Stopford, Ben. Projetando sistemas orientados por eventos. Sebastopol: O'Reilly, 2017.

Valentino, Jason D. "Moving é uma das maiores empresas voltadas para o cliente da Capital One Aplicativos para a AWS." Medium/Capital One Tech, 24 de maio de 2017.

<https://oreil.ly/IEIC3>.

Vaughan, Diane. A decisão de lançamento do Challenger: tecnologia arriscada, Cultura e desvio na NASA. Chicago: Imprensa da Universidade de Chicago, 1996.

Vernon, Vaughn. Design orientado por domínio destilado em Boston: Addison-Wesley, 2016.

Vernon, Vaughn. Implementando o design orientado por domínio. Rio Upper Saddle, NJ: Addison-Wesley, 2013.

Vocke, Ham. "A pirâmide de testes práticos". martinfowler.com, 26 de fevereiro 2018. <https://oreil.ly/6rR0U>.

Webber, Emily. Construindo comunidades de prática bem-sucedidas. San Francisco: Blurb, 2016.

Webber, Jim, Savas Parastatidis e Ian Robinson. REST na prática: Arquitetura de sistemas e hipermídia. Sebastopol: O'Reilly, 2010.

Woods, David D. "Quatro conceitos para a resiliência e as implicações para a Futuro da engenharia de resiliência." Engenharia de confiabilidade e segurança do sistema

141 (setembro de 2015); 5-9. doi.org/10.1016/j.ress.2015.03.018.

Yourdon, Edward e Larry L. Constantine. Design Estruturado. Nova York: Yourdon Press, 1976.

Zimmerman, Adam. "Entrega progressiva, uma história... Condensado." Indústria Insights (blog). LaunchDarkly, 6 de agosto de 2018. <https://oreil.ly/4pVY7>.

Glossário

agregar

Uma coleção de objetos que são gerenciados como uma única entidade, normalmente referindo-se a conceitos do mundo real. Um conceito do DDD.

Amazon Web Services (AWS)

A oferta de nuvem pública da Amazon.

Gateway de API

Um componente que normalmente fica no perímetro de um sistema e rotas chamadas de fontes externas (como interfaces de usuário) para microserviços, entre muitas outras coisas.

autenticação

O processo pelo qual um diretor prova que é quem diz ser são. Isso pode ser tão simples quanto uma pessoa fornecer seu nome de usuário e senha.

autorização

O processo que determina se um diretor autorizado tem permissão para accesse uma determinada funcionalidade.

Azure

A oferta de nuvem pública da Microsoft.

backend para front-end (BFF)

Um componente do lado do servidor que fornece agregação e filtragem para um interface de usuário específica. Uma alternativa a um gateway de API de uso geral.

contexto limitado

Um limite explícito dentro de um domínio comercial que fornece funcionalidade para o sistema mais amplo, mas que também esconde a complexidade. Frequentemente mapas para limites organizacionais. Um conceito do DDD.

antepara

Uma parte do sistema na qual uma falha pode ser isolada, de modo que o resto do sistema pode continuar operando mesmo se ocorrer uma falha.

coreografia

Um estilo de saga, onde a responsabilidade pelo que deveria acontecer e quando é distribuído em vários microserviços, em vez de gerenciado por uma entidade única.

disjuntor

Um mecanismo colocado em torno de uma conexão com um serviço downstream que pode permitir que você falhe rapidamente se o serviço downstream estiver sofrendo de problemas.

coesão

Até que ponto o código que muda em conjunto permanece unido.

propriedade coletiva

Um estilo de propriedade no qual qualquer desenvolvedor pode alterar qualquer parte do sistema.

recipiente

Um pacote de código e dependências que pode ser executado de forma isolada em uma máquina. Conceitualmente semelhante às máquinas virtuais, embora muito mais leve.

entrega contínua (CD)

Uma abordagem de entrega na qual você modela explicitamente o caminho para produção, trate cada check-in como um candidato a lançamento e pode facilmente avaliar a adequação de qualquer candidato a lançamento para ser implantado em produção.

implantação contínua

Uma abordagem em que qualquer construção que passa por todas as etapas automatizadas é implantado automaticamente na produção.

integração contínua (CI)

A integração regular (diária) das mudanças com o resto da base de código, junto com um conjunto de testes para validar se a integração funcionou.

Lei de Conway

A observação de que as estruturas de comunicação das organizações terminam impulsionando o design dos sistemas de computador que essas organizações construir.

acoplamento

Até que ponto a alteração de uma parte do sistema exige uma mudança em outro. Normalmente, é desejável um baixo acoplamento.

requisito multifuncional (CFR)

Uma propriedade geral do sistema, como a latência necessária para operações, segurança de dados em repouso, etc. Também conhecido como não funcional requisito (mas eu prefiro muito mais multifuncional como descrição).

software personalizável pronto para uso (COTS)

Software de terceiros que é altamente personalizado pelo usuário final e é também normalmente funcionam em sua própria infraestrutura. Exemplos típicos incluem

sistemas de gerenciamento de conteúdo e gerenciamento de relacionamento com o cliente
plataformas.

particionamento de dados

Dimensionar um sistema distribuindo a carga com base em alguma faceta dos dados. Para
exemplo de divisão de carga com base no cliente ou no tipo de produto.

controles de detetive

Um controle de segurança que ajudará você a identificar se um ataque está em andamento/tem
aconteceu.

acoplamento de domínio

Uma forma de acoplamento em que um microsserviço é "acoplado" ao domínio
protocolo exposto por outro microsserviço.

design orientado por domínio (DDD)

Um conceito em que o problema/domínio de negócios fundamental é
modelado explicitamente no software.

Docker

Um conjunto de ferramentas para ajudar a criar e gerenciar contêineres.

equipe capacitadora

Uma equipe que apoia equipes alinhadas ao fluxo na realização de seu trabalho. Normalmente,
uma equipe capacitadora tem um foco específico, por exemplo, usabilidade, arquitetura,
segurança.

orçamento de erro

Relaciona-se ao nível aceitável em que um SIQ pode estar fora do limite,
normalmente definido em um grau aceitável de tempo de inatividade para um serviço.

evento

Algo que acontece no sistema que outras partes do sistema podem preocupe-se, por exemplo, com "Pedido feito" ou "Login do usuário".

ramificação de características

Criando uma nova ramificação para cada recurso que está sendo trabalhado, mesclando que voltam para a linha principal quando o recurso é concluído. Algo que eu desencorajar.

Função como serviço (FaaS)

Um tipo de plataforma sem servidor que invoca código arbitrário com base em determinados tipos de gatilhos - por exemplo, lançando código em reação a um HTTP chamada ou uma mensagem sendo recebida

governança

Concordar como as coisas devem ser feitas e garantir que elas sejam feitas assim caminho.

extensibilidade elegante

Como lidamos bem com uma situação inesperada.

GraphQL

Um protocolo que permite ao cliente emitir consultas personalizadas que podem resultar em chamadas feitas para vários microsserviços downstream. Útil para ajudar agregação e filtragem de chamadas para clientes externos sem exigir o uso de gateways BFF ou API.

duplicação horizontal

Dimensionar um sistema com várias cópias de uma coisa.

idempotência

A propriedade de uma função em que, mesmo que seja chamada várias vezes, o resultado é o mesmo. Útil para permitir que as operações em microsserviços sejam tentou novamente com segurança.

capacidade de implantação independente

A capacidade de fazer uma alteração em um microserviço e implantá-lo em produção sem precisar mudar ou implantar qualquer outra coisa.

ocultação de informações

Uma abordagem em que todas as informações estão ocultas por padrão dentro de um limite, e somente o mínimo necessário é exposto para satisfazer o externo consumidores.

infraestrutura como código

Modelando sua infraestrutura em forma de código, permitindo a infraestrutura o gerenciamento deve ser automatizado e o código deve ser controlado por versão.

Token Web JSON

Um padrão para criar uma estrutura de dados JSON que pode ser opcionalmente criptografado. Normalmente é usado para transmitir informações sobre diretores autenticados.

