FullCycle

Full Cycle

Wesley Willians

Esse livro está à venda em http://leanpub.com/fullcycle

Essa versão foi publicada em 2024-03-20

Esse é um livro Leanpub. A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. Publicação Lean é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.

Conteúdo

Introdução

A mudança de perspectiva

Foi em 2015 quando comecei ouvir com frequência o termo microsserviços. A cada palestra e apresentação de cases de grandes empresas, ficava mais evidente de que essa tendência iria perdurar por algum tempo.

Grandes empresas e unicórnios precisavam crescer rapidamente, gerar mais valor a cada dia, contratar mais pessoal e ter mais independência entre seus projetos. Por outro lado, quanto mais ouvia falar sobre microsserviços, também mais era evidente que utilizar tal arquitetura não era trivial. Muitas peças ainda precisavam se encaixar para que esse modelo pudesse se tornar algo mais *natural* nas organizações.

A complexidade de arquitetar, desenvolver, testar, realizar o deploy e monitorar uma única aplicação estava sendo multiplicada pelo número de microsserviços que cada empresa possuia. Apesar de muitos projetos serem pequenos, ainda assim, todos precisavam passar por essas etapas.

Com o número de sistemas crescendo exponencialmente, a área de operações dessas empresas também começou a colapsar. Profissionais que estavam acos-

tumados a receber demandas de devs para realizarem quatro deploys diários, tiveram suas rotinas alteradas para realizar quarenta ou quatrocentos.

O número de aplicações a serem monitoradas também foi se multiplicando, e conflitos entre pessoas desenvolvedoras e sysadmins se intensificaram. Estava muito claro que já estávamos em uma nova era. Uma era que não tinha mais volta.

Full Cycle Developers @Netflix

Em 17 de maio de 2018, alguns profissionais da Netflix que já possuíam anos de casa, também compartilharam suas dores e tentativas que vinham realizando desde 2012.

Naquela época, não muito diferente de outras organizações, eles possuíam papéis extremamente bem definidos para o ciclo de desenvolvimento de um software.

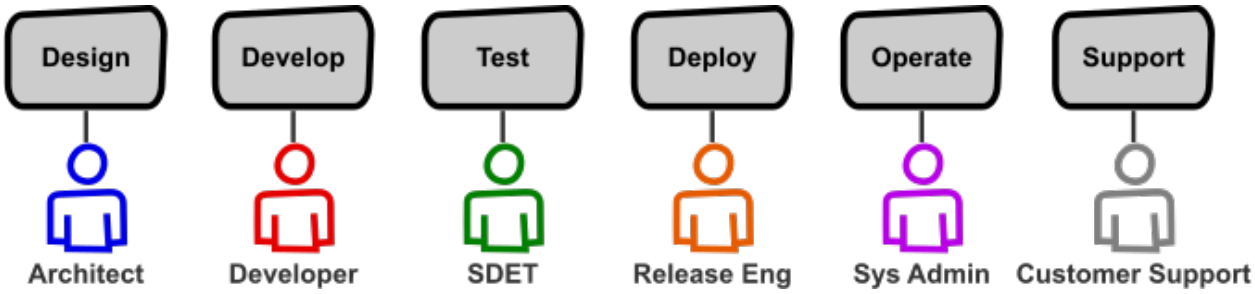


Imagem retirada do Tech Blog da Netflix.

A Netflix deixou de ser apenas um arquivo “war” e também foi dividida em microsserviços.

Depois de muitos erros e acertos, perceberam que um dos principais pontos que sem dúvidas mudaria o jogo, seria transferir totalmente a responsabilidade de cada projeto para seus times de desenvolvimento, ou seja: agora os desenvolvedores fariam parte de todo o ciclo de desenvolvimento de suas aplicações. Da arquitetura ao deploy e monitoramento.

O grande lema se tornou: *"Operate what you build"*, ou opere o que você mesmo constrói. O raciocínio foi remover intermediários de todo processo e fazer com que a equipe de dev fique 100% responsável por seu microsserviço sendo capaz de trabalhar com *feedbacks* curtos de todo processo e aprender rapidamente com isso.

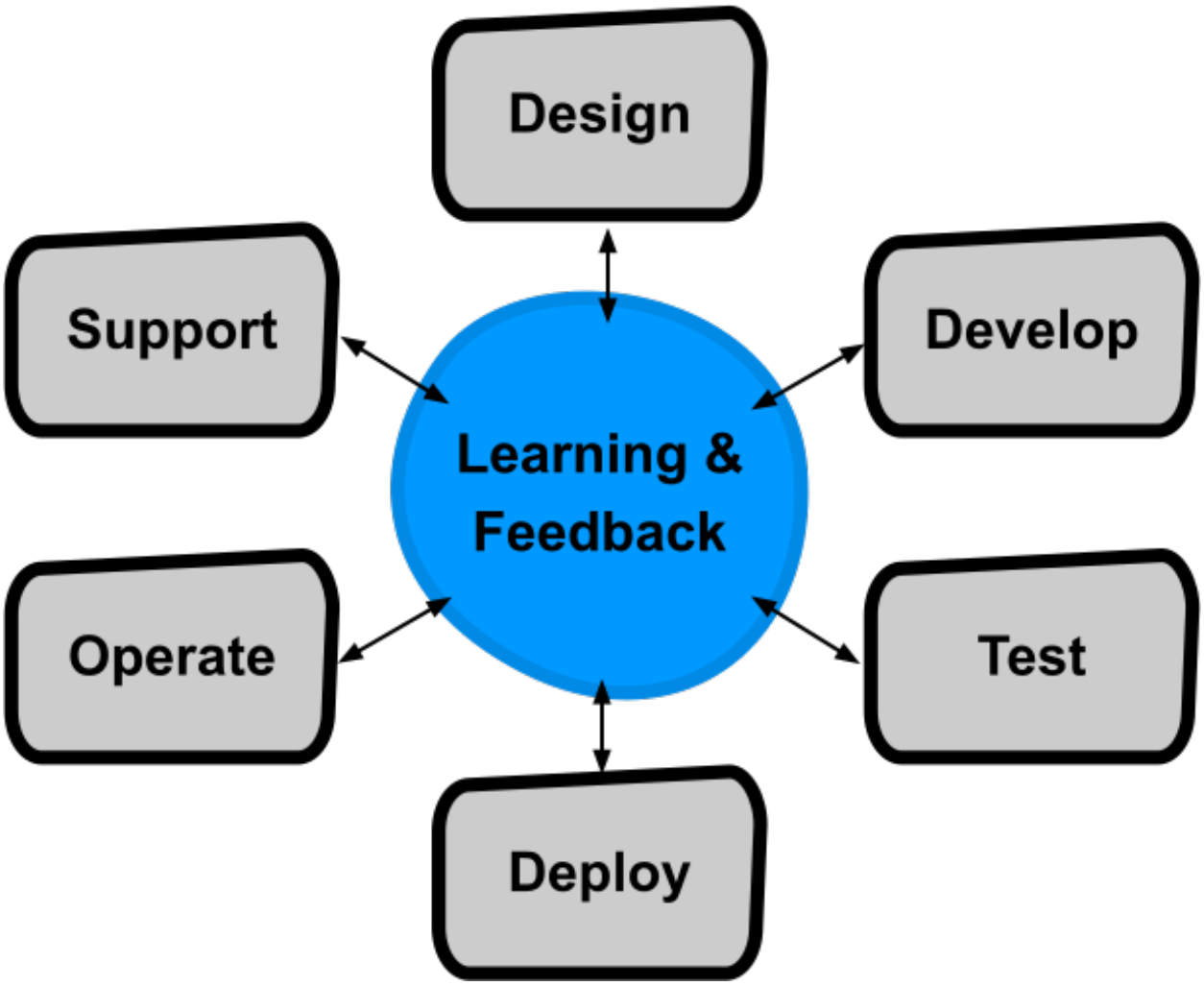


Imagem retirada do Tech Blog da Netflix.

Devs com muitas responsabilidades

Se você é uma pessoa desenvolvedora, com certeza já sabe o número imenso de responsabilidades que possui no dia a dia para entregar software com alta qualidade.

A grande questão é: colocar mais responsabilidades nas “costas” da pessoa

desenvolvedora é realmente a solução?

Foi exatamente essa pergunta que fiz para mim mesmo assim que estava na metade do artigo. Porém, ao continuar com a leitura percebi que os autores deram uma solução para minimizar esse fardo e ao mesmo empoderar devs cansados e estressados com tantos conflitos que resultavam em idas e vindas junto a área de operações para solucionar problemas em produção e para colocar suas aplicações no ar.

A solução criada estava na utilização de ferramentas desenvolvidas especificamente para que a pessoa desenvolvedora tivesse total autonomia para que realizar um deploy e entender em tempo real o comportamento de uma aplicação, de forma simples, rápida e que fundamentalmente fizesse parte do fluxo natural de trabalho.

Times de plataforma

Se você atualmente trabalha em alguma empresa que minimamente possui processos de desenvolvimento bem definidos, bem como possui o mínimo de uma cultura DevOps implementada, acredito que tudo que você leu até o momento representa a sua realidade de trabalho atual.

Grande parte das empresas já possui times de *plataforma*, que têm o objetivo de dar suporte e autonomia para as pessoas desenvolvedoras no dia a dia. Esses times criam ferramentas, padronizam pipelines exatamente para que o processo

de desenvolvimento no dia a dia seja fluido e produtivo, sem tirar o foco na entrega de valor pelos devs.

Você é Full Cycle

Se você participa do fluxo completo de uma aplicação, você é um Full Cycle Developer, todavia, como qualquer profissional de tecnologia, a evolução precisa ser constante; e esse é o objetivo desse livro: abranger os principais aspectos do mundo Full Cycle, e te tornar capaz de desenvolver aplicações de grande porte utilizando as tecnologias mais modernas do mercado onde quer que você esteja.

A partir de agora, vamos partir para uma jornada completa do *Software Development Life Cycle*. Vamos partir do mundo da arquitetura de software até o deploy e monitoramento de aplicações de grande porte.

Introdução à Arquitetura de Software

Neste capítulo explicaremos o que é a arquitetura de software. Os conhecimentos sobre esse tema podem nos auxiliar desde o processo de desenvolvimento de uma solução até sua sustentabilidade.

Um dos pontos fundamentais que, sem dúvidas, diferencia um desenvolvedor, é o quanto ele está preparado para desenvolver softwares sustentáveis, ou seja, aquele software que consegue ao longo do tempo gerar valor para uma organização. O software sustentável é flexível e permite grandes mudanças sem muita necessidade de reescritas.

Assim, gostaria essencialmente de fazer três "provocações" que ajudarão você entender a real essência da Arquitetura de Software:

1. Qual a fórmula para criarmos softwares sustentáveis?
2. O que realmente é Arquitetura de Software?
3. Qual a relação entre a sustentabilidade de um software e a sua arquitetura?

