FACULDADE IMPACTA
MBA FULL STACK DEVELOPER
MICROSERVICE & SERVERLESS ARCHITECTURE
TRABALHO FINAL (TEÓRICO)

ABRIL/2022

ODIMAR DA GAMA BARBOSA
BRUNO MANOEL QUINTAS DE ARCANJO

SUMÁRIO

1. Microservices – O Que É?

O estilo de arquitetura microservice é uma abordagem para desenvolver uma única aplicação como um conjunto de pequenos serviços, cada um executando em seu próprio processo e se comunicando através de mecanismos leves, geralmente uma API baseada HTTP. Esses serviços são desenvolvidos em torno de domínios de negócios e podem ser implantados de forma independente por meio de implantação totalmente automatizadas. Há um mínimo de gerenciamento centralizado desses serviços, que podem ser escritos em diferentes linguagens de programação e usar diferentes tecnologias de armazenamento de dados.

As arquiteturas de microserviços facilitam a escalabilidade e agilizam o desenvolvimento de aplicativos, habilitando a inovação e acelerando o tempo de introdução de novos recursos no mercado.

1.2 Pequenos serviços autônomos que trabalham juntos:

- Autônomo
- Heterogeneidade tecnológica
- Resiliência
- Escalabilidade
- Deployment facilitado
- Alinhamento Organizacional
- Composição
- Otimizado para substituição

1.3 Microservices - também conhecido como microservice architecture - é um estilo arquitetural que define uma aplicação como uma coleção de serviços que são:

- Altamente manutenível e testáveis
- Baixo acoplamento
- Implementáveis de maneira independente
- Organizado ao entorno de capacidades de negócio
- Gerenciado por um time pequeno

2. Microservices vs SOA vs Monolítico

2.1 Definições e diferenças

"...Arquitetura Orientada a Serviços (SOA) é um estilo arquitetural que oferece suporte à orientação a serviços. A orientação para serviços é uma forma de pensar em termos de serviços e desenvolvimento baseado em serviços e seus resultados..."

- Representação lógica de atividades de negócios repetitíveis que possuem um resultado específico (e.g, verificar crédito do cliente, prover informações sobre o clima, relatórios de consolidação de perfuração).
- Autocontido;
- Pode ser composto por outros serviços;
- “Caixa Preta (black box)” para consumidores desse serviço.

A arquitetura monolítica é o modelo tradicional unificado, todos os serviços e funções do sistema em um único self-contained bloco, independente de outras aplicações. Essa abordagem de arquitetura tem suas vantagens, especialmente se considerarmos throughput, facilidade de troubleshooting / logging, e testes. Mesmo o deploy é mais simples, já que não requer coordenação com outros serviços e aplicações.

Entretanto, ao mesmo tempo que a arquitetura monolítica mantém esses benefícios, com as novas tendências que vão desde metodologias (waterfall vs. agile) a novas oportunidades (cloud, 5G, edge computing) e a necessidade de rápidas mudanças e adaptações, os monólitos deparam-se com algumas desvantagens: tamanho e complexidade que podem atingir – o que também impacta a agilidade de mudanças e atualizações, o tempo de restart das aplicações, e o esforço de desenvolvimento contínuo.

Como alternativa para tratar essas desvantagens da arquitetura monolítica surge, no outro extremo, a arquitetura de microsserviços. Essa arquitetura é uma abordagem de desenvolvimento em que uma simples aplicação é composta de uma série de pequenos serviços (microsserviços) cada um rodando com seu próprio processo, conectados com outros microsserviços com um uma comunicação leve, geralmente HTTP/APIs.

3. Microservices – Quando Utilizar?

Não existe uma resposta direta. O que temos, são direcionamentos que devem ser utilizados durante uma discussão acerca desse tema. Esse é um tema complexo. Permita que outras pessoas com experiências diferentes da sua participem dessa decisão.

Em seu livro Architeting for Scale, Lee Atchison propõe quatro aspectos que podem ser a fronteira para decidir utilizar ou não microserviços. São eles:

3.1 A aplicação possui um requisito específico de negócio

Se a sua aplicação armazena ou processa dados críticos de negócio como transações bancárias e ações judiciais, ela é uma forte candidata a ser um microserviço. Por muitas vezes, essas aplicações possuem particularidades intrínsecas ao seu processo.

