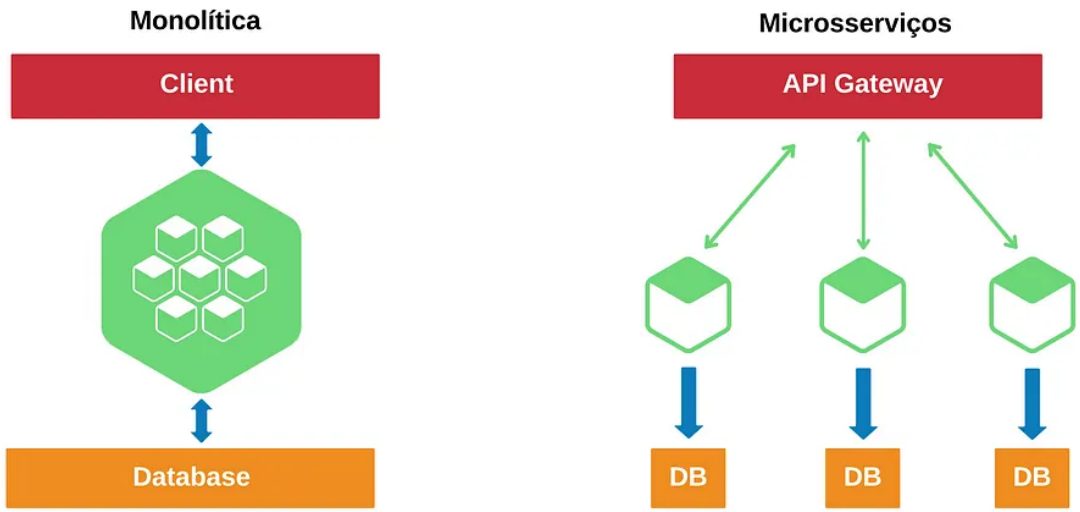
Marcelo M. Gonçalves Follow
Jun 1, 2020 · 9 min read

Arquitetura de Microsserviços



Arquitetura de Microsserviços

Refere-se a um estilo de arquitetura para construção de software a qual decompõe o domínio de negócio em pequenos blocos, transacionalmente consistentes e com contexto próprio. Na prática, a implementação dos microsserviços pode adotar uma natureza de comunicação assíncrona, através de um message broker realizando o desacoplamento dos componentes, bem como uma natureza síncrona, como a exposição de APIs utilizando **HTTP/REST**.

A implementação dos microsserviços consiste em um conjunto de padrões de design conceituais, divididos em categorias como aplicação, redes, infraestrutura, database, etc. sendo cada unidade executando dentro de seu próprio processo, descrevendo características de serviços auto contidos, autônomos e independentes. A colaboração entre os componentes, descritos arquiteturalmente como microsserviços, atua em uma granularidade fina em relação as unidades,

promovendo o baixo acoplamento e proporcionando maior agilidade do ponto de vista de mudanças.

Transações e Divisão de Responsabilidades

A arquitetura de microsserviços propõem descentralizar a responsabilidade dos dados, implicando no gerenciamento de atualizações individualmente, sem transações distribuídas atomicamente (ACID transactions). Cada escopo de transação é tratado pelo microsserviço responsável dentro de seu contexto e limites transacionais adequados, devendo trabalhar com etapas de compensação das operações em mente do início ao fim do processo.

> *Abordar práticas envolvendo componentes distribuídos agrega dificuldade de implementação e rastreabilidade após **deploy**, trazendo dificuldades inerentes a natureza distribuída sob o ponto de vista do controle e gerenciamento destas unidades.*

Microsserviços possuem diversos benefícios, porém estas vantagens, seja em etapa de desenvolvimento ou deploy, vem com um preço. A complexidade operacional relacionada aos sistemas distribuídos, compreensão do sistema como um todo (natureza distribuída), dificuldade para depurar problemas (troubleshooting), logs e tracing distribuídos, entre outras. Obrigando-nos a deixar para trás a noção de conceitos como transação atômica, largamente utilizadas em aplicações monolíticas, as quais bloqueariam toda a comunicação e os recursos de ponta a ponta, ao transcender a cadeia inteira de processamento, e evidentemente tornando-se inviável em sistemas distribuídos.