Kubernetes

Uma plataforma de código aberto que gerencia cargas de trabalho de contêineres em várias máquinas subjacentes.

biblioteca

Um conjunto de código que é empacotado de forma que possa ser reutilizado em vários programas.

implantação em etapas

A necessidade de implantar duas ou mais coisas ao mesmo tempo, porque um ocorreu uma mudança que a exige. O oposto de independente capacidade de implantação. Em geral, evite.

mensagem

Algo enviado para um ou mais microsserviços downstream por meio de um mecanismo de comunicação assíncrona, como um corretor. Pode conter uma variedade de cargas úteis, como uma solicitação, uma resposta ou um evento.

corretor de mensagens

Software dedicado que gerencia a comunicação assíncrona entre processos, normalmente fornecendo recursos como garantidos entrega (para alguma definição da palavra garantida).

microsserviço

Um serviço implantável de forma independente que se comunica com outros microsserviços por meio de um ou mais protocolos de comunicação.

monorepo

Um único repositório que contém todo o código-fonte de todos os seus microsserviços.

multirepositório

Uma abordagem na qual cada microsserviço tem seu próprio código-fonte repositório.

orquestração

Um estilo de saga em que uma unidade central (também conhecida como orquestrador) gerencia o operação de outros microsserviços para realizar um processo de negócios.

informações de identificação pessoal (PII)

Dados que, quando usados isoladamente ou em adição a outras informações, pode ser usado para identificar um indivíduo.

controle preventivo

Um controle de segurança que visa impedir que um ataque aconteça.

diretor

Algo - normalmente uma pessoa, embora também possa ser um programa - que está solicitando a autenticação e a autorização para obter acesso.

requisito

Enviado por um microsserviço para outro solicitando o microsserviço downstream para fazer alguma coisa.

resposta

Transmitido de volta como resultado de uma solicitação.

controle responsivo

Um controle de segurança que ajuda você a responder durante/após um ataque.

robustez

A capacidade de um sistema continuar operando mesmo quando algo está ruim acontece.

saga

Uma forma de modelar operações de longa duração de forma que os recursos não precisam ficar trancados por longos períodos de tempo. Preferido a transações distribuídas ao implementar processos de negócios.

sem servidor

Um termo abrangente para produtos em nuvem que, do ponto de vista do usuário, abstrai os computadores subjacentes, na medida em que o usuário não precisa mais se preocupar com eles. Exemplos desses produtos incluem AWS Lambda, AWS S3 e Azure Cosmos.

contrato de nível de serviço (SLA)

Um acordo entre um usuário final e um provedor de serviços (por exemplo, cliente) e fornecedor) que define a oferta mínima aceitável de serviços, e as penalidades que se aplicam se o acordo não for cumprido.

indicador de nível de serviço (SLI)

Uma medida de como seu sistema está se comportando, por exemplo, uma resposta tempo.

objetivo de nível de serviço (SLO)

Um acordo sobre qual é a faixa aceitável de um determinado SLI.

malha de serviços

Um tipo distribuído de middleware que fornece funcionalidade transversal principalmente para chamadas síncronas ponto-a-ponto, exemplo, mútuas TLS, descoberta de serviços ou disjuntores.

arquitetura orientada a serviços (SOA)

Um tipo de arquitetura em que o sistema é dividido em serviços que pode ser executado em máquinas diferentes. Microserviços são um tipo de SOA que prioriza a capacidade de implantação independente.

aplicativo de página única (SPA)

Um tipo de interface gráfica de usuário em que a interface do usuário é fornecida em uma única painel do navegador, sem a necessidade de navegação para outras páginas da web.

equipe alinhada ao fluxo

Uma equipe focada na entrega de ponta a ponta de um valioso fluxo de trabalho. Essa é uma equipe de longa duração que normalmente estará diretamente focada no cliente e corte dados, código de back-end e front-end.

forte propriedade

Um estilo de propriedade no qual partes do sistema são de propriedade de uma pessoa específica equipes, e mudanças em uma parte específica do sistema só podem ser feitas por a equipe que o possui.

adaptabilidade sustentada

A capacidade de se adaptar continuamente a ambientes em mudança, partes interessadas, e demandas.

modelagem de ameaças

O processo no qual você comprehende as ameaças que podem ser trazidas para controle seu sistema e priorize quais ameaças precisam ser abordadas.

desenvolvimento baseado em troncos

Um estilo de desenvolvimento em que todas as mudanças são feitas diretamente no tronco principal do sistema de controle de origem, incluindo alterações que ainda não foram feitas completo.

linguagem onipresente

Definindo e adotando uma linguagem comum para ser usada em código e em descrevendo o domínio, para auxiliar na comunicação Um conceito do DDD.

escala vertical

Melhorar a escala do sistema com a obtenção de uma máquina mais potente.

máquina virtual (VM)

Uma emulação de uma máquina em que a máquina aparece para todos os efeitos e tem o objetivo de ser uma máquina física dedicada.

widget

Um componente de uma interface gráfica de usuário.

Índice

Símbolos

2PC (algoritmos de confirmação em duas fases), transações distribuídas em duas fases

Confirmações – Transações distribuídas – Confirmações em duas fases

UMA

Testes A/B, testes A/B

ACID (atomicidade, consistência, isolamento e durabilidade), transações ACID

Active Directory, implementações comuns de login único

adaptabilidade, adaptabilidade sustentada, engenharia do caos, resumo, resumo

agregar, agregar e agregar, mapear agregados e contextos limitados
para microserviços, glossário

alarmes, versus alertas, fadiga de alertas

fadiga de alerta, fadiga de alerta

alertando, alertando - Para um melhor alerta

Amazon Web Services (AWS)

Gateways de API, Service Meshes e Gateways de API

equipes autônomas orientadas a produtos, garantindo consistência

escalonamento automático em, gerenciamento do estado desejado, escalonamento automático - escalonamento automático

zonas de disponibilidade, espalhando seu risco

AWS Lambda, contêineres Windows

AWS Secrets Manager, Secrets

Beanstalk, opções de implantação

bibliotecas de cliente, bibliotecas de cliente

CloudWatch, implementações

contêineres e, isolados, de forma diferente

credenciais e segurança, credenciais do usuário, rotacão, backups

definido, Glossário

gerenciamento do estado desejado e, Gerenciamento do estado desejado

registros de serviços dinâmicos, Rolling your own

Elasticsearch e, implementações

abraçando o fracasso, experimentos de produção

Serviços de FaaS, opções de implantação, desafios

desenvolvimento na nuvem, experiência do desenvolvedor

execução isolada, Execução isolada

limitações de, Limitações

VMs gerenciadas ativadas, boas para microsserviços?

participação de mercado de, Multilocação e Federação

corretores de mensagens, Choices

sistemas de provisionamento sob demanda, escalabilidade

ferramentas específicas da plataforma, Infraestrutura como Código (IAC)

Serviço de banco de dados relacional (RDS), implantação e escalabilidade do banco de dados

selecionando, você deve usá-lo?

Disponibilidade do SLA, várias instâncias, contrato de nível de serviço,

Redundância

virtualização tipo 2, custo da virtualização

escalonamento vertical, implementação

deteção de anomalias, The Expert in the Machine

Ansible, qual opção de implantação é ideal para você?

antifragilidade, e o mundo real

Apache Flink, Transmissão

Gateways de API, mantenha suas APIs independentes de tecnologia, Service Meshes e
Gateways de API - O que evitar, glossário

contêineres de aplicativos, contêineres de aplicativos

segurança de aplicativos (veja também segurança)

capacidade de reconstruir, reconstruir

backups, Backups

credenciais, escopo de limitação de credenciais

aplicação de patches, aplicação de patches

estado do aplicativo, hipermídia como motor de, Hipermídia como motor de
estado do aplicativo

arquitetos, O que há em um nome?

(veja também arquitetos evolucionistas)

princípios arquitetônicos

definindo padrões para microserviços, The Required Standard-

Segurança arquitetônica

guiando a arquitetura evolutiva, Guiando uma arquitetura evolutiva

arquitetura heterogênea, Heterogeneidade de tecnologia

camadas internas versus externas, organizacional

princípios e práticas, uma abordagem baseada em princípios - um mundo real

Exemplo

sistemas autônomos (SCSs), quando usá-los

arquitetura de software definida, O que é arquitetura de software?

arquitetura de três camadas, alinhamento de arquitetura e organização-

Alinhamento de arquitetura e organização

arquitetura vertical, alinhamento de arquitetura e organização

segurança arquitetônica, Segurança arquitetônica

criação de artefatos, Criação de artefatos

construção async/await, desvantagens

AsyncAPI, interface explícita

chamadas assíncronas sem bloqueio, Padrão: Asynchronous Nonblocking-\\ Where para usá-lo

confirmações atômicas, versus implantação atômica, Padrão: Monorepo

atomicidade, consistência, isolamento e durabilidade (ACID), transações ACID

autenticação e autorização

centralizada, autorização upstream, centralizada, upstream

Autorização

login único comum (SSO), login único comum

Implementações

problema do deputado confuso, O problema do deputado confuso- O confuso

Problema do deputado

autORIZAÇÃO DESCENTRALIZADA, Autorização descentralizada

definido, Autenticação e autorização, Glossário

autorização refinada, autorização refinada

autenticação humana, Autenticação humana

JSON Web Token (JWT), JSON Web Tokens – Desafios

autenticação mútua, identidade do cliente

autenticação de serviço a serviço, autenticação de serviço a serviço

gateways de login único, gateway de login único - Login único

Gateway

automação

na detecção de anomalias, The Expert in the Machine

como essencial para a segurança, a automação

durante a implantação, concentre-se na automação

autonomia, Sobre autonomia, resumo

escalonamento automático, escalonamento automático

disponibilidade, quanto é demais?, Sacrificando a disponibilidade

Azure

Insights e implementações de aplicativos

Funções do Azure, opções de implantação e limitações

Aplicativos Web do Azure, você deve usá-los?