Sustentabilidade no dia zero

Geralmente o termo sustentabilidade está relacionado à realização de atividades que busquem a preservação do meio ambiente. Porém, quando usamos esse termo para falar do desenvolvimento de um software, sua definição é um pouco diferente. Dizer que um software é sustentável é o mesmo que dizer que esse software foi arquitetado para evoluir dentro de uma organização. Ou seja, continuar existindo pelo maior tempo possível.

De forma geral, desenvolver uma aplicação é extremamente caro e, por isso, traz impactos diretamente na performance da empresa, principalmente na área de investimentos dessa organização. Hoje em dia, sabemos que grande parte das corporações que estão nascendo, e que inclusive estão revolucionando o mundo todo, são empresas de tecnologia, logo, o desenvolvimento de sistemas é uma parte core da companhia. A Lyft, a Uber e o iFood são alguns exemplos de organizações em que grande parte do investimento para que elas possam funcionar é relacionado às tecnologias.

Outro ponto importante que precisamos entender é que o software, de modo geral, busca resolver uma "dor" que a organização tem. Então, utilizando o exemplo do iFood, por ser uma empresa de tecnologia voltada a entrega de alimentos, algo que poderia ser feito por meio de uma ligação telefônica, essa precisou evoluir para assim ter diferenciais que fizessem com que a empresa pudesse se manter no mercado. Por exemplo, seu software tem mais escalabilidade para ele seja

viável no dia a dia e assim se diferenciar de pedidos por ligações telefônicas, conseguindo atender a demanda do mercado.

O software pode ser considerado parte de uma empresa, ou seja, a medida que a organização evoluiu esse software precisa evoluir também. Ele precisa se manter, evoluindo, de forma que o custo dele seja muito menor do que o resultado que ele está trazendo para a corporação. Desse modo, haverá um ponto de equilíbrio, isto é, a solução conseguirá retornar seu investimento.

Quando as empresas se dão conta de que determinado software chegou em seu limite de evolução, novos projetos de "modernização tecnológica" são criados exatamente para dar conta do que o software "legado" não conseguiu. Com certeza se você já está no mercado há alguns anos, você já ouviu algum caso parecido com o citado acima. Agora, a pergunta que podemos nos fazer é: o software parou de evoluir realmente por que a tecnologia evoluiu muito ou por que a forma com que ele foi criado tornou sua evolução cada vez mais caótica?

Sem dúvidas a tecnologia evolui. Novas linguagens, frameworks, bibliotecas são criadas todos os dias, porém, em muitos casos, a forma com que um software é criado tem um peso infinitamente superior do que a tecnologia utilizada pelo mesmo.

O software precisa nascer sustentável e continuar em produção pelo maior tempo possível. Quanto mais tempo ele fica no ar, mais retorno ele gera para a empresa e isso só é possível se sua base for bem feita. Só é possível se no

dia zero cada desenvolvedor, arquiteto, product owner, entre outros, de forma intencional, pensarem em como fazer com que esse software fique no ar e entregue valor por diversos anos.

Então, para que um software seja sustentável, ele precisa ser bem arquitetado, assim poderá se manter mais tempo no ar e conseguir gerar mais retorno para a organização. Ou seja, sem sustentabilidade no dia zero o software pode trazer mais prejuízos do que valor e isso pode significar a razão da empresa continuar ou não suas operações.

Modalidades arquiteturais

Entender sobre arquitetura de software é essencial para desenvolvedores que almejam trabalhar em grandes projetos. Esse entendimento vai expandir sua compreensão sobre o desenvolvimento de sistemas e, assim, aumentar as possibilidades de trabalho em sua carreira como pessoa desenvolvedora.

É importante dizer que, dentro de uma organização, a arquitetura, de forma geral, contribui para que seja gerado valor ao negócio. Nesse sentido, é extremamente relevante para a corporação ter profissionais com experiência na área. Podemos chamar esses profissionais de arquitetos. Existem diferentes tipos de arquitetos, porém apresentaremos neste módulo quatro que são considerados mais relevantes para área de desenvolvimento de sistemas.

No primeiro tópico, falaremos sobre o arquiteto tecnológico, depois sobre o

arquiteto corporativo, no terceiro tópico sobre o arquiteto de solução e, por último, sobre o arquiteto de software.

Os profissionais que trabalham nessas modalidades são de grande relevância para as organizações, pois além de terem um amplo conhecimento técnico em diversas áreas, suas decisões impactam diretamente no processo de planejamento e desenvolvimento de soluções nas organizações como um todo. Lembrando que, dependendo da empresa, essas funções podem sofrer algumas variações ou muitas vezes serem inexistentes (a depender da necessidade).

Arquitetura tecnológica

Neste tópico falaremos um pouco sobre quem é e o que faz uma pessoa arquiteta tecnológica. Daremos exemplos para que se perceba qual sua importância dentro de um negócio.

Essa pessoa é basicamente alguém com especialidade em uma determinada tecnologia. Por possuir conhecimento elevado, conhece detalhes sobre a tecnologia na qual é habilitada.

Primeiramente, é importante destacar que existem diversos tipos de arquitetos para as mais diferentes tecnologias. Tendo isso em mente, usaremos alguns exemplos para aprofundarmos nosso entendimento sobre tal profissional: Ao trabalharmos em um grande projeto, nos deparamos com a necessidade de usarmos diversas tecnologias. E para o bom funcionamento deste projeto, eventualmente,

é necessário que se tenha profissionais com conhecimentos específicos nesses tipos específicos de solução. Desse modo, surge a necessidade da pessoa arquiteta tecnológica. A sua experiência vai gerar valor ao projeto, baseado na expertise que ela tem referente a determinada tecnologia.

A Elastic Stack pode ser considerada um bom exemplo da necessidade de ter técnicos especializados. Essa stack possui um conjunto de ferramentas específicas. Profissionais tecnológicos especialistas nessas ferramentas conhecerão de forma profunda sobre Elasticsearch, Kibana, Beats, Logstash e Fleet, e por isso poderão ser nomeados como pessoa arquiteta tecnológica Elastic. Inclusive, há uma certificação provida pela empresa Elastic para esse tipo de profissional. Desse modo, outra pessoa desenvolvedora pode até conhecer um pouco da Elastic Stack, porém, não compreenderá os detalhes sobre a sua arquitetura.

Outro exemplo que podemos citar é do arquiteto tecnológico Java. Profissional que, sabe muito além de desenvolver em Java, conhece com profundidade sua JVM, sendo possível, do mesmo modo, adquirir certificações específicas para essa tecnologia.

Além desses exemplos, podemos encontrar, ainda, arquitetos focados em bancos de dados, que normalmente também são DBAs.

A Salesforce e a SAP também possuem tecnologias extremamente específicas, logo, caso uma organização queira implementá-las em seu sistema, a presença de um profissional que domine essas tecnologias é fundamental.

Assim, quando uma corporação opta por trabalhar com determinadas tecnologias, por suas complexidades, é evidente a necessidade do trabalho de uma pessoa arquiteta tecnológica para que se possa alcançar êxito em seus projetos.

Arquitetura corporativa

Apesar de sua importância, pouco se ouve falar em arquitetura corporativa, e, por isso, sua definição levanta dúvidas entre pessoas desenvolvedoras.

Quando pensamos em arquitetura corporativa, precisamos ter em mente que estamos nos referindo a políticas e regras que impactam estrategicamente a organização como um todo.

A pessoa arquiteta corporativa, então, poderá fazer a avaliação dos custos que toda a área de desenvolvimento e engenharia terá para desenvolver os projetos que farão sentido para um negócio crescer. Esses custos podem ser com Devs, verticais de desenvolvimento, tipos de tecnologias e licenças. Além disso, essa pessoa está habilitada também para o planejamento de grandes implementações tecnológicas. Por meio de sua análise é possível verificar se é necessário ou não a migração de todos os setores para essa nova tecnologia implementada, por exemplo. Assim, esse profissional poderá indicar uma possível padronização das tecnologias dentro da empresa.

Por meio de alguns exemplos será possível compreender melhor a sua relevância para uma organização. Pensemos em uma situação hipotética em que temos

uma software house ou um grande banco. Nesse contexto, temos milhares de funcionários, logo, se não houver regras e uma governança sólida, o controle pode ser perdido e gerar grandes danos à corporação. Alguns questionamentos devem ser levantados para que seja feito essa governança sólida: como os funcionários utilizarão as tecnologias? Quais serão as tecnologias? Quais serão os principais vendors? Normalmente, esses questionamentos serão respondidos por um arquiteto corporativo.

Outro exemplo prático é quando surge uma nova tecnologia, ou são lançadas novas versões das ferramentas já existentes, e a empresa precisa analisar se a migração faz sentido para aquela situação e contexto. Nesse sentido, a pessoa arquiteta corporativa estará apta para que tal decisão seja tomada da melhor forma possível.

Outro exemplo é de empresas como a Salesforce, que oferece soluções em CRM (gestão de relacionamento com o cliente). Essas soluções, ao serem implementadas dentro de uma corporação, mudarão culturalmente a forma como o pessoal de vendas e pós-vendas trabalhará. Nesse caso, um profissional de arquitetura corporativa avaliará qual impacto isso trará para a companhia, e como aproveitar essa implementação em outros setores dentro da organização, por exemplo, se faz sentido também utilizar a nuvem de marketing da Salesforce.

A pessoa arquiteta corporativa é essencial dentro de uma companhia, pois sem sua avaliação a pluralidade de tecnologias implementadas pode impactar negativamente os projetos da empresa.

Ela ajudará a alinhar e padronizar, estrategicamente, as áreas dentro da corporação. Fazendo com que as implementações tecnológicas façam sentido e contribuam para o crescimento do negócio.

Arquitetura de solução

Neste tópico, nos dedicaremos a entender o que é uma pessoa arquiteta de solução e como essa profissional pode ajudar no dia a dia de uma corporação.

O primeiro ponto que precisamos entender é que o trabalho dessa pessoa fica entre a área de negócios e a área de software, ou seja, fica em uma área técnica que busca transformar requisitos de negócio em soluções de software. Isto é, ela vai enxergar as especificações e as necessidades da organização e fazer isso virar software.

Um de seus papéis diários é fazer desenhos arquitetônicos da solução para reproduzir como ela irá funcionar. Profissionais de solução precisam ter a capacidade de expressar seus pensamentos por meio de desenhos. Obviamente, existem diversas formas desse profissional desenhar e documentar suas ideias. Um exemplo é utilizar desenhos em diagrama C4, UML e BPMN para apresentar como irá transformar algo que está em sua cabeça em solução de software.

Outro papel que também pode ser atribuído a essa pessoa é o de analisar os impactos comerciais em relação a uma escolha tecnológica. Por existir diversas formas de solucionar uma necessidade dentro de uma empresa, ela escolherá,

pelo contexto do cliente, qual será a melhor solução possível, levando em consideração tanto aspectos técnicos quanto comerciais.