Uma aplicação que processa pagamentos por exemplo, deve armazenar de forma segura os dados de cartões de créditos dos clientes. Com acesso restrito. E, possivelmente precisará escalar de forma distinta das demais aplicações em determinado momento do dia, semana, mês ou ano.

Já uma aplicação que confere transações judiciais, certamente deve atender a requisitos técnicos legais para que possa funcionar. Requisitos que não necessariamente devem se aplicar as interfaces que utilizam essa aplicação.

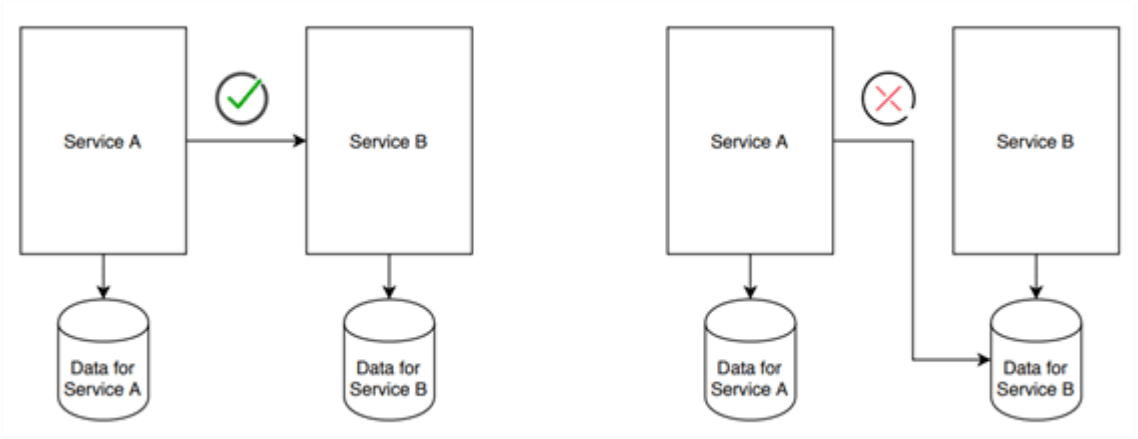
3.2 Times distintos, separados e donos do serviço

Aliado a escalabilidade, essa talvez seja a justificativa mais requisitada para defender essa nova abordagem. Geralmente as aplicações nascem por um propósito e são simples na sua concepção. Porém, a necessidade de evolução tende a torna-las monolíticas. Principalmente quando são colocados vários times para trabalharem no mesmo código.

Voltando um pouco ao conceito discutido no início. A “grande chance” de se ter uma arquitetura baseada em microserviços só se torna concreta, se tivemos apenas um time responsável pelo microserviço.

3.3 Base de dados naturalmente separada

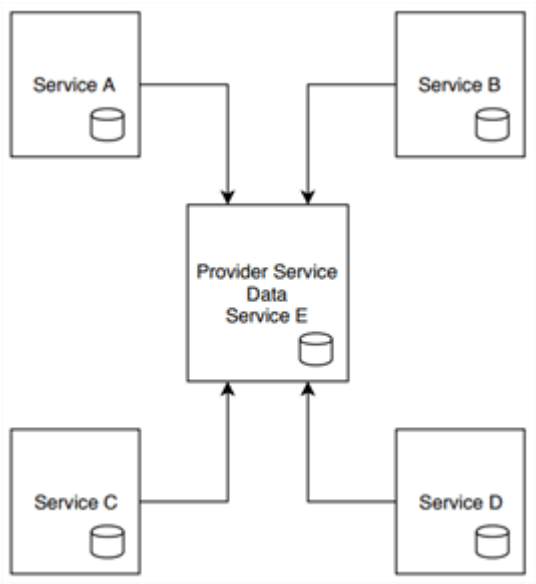
Essa talvez seja a característica mais clara e direta de um microserviço. A figura a seguir nos ajuda a ilustrar essa diretriz.