Cofre de chaves do Azure, segredos

benefícios de, você deve usá-lo?

definido, Glossário

gerenciamento do estado desejado, Gerenciamento do estado desejado

produto de grade de eventos, interface explícita

limitações de, Limitações-Limitações

VMs gerenciadas ativadas, boas para microsserviços?

participação de mercado de, Multilocação e Federação

ferramenta openapi-diff, detecte alterações acidentais de quebra antecipada

serviços de nuvem pública oferecidos por, Public Cloud e Serverless

Disponibilidade de SLA, várias instâncias

B

backend para padrão de frontend (BFF), Padrão: Backend para frontend (BFF) -

Quando usar

Ferramenta de bastidores, The Self-Describing System

backups, Backups, Criptografar backups

compatibilidade com versões anteriores, facilite a compatibilidade com versões anteriores (veja também

mudanças significativas)

recuperação para trás, versus para frente, modos de falha do Saga

Ferramenta Biz Ops, The Self-Describing System

autópsia irrepreensível, Culpa

implantação azul-verde, Parallel Run

contexto limitado

alternativas aos limites do domínio de negócios, Alternativas aos negócios

Limites de domínio - organizacionais

definido, contexto limitado, glossário

definindo limites do sistema, Definindo limites do sistema - Definindo

Limites do sistema

modelos ocultos, Modelos ocultos

mapeamento para microsserviços, mapeamento de agregados e contextos limitados

para microsserviços

modelos compartilhados, modelos compartilhados

Ferramentas de BPM (modelagem de processos de negócios), sagas orquestradas

Brakeman, incorpore a segurança ao processo de entrega

ramificação, modelos de ramificação

mudanças significativas

evitando, evitando alterações interrompidas - Detecte quebras accidentais

Mudanças antecipadas

gerenciamento, gerenciamento de mudanças significativas - medidas extremas

testes de fragilidade, testes de escamação e fragilidade

fragilidade, fragilidade

construa pipelines, construa pipelines e entrega contínua e construa pipelines

e entrega contínua

anteparas, padrões de estabilidade, anteparas, glossário

funcionalidade de negócios, alinhamento da arquitetura com o alinhamento da arquitetura e alinhamento organizacional de arquitetura e organização

ferramentas de modelagem de processos de negócios (BPM), sagas orquestradas

testes voltados para negócios, tipos de testes

C

caching

noções básicas de, Caching

envenenamento por cache, Envenenamento por cache: um conto de advertência

invalidação, invalidação-Write-Behind

benefícios de desempenho, para desempenho

benefícios de robustez, para robustez

benefícios de escalabilidade, For Scale

compensações entre frescor e otimização, a regra de ouro da Atualização do cache versus otimização

onde armazenar em cache, Onde armazenar em cache - Solicitar cache

lançamentos canários, Canary Release, Canary release

Teorema CAP

disponibilidade, sacrificando a disponibilidade

abordagem combinada, não é tudo ou nada

componentes do Teorema CAP - Teorema CAP

consistência, sacrificando a consistência

tolerância de partição, sacrificando a tolerância de partição?

aplicação no mundo real de. E o mundo real

selecionando sua abordagem, AP ou CP?

cardinalidade, baixa versus alta, baixa versus alta cardinalidade

CDCs (contratos voltados para o consumidor), detecte mudanças accidentais

Testes antecipados de contratos e contratos orientados ao consumidor (CDCs) - é sobre conversas

padrão de gateway de agregação central, Padrão: Gateway de agregação central-

Quando usá-lo

CFR (requisitos multifuncionais), testes multifuncionais, quanto

É demais? , Glossário

alterar avaliações, Alterar avaliações

engenharia do caos, engenharia do caos, engenharia do caos - da robustez

para Beyond

Chaos Monkey, experimentos de produção

Kit de ferramentas de caos, da robustez ao mais

Chef, qual opção de implantação é ideal para você?

sagas coreografadas, Implementando sagas, Sagas coreografadas - Devo eu

usa coreografia ou orquestração (ou uma mistura)? , Fluxo de trabalho

coreografia, Glossário

disjuntores, padrões de estabilidade, disjuntores-disjuntores,

Glossário

identidade do cliente, identidade do cliente

bibliotecas de cliente, bibliotecas de cliente

armazenamento em cache do lado do cliente, do lado do cliente

desenvolvimento de código fechado, modelos de ramificação

Fundação de computação nativa em nuvem (CNCF), a computação nativa em nuvem Federação

CloudEvents, interface explícita

COBIT (Objetivos de Controle para Tecnologias da Informação), Governança e a estrada pavimentada

ramificação de código, modelos de ramificação

organização de código

um repositório para vários microsserviços, Pattern: Monorepo-Where to use esse padrão

um repositório por microsserviço, Padrão: um repositório por Microservice (também conhecido como Multirepo) - Onde usar esse padrão

repositório único e gigante, One Giant Repo, One Giant Build-One Giant Repo, uma construção gigante

reutilização de código, DRY e os perigos da reutilização de código em um mundo de microsserviços-

Bibliotecas de cliente, reutilizando código em repositórios

revisões de código, revisões de alterações - código síncrono versus assíncrono revisões

coesão

acoplamento e, A interação de acoplamento e coesão

definido, Coesão, Glossário

colaboração, resumo

propriedade coletiva, Propriedade coletiva-Propriedade coletiva, Glossário

Padrão de Segregação de Responsabilidade por Consulta de Comandos (CQRS), Start Small

comandos, versus solicitações, Padrão: Comunicação entre solicitação e resposta

comentários e perguntas, Como entrar em contato conosco

acoplamento comum, Acoplamento comum-Acoplamento comum

login único comum (SSO), implementações comuns de login único

estilos de comunicação

assíncrono sem bloqueio, Padrão: Asynchronous Nonblocking-Where

para usá-lo

comandos versus solicitações, Padrão: Comunicação entre solicitação e resposta

comunicação por meio de dados comuns, comunicação de padrões por meio de

Dados comuns: onde usá-los

complexidade introduzida por, Prossiga com cuidado

orientado por eventos, Padrão: comunicação orientada por eventos - Onde usá-la

chamadas em processo versus chamadas entre processos, de em processo a entre processos-

Tratamento de erros

estilos de mistura, Mix and Match

visão geral de, Estilos de comunicação por microsserviço, Comunicação

Estilos

chamadas paralelas versus sequenciais, implementação: síncrona versus

Assíncrono

solicitação-resposta, Padrão: Comunicação de solicitação-resposta - Onde

Use-o

bloqueio síncrono, Padrão: Bloqueio síncrono - Onde usá-lo

suporte tecnológico, Tecnologia para comunicação entre processos:

Tantas opções

comunicação, definida, Arquitetura em uma organização alinhada ao fluxo

grupos de comunidade de prática (CoP), Comunidades de Prática

compatibilidade, garantindo versões anteriores, facilitando a compatibilidade com versões anteriores (consulte também interrompendo mudanças)

compensando transações, reversões da Saga

composabilidade, Composabilidade

GETs condicionais, GETs condicionais

Registro de esquema confluente, detecte alterações acidentais de interrupção antecipada

problema do deputado confuso, O problema do deputado confuso- O confuso

Problema do deputado

consistência, sacrificando a consistência, garantindo consistência

Cônsul, Cônsul

contratos orientados ao consumidor (CDCs), detecte mudanças acidentais

Testes antecipados de contratos e contratos orientados pelo consumidor (CDCs) - O final

Palavra

abordagem que prioriza o consumidor, O Contrato Social

orquestração de contêineres, The Case for Container Orchestration (veja também Kubernetes)

virtualização baseada em contêineres, custo da virtualização

contêineres (veja também contêineres de aplicativos)

noções básicas de, Isolado, isolado de forma diferente, diferentemente

definido, Glossário

Docker, Docker

desvantagens de, Não é perfeito

aptidão para microserviços, Aptidão para microserviços

papel em microserviços, contêineres e Kubernetes, contêineres

questões de segurança, aplicação de patches

Windows e, contêineres Windows

acoplamento de conteúdo, acoplamento de conteúdo - acoplamento de conteúdo

contexto, fornecimento de ferramentas de monitoramento, fornecimento de contexto

entrega contínua (CD), Build Pipelines e Continuous Delivery-Build

Pipelines e entrega contínua, glossário

implantação contínua, construção de pipelines e entrega contínua, glossário

integração contínua (CI), uma breve introdução à integração contínua-

Modelos de ramificação, glossário

quebras de contrato

evitando mudanças significativas, evitando mudanças significativas - Catch

A quebra acidental muda precocemente

gerenciamento, gerenciamento de mudanças significativas - medidas extremas

estrutural versus semântico, Contrato estrutural versus semântico

Quebras

testes de contrato, testes de contrato e contratos orientados ao consumidor (CDCs)

Objetivos de controle para tecnologias da informação (COBIT), governança e
a estrada pavimentada