Vamos imaginar, por exemplo, que todo software legado de uma companhia está utilizando a AWS, não fará sentido fazer esse cliente migrar toda sua infraestrutura para o Google, somente por uma opinião de que o Google é melhor, a não ser que exista, por exemplo, um ponto financeiro que será impactado positivamente com a mudança. Ou seja, toda escolha vai depender do contexto da organização. Outra situação, seria se toda base de dados de um sistema de uma empresa estivesse utilizando Oracle; não podemos mudar para SQL Server se não fizer muito sentido para o crescimento da corporação.

Outra função que a pessoa de solução pode assumir é a de participar de reuniões com clientes na etapa de pré-venda e/ou venda para apresentar detalhes técnicos de como a solução poderá agregar valor ao negócio. Geralmente, a empresa envia para essas reuniões consultores que irão negociar com os clientes, porém, esses não são profissionais extremamente técnicos. Eles compreendem o funcionamento do software, o sistema de precificação e a negociação, porém caso seja necessário apresentar detalhes mais técnicos, arquitetos de solução poderão demonstrar ao cliente como que, tecnicamente, a solução poderá ser implementada. Além disso, arquitetos poderão analisar os impactos de custos da implementação deste software para o negócio. Um exemplo dessa análise é quando uma empresa opta por implementar um sistema CRM. O consultor poderá apresentar a solução de forma geral, algo menos técnico, porém, o

entendimento profundo de como a solução pode impactar a organização, a metrificação de algo mais técnico como integrações e migrações, será melhor apresentada por uma pessoa arquiteta de solução, já que ela conseguirá gerar um pouco mais de previsibilidade sobre todos esses pontos que a empresa terá que trabalhar durante a implementação da solução.

O que é arquitetura de software

Para compreendermos um pouco mais sobre arquitetura de software é importante sabermos que esta é uma disciplina da engenharia de software. Quando falamos em engenharia de software, estamos nos referindo a todos os processos de desenvolvimento de uma solução, desde as metodologias até o formato em que o software será desenvolvido. Ou seja, essa engenharia compreende que existe um ciclo de desenvolvimento para criação de um software. E a arquitetura de software contribuirá para que esse ciclo aconteça.

Existem diversas definições válidas para arquitetura de software. Desde definições mais formais até algumas mais informais. Inicialmente, vamos definir essa arquitetura como a relação entre os objetivos de um negócio e suas restrições com os componentes a serem criados e suas responsabilidades, visando sua evolução.

Todo negócio, ao implementar um software, tem um objetivo a ser atingido, mas restrições financeiras, de equipe e tecnológica, por exemplo, podem afetar o

processo de construção dessa aplicação. A pessoa com habilidades arquitetônicas poderá contribuir no processo de construção da solução para que essa se adapte às restrições presentes na corporação.

Além disso, essa pessoa poderá contribuir para que se possa desenvolver uma solução que mantenha uma relação bem estruturada entre seus componentes, fazendo com que esses atendam os objetivos do negócio. Isto é, ela poderá fazer com que esses componentes, em conjunto, apesar das restrições, consigam gerar uma solução de alta qualidade que irá atender as necessidades da corporação.

Outra definição mais formal é a da ISO/ IEC/IEEE 42010. Este instituto define arquitetura de software como "*...a relação de um sistema e seus componentes, suas relações, seu ambiente, bem como os princípios que guiam seu design e evolução*". Normalmente quando desenvolvemos um software não pensamos apenas em curto prazo, pensamos, ou deveríamos pensar, em como esse software vai evoluir a médio e a longo prazo. Logo, a pessoa arquiteta de software vai pensar nos componentes atuais da corporação e em como esses poderão evoluir dentro de uma solução. Ou seja, apresentará o desenho de um software, de fato, sustentável.

A lei de Conway pode nos ajudar a entender um pouco mais como funciona a arquitetura de software. Melvin Conway diz que "*organizações que desenvolvem sistemas de software tendem a produzir sistemas que são cópias das estruturas de comunicação dessas empresas*". Vamos imaginar um exemplo prático: uma companhia decide implementar um sistema de software. Nessa corporação, há um

desenvolvedor Backend, um desenvolvedor Frontend e um DBA. Provavelmente, quando essa aplicação for desenvolvida, teremos uma solução Backend que vai se comunicar com o banco de dados e certamente, também uma SPA (Single Page Application) Frontend para se comunicar com Backend. Porém, se na organização houver somente um profissional Backend e um DBA, dependendo da situação, o frontend da aplicação será uma parte do projeto principal rodando em conjunto com backend. Então, resumidamente, a arquitetura do software vai ser desenvolvida de acordo com os times disponíveis para o projeto.

Assim, podemos dizer que o processo de arquitetar um software estabelece que o que está sendo desenvolvido faça parte de um conjunto maior que, normalmente, é o negócio.

Arquitetura de software x arquitetura de solução

Por existir um ponto de intersecção entre a arquitetura de software e a arquitetura de solução, seus conceitos são, muitas vezes, confundidos. O que precisamos ter em mente é que a arquitetura de software representa algo mais baixo nível se comparada com arquitetura de solução. A arquitetura de software está diretamente ligada ao processo de desenvolvimento do software, seus padrões, qualidade de código, boas práticas, bem como sua estrutura.

A pessoa arquiteta de software poderá contribuir diretamente na estrutura organizacional de uma empresa, ou seja, dependendo da arquitetura de um sistema, ela poderá definir a divisão dos times dentro da organização. Em

outras palavras, dependendo dos componentes que o software terá, essa pessoa auxiliará na seleção dos profissionais que farão parte do projeto. Diferentemente da pessoa arquiteta de solução que quase sempre aborda aspectos de alto nível, sendo que raramente chegará a trabalhar diretamente com código.

O papel do arquiteto de software

Apesar de nem todas as organizações possuírem o cargo de pessoa arquiteta de software, normalmente profissionais mais experientes como desenvolvedores seniors e tech leads acabam realizando esse papel baseado em suas experiências anteriores.

O principal papel que essa pessoa pode assumir é a função de transformar requisitos de negócios em padrões arquitetônicos. Ou seja, ela vai pensar em como atender a alguns requisitos da empresa os transformando em uma solução. Para isso irá utilizar seus conhecimentos sobre padrões arquitetônicos.

A pessoa arquiteta é, na maioria das vezes, um desenvolvedor e em seu dia a dia pode orquestrar o fluxo de comunicação entre pessoas desenvolvedoras e experts de domínio. A necessidade de ter um expert de domínio trabalhando junto com o desenvolvedor surge por existir, normalmente, uma dificuldade em alinhar o que vai ser desenvolvido com o que o cliente precisa. Esse expert é uma pessoa que sabe da necessidade da organização, ou na maioria das vezes é a pessoa que vai utilizar o software no dia a dia. A pessoa arquiteta poderá facilitar o fluxo de

comunicação entre esses dois profissionais, para que a aplicação seja produzida do jeito esperado.

Em outras palavras, por ser cada vez mais natural que as equipes tenham autonomia para tomar decisões de design e arquitetura, é muito evidente que a grande pressão do dia a dia faça com que o software tome caminhos arquiteturais diferentes do que se havia planejado para garantir sua sustentabilidade, assim como na garantia do atendimento dos atributos de negócio; por conta disso, um arquiteto ou arquiteta de software deve se fazer presente nos projetos.

Ainda sobre o papel dessa pessoa, entender profundamente sobre conceitos e modelos arquiteturais é essencial para que a pessoa arquiteta consiga auxiliar na resolução de problemas. Por vezes é comum que um profissional queira solucionar um problema baseado apenas em sua experiência. Um bom exemplo disso é quando um desenvolvedor costuma criar sistemas monolíticos. É comum que nos próximos desafios esse profissional queira resolver as situações de forma monolítica. Porém, quanto mais uma pessoa entende de modelos arquiteturais, maior vai ser a diversidade de possibilidades para resolução de desafios, pois ela conseguirá entender como se pode contextualizar o desafio para conseguir resolvê-lo da melhor forma possível. Isso tudo, não apenas pela nomenclatura de arquiteto, que algumas vezes nem é presente na organização, mas porque normalmente esse arquiteto já tem muita experiência, isto é, já vivenciou muitas situações e quando se vê frente a um desafio, como por exemplo um atraso na entrega ou insatisfação do cliente. Essa pessoa é capaz de se reunir com o time e,

baseado em sua experiência, poderá propor uma variedade de possibilidades que, tecnicamente falando, poderão ajudar a equipe. É importante dizer que muitas dessas situações não estão previstas no negócio, nem no calendário de entrega, mas fazem parte do processo.

Outro papel importante da pessoa arquiteta é reforçar boas práticas de desenvolvimento. Isso é feito através de testes, trabalho com SOLID, conexões e com bancos de dados. Essa profissional poderá fazer os seguintes questionamentos: Como essa solução irá funcionar? Vai utilizar clean architecture? Vai trabalhar com eventos? Ou seja, nitidamente, por ser uma pessoa que está diretamente ligada ao software que está sendo desenvolvido e, apesar de sua visão não abordar aspectos de alto nível como a do arquiteto de solução, ela está preocupada com o "if" que o desenvolvedor está fazendo. Mesmo tendo criado e organizado os componentes arquiteturais ela irá contribuir, durante o processo de desenvolvimento, para que essa solução tenha um padrão de qualidade garantido.

Code reviews

Algo comum na carreira de um arquiteto de software é que, por ser além de desenvolvedor, ele poderá participar de mais de um projeto ao mesmo tempo e por esse motivo não tem disponibilidade para ficar codificando com outros devs. Nessa situação, ele pode ser uma das pessoas atribuídas para fazer code reviews, para que dessa forma ele possa validar se os componentes e os requisitos de arquitetura definidos estão presentes na solução.

Departamento de arquitetura na empresa

Há empresas que apesar de não possuírem formalmente o cargo de arquiteto de software, possuem um departamento de arquitetura que auxilia os diversos times da organização com questões arquiteturais. Esse departamento pode dar suporte aos desenvolvedores. Os devs desse departamento conseguem visualizar todos os projetos e microsserviços funcionando e como a empresa está avançando tecnologicamente. Ao surgir um novo projeto na organização essa equipe de suporte arquitetural poderá auxiliar os desenvolvedores por meio de uma avaliação. Nessa avaliação, será analisado se aquele projeto já existe, ou existiu, e nestes casos poderá sugerir a utilização desses projetos já existentes ao invés de criação de novos. Além disso, esse setor avalia também toda documentação que descreve o software, podendo dizer se está falha ou com padrões inadequados.

Logo, mesmo não tendo um arquiteto de software como cargo específico, a organização tem devs que trabalham nessa área arquitetural para aprovar e auxiliar as equipes de projetos a tomarem as melhores decisões, alinhadas ao contexto da organização.