Em resumo, se sua aplicação possui sua própria base de dados acessível apenas via APIs, ela é uma forte candidata a ser um microserviço. Porém, se a base da sua aplicação pode ser acessada diretamente por outros serviços, ela não é uma boa candidata.

3.4 Necessidade de compartilhamento de dados

Se você precisa compartilhar informações a cerca de um domínio específico entre diversos serviços, na verdade, talvez exista uma necessidade de um microserviço. Um serviço simples. Sem lógica de negócio complexa que simplesmente compartilhe dados gerais de fornecedores seria um bom exemplo.



É importante ressaltar que essa é uma diretriz e não uma regra. Nesse exemplo, a centralização pode ser útil para manter as informações em apenas um lugar. Porém, podem existir dados de fornecedores que são específicos para o Serviço A ou para o Serviço C. Nesses casos, talvez fosse melhor hospedar essas regras nos próprios serviços A e C do que no serviço E, que só seria consumido em algumas ocasiões.

Como Daniel Stori nos conta, essa nova abordagem é atraente. Principalmente quando vemos cases como a Spotify e a Netflix.

Porém, não deve ser adotada as cegas. Cada vez que dividimos um serviço ou um domínio, diminuímos a complexidade daquela aplicação e aumentamos a complexidade do todo.

Isso nos traz alguns problemas—ou desafios—como por exemplo:

- Fica cada vez mais difícil visualizar a arquitetura e comunicações entre as aplicações.
- Se temos mais serviços comunicando entre si, temos mais requisições e consequentemente mais probabilidade de falhas.
- Dependendo da frequência de utilização e quantidade de clientes, alterações bruscas nos microserviços como troca de base e ou tecnologia se tornam mais complexas pela possibilidade de afetarem serviços críticos.
- Com base proprietária, os microserviços tentem a aumentar sua dependência com outras APIs. Mais dependências, maior a probabilidade de falhas.

Diante disso, talvez o segredo esteja em entender a situação atual do seu ambiente organizacional e qual alvo o seu time deve buscar aliado a um objetivo maior traçado pela organização.

4. Microservices – Quando Não Utilizar?

Em resumo, se sua aplicação possui sua própria base de dados acessível apenas via APIs, ela é uma forte candidata a ser um microserviço. Porém, se a base da sua aplicação pode ser acessada diretamente por outros serviços, ela não é uma boa candidata.

4.1 Muitas empresas já utilizam com sucesso este estilo arquitetural, por que eu devo evitar começar a desenhar minha arquitetura neste estilo?

Conforme a tecnologia avança ela se torna cada vez mais complexa e requer mais conhecimento técnico de quem vai implementar. No cenário de grande escassez técnica (principalmente no Brasil) percebo que muitos desenvolvedores não estão aptos tecnicamente para entregar uma solução baseada na arquitetura de microservices.

Isso sempre foi um problema, porém a arquitetura de microservices exige muito mais conhecimento do que projetar uma aplicação monolítica tradicional. Sendo assim é altamente recomendável que o arquiteto da futura aplicação em microservices esteja ciente de todos os pré-requisitos e habilidades antes de começar o projeto.

4.2 Como implementar uma arquitetura de microservices de forma responsável e dentro dos pré-requisitos técnicos esperados?

Não existe uma receita para isto. Tudo depende dos skills técnicos da equipe e das plataformas / tecnologias utilizadas na empresa. O que podemos fazer é observar e até mesmo gerar uma lista com pontos cruciais que realmente irão fazer toda a diferença.

Esta lista dos 10 motivos não foi obtida de algum livro / artigo / material. Eu a criei baseada em minhas “cicatrizes” de anos de consultoria, portanto trata-se de minha visão técnica e de consultor, portanto recomendo que assista meu vídeo para compreender melhor sobre estes pré-requisitos/motivos e o por que eles são tão fundamentais.

5. Microservices – Principais Benefícios

5.1 Quais são os benefícios da arquitetura de microservices?

- Lançamento no mercado com mais rapidez. Como os ciclos de desenvolvimento são reduzidos, a arquitetura de microsserviços é compatível com implantações e atualizações mais ágeis;
- Altamente escalável;
- Resiliência;
- Acessível;
- Mais open source;
- Autônomo;
- Heterogeneidade tecnológica;
- Escalabilidade;
- Deployment facilitado;
- Alinhamento Organizacional;
- Composição;
- Otimizado para substituição.