Lei de Conway, Alinhamento de Arquitetura e Organização, Lei de Conway-

Netflix e Amazon, a lei de Conway ao contrário, glossário

Grupos de CoP (comunidade de prática), Comunidades de Prática

comitês principais, papel interno de código aberto dos comitês principais

custos, Custo

COTS (software personalizável pronto para uso), vantagens, glossário

acoplamento

coesão e, A interação de acoplamento e coesão

acoplamento comum, Acoplamento comum-Acoplamento comum

acoplamento de conteúdo, acoplamento de conteúdo - acoplamento de conteúdo

definido, Glossário

acoplamento de domínio, acoplamento de domínio

solto versus apertado, acoplamento

acoplamento pass-through, acoplamento pass-through - acoplamento pass-through

acoplamento patológico, acoplamento de conteúdo

acoplamento de tecnologia, acoplamento de tecnologia

acoplamento temporal, acoplamento de domínio

tipos de, Tipos de acoplamento

Padrão CQRS (Command Query Responsibility Segregation), Start Small

credenciais

incluindo acidentalmente chaves no código-fonte, Revogação

desafios de microsserviços, credenciais

limitando o escopo de, Limitando o escopo limitador

revogação, revogação

girando com frequência, rotação

segredos, Segredos-Segredos

credenciais do usuário, credenciais do usuário

requisitos multifuncionais (CFR), testes multifuncionais, quanto
é demais? , Glossário

testes interfuncionais, testes multifuncionais - testes de robustez

definições de recursos personalizados (CRDs), Helm, Operators e CRDs, Oh My!

software personalizável pronto para uso (COTS), vantagens, glossário

cibersegurança, cinco funções de, As cinco funções da cibersegurança-

Recuperar

(veja também segurança)

D

dados

comunicação por meio de dados comuns, Padrão: Comunicação Através

Dados comuns: onde usá-los

preocupações com a decomposição, relatórios de preocupações com a decomposição de dados

Banco de dados

durabilidade de, Quanto é demais?

dados de alta cardinalidade, baixa versus alta cardinalidade

influência na decomposição, Dados

mantendo a consistência de, consistência de dados

protegendo, protegendo backups com criptografia de dados

particionamento de dados, Limitações de particionamento de dados, Glossário

transações de banco de dados

Transações ACID, Transações ACID

definido, Transações de banco de dados

sem atomicidade, ainda ácido, mas sem atomicidade? -Ainda ácido, mas
Falta de atomicidade?

bancos de dados

questões de implantação, The Database-Environments

ocultando, O que são microserviços?

preocupações de integridade, integridade de dados

preocupações com o desempenho, Desempenho e Desempenho

bancos de dados de relatórios, Banco de dados de relatórios

compartilhado, possuindo seu próprio estado

preocupações com ferramentas, Ferramentas

questões de transação, Transações

Debezium, Transmissão

descentralização, organizações fracamente acopladas

autorização descentralizada, Autorização descentralizada

decomposição (veja também em interfaces de usuário (UIs)),

abordagem combinada de, The Monolith Is Rarely the Enemy

perigos do prematuro, Os perigos da decomposição prematura

preocupações com dados, banco de dados de relatórios de preocupações com a decomposição de dados

definição de metas, Tenha uma meta, Metas estratégicas

migrações incrementais, Migração incremental

influência dos dados em, Data

camadas de, Decomposição por Layer-Data First

padrões para, Padrões de decomposição úteis

selecionando um ponto de partida, o que dividir primeiro? - O que dividir primeiro?

baseado em volatilidade, volatilidade

defesa em profundidade, Defesa em profundidade

gargalos de entrega, gargalos de entrega, conectáveis, modulares

Microsserviços

contenção de entrega, monólitos e contenção de entrega

implantação

contêineres de aplicativos, contêineres de aplicativos

implantação azul-verde, Parallel Run

orquestração de contêineres e Kubernetes, Kubernetes e Container

Orquestração - você deve usá-la?

contêineres, Containers-Fitness para microsserviços

implantação contínua, construção de pipelines e entrega contínua,

Glossário

preocupações com o banco de dados, The Database

facilidade de uso de microsserviços, facilidade de implantação

ambientes, Ambientes-Ambientes

Função como serviço (FaaS), função como serviço (FaaS) - O caminho

para frente

no Kubernetes, uma visão simplificada dos conceitos do Kubernetes

implantação em lockstep, Lockstep Deployment

passando do lógico para o físico, do lógico para o físico.

várias instâncias de cada serviço, várias instâncias

opções para, Opções de implantação

visão geral de, Implantação

máquinas físicas, Máquinas físicas

Plataforma como serviço (PaaS), Plataforma como serviço (PaaS)

princípios de, Princípios de implantação de microserviços - GITOps,

Resumo

entrega progressiva, entrega progressiva - execução paralela

papel do Puppet, Chef e outras ferramentas, qual é a opção de implantação

Ideal para você?

selecionando a plataforma de implantação correta, qual é a opção de implantação

Ideal para você?, Resumo

máquinas virtuais (VMs), máquinas virtuais - boas para microserviços?

implantação sem tempo de inatividade, implantação sem tempo de inatividade

gerenciamento do estado desejado, gerenciamento do estado desejado - Gitops

controles de detetive, Defesa em profundidade, Glossário

acontecimento

criação de artefatos, Criação de artefatos

construa pipelines, construa pipelines e entrega contínua-construção

Pipelines e entrega contínua

incorporando segurança na entrega de software, incorporando segurança no

Processo de entrega

desafios da Experiência do Desenvolvedor

organização de código, mapeamento de código-fonte e compilações para microsserviços-
Qual abordagem eu usaria?

entrega contínua (CD), Build Pipelines e Entrega Contínua-

Crie pipelines e entrega contínua, glossário

integração contínua (CI), uma breve introdução ao contínuo

Modelos de integração e ramificação, glossário

ferramentas, Ferramentaria

compensações e ambientes, compensações e ambientes

serviços de diretório, implementações comuns de login único

monólitos distribuídos, O monólito distribuído

rastreamento distribuído, agregação de registros e rastreamento distribuído, distribuído

Rastreamento - Implementação do rastreamento distribuído

transações distribuídas

evitando transações distribuídas - basta dizer não

versus sagas, Sagas versus transações distribuídas

confirmações bifásicas, transações distribuídas - confirmações bifásicas-

Transações distribuídas - confirmações em duas fases

DNS (Sistema de Nomes de Domínio), Sistema de Nomes de Domínio (DNS)

Docker, Docker, Glossário

Dockerfiles, qual opção de implantação é ideal para você?

documentação, de serviços, serviços de documentação - A autodescrição

Sistema

acoplamento de domínio, acoplamento de domínio, glossário

eventos de domínio, O processo

Sistema de nomes de domínio (DNS), Sistema de nomes de domínio (DNS)

design orientado por domínio (DDD), modelado em torno de um domínio comercial, apenas

Design orientado por domínio suficiente - O caso do design orientado por domínio para

Microserviços, o que são microsserviços? Glossário

DRY (não se repita), DRY e os perigos da reutilização de código em um

Mundo dos microsserviços

registros de serviços dinâmicos, registros dinâmicos de serviços - não esqueça o

Humanos!

E

facilidade de implantação, facilidade de implantação

EEMUA (Associação de Usuários de Equipamentos e Materiais de Engenharia), Toward

melhor alerta

Elasticsearch, Implementações

empatia, resumo

capacitando equipes, compartilhando especialistas, capacitando equipes - A estrada pavimentada,

Arquitetura em uma organização alinhada ao fluxo

testes de ponta a ponta

alternativas para, Você deve evitar testes de ponta a ponta? -A palavra final

implementando, implementando (aqueles complicados) testes de ponta a ponta - Falta de

Testabilidade independente

escopo de testes de ponta a ponta

Associação de Usuários de Equipamentos e Materiais de Engenharia (EEMUA), Toward

melhor alerta

engenheiros, o que há em um nome?

(veja também arquitetos evolucionistas),

programação em conjunto, Programação em conjunto

ambientes

durante a implantação, Ambientes-Ambientes

durante o desenvolvimento, compensações e ambientes

orçamentos de erro, Orçamentos de erro, Glossário

tratamento de erros, tratamento de erros, tratamento de exceções

etcd, etcd e Kubernetes

fornecimento de eventos, Start Small

invasão de eventos, tempestade de eventos - O processo

comunicação orientada por eventos, Padrão: Comunicação orientada por eventos - Onde

para usá-lo, estilos de comunicação

eventos

componentes de, O que há em um evento? -Eventos totalmente detalhados

definido, Glossário

versus mensagens, Padrão: comunicação orientada por eventos

arquitetos evolucionários

arquitetura em organizações alinhadas ao fluxo, Arquitetura em um fluxo-

Organização alinhada

construindo equipes, Construindo uma equipe

conceito de, Arquitetura

principais responsabilidades de, Resumo

definindo padrões para microserviços, The Required Standard-

Segurança arquitetônica

definindo limites do sistema, Definindo limites do sistema - Definindo

Limites do sistema

tratamento de exceções, Tratamento de exceções

governança e a estrada pavimentada, Governança e a estrada pavimentada-A

Estrada pavimentada em grande escala

guiando a arquitetura evolutiva, Guiando uma arquitetura evolutiva

Habitabilidade e, Habitabilidade

tornando a mudança possível, tornando a mudança possível

princípios e práticas, uma abordagem baseada em princípios - um mundo real

Exemplo

papel de, O que há em um nome? - O que há em um nome?

construção social de, Uma construção social

arquitetura de software definida, O que é arquitetura de software?