Além disso, a área de arquitetura monitora, gerencia os processos de mudança e garante a governança, reforçando assim um padrão de qualidade em todos os projetos.

Apesar de muitos entenderem que áreas como essa podem burocratizar o processo de desenvolvimento e retirar a liberdade dos devs, é evidente que

quando há uma grande quantidade de equipes, sistemas e tecnologias envolvidas, é imperativo que haja controle para garantir a sustentação de todo o ecossistema a médio e a longo prazo. Isso reforça a ideia de que a criação desse setor em uma corporação não tem a intenção de controlar o trabalho de outros desenvolvedores. A ideia é que os profissionais dessa área consigam consolidar seus conhecimentos sobre o contexto da empresa para ajudar e apoiar os devs que estão criando projetos no dia a dia.

Pelo fato das corporações não utilizarem a nomenclatura de arquiteto é comum vermos tech leads assumindo essa posição por tomarem decisões totalmente arquiteturais.

As vantagens de aprender arquitetura de software

Mesmo que eventualmente não exista o cargo de pessoa arquiteta formalmente na maioria das empresas, pessoas desenvolvedoras atribuídas a cargos de liderança técnica muitas vezes acabam tomando decisões arquiteturais fazendo assim, informalmente, esse papel.

Ter conhecimento sobre arquitetura permite navegar da visão macro para a visão micro de um ou mais software. Dessa forma, é possível visualizar aspectos de alto nível e de baixo nível dentro da solução. Com isso, podemos perceber que o código é um componente que se relaciona com outros. Podemos garantir também que esses componentes sejam construídos de modo que possam ser desacoplados caso isso seja necessário para sua evolução.

A pessoa desenvolvedora, ao estudar arquitetura, compreende quais são os diversos protocolos para desenvolver a mesma solução, e partindo disso ela poderá escolher a melhor opção para determinada situação e contexto. Um dev que entende como os componentes se relacionam saberá quais são as formas arquitetônicas para trabalhar como por exemplo CQRS, Arquitetura baseada em eventos, etc. Também, essa pessoa poderá avaliar quando um sistema deve ser ou não monolítico. Esses conhecimentos possibilitam, ainda, que a pessoa desenvolvedora pense no projeto a longo prazo, isto é, na sustentabilidade; nos aprofundaremos neste tema mais adiante. Quando desenvolvemos um software é muito comum que tenhamos prazos pouco flexíveis e isso acaba criando uma tendência pela busca de resoluções a curto prazo. Através dos conhecimentos sobre arquitetura, mesmo com prazos curtos, é possível modelar o software de forma que seja mais fácil mantê-lo a longo prazo. Lembrando que o papel do desenvolvedor é conseguir fazer com que esse software retorne o valor para a empresa. Se ao passar dos anos essa solução não puder evoluir, eventualmente a solução pode ter trazido mais prejuízo do que valor para a organização.

Trabalhar com arquitetura de software de forma intencional pode definir o sucesso ou fracasso de um projeto. Pois o sucesso de um projeto não está na primeira entrega, mas sim nas evoluções subsequentes.

Por não conhecerem muitas possibilidades para solucionar desafios, é comum que desenvolvedores pouco experientes fiquem interessados em implementar novas tecnologias assim que são lançadas. Eles acreditam que essa nova tecno-

logia é a melhor forma de solução para as diversas situações presentes em uma organização. Porém, é necessário que se compreenda se essa nova tecnologia atende os objetivos daquela corporação. Isso só é possível caso a pessoa tenha uma visão macro e micro do negócio para saber exatamente qual será impacto daquela tecnologia no projeto, e se vale a pena usá-la ou não naquele momento e contexto. Sem dúvidas ter conhecimentos sobre arquitetura faz com que essa pessoa desenvolvedora consiga tomar decisões de forma mais racional, evitando assim ser influenciado por "hypes" de mercado.

Ao aprender sobre a visão macro e micro de uma solução a pessoa poderá ter mais clareza do impacto que o software possui na organização como um todo e não apenas em uma área. Esse senso de pertencimento, ou seja, de saber exatamente como seu trabalho afeta toda corporação, tem benefícios para a sua carreira, bem como para manter sua motivação profissional por saber que seu trabalho é significativo.

Quando um dev não compreende totalmente os conceitos de arquitetura, isso pode limitá-lo ao processo de solucionar problemas, gerando eventualmente insegurança na tomada de decisão. O entendimento de arquitetura não será necessariamente uma "bala de prata", porém sem dúvidas será um ponto de partida na busca por soluções.

É importante saber também que aprender sobre arquitetura nos traz a necessidade de mergulharmos em padrões de projetos e de desenvolvimento e suas boas práticas. Ou seja, a arquitetura nos força estudar quais foram os padrões que

outras pessoas já utilizaram para resolver diversos tipos de problemas, fazendo com que possamos ganhar tempo no processo de desenvolvimento, além de termos a possibilidade de padronização da solução, trazendo mais clareza a outros profissionais que eventualmente poderão ter contato com a aplicação.

Arquitetura vs Design de software

Existe uma linha de pensamento que afirma que arquitetura e design de software são a mesma coisa. Neste tópico, faremos algumas reflexões sobre essas duas áreas e, assim, iremos perceber que, de certo modo, essas áreas podem ser consideradas distintas.

Quando falamos em arquitetura de software, estamos nos referindo ao escopo global de um software, ou seja, visualizar esse software em um nível mais alto. Conseguimos ver a componentização, a comunicação e as abstrações dessa solução. Por outro lado, quando falamos em design de software estamos apontando para um escopo mais local, isto é, mais baixo nível. Podemos pensar nos seguintes questionamentos: como deixar uma classe com menos responsabilidade? Como implantar patterns para facilitar nossa estratégia? As respostas para essas questões podem ser consideradas decisões de design de software.

A citação do Elemar Jr pode nos ajudar a entender melhor os conceitos sobre essas duas áreas: "*Atividades relacionadas à arquitetura de software são sempre de design. Entretanto, nem todas as atividades de design são sobre arquitetura.*

O objetivo primário da arquitetura de software é garantir que os atributos de qualidade, restrições de alto nível e os objetivos de negócio sejam atendidos pelo sistema. Qualquer decisão de design que não tenha relação com este objetivo não é arquitetural. Todas as decisões de design para um componente que sejam "visíveis" fora dele, geralmente, também não são". Logo, sempre que falarmos de arquitetura estaremos nos referindo também à design, pois até mesmo para desenvolver um componente é necessário utilizar aspectos de design, entretanto, nem todas as atividades de design podem ser consideradas arquiteturais. Normalmente, ao pensar em arquitetura, estamos falando de requisitos não funcionais e, ainda, restrições de alto nível.

Por exemplo, para que logs de um sistema possam ser centralizados e facilmente recuperados, a Elastic Stack será utilizada. Podemos perceber que tal decisão afetará a aplicação como um todo, além da possível contratação de mais infraestrutura para que a Elastic Stack seja instalada, ou mesmo, uma eventual contratação de um serviço gerenciado na Elastic Cloud. Decisões como essa podem afetar todos os sistemas de uma organização, seu orçamento, o tempo em que um possível bug pode ser corrigido, a forma com que cada time trabalhará no dia a dia com a observabilidade, além do conhecimento básico na Stack que será requerida por cada desenvolvedor e eventuais treinamentos que os mesmos deverão receber para operar ferramentas. Decisões que impactam diretamente em quais componentes e vendors que um projeto utilizará são decisões arquiteturais.

Por outro lado, quando tomamos decisões de quais patterns GoF (Gang of Four) o projeto utilizará; SOLID, DRY, Clean code, a quantidade de camadas de uma aplicação, estamos nos referindo fundamentalmente ao design do software.

Pilares da arquitetura de software

Para compreendermos o processo de arquitetar um software é importante separarmos alguns de seus conceitos em pilares, isso facilitará nosso entendimento. Vamos organizar nossos estudos nos seguintes tópicos: organização, estruturação, componentização, relacionamento entre sistemas e governança.

Organização

Quando falamos em arquitetura, falamos em organizar um sistema (não o software em si) que possibilite a fácil componentização, evolução, bem como um fluxo rico para que possamos atender os objetivos de negócio gerando um produto para o cliente final.

Estruturação

Estruturar um software significa organizá-lo para que este seja de fácil evolução e componentização. Além disso, a solução precisa atender os objetivos de negócios, tendo componentes com estruturas claras. Sem isso não é possível criar um software de qualidade e que consiga evoluir com o passar do tempo.

Componentização

Uma solução pode ser o conjunto de diversos sistemas e, dependendo da interpretação, eventualmente esses sistemas precisam se comunicar, por isso é necessário compreender como componentizar uma solução. Isto é, entender como um sistema pode se relacionar com os outros. É através da junção de componentes que nós conseguimos atingir os atributos de qualidade do sistema. Devemos pensar em todo ecossistema que existe em torno do processo que nós pretendemos fazer até o resultado final, já que esses componentes serão usados em diversos momentos do nosso trabalho.

Grande parte do trabalho no mundo da arquitetura de software envolve dominar as formas de realizar a componentização dos processos para que eles operem com eficácia, evitando o retrabalho. Podemos dizer então que a componentização é a base na criação de um software.

Relacionamento entre sistemas

Dentro de uma corporação é comum que se tenha mais de um software. Por isso é importante que, ao desenvolvermos uma aplicação, saibamos como preparar seus componentes para que esses consigam se integrar de maneira eficiente dentro de um processo maior. É necessário, então, observar se os protocolos estão apropriados, se as redes estão sendo usadas de modo necessário e se as regras de segurança estão sendo efetivas.

Governança

Principalmente nas grandes empresas a vasta quantidade de sistemas e integrações são cada vez maiores. Muitas pessoas veem a governança como uma forma de burocratizar o processo de desenvolvimento; porém, na grande maioria das vezes, o software a ser desenvolvido é apenas mais um dentre as centenas de outros que já estão em operação. Logo, é necessário que se tenha padronização, regras e documentação. Ou seja, definições que fiquem claras para todos os colaboradores. Por exemplo, quais linguagens serão utilizadas? Quais protocolos serão aplicados? Quais sistemas serão utilizados para se comunicar? É necessário ter o mínimo de governança para que essa solução se integre às demais e consiga evoluir naquele ambiente.

A governança busca a garantia de que o software continue funcionando independente de equipe. Tendo um requisito base de governança, teremos a segurança de que as pessoas que trabalham nessa aplicação sejam dispensáveis. Quando falamos dispensáveis, não estamos falando do valor dessas pessoas dentro da corporação, mas sim sobre poder substituir esses profissionais em caso de eventuais mudanças. Com regras e protocolos definidos se torna mais rápida a adaptação de novos devs, caso seja necessário, para o projeto não se perder. Além disso, ao se fazer tão necessário em um projeto um desenvolvedor poderá perder uma possível promoção dentro de sua organização. Quando isso acontece, sua carreira é afetada negativamente, pois a dependência dessa pessoa no projeto

impossibilita sua evolução.