6. Microservices – Desafios do Estilo Arquitetural

As grandes vantagens desta arquitetura estão no fato de os microsserviços serem independentes entre si, o que facilita a implementação, manutenção, desenvolvimento contínuo e escalabilidade. Neste último item, se um microserviço precisa de mais recursos do que o outro, ele pode ser escalado individualmente. Cada microserviço é gerenciado por um pequeno time e não há a necessidade de todos os microsserviços serem escritos na mesma linguagem, nem usar a mesma tecnologia para armazenar os dados – algo que facilita e acelera desenvolvimento e atualizações.

Entretanto, apesar de ser alternativa para as desvantagens da arquitetura monolítica ou distribuída, a arquitetura de microsserviços não é a solução para todos os desafios de desenvolvimento de software e nem deve ser considerada a bala de prata para tal.

O primeiro desafio que deve ser considerado ao se pensar em microsserviços é o quão "micro" eles devem ser – e nesta questão, podemos incluir: quantos e qual tamanho deles?

6.1 Os fatores que devem ser considerados ao responder esta questão são:

- Qual a capacidade da equipe de administrar os microsserviços? À medida que o número de microsserviços aumenta, diminui a capacidade de administrá-los sem alterar a quantidade de times envolvidos. Aumentando o time, o custo também aumenta.
- Qual é o melhor número de microsserviços para garantir a melhor performance, antes que a latência de rede e problemas de comunicação entre eles comece a causar mais impactos que benefícios?

O segundo desafio que precisa ser considerado é como implementar um software baseado em microsserviços. Se considerarmos que um dos problemas que eles vêm tratar é a escalabilidade, no início do desenvolvimento e implementação, um monólito ainda é facilmente escalável. À medida que novas funcionalidades são adicionadas ao software, esse monólito cresce até que chega a um ponto que a escalabilidade não é mais tão simples. Esta é a hora de se "quebrar" o monólito em partes menores, os microsserviços.

Finalmente, o terceiro desafio: quando usar microsserviços. Nem sempre o uso de monólitos é ruim, e nem sempre os microsserviços resolvem todos os desafios de uma implementação. Além de escalabilidade, outro ponto que a arquitetura de microsserviços tenta tratar, é a complexidade. É necessário avaliar para cada situação, em que momento um monólito deixa de ser a solução menos complexa e os microsserviços tornam-se a melhor arquitetura.

Dados os contextos de cada ambiente, cada negócio e cada necessidade, uma solução diferente ou mesmo híbrida pode ser a melhor para aquela situação e momento.

Existe uma tendência de se explorar a arquitetura de microsserviços devido sua facilidade de adaptação ao ambiente em cloud, e à agilidade de implementação, manutenção e escalabilidade.

Mas não existe a resposta certa. Apenas a análise detalhada e criteriosa de cada situação, foi e sempre será, a melhor resposta para a definição de qual arquitetura adotar.

6.2 Microservices: 10 novos desafios em relação à segurança

Para as equipes de DevOps que atuam com arquitetura de microservices, os desafios sempre se reinventam, e elas precisam estar preparadas. Para garantir que a arquitetura de microsserviços funcione bem para o negócio em relação às versões de ameaças já comuns, e até mesmo as em evolução de nova geração, perspectivas precisam ser renovadas e, ao mesmo tempo, intuitivas.

Ao lidar com microsserviços, estamos falando de códigos. Mais linhas de códigos significam maior vulnerabilidade. Essa abordagem envolve muito mais complexidade quando se trata de segurança, e é aí que entra o DevSecOps.

O papel do DevSecOps é proporcionar a segurança da informação integrada em todos os níveis da aplicação, automatizando o trabalho de segurança para que o fluxo não fique lento, selecionando as ferramentas certas e construindo uma nova vertente sobre a cultura DevOps dentro da empresa.