dívida técnica e, Dívida técnica

visão para, Uma visão evolutiva para o arquiteto

esquemas explícitos, torne sua interface explícita, você deve usar esquemas? ,

Interface explícita, esquemas explícitos

testes exploratórios, tipos de testes

extensibilidade, graciosa, extensibilidade graciosa

F

ramificação de recursos, modelos de ramificação, glossário

alternâncias de recursos, alternância de recursos, alternância de recursos

federação (Kubernetes), multilocação e federação-multilocação e Federação

autorização refinada, autorização refinada

Firecracker, contêineres Windows

funções de condicionamento físico, orientando uma arquitetura evolutiva

testes de escamação, testes de escamação e fragilidade

flexibilidade, flexibilidade

Fluentes, implementações

foco na automação, foco na automação

Modos de falha do Saga de recuperação para frente versus para trás

fragilidade, e o mundo real

frameworks, modelo de microsserviço personalizado

equipes de front-end, motoristas para equipes dedicadas de front-end

equipes completas, rumo a equipes alinhadas ao fluxo

virtualização completa, isolada, de forma diferente

Função como serviço (FaaS), nuvem pública e sem servidor, função como

Serviço (FaaS) - O caminho a seguir, Glossário

decomposição funcional, Decomposição funcional - Limitações

G

exercícios de dia de jogo, dias de jogo

Modelo de desenvolvimento GitFlow, modelos de ramificação

GitOps, GitOps

consistência global, modelos de balanceamento

regra de ouro do armazenamento em cache, A regra de ouro do armazenamento em cache

governança, governança e estrada pavimentada - A estrada pavimentada em escala,

Resumo, Glossário

extensibilidade elegante, Extensibilidade elegante, Glossário

Grafite, Implementações

GraphQL, GraphQL-Onde usá-lo, GraphQL-GraphQL, Glossário

entrega garantida, entrega garantida

H

habitabilidade, Habitabilidade

código de autenticação de mensagens baseado em hash (HMAC), manipulação de dados

HATEOAS (hipermídia como mecanismo do estado do aplicativo), Hypermédia como o mecanismo do estado do aplicativo

Helm, Helm, Operators e CRDs, oh meu Deus!

Os Doze Fatores e Princípios de Heroku

arquiteturas heterogêneas, Heterogeneidade de tecnologia

Padrão de arquitetura hexagonal, microserviços em um piscar de olhos

modelos ocultos, Modelos ocultos

dados de alta cardinalidade, baixa versus alta cardinalidade

HMAC (código de autenticação de mensagens baseado em hash), manipulação de dados

Honeycomb, agregação de registros e rastreamento distribuído, implementações

arquitetura horizontal, Organizacional....

duplicação horizontal, Duplicação horizontal - Limitações, Glossário

HTTP (Protocolo de Transferência de Hipertexto)

REST e, REST e HTTP

autenticação humana, Autenticação humana

registro humano, O Sistema de Autodescrição...

Contêineres Hyper-V, contêineres Windows

hipermídia como motor do estado do aplicativo (HATEOAS), Hypermídia como o mecanismo do estado do aplicativo

hipervisores, custo da virtualização...



idempotência, Idempotência, Glossário

provedores de identidade, implementações comuns de login único

implementação

Gateways de API, Service Meshes e Gateways de API - O que evitar

reutilização de código, DRY e os perigos da reutilização de código em um microsserviço

Bibliotecas World-Client

serviços de documentação, Serviços de documentação - A autodescrição

Sistema

definição de metas, procurando a tecnologia ideal - Hide Internal

Detalhe da implementação, metas estratégicas

lidar com mudanças entre microsserviços, lidar com mudanças entre

Microsserviços

esquemas, esquemas - você deve usar esquemas?

formatos de serialização, formatos de serialização

descoberta de serviços, descoberta de serviços - não esqueça os humanos!

malhas de serviços, Service Meshes e API Gateways - Service Meshes e gateways de API, malhas de serviços - e quanto aos outros protocolos?

opções de tecnologia, Opções de tecnologia-Kafka

confiança implícita, confiança implícita

chamadas em processo versus chamadas entre processos, de em processo a entre processos-

Tratamento de erros

testes em produção, tipos de testes em produção, testes em produção-

Engenharia do caos

capacidade de implantação independente, capacidade de implantação independente, possuir sua própria

Estado, falta de testabilidade independente, o que são microsserviços? , Glossário

ocultação de informações, microsserviços em um piscar de olhos, possuir seu próprio estado,

Ocultação de informações, o que são microsserviços? , Glossário

infraestrutura como código (IAC), Infraestrutura como código (IAC), Glossário

testes de integração, testes de ponta a ponta

interfaces, expondo explicitamente, torne sua interface explícita, você deve usar

Esquemas? , Interface explícita, Esquemas explícitos

estrutura interna de código aberto, ferramentas internas de código aberto

invalidação

GETs condicionais, GETs condicionais

baseado em notificação, baseado em notificação

hora de viver (TTL), Hora de viver (TTL)

gravação por trás de caches, gravação por trás

caches de gravação, gravação

execução isolada, Execução isolada-Execução isolada

isolamento, isolamento

Istio, como eles funcionam e Knative

Arquitetos de TI, uma visão evolutiva para o arquiteto

(veja também arquitetos evolucionistas)

J

Jaeger, agregação de registros e rastreamento distribuído

JSON Web Token (JWT), JSON Web Tokens - Desafios, Glossário

json-schema-diff-validator, detecte alterações acidentais de interrupção antecipada

K

Kafka, Kafka

indicadores-chave de desempenho (KPIs), rumo a equipes alinhadas ao fluxo

chaves

incluindo acidentalmente no código-fonte, Revogação

digitalização, Revogação

armazenamento seguro de, tudo gira em torno das chaves

Kibana, Implementações

Kinesis, Escolhas

Knative e Knative

KSQLDB, Transmissão

Kubernetes

antecedentes de, Multilocação e Federação

noções básicas de, Uma visão simplificada dos conceitos do Kubernetes - uma visão simplificada

Visão dos conceitos do Kubernetes

vantagens e desvantagens de, você deve usá-lo?

Fundação de Computação Nativa em Nuvem (CNCF), The Cloud Native

Federação de Computação

gerenciamento de configuração em, etcd e Kubernetes

definido, Glossário

desenvolvimento futuro, O futuro

Knative e Knative

gerenciando aplicativos de terceiros, Helm, Operators e CRDs, Oh My!

multilocação e federação, multilocação e federação-multilocação

e Federação

plataformas e portabilidade, Plataformas e portabilidade

papel em microserviços, contêineres e Kubernetes

gerenciamento de segredos com, Secrets

L

latência, latência, quanto é demais?

bibliotecas

bibliotecas de cliente, bibliotecas de cliente

definido, Glossário

contribuições externas por meio de, Contribuição externa por meio de bibliotecas

compartilhando código via, Compartilhando código via bibliotecas

Lightstep, agregação de registros e rastreamento distribuído, implementações

Lightweight Directory Access Protocol (LDAP), login único comum

Implementações

eliminação de carga, anteparas

chamadas locais, versus chamadas remotas, chamadas locais não são como chamadas remotas

otimização local, modelos de balanceamento

implantação em etapas, implantação em etapas, glossário

agregação de registros

noções básicas de agregação de logs-agregação de registros

formato comum para, Formato comum

correlacionando linhas de registro, Correlacionando linhas de registro - Correlacionando linhas de registro

implementações, Implementações

papel em microsserviços, agregação de registros e rastreamento distribuído

deficiências de, Deficiências

cronometragem, cronometragem

transações de longa duração (LLTs), Sagas

acoplamento solto, Implantabilidade independente, Acoplamento

organizações fracamente acopladas, Organizações fracamente acopladas, Vagamente e
organizações fortemente acopladas

corretores gerenciados, Choices

testes exploratórios manuais, tipos de testes

tempo médio entre falhas (MTBF), tempo médio de reparo acima do tempo médio

Entre falhas?

tempo médio de reparo (MTTR), tempo médio de reparo acima do tempo médio

Entre falhas?

MELT (métricas, eventos, registros e tracos), Os pilares da observabilidade? Não

Tão rápido

corretores de mensagens, Corretores de mensagens-Kafka, Glossário

mensagens, Glossário

agregação de métricas, agregação de métricas - implementações

padrão de micro frontend, Padrão: Micro front-ends - quando usá-lo

microsserviços (veja também modelagem de microsserviços)

abordagem à aprendizagem, Navegando neste livro - Parte III, Pessoas

benefícios de, Prefácio, Vantagens da capacidade de composição de microsserviços, e

Arquitetura de microsserviços

melhores usos para, Devo usar microsserviços? -Onde eles funcionam bem

desafios dos pontos problemáticos do microsserviço - consistência de dados

definido, O que são microsserviços? Glossário

definindo padrões para, O padrão exigido de segurança arquitetônica

direções futuras, Olhando para o futuro

definição de metas, tenha uma meta, metas estratégicas, migração para microsserviços

conceitos-chave, conceitos-chave de microsserviços - alinhamento de

Arquitetura e organização

sistemas monolíticos, as vantagens monolíticas dos monólitos.....

visão geral de Microservices at a Glance-Microservices at a Glance

papel da tecnologia em, Habilitando a Tecnologia - Nuvem Pública e

Sem servidor

versus arquitetura orientada a serviços, microserviços em um piscar de olhos

middleware, Middleware

programação de multidões, programação em conjunto

zombando de colaboradores posteriores, Mocking or Stubbing

modelagem de microserviços (veja também desenvolvimento; microserviços)

alternativas aos limites do domínio de negócios, Alternativas aos negócios

Limites de domínio - organizacionais

design orientado por domínio, design orientado por domínio apenas o suficiente - The Case

para design orientado por domínio para microserviços

limites de microserviços, o que constitui um bom limite de microserviço?