Requisitos Arquiteturais (Ras)

Quando desenvolvemos um software, principalmente na parte de planejamento, é visível que, para que essa solução seja bem construída, necessitamos de requisitos bem definidos. Para isso, pensamos em como esses aspectos irão impactar diretamente a arquitetura do nosso software. E caso esses não sejam funcionais, muitas vezes podem ser considerados como requisitos arquiteturas (Ras).

Hoje em dia, com o modelo de Squad, isto é, cada time criando seu próprio software e entendendo como as coisas funcionam, raramente vemos Ras de um modo formalizado. Antigamente, trabalhávamos muito com vários documentos do excel, um requisito por vez, ou seja, cada detalhe de todos os requisitos arquiteturais separadamente e podíamos ver RAs formalmente dentro das organizações.

Atualmente, apesar desses requisitos serem mais vistos quando falamos em arquitetura de solução, é importante, ao desenvolvermos um software, que tenhamos conhecimento, ao menos o básico, de requisitos arquiteturais. Quando compreendemos esses requisitos, podemos planejar a arquitetura da aplicação da melhor forma possível.

Para facilitar nosso entendimento, separamos alguns requisitos que conside-

ramos essenciais ao nosso estudo: performance, armazenamento de dados, escalabilidade, segurança, legal, audit (auditoria) e marketing.

Performance

Dizemos que se trata de um requisito de performance quando, em uma aplicação, temos uma regulação com o limite de requisições preestabelecido, por exemplo, de 500 milissegundos. Outro ponto importante é o throughput dessa aplicação - se tivermos uma máquina com 1000 millicores rodando e precisarmos aguentar 50 transações por segundo, esse também será um requisito de performance.

Armazenamento de dados

Quando uma empresa, eventualmente fecha contrato com a AWS, por exemplo, e a equipe precisa se adaptar para utilizar seus bancos de dados, inclusive o DynamoDB. Dizemos que essa adaptação está relacionada a um requisito arquitetural de armazenamento de dados.

Outra situação ocorre quando os dados precisam cumprir regulações, ou seja, se o software estiver rodando na Europa precisa ter um datacenter europeu. Não é apenas uma questão tecnológica mas sim de praticar a compliance dentro da organização. Ou seja, estar de acordo com uma regra e com a legislação vigente. Isso vai gerar valor ao negócio e poderá contribuir para sua permanência no mercado.

Escalabilidade

Para pensarmos em escalabilidade precisamos ter em mente como esse software vai escalar. Verificaremos se ele vai escalar horizontal ou verticalmente. Outro ponto que veremos neste requisito é se podemos optar pelo uso do load balancer para facilitar o bom funcionamento do sistema em caso de um congestionamento.

Segurança

Quando trabalhamos com E-commerce é muito comum recebermos transações via cartão de crédito. Estas precisam estar com certificações PCI, por exemplo. Além disso, pode surgir a necessidade de um sistema rodar criptografado ou a comunicação entre os microsserviços precisarem rodar usando Mutual TLS. Essas situações podem ser consideradas de requisito arquitetural relacionada à segurança.

Legal

Como já foi dito, precisamos observar quais são os requisitos legais para que nós possamos cumprir a legislação vigente de cada país. No Brasil, temos a LGPD (Lei Geral de Proteção de Dados), extremamente necessária para que seja possível construir mecanismos que evitem ao máximo o vazamento de dados.

Audit

Ao criarmos projetos precisamos garantir que existam parâmetros e formas de auditorias. Isso nos permite entender como fazer verificações de maneira eficiente.

Alguns questionamentos podem nos guiar em relação a esse requisito: onde a aplicação estará logada? Como conseguir logar? Por quanto tempo o dado ficará retido? Esses aspectos são importantes, pois tudo o que acontece em um sistema precisa estar em logs.

Marketing

Caso nossa aplicação precise participar de campanhas de marketing, ela precisará ter disponibilidades específicas para sustentar os diversos pontos ligados ao marketing. Precisaríamos, por exemplo, ter regras para garantir que nossa solução consiga trackear de onde vem cada acesso. Desse modo, será possível garantir que cada tipo de acesso em nosso sistema esteja cacheado mais próximo do usuário para garantir um acesso mais personalizado.

Requisitos arquiteturais na prática

A forma mais comum de fazer/organizar os requisitos arquiteturais é através de planilhas. Podemos nos reunir com experts de domínio e com o executivo

para fazermos alguns questionamentos que poderão nos auxiliar a montar os documentos que nos guiarão na tomada de decisão de RAs. Durante a reunião, podemos fazer algumas perguntas para, assim, adequar nossa solução à melhor arquitetura. Por exemplo, quais setores irão utilizar a solução? Os clientes que irão acessar o software serão somente internos, ou internos e externos? Esses questionamentos também podem ser feitos a pessoas de departamentos específicos da empresa, por exemplo, fazer esse levantamento no setor jurídico. Isso vai depender da funcionalidade da aplicação que pretendemos desenvolver.

No dia a dia, não vemos esse tipo de organização sendo feita com frequência. Geralmente as coisas acontecem de maneira mais orgânica e flexível, mesmo em corporações de grande porte com diversas restrições como os bancos. Porém, de uma forma ou de outra, requisitos existem e muitas vezes ficam subentendidos. O que devemos levar em consideração é que quanto mais clareza nós tivermos do tipo de software que iremos criar, ou seja, dos requisitos arquiteturais que teremos que utilizar, mais clareza teremos no processo de desenvolvimento. Desse modo, poderemos evitar ruídos quando nossa aplicação estiver no ar.

Características Arquiteturais

Sempre que desenvolvemos um sistema, de uma forma ou de outra, este possui características arquiteturais. E algumas vezes essas características são ruins. Isso acontece quando não se pensa de maneira intencional em aspectos específicos ao projetar a solução. Quando estamos trabalhando com arquitetura de software,

isto é, desenvolvendo algum sistema é importante entender como estruturá-lo de maneira intencional.

Compreender a estrutura do software facilita a visão de forma intencional nos pontos que poderão impactar a solução. Se não tivermos uma visão geral de como trabalhar intencionalmente com a arquitetura de um sistema, podemos ter dificuldade para resolver alguns problemas que eventualmente surjam durante o seu processo de evolução.

Trabalhar baseado em uma intenção significa estar preparado para resolver determinados problemas. Caso não tenhamos uma base de arquitetura no processo de desenvolvimento talvez até consigamos resolver um problema indiretamente. Porém, isso significaria contar com a sorte, e nisso não existe garantia de que o nosso sistema irá funcionar conforme o esperado.

Por exemplo, sabemos que um software precisa ser resiliente, ou seja, ele precisa se adaptar em momentos de crise e se recuperar rapidamente, ou ter um plano "B" para que não deixe de funcionar diante das dificuldades (adversidades). Vamos imaginar que quando estamos desenvolvendo um sistema não tenhamos pensado em resiliência de maneira explícita, ou seja, de forma intencional. Mas ao utilizarmos bibliotecas, com infraestrutura montada e com muitos aspectos pré-definidos, essa resiliência acaba sendo feita, de maneira embutida e de graça, podemos dizer de modo natural. A conclusão é que nosso sistema pode até ser resiliente, mas isso foi conseguido de modo não intencional. Isso aconteceu por sorte, funcionou mas poderia não ter funcionado. Nesse caso, sabemos

que não é interessante dependermos de sorte em nossas aplicações. A frase de Thomas Jefferson pode nos ajudar a refletir sobre a importância de estar preparado: "*quanto mais trabalho para resolver determinado problema mais eu tenho sorte*" (adaptado). Então, se queremos ter a garantia de "sorte" quando vamos desenvolver software, precisamos estar preparados para verificar pontos e características que devem ser levadas em consideração e são essenciais ao bom funcionamento dessa solução, ou seja, considerar as características arquiteturais.

É importante dizer que, normalmente, muitas dessas características são requisitos não funcionais do seu sistema, isto é, não são requisitos de necessidade explícita e de regras de negócio focadas no que o cliente pediu, mas sim requisitos que vão conseguir nos trazer a garantia de que o sistema trabalhe e suporte a carga. Conseguir de forma geral se manter online da melhor forma possível.

Para compreendermos melhor essas características, vamos dividi-las em três áreas. A primeira área que veremos são as operacionais, em seguida abordaremos a área de características estruturais e por último as Cross-Cutting. Assim veremos especificamente aspectos que permeiam todo o software.

Características Operacionais

A forma como desenvolvemos uma solução poderá impactar na operação desse sistema. Por esse motivo abordaremos, neste tópico, características arquiteturais focadas em aspectos operacionais. Nesse contexto, o nosso objetivo aqui não é

que você seja um expert em criar subnets ou, ainda, um expert em gerenciar backups, por exemplo. Porém, é importante que você saiba como facilitar essas operações. Outro ponto importante, é que possamos compreender como um possível backup pode afetar a performance de nossa aplicação. Resumidamente, tudo que é operacional, normalmente, são coisas que não iremos fazer mas que devemos permitir que sejam operadas em nosso software.

Disponibilidade

O primeiro item que podemos destacar relacionado a características operacionais é a disponibilidade.

Geralmente, quando desenvolvemos uma aplicação, pensamos apenas em como colocá-la no ar, mas não pensamos em o quão disponível deixaremos essa solução. E focar especificamente nisso é algo que pode fazer muita diferença na qualidade do nosso sistema.

Sempre que vamos criar um software, podemos pensar em como garantir que esse esteja disponível. Podemos pensar, por exemplo, se essa solução ficará no ar 24/7. O nível de SLA, que é o quanto combinamos com o cliente, e de SLO, que são os objetivos que queremos garantir para o cliente.

Assim, pensar em como verificar a disponibilidade pode desencadear uma série de aspectos, por exemplo, a observabilidade. Neste contexto, podemos usar técnicas de SRE e, desse modo, fazer um nível de budget de indisponibilidade. Ou

seja, se hipoteticamente podemos ficar indisponíveis somente uma vez por ano durante 1h. Caso fiquemos 20 min indisponíveis, logo teremos somente mais 40 min para ter essa indisponibilidade durante o restante do ano. Assim, precisamos saber como verificar esses incidentes.

Recuperação de desastres

Outro aspecto relacionado à disponibilidade é a recuperação de desastres. Precisamos pensar intencionalmente em como recuperar um sistema quando esse estiver fora do ar. Isso é importante, pois conforme estivermos trabalhando com sistemas, estes podem passar a ter missões cada vez mais críticas. Por exemplo, se estivéssemos trabalhando em um site de uma padaria e o negócio ficasse fora do ar por uma hora, muito provavelmente o dono do negócio nem perceberia. Porém, se um sistema tem missão mais crítica e começa a ficar indisponível, isso pode gerar um problema muito grande para a empresa. No segundo caso, é necessário, então, ter processos específicos para amenizar as consequências da indisponibilidade. Além disso, criar estratégias para mitigar o problema e conseguir evitar que este ocorra novamente.