A abordagem de microsserviços poderia ser descrita simplesmente como um estilo arquitetônico, mas a complexidade é a sua marca distintiva. Vamos apresentar dez desafios que os microsserviços podem apresentar para a segurança da informação:

1. Quanto mais complexa a aplicação, maior a expansão da superfície de ataque

Microsserviços se comunicam entre si via APIs independente da linguagem de programação, e, quanto maior a comunicação e a quantidade de aplicações interativas, mais chances de ataques, por isso, é importante manter a equipe de DevSecOps sempre um passo à frente para avaliar se algum microserviço apresenta falhas em relação à segurança ou não.

Quando um microserviço apresenta um problema e necessita de tempo para ser operado novamente, não é fácil ver como estão contribuindo para ameaças à segurança e ao sistema como um todo, por isso, o time de DevSecOps precisa estar alinhado e pronto para solucionar os problemas.

2. Modificação de aplicações monolíticas para microsserviços

A transição de aplicações monolíticas para ambientes que trabalham com a arquitetura de microsserviços está se tornando cada vez mais comum nos ambientes organizacionais. A necessidade de ciclos de vida variáveis para APIs está no cerne dessas mudanças em muitas empresas.
Como se torna muito mais difícil manter uma configuração de microsserviços do que uma monolítica, cada uma delas pode evoluir para uma ampla variedade de estruturas e linguagens de codificação. As complexidades influenciarão as decisões estratégicas sempre que sistemas monolíticos

forem modificados. Cada linguagem adicional usada para criar novos microsserviços gera um impacto na segurança, garantindo estabilidade com a configuração existente.

3. O *log* tradicional torna-se ineficaz

Como os microsserviços são independentes, haverá mais *logs*, e o desafio desta questão é que esses *logs* podem camuflar problemas à medida que surgem.
Com os microsserviços funcionando em inúmeros *hosts*, é necessário enviar *logs* por todos eles para uma localização única, externa e centralizada. Para que a segurança dos microsserviços seja eficaz, o *log* do usuário precisa correlacionar eventos em várias plataformas diferentes, o que exige um ponto de vista mais alto para observação, independentemente de qualquer API ou aplicação.

4. Novas complicações de monitoramento

O monitoramento apresenta um novo problema com a arquitetura de microsserviços. À medida que novos serviços são adicionados ao sistema, manter e configurar o monitoramento para todos eles se torna novo desafio.
A automação será necessária apenas para oferecer suporte ao monitoramento de todas essas alterações em escala dos serviços afetados, e o balanceamento de carga faz parte da conscientização de segurança de que este monitoramento deve ser responsável.

5. A segurança do aplicativo ainda é um fator importante

Os aplicativos são, de uma maneira ou de outra, o básico das equipes de microsserviços. A segurança da API simplesmente não desaparece com a segurança dos microsserviços. Com cada novo serviço adicional, surge o mesmo antigo desafio de manutenção, configuração e monitoramento da API. Se o monitoramento de aplicativos não for de ponta a ponta, torna-se muito cansativo isolar ou resolver problemas. Sem a automação, é menos provável que as equipes possam monitorar mudanças e ameaças em escala em todos os serviços expostos.

6. Mais pedidos

O método de usar cabeçalhos de solicitação para permitir que serviços comuniquem dados é comum. Isso pode simplificar o número de solicitações feitas, porém, quando um grande número de serviços estiver utilizando esse método, a coordenação da equipe precisará aumentar e se tornar eficiente, e ela mesma também deverá ser simplificada.
Além disso, com um número maior de solicitações, os desenvolvedores deverão ser capazes de compreender o tempo necessário para processá-las.

É provável que a serialização e deserialização de solicitações de serviços se desenvolvam e tornem-se difíceis de trabalhar, sem ferramentas e métodos adequados, apenas para acompanhar as solicitações e vinculá-las a um aparato de segurança autônomo, capaz de funcionar em escala.

7. Tolerância a falhas

A tolerância a falhas no modelo da arquitetura de microsserviços se torna mais complexa do que em um sistema monolítico legado. A equipe de DevOps deve ser capaz de lidar com essas falhas e com outros problemas que ocorram por motivos desconhecidos nas aplicações.
Quando elas se acumulam, isso pode afetar os demais serviços, criando outras falhas no *cluster*. Os microsserviços exigem um novo foco na interdependência e um novo modelo de segurança para garantir a estabilidade entre as aplicações.