Ao trabalharmos com microsserviços, a divisão das equipes pode conter menos integrantes (owned by a small team) possibilitando maior agilidade. Estas equipes poderiam trabalhar em bounded contexts distintos, responsabilizando-se pela entrega de microsserviços relacionados a um determinado contexto. Podemos nos tornar produtivos mais rapidamente além de possibilitar o desacoplamento entre as equipes e suas entregas, realizando a adoção do formato de deploys mais consecutivos.

Desta forma, observamos diversas vantagens em adotarmos componentes de software sendo entregues utilizando uma arquitetura distribuída de microsserviços. Arquiteturas distribuídas podem ser observadas como uma maneira de mover-se rapidamente, permitindo entregarmos valor mais facilmente ao negócio.

A independência contribui para agilidade, trazendo maior liberdade para reagir rapidamente a mudanças e tomar decisões. Tecnologias heterogêneas podem ser utilizadas e experimentadas com maior facilidade, habilitando o continuous delivery, on-demand virtualization, infrastructure automation, deployment, manutenção, confiabilidade e escalabilidade horizontal, além de podermos rodar múltiplas instâncias de seu serviço em múltiplas máquinas na nuvem.

Antecessores dos Microsserviços

A ideia dos microsserviços não se trata de algo novo, tendo origem nos sistemas distribuídos da década de 70s. Trabalhamos com os conceitos principais por trás da arquitetura de microsserviços, como arquitetura orientada a serviços (**SOA — service-oriented architecture**), o modelo de atores (actor model) ou mesmo ideias do modelo original da orientação a objetos a mais de 50 anos.

Todos estes modelos predecessores, principalmente o SOA, já possuía os aspectos principais relacionados arquitetura de microsserviços que vemos hoje em dia, trazendo os bons aspectos dos microsserviços já estavam previstos no SOA. Adicionalmente, do ponto de vista conceitual envolvendo o SOA (predecessor direto dos microsserviços), não há nada de errado em criarmos serviços e hospedá-los no mesmo servidor, em um mesmo processo, ou mesmo acrescentar diversas funcionalidades coerentes dentro de um mesmo componente e realizar o deploy do bloco inteiro.

Pelo contrário, nesta abordagem de arquitetura uma camada conhecida como service mesh, envolvendo serviços de redes e infraestrutura, é resolvida pelo próprio servidor de aplicação (weblogic, websphere, jboss, glassfish). Um dos grandes desafios que enfrentaremos ao migrarmos para uma arquitetura de microsserviços, seria ter de lidar com a camada de **service mesh** individualmente com cada instância de nossos microsserviços. De qualquer forma, independente da camada de service mesh, adotar um design de aplicações utilizando uma arquitetura diferente (**microservices**) nos obriga a pensar diferente o processo de **dev, build e deploy**.

Camada de Service Mesh

A camada de service mesh refere-se a uma abstração para a comunicação a nível de redes entre os microsserviços, oferecendo recursos de infraestrutura dedicados a

resolver problemas comuns e conhecidos quando tratamos da comunicação entre diversos componentes distribuídos compondo uma solução.

> *Em uma camada de **service mesh**, a maior parte destas abstrações se referem a serviços (**shared libraries**) como balanceamento de carga, circuit breaker, bulkhead, novas tentativas (**retry**), tempo limite (**timeout**) e roteamento inteligente.*

A arquitetura de microsserviços sugere o isolamento dos componentes em processos próprios, rodando em containers visando maior mobilidade e possuindo seu próprio ciclo de vida. Em contra partida, devemos assumir que a responsabilidade de resolver estas questões será inteiramente de quem desenvolve o código, pois não teremos mais o apoio de implementações que antes estavam disponíveis no servidor de aplicação.

> *Algumas libraries famosas podem nos ajudar com a implementação destas responsabilidades, **netflix ribbon, hystrix**. Vale lembrar ainda que outra característica importante, relacionada ao service mesh, trata-se da telemetria (observabilidade e tracing distribuído), resolvidas facilmente por libraries como jaeger, zipkin e prometheus.*

Shared libraries abstraem características essencialmente complexas, nos poupando grande esforço, porém a grande desvantagem do uso das shared-client libraries, conforme mencionado, seria que estes recursos estariam atrelados a tecnologias específicas. Desta forma, obrigando-nos buscar individualmente implementações confiáveis em cada linguagem que escolhermos utilizar.