Outro exemplo é se, por algum motivo, caísse uma região AWS em que nosso sistema está. O quanto estaríamos dispostos a pagar para trabalharmos com multirregião ou para trabalharmos com multicloud. Todos esses aspectos devem ser levados em conta.

Performance

Ao projetarmos uma aplicação é importante que pensemos que esta precisa ser performática. Nesse sentido, falaremos basicamente sobre throughput, que a capacidade de receber e processar requisições. É essencial pensarmos intencionalmente no quanto de performance queremos ter, ou seja, o quanto nosso sistema precisa suportar. Por exemplo, digamos que temos duas situações, na primeira um sistema que precisa suportar 5 mil requisições por segundo e na segunda situação um que precisa suportar 50 requisições por segundo. Esses sistemas precisam ser arquitetados de maneira diferente. Talvez, a segunda situação não precise que trabalhemos com CQRS, mas na primeira situação essa opção é necessária. Pensamos em performance principalmente quando trabalhamos de forma intencional.

Recuperação (backup)

Pode existir uma dificuldade de pensar nesse aspecto, principalmente por ser comum surgir a ideia de que a necessidade do backup está associada a algo ruim que aconteceu em nossa aplicação. O problema é que quando não temos essa reflexão podemos precisar do backup e este não estar disponível. Ultimamente nós podemos ficar, de certo modo, tendenciosos a confiar na nuvem como que sempre garantirá toda disponibilidade que você precisa, por outro lado, temos que lembrar que trabalhar com computação em nuvem significa trabalharmos

com um sistema de responsabilidade compartilhada.

Assim, é importante que criemos regras e políticas específicas onde a cada quantidade de tempo pré-determinado exista um teste de backup. Podemos, ainda, criar uma política para deixar o backup em redes separadas. Isso será eficiente pois caso nossa aplicação sofra um ataque de ransomware, nosso backup não será criptografado junto. Lembrando que, nossa aplicação pode ter missão crítica para empresa e por isso é extremamente importante pensarmos em como garantir o backup nos dias atuais.

Confiabilidade e segurança

Provavelmente, teremos que pensar em aspectos específicos de confiabilidade e segurança quando estivermos trabalhando com sistemas de missão crítica. Para isso, é preciso que saibamos onde estão os pontos de dificuldade e onde ocorrem mais incidentes relacionados à defesa da nossa aplicação. Em algumas situações, por considerarmos esses aspectos relativamente básicos, podemos negligenciá-los. Por exemplo, digamos que para criação de uma conta em determinada solução que estamos desenvolvendo fosse necessário apenas login por uma rede social ou digitar um e-mail para que a conta seja criada automaticamente, isto é, o usuário precisa se esforçar minimamente para criar a conta. Caso o usuário, por algum motivo, digitasse o email errado, e não fosse validado naquele momento, sabemos que iria entrar um “lixo” no banco de dados, mas com rotinas esporádicas para limpeza conseguiríamos eliminar esse email. O

grande problema é se, em algum momento, tivermos pessoas mal intencionadas que resolvam fazer robôs para ficar acessando nosso endpoint para criar contas fakes em nosso sistema. Isso tudo tem a ver com confiabilidade.

Quando pessoas tentam fazer brute force para entrar e fazer login em uma plataforma, também podemos considerar como um exemplo de como aspectos de segurança precisam ser pensados de maneira intencional. Essas pessoas podem colocar robôs para rodar e tentar descobrir a senha dos usuários, o que acaba gerando um número grande de requisições. Sabemos que a maioria dos sites possuem regras e políticas de senhas fortes que dificultam a quebra de senhas. Porém, quando essa pessoa mal intencionada faz vários brute force acaba afetando o banco de dados, a velocidade, utilização da CPU, etc. Imaginemos que uma situação semelhante a essa aconteça durante um evento, ou seja, comecemos a receber vários acessos a ponto de percebermos que nosso POD no kubernetes começou a escalar descontroladamente. Isso pode significar que estamos recebendo milhões de requisições de robôs, vindo de vários lugares diferentes. A primeira solução seria colocarmos um Captcha e de fato é uma ótima solução. Porém essa alternativa não resolve tudo. Poderá resolver o problema de segurança mas já teríamos recebido as milhares de requisições e nosso sistema já estaria afetado. Imaginemos, então, um cenário mais complicado: se durante um evento de vendas ficássemos indisponíveis, geraria um transtorno imenso para empresa, por não ser possível realizar as vendas. Observe que nossa aplicação precisa ter camadas adicionais de segurança para continuar funcionando mesmo frente a esses tipos de ataques.

Robustez

Falamos em sistemas robustos quando a aplicação além de confiável está em uma estrutura robusta o suficiente para conseguir escalar, caso seja necessário. Inicialmente, é importante termos em mente as seguintes informações, primeiro a cloud não é infinita. E segundo, existem muitas empresas que rodam mais de 100 mil instâncias simultaneamente. Com essas informações, vamos imaginar que caiu uma região da AWS em que nossa aplicação estava, e essas máquinas precisam evacuar de uma região para outra para conseguir manter a disponibilidade. O problema é que em outra região não havia capacidade operacional de ter tantas máquinas no ar rodando. Então, mesmo que tenhamos uma AWS por trás do sistema, podemos ficar sem região. Nesse contexto, não conseguiremos fazer o deploy de todo nosso serviço.

Lembrando que não é incomum uma zona de disponibilidade ficar indisponível. Por exemplo, digamos que temos a região Norte Virgínia e nessa existam diversos datacenters que se comunicam rapidamente. Se um datacenter cair, logicamente vamos tentar jogar nossos recursos computacionais para outro datacenter. Mas devemos pensar no que pode acontecer se não tivermos a quantidade de IP's necessários para isso. Acredite grandes empresas que trabalham com uma enorme quantidade de máquinas podem sofrer com esse tipo de desafio. Ou seja, até a divisão das subnets precisam prever esses tipos de adversidades.

Escalabilidade

Neste tópico, vamos pensar um pouco no poder que nossa aplicação consegue escalar. Temos, basicamente, duas formas de escalabilidade: vertical e horizontal.

Vertical quando aumentamos os recursos computacionais da nossa máquina. E horizontal quando adicionamos mais máquinas.

É necessário que possamos garantir que nosso sistema ficará o mínimo escalável possível, principalmente de forma horizontal. Para isso é importante trabalhar de forma stateless, bem como seguir boas práticas no processo desenvolvimento como por exemplo o famoso “The Twelve-Factor App”.

Características Estruturais

No tópico anterior falamos um pouco sobre como fazer para que nossa aplicação seja operada mais facilmente, isto é, sobre as características operacionais.
Neste tópico, falaremos sobre as características estruturais, que estão mais relacionadas aos pontos de atenção que devemos ter no software para que este funcione de forma cada vez mais flexível.

As características estruturais estão ligadas ao processo de desenvolvimento de um sistema, ou seja, de como nossa aplicação será desenvolvida. E para facilitar nossos estudos, dividimos essas características em aspectos que consideramos

essenciais para o conhecimento de todos que estão envolvidos no processo de construção de um software. São eles: configurável, extensibilidade, fácil operação, reuso de componentes, internacionalização, fácil manutenção, portabilidade e fácil suporte.

Configurável

Apesar de parecer algo simples, muitas aplicações são de difícil configuração. Caso precisemos "setar" uma conexão com banco de dados, por exemplo, teremos que decidir se colocaremos nosso código de uma forma hard coded ou vamos trabalhar com variáveis de ambiente?

As respostas a esses questionamentos nos permite identificar se uma aplicação é configurável ou não. Dizemos que a resposta será positiva se não precisarmos fazer alterações no código fonte da solução para conseguir rodá-la em ambientes diferentes.

Por exemplo, sobre formas de pagamentos. Se temos uma gateway de pagamento como padrão, caso esta falhe é fácil fazer a alteração para uma outra?

Ao tentarmos subir uma solução, se precisarmos fazer uma só mudança do código fonte para que esta possa rodar em diferentes ambientes, já perceberemos que a aplicação não é configurável. Então, quando formos criar um software, por padrão, é necessário que pensemos nisso para que nossa aplicação seja cada vez mais configurável.

Extensibilidade

É fato que uma aplicação deve ser pensada para que consiga crescer. Ou seja, ela precisa crescer de certo modo que as coisas possam ser "plugadas" nela.

Por exemplo, vamos imaginar que iremos utilizar a gateway de pagamento "X" em nossa solução. Faremos, logicamente, a implementação dessa gateway em nosso sistema. Porém, imaginemos que nosso superior solicite, posteriormente, que façamos uma mudança para gateway "Y". Agora, obviamente precisaremos fazer uma nova implementação. Mas caso, durante essa nova implementação, precisemos mudar pontos estruturais de nossa aplicação para conseguir adicionar a nova gateway, é bem provável que tenhamos projetado nosso sistema de maneira errada.

Precisamos conseguir trabalhar com interfaces, adaptadores etc. para que possamos simplesmente adicionar coisas a nossa aplicação, de modo que não fiquemos reféns dos vendors que trabalhamos.

Nesse contexto, podemos falar sobre conceitos como o de camadas de anticorrupção, em que conseguimos separar nossa aplicação em camadas finas com mundo externo. Então, se precisarmos trocar de bancos de dados ou o Message Brocker, podemos apenas adicionar novos módulos, sem precisar mudar a base do que estamos criando.

Assim, nossa aplicação precisa ser extensível tanto nos lados de vendors, que vão

ser plugados, como também ao ponto de conseguirmos adicionar novos módulos nela. Caso seja difícil adicionar novos módulos, se precisarmos refatorar muita coisa sempre que tentarmos adicionar algo em nossa solução, é muito provável que precisemos rever essa estruturação.

Fácil instalação

Se todas as vezes que formos fazer o deploy, criar um ambiente de testes, um ambiente staging, um de produção ou criar um ambiente na máquina de algum companheiro de trabalho e isso for algo extremamente demorado, sabemos que poderá dificultar o nosso trabalho. Por esse motivo nossa aplicação precisa ser de fácil instalação.

Mas quais são os principais problemas que enfrentaremos para fazer a instalação de uma solução?

Em primeiro lugar, como dificuldade de instalação, temos a padronização do ambiente. No contexto atual, para que possamos padronizar um ambiente no qual sua aplicação vai rodar, a melhor alternativa é trabalhar com containers, Docker, etc, vai garantir que sua aplicação trabalhe até com o mesmo kernel que você escolheu na imagem.

Em segundo lugar, voltaremos a falar que a aplicação precisa ser de fato configurável, isto é, se for muito muito difícil de configurar seu sistema, obviamente, será dificultoso fazer a instalação.