8. Armazenamento em Cache

Embora o armazenamento em cache ajude a reduzir a grande frequência de solicitações feitas, isso é uma "faca de dois gumes". Com o recurso aprimorado que o cache traz para o ambiente de DevOps, essas solicitações aumentam inevitavelmente para lidar com um número crescente de serviços.
O excesso de armazenamento em cache de reserva pode aumentar a complexidade e a enorme necessidade de comunicação das equipes. Automatizar, ordenar e otimizar essa comunicação se torna um novo requisito que talvez não existisse antes, com sistemas monolíticos.

9. Esforços de segurança colaborativos com o DevOps

À medida que a responsabilidade pela integridade dos microsserviços se espalha entre as equipes, o DevSecOps entra em ação. Pensar com um "cérebro de segurança" se torna um empreendimento coletivo e não hierárquico.
A colaboração e os pontos de contato regulares em todos os níveis tornam-se uma necessidade e devem ser trabalhados na cultura de dados/desenvolvimento. Um bom design de segurança é uma questão de colaboração, necessidades e desejos conflitantes em relação à linha de base de ameaças conhecidas e emergentes ao microserviço e às estruturas virtualizadas que eles combinam para criar.

10. Projeto de segurança prospectivo

Por todos os motivos vistos acima, chegamos à prioridade do design de segurança prospectivo. Longe de indicar uma abordagem única de uma equipe de autoridade de segurança em silos, chegamos a um ponto na cultura DevOps em que todos os fatores possíveis contribuem para considerações

que formam a base de uma política de segurança estável e um processo de formação de protocolo. Sem saber como todos serão impactados pelas mudanças nas preocupações da equipe no dia a dia e pelas práticas de segurança, é quase impossível criar um bom design de segurança. O *scrum* ágil pode se tornar a substituição do silo de autoridade no modelo de microsserviços, mas como as equipes podem dedicar mais tempo a eles em questões de segurança, sem prejudicar as suas tarefas diárias? Devido às muitas maneiras pelas quais a complexidade está mudando a cena do DevOps e fazendo com que os sistemas repensem o seu *modus operandi*, a autonomia precisará substituir o fator humano, não apenas pela integridade do serviço, mas por todo o campo de preocupações com segurança.

6.3 Abordagens ao gerenciamento de dados

Não há nenhuma abordagem única que seja correta em todos os casos, mas aqui estão algumas diretrizes gerais para gerenciar os dados em uma arquitetura de microsserviços.

- Adote consistência eventual quando possível. Entenda os lugares no sistema onde uma consistência forte ou as transações ACID são necessárias e os locais onde a consistência eventual é aceitável.
- Quando você precisa de garantias de consistência forte, um serviço pode representar a fonte de verdade para determinada entidade, exposta por meio de uma API. Outros serviços podem conter sua própria cópia ou um subconjunto de dados, que eventualmente são consistentes com os dados mestres, mas não são considerados como fonte da verdade. Por exemplo, imagine um sistema de comércio eletrônico com um serviço de pedido de cliente e um serviço de recomendação. O serviço de recomendação pode escutar eventos do serviço de pedidos, mas se um cliente solicitar reembolso, será o serviço de pedidos, e não o serviço de recomendação, que terá todo o histórico da transação.
- Para transações, use padrões como Supervisor de Agente do Agendador e transações de compensação para manter os dados consistentes em vários serviços. Talvez seja necessário armazenar dados adicionais que capturem o estado de uma unidade de trabalho que abrange vários serviços, para evitar uma falha parcial entre os serviços. Por exemplo, manter um item de trabalho em uma fila durável, enquanto uma transação de várias etapas está em andamento.
- Armazene apenas os dados que um serviço precisa. Um serviço pode precisar apenas de um subconjunto de informações sobre uma entidade de domínio. Por exemplo, no contexto de envio limitado, precisamos saber qual cliente está associado a uma entrega específica. Mas não precisamos do endereço de cobrança do cliente, que é gerenciado pelo contexto limitado de Contas. Pensar cuidadosamente o domínio e usar uma abordagem DDD, pode ajudar.
- Considere se os serviços são coerentes e acoplados de forma flexível. Se dois serviços trocam informações continuamente entre si, resultando em APIs de conversa, será necessário redigir os limites do serviço, mesclando dois serviços ou refatorando suas funcionalidades.