Outra característica no uso das shared libraries seria a necessidade de empacotar uma cópia em cada instância de nossos microsserviços, tornando nosso container em execução um pouco mais inchado. Lembrando que não seria possível fazer deploy de uma shared library isoladamente e sim do microsserviço inteiro reduzindo capacidade de isolar mudanças em nível mais granular, pois seu uso na comunicação entre microsserviços torna-se um ponto de acoplamento entre a library e a linguagem escolhida.

Alternativamente, para lidarmos com os problemas apresentados na adoção das shared libraries, existem plataformas como istio e linkerd, as quais apresentam maneiras agnósticas a tecnologia e infraestrutura para lidar com os problemas apresentados.

Isolamento e Independência

A computação distribuída trata-se de possibilitar maior disponibilidade e alto poder de escalabilidade quando necessário. Entretanto, nem todo o sistema precisa ser distribuído, somente quando necessário e não sendo apropriado para na maior parte dos casos. Uma das necessidades que teríamos seria a de isolamento entre os componentes, pois ao executarmos cada microsserviço em seu próprio processo dentro de um servidor, adquirimos maior agilidade.

Uma das razões para querermos o isolamento (independência) seria para lidar com falhas separadamente, pois quando rodamos todas nossas instâncias dentro de um mesmo processo, em caso de um restart, todas nossas instâncias seriam destruídas e recriadas desnecessariamente. Portanto, a execução de cada microsserviço em seu próprio servidor, possibilitam restarts com menos efeitos colaterais relacionados a disponibilidade dos nossos componentes.

Sistemas distribuídos são complexos e devemos assumir que existirão falhas em sua execução. Desta forma, precisam ser construídos para cenários de incerteza, possuindo seu design pensado para falhas. Neste caso, precisamos desenhar nossos microsserviços para lidar com falhas e não tentar prevê-las. Serviços necessitam ser resilientes, além de não permitir que falhas se propaguem comprometendo toda uma cadeia de comunicação.

Ao modelarmos nossos componentes de microsserviços existe um conceito indispensável, independente da natureza de comunicação adotada, que refere-se a etapas de compensação para os casos de falha. Neste caso, todo o caminho realizado para implementar determinado fluxo, deve ser desfeito em uma ordem inversa, aplicada a cada fase do processo, possibilitando com que o componente possa assumir seu estado inicial em relação a ocorrência do erro. Desta forma, ações de rollback serão efetuadas somente dentro do contexto transacional de cada microsserviço individualmente, devendo cada componente resposabilizar-se em desfazer suas alterações no domínio de dados.

Transações em Sistemas Distribuídos

Ao adotarmos arquiteturas distribuídas, empregando técnicas de coreografia em sua comunicação, será necessário aplicar o conceito BASE (basically available, soft state, eventual consistency) transactions. Atribuindo ao fluxo do processo principal, processos de compensação de erros. Processos de compensação devem ser disparados para reparar falhas, desfazendo as alterações de estado no domínio, retornando-o a um estado consistente novamente.

Processos de compensação no contexto das aplicações em um projeto, podem ser realizados através das sagas, flexibilizando e facilitando a orquestração do life cycle (ciclo de vida) destes processos de natureza compensatória. Cada uma destas camadas de isolamento adicionais possuem custos e complexidade, ainda assim é muito mais vantajoso lidarmos com diversos nós e instâncias em um servidor do que com um servidor único de difícil manuseio.

Vale lembrar que nenhuma destas abordagens estaria certa ou errada, sendo apenas trade-offs. Desta forma, um dos principais benefícios para determinada empresa em adotar uma arquitetura de microsserviços seria não ser obrigado a tomar decisões arquiteturais verticais, envolvendo a cadeia de processos como um todo, e sim, adquirindo o isolamento através de cada pequeno componente, possibilitando redução de impacto sobre decisões sobre o rumo dos mesmos.

Processamentos envolvendo base de dados as quais utilizam transações do tipo ACID (**atomic, consistent, isolated and durable**) são bem conhecidas por parte dos arquitetos e desenvolvedores. **ACID transactions** são adotadas em cenários onde necessitamos de consistência imediata, de forma atômica, com possibilidade de rollback em casos de falha.