-A interação de acoplamento e coesão

misturando modelos e exceções, Misturando modelos e exceções

tipos de acoplamento, Tipos de acoplamento e acoplamento de conteúdo

modelos

ramificação durante o desenvolvimento, modelos de ramificação

modelos ocultos, ocultos

compartilhados, modelos compartilhados

microserviços modulares, conectáveis, conjuntos modulares de microserviços

programação

monólitos modulares, O monólito modular

monitoramento (veja também observabilidade),

deteção automatizada de anomalias, The Expert in the Machine

desafios de microserviços, monitoramento e solução de problemas,

Disrupção, pânico e confusão

definindo padrões para, Monitoramento

começando, Começando

vários servidores, design de vários servidores, Vários serviços, Vários

Servidores

versus observabilidade, observabilidade versus monitoramento Os pilares da

Observabilidade? Monitoramento e observabilidade não tão rápidos

monitoramento de usuário real, monitoramento de usuário real

monitoramento semântico, monitoramento semântico-monitoramento de usuários reais

microserviço único, design de servidor único, microserviço único, único

Servidor

serviço único, design de vários servidores, microserviço único, vários

Servidores

padronização, Padronização

seleção de ferramentas, seleção de ferramentas adequadas para sua balança

padrão de front-end monolítico, Padrão: Frontend monolítico - Quando usá-lo

sistemas monolíticos (veja também decomposição)

vantagens de, Vantagens dos monólitos

coexistindo com microservices, o monólito raramente é o inimigo

definido, O Monólito

contenção de entrega e, Monólitos e contenção de entrega

distribuído, The Distributed Monolith

versus arquitetura antiga, vantagens dos monólitos

modular, o monólito modular

processo único, o monólito de processo único

abordagem monorepo, Padrão: Monorepo - Onde usar esse padrão, Glossário

servidor mountebank stub/mock, um serviço de stub mais inteligente

MTBF (tempo médio entre falhas), tempo médio de reparo acima do tempo médio
Entre falhas?

MTTR (tempo médio de reparo), tempo médio de reparo acima do tempo médio

Entre falhas?

abordagem multirepositório, padrão: um repositório por microserviço (também conhecido como

Multirepo) - Onde usar esse padrão, Glossário

multilocação (Kubernetes), multilocação e federação-multiterâncias e

Federação

autenticação mútua, identidade do cliente

TLS mútuo, identidade do cliente

N

Instituto Nacional de Padrões e Tecnologia (NIST), As Cinco Funções
da cibersegurança

requisitos não funcionais, testes multifuncionais

invalidação baseada em notificação, baseada em notificação

O

observabilidade (veja também monitoramento)

alertando, alertando - Para um melhor alerta

blocos de construção para, Blocos de construção para observabilidade

rastreamento distribuído, rastreamento distribuído - implementação da distribuição

rastreamento

agregação de registros, Deficiências da agregação de registros

agregação de métricas, agregação de métricas - implementações

versus monitoramento, observabilidade versus monitoramento - os pilares do

Observabilidade? Monitoramento e observabilidade não tão rápidos

monitoramento semântico, monitoramento semântico-monitoramento de usuários reais

saúde do sistema, estamos bem? -Orçamentos de erro

testes em produção, testes em engenharia de caos de produção

sistemas de provisionamento sob demanda, escalabilidade

desenvolvimento de código aberto, modelos de ramificação, código aberto interno

OpenAPI, desafios, interface explícita, esquemas explícitos

openapi-diff, detecte alterações acidentais mais cedo

OpenID Connect, implementações comuns de login único

Operador, leme, operadores e CRDs, meu Deus!

sagas orquestradas, implementação de sagas orquestradas, mistura de estilos-

Devo usar coreografia ou orquestração (ou uma mistura)? , Fluxo de trabalho

orquestração, Glossário

alinhamento organizacional, Alinhamento Organizacional, Organizacional-

Organizacional

estruturas organizacionais

benefícios da autonomia, Sobre a autonomia

benefícios de organizações fracamente acopladas, entendendo a de Conway

Lei

estudo de caso, Estudo de caso: Realestate.com.au-Estudo de caso: realestate.com.au

Lei de Conway, Lei de Conway-Netflix e Amazon

capacitando equipes, capacitando equipes - A estrada pavimentada

distribuição geográfica, Distribuição geográfica

dinâmica humana e, Pessoas

impacto do design do sistema nas organizações, a Lei de Conway em sentido inverso

código aberto interno, ferramentas internas de código aberto

organizações fracamente acopladas, Organizações fracamente acopladas

serviços órfãos, The Orphaned Service

visão geral de, Organização

microserviços conectáveis e modulares, conectáveis, microserviços modulares-

Programação em conjunto

microserviços compartilhados, gargalos de entrega de microserviços compartilhados

equipes pequenas, grande organização, equipes pequenas, grande organização-

Equipes pequenas, grande organização

propriedade forte versus propriedade coletiva, Strong Versus Collective

Modelos de equilíbrio de propriedade

tamanho da equipe, tamanho da equipe

serviços órfãos, The Orphaned Service

modelos de propriedade

mudando de propriedade, Mudando de propriedade

visão geral de, modelos de propriedade - drivers para front-end dedicado

Equipes

programação em pares, revisões de alterações - síncrona versus assíncrona

revisões de código

propriedade forte versus propriedade coletiva, Glossário, Glossário

P

Ferramenta de teste Pact, Pact

decomposição baseada em página, Padrão: Decomposição baseada em página - Onde

Use-o

programação em pares, revisões de alterações - código síncrono versus assíncrono

revisões

padrão de execução paralela, execução paralela, execução paralela, execução paralela

chamadas paralelas versus chamadas sequenciais, Implementação: Síncrona Versus

Assíncrono

tolerância de partição, sacrificando a tolerância de partição?

partições, Implementação

acoplamento pass-through, acoplamento pass-through - acoplamento pass-through

senhas, hashing de senha salgado, escolha o conhecido

aplicação de patches, aplicação de patches

acoplamento patológico, acoplamento de conteúdo

conceito de estrada pavimentada, A estrada pavimentada, Governança e a Estrada Pavimentada-A

Estrada pavimentada em grande escala

testes de desempenho, testes de desempenho

informações de identificação pessoal (PII), Glossário

Plataforma como serviço (PaaS), Plataforma como serviço (PaaS)

equipes da plataforma, A equipe da plataforma

pods (Kubernetes), uma visão simplificada dos conceitos do Kubernetes

validação de pré-produção, tipos de testes

controle preventivo, Defesa em profundidade, Glossário

principal, Autenticação e autorização, Glossário

princípio da defesa em profundidade, Defesa em profundidade

princípio do menor privilégio, Princípio do menor privilégio

chaves privadas, verificação, revogação

experimentos de produção, Experimentos de produção

ferramentas de monitoramento de produção, implementações

(veja também monitoramento)

abordagens de programação

programação em conjunto, Programação em conjunto

programação de multidões, programação em conjunto

programação estruturada Tipos de acoplamento

entrega progressiva, entrega progressiva - execução paralela

Prometheus, Implementações

Protolock, detecte alterações accidentais de interrupção antecipada

Puppet, qual opção de implantação é ideal para você?