- Use um estilo de arquitetura baseado em eventos. Nesse estilo de arquitetura, um serviço publica um evento quando há alterações em suas entidades ou modelos públicos. Serviços interessados podem assinar esses eventos. Por exemplo, outro serviço pode usar os eventos para construir uma visão materializada dos dados que seja mais adequada a consultas.
- Um serviço que tem eventos deve publicar um esquema que pode ser usado para automatizar a serialização e desserialização de eventos, para evitar acoplamento entre publicadores e assinantes. Considere o esquema JSON ou uma estrutura como Microsoft Bond, Protobuf ou Avro.
- Em grande escala, os eventos podem se tornar um gargalo no sistema, portanto, considere usar agregação ou processamento em lote para reduzir a carga total.

6.4 Resiliência e a alta disponibilidade em microsserviços

Lidar com falhas inesperadas é um dos problemas mais difíceis de se resolver, especialmente em um sistema distribuído. Grande parte do código que os desenvolvedores gravam envolve tratamento de exceções, e também é nisso que a maior parte do tempo é gasta no teste. O problema é mais complicado do que escrever código para tratar de falhas. O que acontece quando o computador onde o microserviço está em execução falha? Você não precisará apenas detectar essa falha do microserviço (um problema difícil por si só), mas também precisará de algo para reiniciar o microserviço.

Um microserviço precisa ser resiliente a falhas e conseguir reiniciar geralmente em outro computador para disponibilidade. Essa resiliência também se resume ao estado que foi salvo em nome de microserviço, do qual os microserviço podem recuperar esse estado, e a se o microserviço pode reiniciar com êxito. Em outras palavras, deve haver resiliência na capacidade de computação (o processo pode reiniciar a qualquer momento), bem como resiliência no estado ou nos dados (sem perda de dados e os dados permanecem consistentes).

Os problemas de resiliência são abordados durante outros cenários, como quando ocorrem falhas durante uma atualização de aplicativo. O microserviço, trabalhando com o sistema de implantação, precisa determinar se pode continuar a avançar para a versão mais recente ou, em vez disso, reverter para uma versão anterior para manter um estado consistente. É preciso considerar perguntas como se há computadores suficientes disponíveis para continuar avançando e como recuperar versões anteriores do microserviço. Essa abordagem requer o microserviço para emitir informações de integridade para que o aplicativo e o orquestrador em geral possam tomar essas decisões.

Além disso, a resiliência está relacionada a como sistemas baseados em nuvem devem se comportar. Conforme mencionado, um sistema baseado em nuvem deve compreender falhas e tentar se recuperar automaticamente delas. Por exemplo, no caso de falhas de rede ou um contêiner, aplicativos cliente ou serviços cliente devem ter uma estratégia de repetição de envio de mensagens ou novas tentativas de solicitações, já que, em muitos casos, as falhas na nuvem são parciais. A seção Implementando aplicativos resilientes neste guia aborda como lidar com falhas parciais. Ele descreve técnicas como repetições com retirada exponencial ou o padrão de Disjuntor no .NET usando bibliotecas

como a Polly, que oferece uma grande variedade de políticas para lidar com esse assunto.

6.5 Gerenciamento de integridade e diagnóstico em microsserviços

Pode parecer óbvio, e isso muitas vezes é negligenciado, mas um microserviço deve informar sua integridade e seu diagnóstico. Caso contrário, há pouca percepção de uma perspectiva de operações. Correlacionar os eventos de diagnóstico em um conjunto de serviços independentes e lidar com a defasagem do relógio do computador para entender a ordem dos eventos é um desafio. Da mesma maneira que você interage com um microserviço sobre protocolos e formatos de dados estabelecidos, existe a necessidade de padronização de como registrar em log a integridade e eventos de diagnóstico que, por fim, terminam em um repositório de eventos para consulta e exibição. Em uma abordagem de micros serviços, ele tem chave que diferentes equipes concordem com um único formato de log. Deve haver uma abordagem consistente para exibir eventos de diagnóstico no aplicativo.