Neste caso, para aplicações distribuídas não podemos aplicar transações desta natureza, portando devemos explorar o uso de **BASE transactions**, descrita abaixo em maiores detalhes. O acrônimo, representado pela composição BASE, definido por Eric Brewer, refere-se a transações as quais não podem entregar, estruturalmente, atomicidade de forma semelhante a transações ACID. Ao propor-nos a utilizar BASE transactions, precisamos ter ciência de que estaremos trabalhando com dados ligeiramente obsoletos (stale) pelo tempo de latência de rede.

> *O teorema de **CAP** nos diz que não temos escolha, se desejamos aumentar a escala de disponibilidade de nossos componentes (microsserviços), devemos abrir mão da consistência forte em favor da eventual.*

Foco no Domínio de Dados

Ao desenvolvermos microsserviços, precisamos praticar o design de composição dos componentes com o domínio em mente. Modelos de domínio fazem mais ou menos sentido dependendo do contexto ao qual eles são observados, sendo assim,

praticas de domain-driven design nos ajudam a compreender e construir modelos mais fiéis ao domínio de negócio.

O principal problema atualmente trata-se da ambiguidade e contradições em modelos que residem na cabeça de cada envolvido no projeto, podendo alterar-se com determinada frequência. Quando unificamos a linguagem entre negócio e time de software, sempre que uma mudança ocorre podemos compreender mais rapidamente e claramente como esta mudança afetará nosso software.

Um modelo desacoplado e independente pode ser testado unitariamente com maior facilidade, além acrescentar maior flexibilidade em torno das necessidades de mudanças no negócio. Ao possuir componentes com deploy independente, a escalabilidade horizontal executada sob demanda, envolvendo a computação em nuvem no gerenciamento das instâncias dos microsserviços, são operadas de forma mais inteligente e econômica, evitando assim excessos e desperdícios de recursos (scale up and down).

Componentes menores e mais independentes possuem vantagem do ponto de vista da mobilidade ao rodarem dentro de containers docker, neste caso, tanto o resizing (redimensionamento) como o downsizing (redução de tamanho) na quantidade de instâncias simultâneas são executados on-demand (sob demanda) em intervalos de sazonalidade com maior ou menor exigência de recursos.

Considerações Finais

A arquitetura de microsserviços acrescenta como vantagem a possibilidade de utilizarmos tecnologias heterogêneas (**java, ruby, nodeJS, python**) na composição e construção dos componentes, e ainda assim possibilitando que a comunicação ocorra de forma natural, permitindo facilidade no gerenciamento destas instâncias dos containers.

Componentes distribuídos escalam horizontalmente instâncias e réplicas com muito mais facilidade em ambientes gerenciados pelo kubernetes, por exemplo em conjunto com o docker para execução em containers. Em contrapartida, existe grande diferença entre escalar uma aplicação monolítica horizontalmente, pois este tipo de escalabilidade envolve instâncias de um servidor de aplicação, neste caso envolvendo muito mais recursos necessários tornando as possibilidades de escalabilidade inviável e com alto custo.

Como em qualquer movimento envolvendo decisões arquiteturais, existem vantagens e desvantagens em todas as direções, ocasionando inevitavelmente em trade-offs. Em geral, o estilo de comunicação em uma arquitetura de microsserviços trata-se de algo complexo, podendo envolver diferentes padrões como **REST** e **gRPC** sobre **HTTP** ou **AMQP**, podendo ter natureza síncrona ou assíncrona, como na adoção de padrões arquiteturais voltados à eventos (**EDA — event-driven architecture**).

Como podemos observar, microsserviços são sistemas distribuídos, e assim sendo, carregam consigo toda a complexidade e os cuidados necessários que precisamos nos preocupar ao evoluir nesta direção. De uma forma ou de outra, seja usando **SOA**, **Actor Model** ou **Microsserviços** o conceito básico por trás destes modelos sempre foi a busca em tentarmos quebrar nossos serviços em pedaços cada vez menores, com menos responsabilidades, mais autônomos com o objetivo de promovermos cada vez mais o baixo acoplamento entre eles.

Microservices Microsserviços Microservice Architecture