Q

perguntas e comentários, Como entrar em contato conosco

sistemas baseados em filas, tópicos e filas

R

monitoramento de usuário real, monitoramento de usuário real

monitoramento em tempo real, em tempo real

rebote, capacidade de, rebote

redundância, Redundância

chamadas de procedimento remoto (RPCs), chamadas de procedimento remoto - Onde usá-las

conjuntos de réplicas (Kubernetes), uma visão simplificada dos conceitos do Kubernetes

bancos de dados de relatórios, Banco de dados de relatórios

relatórios, desafios dos microserviços, relatórios

Transferência de estado representacional (REST), REST - onde usá-la

caches de solicitação, cache de solicitações

comunicação solicitação-resposta, Padrão: Solicitação-Resposta

Comunicação - Onde usá-la, estilos de comunicação

pedidos

versus comandos, Padrão: Comunicação entre solicitação e resposta

definido, Glossário

resiliência

autópsia irrepreensível, Culpa

Teorema CAP, Teorema CAP - e o mundo real

desafios de, o fracasso está em toda parte

engenharia do caos, engenharia do caos - da robustez ao além

conceitos básicos de, O que é resiliência? -E arquitetura de microserviços

requisitos multifuncionais e, Quanto é demais?

impacto na funcionalidade, degradando a funcionalidade

visão geral de, Resiliência

gerenciamento de riscos, espalhando seu risco

padrões de estabilidade, Padrões de estabilidade - Idempotência

tempo de resposta, quanto é demais?

respostas, Glossário

responsabilidades, entendendo a arquitetura em um ambiente alinhado

Organização

controle responsável, Defesa em profundidade, Glossário

APIs baseadas em REST-over-HTTP, onde usá-las

Tentativas, Tentativas

gerenciamento de riscos, espalhando seu risco

robustez

assegurador, Robustez

benefícios do armazenamento em cache, para robustez

desafios de melhorar a robustez

engenharia do caos e, Engenharia do Caos

definido, Glossário

determinando quais opções usar, Resumo

princípio do Leitor Tolerante

testes de robustez, testes de robustez

reversões

reduzindo, reordenando as etapas do fluxo de trabalho para reduzir as reversões

reversões semânticas, reversões da Saga

ao usar sagas, Saga Rollbacks mistura fail-backward e fail-situações futuras

análise da causa raiz, Blame

RPCs (chamadas de procedimento remoto), chamadas de procedimentos remotos - Onde usá-las

S

sagas

benefícios de, Sagas

conceito central de, Sagas

definido, Glossário

versus transações distribuídas, Sagas versus transações distribuídas

modos de falha, Saga Failure Modes - Misturando fail-backward e fail-situações futuras

implementando, implementando Sagas - Devo usar coreografia ou orquestração (ou uma mistura)?

falta de atomicidade em, Sagas

limitações dos modos de falha do Saga

papel em microserviços, Sagas, fluxo de trabalho

Salt, qual opção de implantação é ideal para você?

hashing de senha salgado, escolha o conhecido

dimensionamento

escalonamento automático, escalonamento automático

evitando a otimização prematura, Start Small

benefícios do escalonamento

armazenamento em cache, envenenamento por cache: um conto de advertência

CORS e fornecimento de eventos, Start Small

particionamento de dados, limitações de particionamento de dados

bancos de dados durante a implantação, implantação e escalabilidade do banco de dados

quatro eixos de, Os quatro eixos da escala

decomposição funcional, Decomposição funcional - Limitações

duplicação horizontal, Duplicação horizontal - Limitações

modelos de microserviços e, The Paved Road at Scale

monitoramento e, adequado para sua escala

visão geral de, Scaling

redesenho do sistema e, começando de novo

usando modelos combinados para, Combinando modelos

escala vertical, Limitações de escala vertical

esquemas, torne sua interface explícita, Esquemas - você deve usar esquemas?

Backup da Schrödinger, Backups

SCSs (sistemas autônomos), quando usá-los

segredos

aspectos que exigem gerenciamento, Segredos

exemplos de, Secrets

ferramentas para gerenciar, Secrets

segurança

áreas de preocupação, Segurança

autenticação e autorização, Autenticação e autorização-

Desafios

desafios de microserviços, segurança, segurança

princípios fundamentais, princípios fundamentais: incorpore a segurança na entrega

Processo

cinco funções da segurança cibernética, As cinco funções da segurança cibernética-

Recuperar

fundamentos da segurança de aplicativos, Fundamentos do aplicativo

Reconstrução da segurança

confiança implícita versus confiança zero, confiança implícita versus confiança zero - é uma

Espectro

visão geral de, Segurança

protegendo dados, Protegendo backups com criptografia de dados

tipos de controles de segurança, Defesa em profundidade

sistemas autônomos (SCSs), quando usá-los

sistemas de autodescrição, O Sistema de Autodescrição

quebras semânticas, quebras estruturais versus semânticas do contrato

monitoramento semântico, monitoramento semântico-monitoramento de usuários reais

reversões semânticas, reversões da Saga

controle de versão semântico, interface explícita

chamadas sequenciais versus chamadas paralelas, implementação: síncrona versus

Assíncrono

serialização

formatos binários, formatos binários

formatos textuais, Formatos textuais

identidade do servidor, identidade do servidor

cache do lado do servidor, do lado do servidor do lado do servidor

sem servidor

benefícios da nuvem pública e sem servidor

definido, Plataforma como serviço (PaaS), Glossário

Função como serviço (FaaS), função como serviço (FaaS) - O caminho para frente

descoberta de serviços, descoberta de serviços - não esqueça os humanos!

malhas de serviço, malhas de serviço e gateways de API - Malhas de serviços e API

Gateways, malhas de serviços - e quanto aos outros protocolos?, E Knative,

Glossário

provedores de serviços, implementações comuns de login único

testes de serviço, testes de serviço, implementação de testes de serviço - um esboço mais inteligente

Serviço

contrato de nível de serviço (SLA), contrato de nível de serviço, glossário

indicador de nível de serviço (SLI), indicadores de nível de serviço, glossário

objetivo de nível de serviço (SLO), objetivos de nível de serviço, Quanto custa demais

Muito? , Glossário

arquitetura orientada a serviços (SOA), microsserviços em um piscar de olhos, glossário autenticacão de servico a servico, autenticacão de servico a servico serviços (Kubernetes), uma visão simplificada dos conceitos do Kubernetes serviços, documentação, documentação de serviços - O sistema de autodescrição fragmentos, implementação

modelos compartilhados, modelos compartilhados, gargalos de entrega de microsserviços compartilhados Serviço de Notificação Simples (SNS), Opções

Simple Queue Service (SQS), Opções

login único (SSO), implementações comuns de login único - login único No Gateway

aplicativo de página única (SPA), glossário

monólitos de processo único, O monólito de processo único

tamanho, tamanho

testes de fumaça, testes de fumaça

Snyk, incorpore a segurança ao processo de entrega

contrato social, O contrato social

arquitetura de software, O que é arquitetura de software?

Software como serviço (SaaS), onde eles funcionam bem

engenharia de software, O que há em um nome?

ramificação de código-fonte, modelos de ramificação

SPA (aplicativo de página única), Glossário

extensões (rastreamento distribuído), como funciona

Contrato Spring Cloud, outras opções

SSO (login único), implementações comuns de login único - login único

No Gateway

padrões de estabilidade

anteparas, anteparas

estudo de caso, padrões de estabilidade

disjuntores, disjuntores-disjuntores

idempotência, idempotência

isolamento, isolamento

middleware, Middleware

redundância, Redundância

Tentativas, Tentativas

intervalos, intervalos de tempo limite

padrão de figo estrangulador, padrão de figo Strangler, padrões de estabilidade

objetivos estratégicos, Objetivos estratégicos

equipes alinhadas ao fluxo, alinhamento da arquitetura e da organização, rumo

Equipes alinhadas ao fluxo que superam desafios técnicos, de forma vaga

Organizações acopladas, arquitetura em uma organização alinhada ao fluxo

Arquitetura em uma organização alinhada ao fluxo, glossário

streaming, papel em microsserviços, Streaming

propriedade forte, propriedade forte - Até onde vai uma propriedade forte?

Glossário

quebras estruturais, quebras estruturais versus semânticas de contratos

programação estruturada, Tipos de acoplamento

destruindo colaboradores, Mocking or Stubbing

adaptabilidade sustentada, Adaptabilidade sustentada, Engenharia do Caos, Resumo, Glossário

chamadas de bloqueio síncronas, Padrão: Bloqueio síncrono - Onde usá-lo

transações sintéticas, transações sintéticas - implementação de transações sintéticas transações

saúde do sistema, estamos bem? -Orçamentos de erro

arquitetos de sistemas (veja arquitetos evolucionários)

T

modelos de microserviços personalizados, Modelo de microserviço personalizado

público-alvo, quem deve ler este livro

equipes

construindo, construindo uma equipe

grupos de comunidade de prática (CoP), Comunidades de Prática

capacitando equipes, compartilhando especialistas, capacitando equipes - A estrada pavimentada,

Arquitetura em uma organização alinhada ao fluxo

garantindo consistência em todos os setores, garantindo consistência

equipes completas, rumo a equipes alinhadas ao fluxo

distribuição geográfica, Distribuição geográfica

equipes da plataforma, A equipe da plataforma

tamanho de, Tamanho da equipe

equipes pequenas, grande organização, equipes pequenas, grande organização-

Equipes pequenas, grande organização.

equipes alinhadas ao fluxo, rumo ao trabalho contínuo de equipes alinhadas
Desafios técnicos. organizações fracamente acopladas. arquitetura em um
Arquitetura organizacional alinhada ao fluxo em uma arquitetura alinhada ao fluxo

Organização

propriedade forte versus propriedade coletiva, em nível de equipe versus

Nível organizacional

APIs de equipe, equipes pequenas, grandes organizações

duas equipes de pizza, Netflix e Amazon

entendendo as responsabilidades, Arquitetura em um fluxo alinhado

Organização

dívida técnica, dívida técnica

opções de tecnologia

GraphQL, GraphQL - onde usá-lo

corretores de mensagens, corretores de mensagens-Kafka

opções exploradas, Opções de tecnologia

chamadas de procedimento remoto (RPCs), chamadas de procedimento remoto - Onde usar