Outro ponto que podemos abordar é que muitas vezes a aplicação tem dependências que são extremamente complexas para se trabalhar. Um exemplo disso é quando o sistema vai trabalhar com Elasticsearch, que é algo extremamente complexo, principalmente em nível de infraestrutura. Nesse caso, podemos pensar em alguns questionamentos: Como fazer para testar isso? Podemos trabalhar com docker e com Elasticsearch? Quando formos trabalhar em produção, como trabalhar com processos de configuração e conexão? Como vai ser a criação dos índices? Serão criados pela aplicação ou manualmente no servidor de produção?

Podemos pensar também sobre tópicos do Kafka. Precisamos ter em mente se nossa solução irá criar um tópico ou se esse tópico já vai vir criado.

Ainda sobre essa reflexão podemos nos lembrar do RabbitMQ e a criação de filas, criaremos a fila ou esta já vai vir?

Esses tipos de coisas são extremamente importantes quando pensamos em fazer uma instalação, principalmente se dependermos de itens de terceiros, inclusive banco de dados.

Reuso de componentes

Usar componentes para facilitar nosso dia a dia, pode mudar completamente a forma como iremos trabalhar. Porém devemos levar em consideração que existem alguns aspectos que não são tão simples de lidar.

Vamos imaginar, por exemplo, que temos um sistema monolítico. Sabemos

que uma grande vantagem desse sistema é que não existe latência de rede, isto é, não temos problemas de conexão, pois tudo está dentro de um mesmo sistema. E uma vez que estamos dentro de um mesmo sistema podemos ter frameworks e boas bibliotecas para facilitar nosso trabalho. Quando temos um mundo um pouco mais distribuído, com microsserviços e diversos sistemas, muitas vezes as equipes acabam criando soluções iguais para resolver o mesmo problema. Ou seja, digamos que a equipe "A" crie uma biblioteca de validação e a equipe "B" crie outra biblioteca de validação; eventualmente o que poderá acontecer é que teremos duas coisas para serem mantidas. Então, nesse momento poderíamos pensar na possibilidade de ter uma vertical dentro da empresa, onde adicionaremos todas as todas as bibliotecas que podem ser compartilhadas e criaremos times paralelos para manter essas bibliotecas, assim todos podem utilizá-las.

Internacionalização

Esse aspecto não é visto com tanta frequência no Brasil, pois não é comum que precisemos internacionalizar nossos softwares. Obviamente, pode surgir esse tipo de trabalho e, por isso, é importante que saibamos minimamente como desenvolver nossa solução possibilitando sua internacionalização.

Inicialmente, é importante termos em mente que a maior dificuldade desse aspecto não fica no Backend. Já que o Backend possui muitos frameworks, isto é, muitas formas maduras para trabalharmos com internacionalização. A

maior dificuldade pode ser outra. Por exemplo, na área do Frontend podemos desenhar um layout de certa forma, mas quando precisarmos trocar a linguagem do sistema, este poderá ficar totalmente desconfigurado e consequentemente as coisas não irão funcionar do modo que planejamos. Outra situação, é o fato da cultura de quem passará a usar, após internacionalização, ser totalmente diferente. Estes aspectos dificultam o manuseio do software principalmente pela tradução.

Ao criarmos um projeto, precisamos refletir se em algum momento este precisará se internacionalizar. Precisamos pensar quais pontos serão impactados diretamente nesse processo. Por exemplo, se vamos trabalhar com moedas, qual será a moeda base? Se for o real, o que acontecerá quando mudarmos para o dólar? Nesse caso, provavelmente mudaremos a gateway de pagamento. Podemos trabalhar então com a Paypal, pois esta empresa trabalha com transações internacionais. Outro ponto é pensar em como vai funcionar o processo de conversão. Caso não dê para parcelar, como vai funcionar a cobrança recorrente? Como será a política de definição de preço? O software vai fazer a conversão de forma automática ou teremos um local do sistema onde o administrador, em algum momento, vai setar o valor de cada moeda?

Então, podemos perceber que esse aspecto é extenso e demandaria muito tempo para abordarmos de forma mais detalhada. Porém, o que podemos destacar é que sempre que criarmos uma aplicação é importante pensarmos se existe a possibilidade de internacionalização. Caso a resposta seja sim, devemos começar

a levantar as principais possibilidades de falhas na solução ao precisarmos internacionalizar-lá.

Portabilidade

Para compreendermos a portabilidade, falaremos um pouco sobre mudanças no banco de dados. Nessa situação, surge o questionamento: é possível alterar o banco de dados sem impactar de uma forma muito grande o código da sua aplicação?

A verdade é que nunca vai ser tranquilo mudar o banco de dados sem impactar nossa solução. Mas podemos dizer que, tecnicamente, é possível mudar o banco de dados sem impactar de uma forma muito significativa o código de nossa solução. Por exemplo, se estivermos trabalhando com Elastic Stack e precisarmos mudar para New Relic ou para Datadog, devemos pensar se esta mudança será fácil para nosso sistema.

Outro questionamento, é se vale a pena trabalhar com open telemetry. Será que poderá facilitar nosso trabalho com vendors? Além disso, saber se está fácil mudar um gateway de pagamento também ajuda a pensar em portabilidade. Assim, pensando na portabilidade, podemos fazer com que os sistemas fiquem menos dependentes dos vendors.

Fácil suporte (logs e debugging)

É muito importante que consigamos entender onde estão acontecendo os problemas, caso eles aconteçam, em nossa solução. Precisamos que, no momento em que nossa aplicação estiver em operação, fique fácil entender exatamente o que está acontecendo para que seja possível solucionarmos as adversidades. Isso significa suportar a aplicação. E o suporte do sistema não é somente o call center. O suporte é realmente garantir que nosso software está rodando de forma aceitável e que conseguiremos ver rapidamente se está acontecendo alguma dificuldade, antes mesmo de nosso cliente ligar falando sobre a situação. Nesse ponto, precisamos pensar em como logar e em como conseguir criar diversas formas para "debugar". Além disso, trabalhar com observabilidade e centralizar logs, criar métricas; tudo isso faz parte da observabilidade.

Diante de tudo isso, podemos trazer algumas dicas: primeiro, foque em observabilidade e em padrão de geração de logs. Cada framework tem um padrão de logs, logo, é extremamente importante que tentemos consolidar os nossos logs em um único padrão e, dessa maneira, ficará muito mais fácil todo processo de operação.

Características Cross-Cutting

As características de Cross-cutting são aspectos que irão cruzar a aplicação de forma geral, ou seja, são coisas que precisamos sempre levar em consideração no dia a dia. Lembrando que pensar nos aspectos de uma solução de modo intencional traz, de todo modo, "sorte" ao desenvolvedor.

Acessibilidade

Precisamos estar cientes de que nossa aplicação poderá ser acessada por um grupo diversificado de pessoas, com necessidades de acessibilidade distintas.

Normalmente, quando falamos nesse aspecto acabamos mantendo nosso foco mais no Frontend. Isso acontece pois existem bibliotecas que podem nos ajudar a contornar muitas das complexidades técnicas no momento em que pretendemos deixar nossa solução mais acessível para que leitores de tela possam trabalhar.

Nesse contexto, mesmo havendo diversos padrões que podem nos ajudar, devemos focar sempre em um ponto: outras pessoas conseguem acessar facilmente nossa plataforma, isto é, pessoas com comorbidades, deficiência visual ou auditiva e etc. conseguem ter acesso a nossa aplicação?

Existe um movimento muito grande em torno da acessibilidade, e é muito importante adotarmos boas práticas quando formos desenvolver uma solução.

Processo de retenção e recuperação de dados

Ao construirmos um software, é importante termos em mente que os dados não devem ser ignorados. Podemos pensar que nossos storages são caros, e existem diversas formas de trabalharmos retenção de dados. Por exemplo, se estamos trabalhando com um sistema de stream de dados como o kafka, cada tópico que criamos, podemos especificar que aqueles dados ficarão disponíveis por um período de tempo predeterminado (por 7 dias, por exemplo). E depois desse período esses dados serão apagados.

Então se nos questionarmos sobre o que temos de dados hoje, realmente esses precisam existir a longo prazo? Caso a resposta seja positiva, como poderíamos guardar esses dados? E como mantê-los?

Hoje em dia, existem técnicas muito interessantes para trabalharmos com dados. Por exemplo, o que poderíamos fazer em caso de dados que precisamos com frequência? Esses dados, mais conhecidos como dados quentes, estarão ali em nosso banco de dados. Conforme tudo está rodando, vamos acessando e consultando essas informações. Já dados menos acessados, ou seja, dados frios, podem ser armazenados em um outro tipo de storage com um menor custo.

Autenticação e autorização

De um modo geral este tema parece algo simples, porém, se trabalhamos com arquitetura distribuída existem muitas formas possíveis de autenticar e autorizar

requisições. Por isso, de certo modo, torna-se algo mais complexo do que parece.

Por exemplo, ao trabalharmos com uma arquitetura distribuída normalmente teremos um identity provider (ex: Keycloak).

Outro ponto que merece nossa atenção é a possibilidade de utilizarmos API Gateway em nossa solução, um serviço que poderá fazer o processo de autenticação. Existem muitas empresas hoje em dia em que o sistema não tem mais validação de autenticação, justamente porque a autenticação acontece na API Gateway. Devido a isso sabemos que o usuário que está chegando naquele sistema já passou por uma autenticação.

Neste ponto, é importante conceituarmos API Gateway, para que possamos prosseguir em nossos estudos. Podemos dizer, resumidamente, que API Gateway é um mecanismo que fica mais na borda de nossa aplicação e quando os usuários acessam a solução caem primeiramente nessa borda. Nesse mecanismo é possível ter políticas de autenticação, políticas de timeout, políticas de quantidade de requisições etc. Ou seja, existem diversos plugins, de forma geral, que podemos trabalhar em uma API Gateway.

Por outro lado, quando trabalhamos com sistemas monolíticos esse processo de autenticação e autorização do usuário é algo mais simples. Pois é possível encontrarmos diversos frameworks que nos ajudarão a resolver desafios dessa natureza. Quando estamos no mundo distribuído isso realmente será um pouco mais complexo. Nesse segundo caso, é sempre importante pensarmos em um

servidor de identidade que nos ajude nestes aspectos.

Então, é importante pensarmos se vale a pena implementar autenticação em nossos microsserviços ou se o ideal é trabalhar com algum mecanismo de autenticação, que faça essa autenticação antes mesmo da requisição chegar em nossos microsserviços.

Legal

Anteriormente, falamos em dados. Sobre a importância de pensar onde e o tempo que nossos dados serão mantidos em nosso sistema.