6.6 Verificações de integridade

Integridade é diferente de diagnóstico. Integridade significa que o microserviço reporta seu estado atual para tomar as devidas ações. Um bom exemplo é trabalhar com mecanismos de atualização e implantação para manter a disponibilidade. Embora um serviço possa não estar íntegro no momento devido a uma falha de processo ou reinicialização do computador, o serviço ainda pode estar operacional. A última coisa de que você precisa é piorar a situação executando uma atualização. A melhor abordagem é fazer uma investigação primeiro ou aguardar a recuperação do microserviço. Os eventos de integridade de um microserviço permitem tomar decisões bem-informadas e ajudam realmente a criar serviços que recuperam a si próprios.

Na seção Implementação de verificações de integridade nos serviços ASP.NET Core deste guia, explicamos como usar uma nova biblioteca ASP.NET HealthChecks em seus microsserviços para que eles possam relatar o próprio estado a um serviço de monitoramento para executar as ações apropriadas.

Você também tem a opção de usar uma excelente biblioteca de software livre chamada AspNetCore.Diagnostics.HealthChecks, disponível em GitHub e como um pacote NuGet. Essa biblioteca também faz verificações de integridade, mas com uma diferença, pois ela lida com dois tipos de verificação:

- **Liveness**: verifica se o microserviço está vivo, ou seja, se ele é capaz de aceitar solicitações e responder.
- **Preparação**: verifica se as dependências do microserviço (banco de dados, serviços de fila etc.) estão prontas, para que o microserviço possa fazer o que deveria fazer.

6.7 Usando fluxos de eventos de logs e diagnóstico

Logs fornecem informações sobre como um aplicativo ou serviço está sendo executado, incluindo exceções, avisos e mensagens informativas simples. Normalmente, cada log está em um formato de texto com uma linha por evento, embora exceções também muitas vezes mostrem o rastreamento de pilha em várias linhas.

Em aplicativos monolíticos baseados em servidor, você pode gravar logs em um arquivo em disco (um logfile) e analisá-lo com qualquer ferramenta. Uma vez que a execução do aplicativo está limitada a um servidor ou VM fixo, geralmente não é complexo demais analisar o fluxo de eventos. No entanto, em um aplicativo distribuído em que vários serviços são executados em vários nós em um cluster do orquestrador, poder correlacionar eventos distribuídos é um desafio.

Um aplicativo baseado em microserviço não deve tentar armazenar o fluxo de saída de eventos nem arquivos de log por si só, nem tentar gerenciar o roteamento de eventos para um local central. Ele deve ser transparente, o que significa que cada processo deve gravar apenas seu fluxo de eventos em uma saída padrão que, por baixo, será coletado pela infraestrutura do ambiente de execução em que está sendo executado. Um exemplo desses roteadores do fluxo de evento é Microsoft.Diagnostic.EventFlow, que coleta os fluxos de eventos de várias fontes e publica-os em sistemas de saída. Eles podem incluir uma saída simples padrão para um ambiente de desenvolvimento ou sistemas de nuvem como o Azure Monitor e o Diagnóstico do Azure. Também há boas plataformas e ferramentas de análise de log de terceiros que podem pesquisar, alertar, relatar e monitorar logs, inclusive em tempo real, como Splunk.

6.8 Orquestradores gerenciando informações de integridade e diagnóstico

Quando você cria um aplicativo baseado em microserviço, precisa lidar com a complexidade. Logicamente, é simples lidar com um único microserviço, mas dezenas ou centenas de tipos e milhares de instâncias de microsserviços são um problema complexo. Não envolve apenas criar sua arquitetura de microserviço: você também precisará de alta disponibilidade, capacidade de endereçamento, resiliência, integridade e diagnóstico se você quiser ter um sistema estável e coeso.

7. Referências

7.1 Internet e material da disciplina.