Isso

Transferência de estado representacional (REST), REST - onde usá-la

equipes alinhadas e superando desafios técnicos

acoplamento de tecnologia, acoplamento de tecnologia

heterogeneidade de tecnologia, Heterogeneidade de tecnologia, Tecnologia

sobrecarga de tecnologia, sobrecarga de tecnologia

testes voltados para a tecnologia, tipos de testes

modelos, modelo de microserviço personalizado

acoplamento temporal, acoplamento de domínio

testando

Testes A/B, testes A/B

lançamentos canários, lançamento canário

desafios dos microserviços, testes

engenharia do caos, engenharia do caos

testes de contratos orientados pelo consumidor, detecte mudanças accidentais

Cedo

contratos orientados ao consumidor (CDCs), testes de contrato e consumidores

Contratos conduzidos (CDCs) - A palavra final

testes de contrato, testes de contrato e contratos orientados ao consumidor (CDCs)

testes interfuncionais, testes multifuncionais - testes de robustez

experiência do desenvolvedor, Experiência do desenvolvedor

alternativas de teste de ponta a ponta, você deve evitar testes de ponta a ponta? -O

Palavra final

implementação de teste de ponta a ponta, implementação (aqueles complicados) de ponta a ponta

Fim dos testes - falta de testabilidade independente

abordagem holística para, Resumo

testes exploratórios manuais, tipos de testes

visão geral de, Testing

Ferramenta de teste Pact, Pact

padrão de execução paralela, corrida paralela

testes de desempenho, testes de desempenho

pré-produção até testes em produção, da pré-produção à entrada

Teste de produção - tempo médio de reparo acima do tempo médio entre

Falhas?, Testes em engenharia de produção e caos

testes de robustez, testes de robustez

implementação de teste de serviço, implementação de testes de serviço - A Smarter

Serviço de esboço

testes de fumaça, testes de fumaça

transações sintéticas, transações sintéticas - implementação de transações sintéticas

transações

escopo do teste, Test Scope-Trade-Offs

tipos de testes, Tipos de testes - Tipos de testes

modelagem de ameaças, Glossário

três pilares da observabilidade, Os pilares da observabilidade? Não tão rápido

acoplamento estreito, Acoplamento

organizações fortemente acopladas, Organizações fracamente e estreitamente acopladas

tempo de vida (TTL), Sistema de nomes de domínio (DNS), Tempo de vida (TTL)

tempos limite, padrões de estabilidade - tempos limite

ferramentas, Ferramentaria

sistemas baseados em tópicos, tópicos e filas

tracos (rastreamento distribuído), como funciona

transações (consulte transações de banco de dados)

desenvolvimento baseado em troncos, modelos de ramificação, glossário

confiança, confiança

algoritmos de confirmação de duas fases (2 PCs), transações distribuídas - duas fases

Confirmações - Transações distribuídas - Confirmações em duas fases

duas equipes de pizza, Netflix e Amazon

virtualização tipo 2, custo da virtualização

U

linguagem ubíqua, linguagem ubíqua, glossário

testes unitários, testes unitários

autorização upstream, centralizada, autorização upstream

uso, rastreamento, rastreamento de uso

credenciais do usuário, credenciais do usuário

experiência do usuário, simplifique seu serviço para os consumidores

interfaces de usuário (UIs)

adaptando-se às restrições do dispositivo, restrições

backend para padrão de frontend (BFF), Padrão: Backend para frontend

(BFF) - Quando usar

padrão de gateway de agregação central, Padrão: Agregação central

Gateway - Quando usá-lo

desenvolvimentos ao longo do tempo, interfaces de usuário

GraphQL, GraphQL-GraphQL

abordagem híbrida, uma abordagem híbrida

fusão de interfaces de usuário digitais e móveis, Rumo ao digital

padrão de micro frontend, Padrão: Micro front-ends - quando usá-lo

padrão de front-end monolítico, Padrão: Frontend monolítico - Quando usar

É

visão geral de, Interfaces de usuário

modelos de propriedade, modelos de propriedade - drivers para front-end dedicado

Equipes

decomposição baseada em página, Padrão: Decomposição baseada em página-Onde

para usá-lo

equipes alinhadas ao fluxo, rumo ao trabalho contínuo de equipes alinhadas

Desafios técnicos, organizações fracamente acopladas

decomposição baseada em widget, Padrão: Decomposição baseada em widget-

Quando usá-lo

V

variações, execução de várias variações, execução de várias variações

Cofre, segredos

controle de versão, tratamento de alterações entre microsserviços, interface explícita,

Coexistem versões incompatíveis de microsserviços, Infrastructure as Code (IAC)

escala vertical, Limitações de escala vertical, Glossário

VFS para Git, ferramentas

máquinas virtuais (VMs), máquinas virtuais - boas para microsserviços?

Glossário

decomposição baseada em volatilidade, Volatilidade

fase de votação, transações distribuídas - confirmações em duas fases

W
W

Wasm (WebAssembly), Limitações

Web Component Standard, comunicação entre widgets na página

Interface do sistema WebAssembly (WASI), limitações

decomposição baseada em widget, Padrão: Decomposição baseada em widget- Quando para usá-lo

widgets, Glossário

Contêineres Windows, contêineres Windows

fluxo de trabalho

transações de banco de dados, transações de banco de dados - ainda são ACID, mas faltam

Atomicidade?

transações distribuídas, evitando transações distribuídas - basta dizer

Não

transações distribuídas, confirmações em duas fases, transações distribuídas

-Compromissos em duas fases - Transações distribuídas - Confirmações em duas fases

visão geral do fluxo de trabalho

sagas, Sagas-Sagas versus transações distribuídas

gravação por trás de caches, gravação por trás

caches de gravação, gravação

Z

Zed Attack Proxy (ZAP), incorpore a segurança ao processo de entrega

confiança zero, confiança zero

implantação sem tempo de inatividade, implantação sem tempo de inatividade

ZooKeeper, ZooKeeper

Sobre o autor

Sam Newman é consultor independente, autor e palestrante. Em mais

Há 20 anos no setor, ele trabalhou em diferentes pilhas de tecnologia e

em diferentes domínios com empresas em todo o mundo. Seu foco principal está em

ajudando as organizações a colocar o software em produção com mais rapidez e segurança,

e ajudando-os a lidar com as complexidades dos microserviços. Ele também é o

autor de *Monolith to Microservices*, também da O'Reilly.

Colofão

Os animais na capa da Building Microservices, segunda edição, são abelhas (do gênero Apis). Das mais de 20.000 espécies de abelhas, há são apenas oito espécies de abelhas. Essas abelhas que nidificam socialmente são únicas em como eles produzem e armazenam mel coletivamente, bem como constroem colmeias a partir de cera. A apicultura para coletar mel tem sido uma atividade humana em todo o mundo por milhares de anos.

As colmeias de abelhas contêm milhares de indivíduos e têm uma forma muito organizada estrutura social composta por uma rainha, drones e trabalhadores. Cada colmeia tem uma rainha, que permanece fértil por 3-5 anos após seu voo de acasalamento e se deita até 2.000 ovos por dia. Os drones são abelhas machos que acasalam com a rainha (e morrem em flagrante por causa de seus órgãos sexuais farrapos). As abelhas operárias são estéreis mulheres que desempenham muitas funções durante a vida, como enfermeira, operário da construção civil, merceiro, guarda, agente funerário e coletor. Carregado de pólen abelhas operárias que retornam à colmeia "dançam" em padrões estabelecidos para se comunicar informações sobre comida nas proximidades.

Embora as rainhas sejam um pouco maiores, as abelhas melíferas são semelhantes em aparência, com asas transparentes, seis pernas e um corpo segmentado em um cabeca, tórax e abdômen. Eles têm cabelos curtos e felpudos em um amarelo listrado. e padrão preto. A dieta dos adultos é composta exclusivamente de mel, que é criado por um processo de digestão parcial e depois regurgitação rica em açúcar, néctar de flores.

As abelhas são cruciais para a agricultura; ao coletarem seus alimentos, elas polinizam colheitas. As colmeias comerciais são transportadas pelos apicultores para onde as colheitas precisam ser polinizados. Em média, cada colmeia de abelhas reúne 66 libras de pólen por ano. Nos últimos anos, no entanto, o transtorno do colapso das colônias, trouxe devido a uma variedade de doenças e outros fatores de estresse, tem causado um alarmante declínio entre as espécies de abelhas.

As abelhas são vulneráveis aos mesmos pesticidas e parasitas introduzidos e doenças que reduziram o número de abelhas selvagens e outras polinizadoras, mas as abelhas têm algum apoio e proteção humanos.

porque eles são fundamentais na agricultura. Muitos dos animais nas capas de O'Reilly
estão em perigo; todos eles são importantes para o mundo.

A ilustração colorida da capa é de Karen Montgomery, baseada em preto e
gravura branca do Museu Pictórico da Natureza Animada. A capa
as fontes são Gilroy e Guardian Sans. A fonte do texto é Adobe Minion Pro; a
a fonte do título é Adobe Myriad Condensed; e a fonte do código é Dalton
Ubuntu Mono de Maag.