É importante termos em mente, além disso, que tudo o que acontece em nossa aplicação precisa estar em conformidade com as leis do país onde ela está rodando, inclusive temas relacionados ao uso de dados. Isso deve ser pensado em todas as aplicações que formos desenvolver. Mas normalmente quando uma organização está muito ligada a esse ponto é comum que essa corporação transfira esses aspectos para nós desenvolvedores.

Privacidade

Ao criarmos um software é essencial que pensemos na perspectiva da LGPD (Lei Geral de Proteção de Dados Pessoais), para sabermos como minimizar problemas em relação a possibilidade de vazamento de dados dos usuários. Existem alguns pontos extremamente importantes que não podemos deixar de pensar e essa

é uma cultura que nós, enquanto desenvolvedores, devemos ter. Por exemplo, quando vamos testar uma aplicação é muito mais fácil testar se tivermos uma cópia do banco de dados em produção para validar as coisas. Neste momento, é importante sabermos que quando precisarmos de uma forma a mais para validar vazamento de dados, mesmo nomes e emails já são considerados dados.

Existem muitas "manobras" que as empresas estão fazendo para ajudar nessa questão. Por exemplo, todos os dados sensíveis de usuários estão sendo separados em outro banco de dados, ou seja, em outro nível de serviço; mas eventualmente podem ficar criptografados. Assim, o sistema principal só roda com dados muito básicos para que o usuário consiga carregar. A organização faz isso porque quando separamos a base de dados, podemos mitigar os riscos de vazamento. Essa manobra pode garantir a privacidade do usuário, em conformidade com as leis vigentes.

Nos dias atuais, falar em privacidade é algo muito crítico na maioria das organizações. É possível, e provável, que tenhamos clientes que nos façam assinar diversos contratos sobre políticas de privacidade. Isso ocorre pois a empresa responde legalmente pelos dados dos usuários.

Segurança

É essencial que a segurança de um sistema seja feita de "ponta a ponta". Então, podemos dizer que a primeira sugestão sobre esse aspecto é para pensarmos em segurança desde a borda da aplicação, ou seja, bem antes do usuário acessar o

servidor. Além disso, é interessante que trabalhemos com web firewall. Vamos, assim, criar regras e mecanismos que consigam identificar os robôs para ficar batendo em nossa aplicação. Dessa maneira, com o uso de web firewall já conseguiremos barrar tentativas de SQL injection, XSS e provavelmente as principais tentativas de exploração que podem ser encontradas na OWASP.

Outra sugestão muito importante nesse aspecto é que devemos usar preferencialmente padrões abertos em nosso software. Por exemplo, não é interessante tentarmos criar formas próprias de criptografia. Resumidamente, é melhor evitarmos criar qualquer coisa que envolva segurança. Ao invés disso, é sempre mais indicado que usemos um padrão aberto. Pois esses padrões foram criados por pessoas que se dedicaram durante muitos anos a pesquisas e especializações para criar boas práticas.

Outra prática que pode nos auxiliar é a de manter o backup em outras redes.

Usabilidade

É consenso entre a maioria dos devs que para usabilidade no Frontend existem diversas ferramentas e profissionais que podem nos ajudar a entender um pouco melhor a navegação do usuário. Existem ferramentas que mostram o usuário navegando, isto é, conseguimos ver o seu comportamento. Mas quando falamos em usabilidade, não podemos nos limitar ao Frontend.

Devemos pensar também no Backend. Assim, ao trabalharmos com API pre-

cisamos pensar: “Ela está organizada?”, “Como está organizada?”, “Tem documentação?”, “Como estamos documentando tudo?”, “É de fácil utilização?”, “Estamos trabalhando com padrões OpenAPI?”, “Possuímos um contrato claro da nossa API, que possa disponibilizar para outras pessoas?”, “Como estamos documentando?”, “Temos um README?”.

Além de tudo isso, é necessário que nos perguntemos também: quem é nosso cliente? Quem vai usar nossa aplicação? Vai ser outra aplicação? Como podemos desenvolver de modo a facilitar o trabalho com nossa aplicação? Como proporcionar a melhor experiência possível para o meu cliente?

Quem é de Frontend vai sim ter que pensar em muita coisa. Até dar nomes de eventos para conseguirmos mapear e trackear os principais eventos, uma vez que quando estamos trabalhando com algum APM (Application Performance Monitoring) todas essas informações são armazenas e sem dúvidas vão nos ajudar em eventuais comportamentos inesperados pelo lado do client.

Perspectivas para arquitetar software de qualidade

Neste tópico, falaremos sobre algumas perspectivas que nós, como pessoas desenvolvedoras, devemos ter para conseguirmos arquitetar um software de qualidade. Nesse caso, nós podemos citar três perspectivas básicas que nos

ajudam a perceber se nosso software está mais propenso a ter sucesso quando for ao ar.

Primeiramente vamos falar sobre a perspectiva relacionada à performance. Compreenderemos o que significa ser performático. Além disso, veremos quais métricas devemos usar para que consigamos ter performance em nossos sistemas no dia a dia.

A segunda perspectiva é relacionada à escalabilidade. Precisamos saber o que fazer para que o nosso software seja escalável, ou seja, para que ele consiga se manter estável conforme o número de requisições cresce.

A terceira e última perspectiva é em relação à resiliência. Esse ponto é extremamente importante para nós, porque sem isso não conseguiremos partir do pressuposto que o nosso software vai falhar. Todo software nasceu para falhar; nasceu para ter bugs e para ter problemas, inclusive onde nós não temos controle.

Performance

Este é um assunto que praticamente todas as pessoas desenvolvedoras gostam de falar. Porém, existem algumas dificuldades presentes nesse tema: muitas vezes as pessoas não conseguem medir e compreender o que realmente é a performance de uma solução.

Antes de tudo, é importante termos bem claro o seu conceito. Performance é o desempenho que um software possui para completar um determinado workload.

Conseguiremos, por meio desse aspecto, verificar o desempenho que o sistema está tendo para desempenhar uma ação, que é o seu papel no dia a dia.

Partindo do princípio que sabemos disso, obviamente precisamos de dados para conseguirmos avaliar a performance de nosso sistema. Lembrando sempre que não iremos verificar o desempenho do nosso software o comparando com a do "sistema vizinho". Devemos comparar a performance da nossa solução com ela mesma.

Métricas para medir a performance

Todas as vezes que quisermos avaliar um software, logicamente, teremos diversos aspectos para olhar. Mas existem dois aspectos que devemos nos atentar necessariamente, pois estes poderão definir realmente como analisar a performance de uma solução.
Sendo assim, as principais unidades de medida para avaliarmos a performance de um software são: latência e throughput.

A latência ou "response time" pode ser definida como o tempo de resposta que vamos receber. Por exemplo, sempre que vamos fazer uma requisição, o tempo até o software processar essa chamada e ele nos retornar o resultado é considerado a latência. É mais comum ouvirmos o termo "response time", mas saiba que tem o mesmo significado que latência.

O segundo indicador que devemos utilizar é o throughput. Este nos mostra o

quanto de requisição nosso software consegue suportar. Mais adiante, falaremos deste tópico com mais detalhes.

Antes de darmos continuidade, é importante termos em mente que ter um software performático é diferente de ter um software escalável. Normalmente, é comum misturarmos esses termos. Porém devemos ter claro que são aspectos diferentes. Ou seja, podemos ter um sistema performático que não é escalável ou vice-versa.

Como melhorar a performance do software

Para melhorarmos a performance do nosso software, o primeiro passo que devemos seguir é diminuir a latência. Ou seja, o "response time" precisa ser menor.

Precisamos ter em mente que nossa latência é afetada pelo tempo de processamento da aplicação, rede e chamadas externas. Isso é algo extremamente importante e não podemos deixar de levar em consideração. Por vezes ficamos otimizando nossa aplicação, sem considerar o tempo da chamada que o usuário faz até chegar em nossa aplicação. Quanto mais longe estiver do datacenter, quanto pior for a rede, consequentemente pior será a performance da nossa solução. Muitas vezes, nossa aplicação depende de chamadas externas para rodar. Vamos imaginar que o usuário coloque um CEP para que possamos trazer o seu endereço. Nessa situação, teremos que acessar a API dos Correios, mas se essa API estiver muito lenta naquele momento, isso irá afetar diretamente

a performance do nosso sistema. Muitas vezes, pessoas desenvolvedoras não conseguem perceber isso. Assim, o software pode até estar todo otimizado, mas a chamada externa está aumentando muito o “response time”. Normalmente isso acontece devido a falta de observabilidade para conseguir perceber que o obstáculo está em uma chamada externa.

Outro ponto que merece nossa atenção, ao falarmos de como melhorar a performance, é o aumento do throughput. Fazemos isso quando permitimos que nosso software consiga lidar com mais requisições. Por exemplo, se o nosso sistema está aguentando dez requisições simultaneamente, e conseguirmos fazer com que ele receba vinte, trinta… cem requisições de modo simultâneo, com certeza ele vai conseguir ser mais performático. Porque quanto mais requisições ele aguentar e quanto mais rápido ele deve retornar para o usuário final, mais performático nosso software vai ser.

O throughput de uma forma ou de outra está totalmente ligado à latência. Se nosso “response time” está demorando, muito provavelmente está tendo uma conexão presa em nossa aplicação. Quanto mais requisições estiverem presas em nosso software, raramente nossa aplicação vai conseguir lidar com mais conexões ao mesmo tempo, diminuindo assim o throughput.

Além disso, é essencial observarmos que se nossa aplicação não for boa, isto é, não for bem feita, ela também deixará de ser performática.

Principais razões para baixa performance

Se não soubermos para onde olhar, podemos acabar arriscando nas tentativas e erros para realizar o aumento da performance em nosso sistema. Neste tópico, queremos mostrar "o caminho das pedras" para que você saiba um pouco mais sobre os pontos onde normalmente podemos encontrar dificuldades em aumentar a performance, isto é, as principais possíveis razões para que a performance de uma solução esteja baixa.

Primeiramente, é necessário que falemos sobre o processamento ineficiente. Normalmente quando temos um sistema que está trabalhando de modo muito ineficiente, é bem possível que seus próprios algoritmos podem estar causando tais problemas.

Outro ponto importante a ser abordado é sobre recursos computacionais. É provável que se estivermos rodando em um hardware ruim, nossa performance seja menor. Então, podemos pensar no seguinte trade-off: Quanto maior nosso hardware, maior será nosso custo. Por outro lado, quanto menor for nosso hardware, menor será nosso custo, porém, menor será também nossa performance. Quando começarmos a falar sobre escalabilidade, veremos que precisamos conseguir alinhar esse aspecto pois, eventualmente, precisaremos adicionar mais poder computacional ao nosso sistema (mais adiante veremos duas formas de fazer isso).

O próximo ponto que iremos observar é algo que está, muitas vezes, totalmente

visível para os devs. Porém, algumas vezes pode não ser tão perceptível. Estamos falando sobre trabalhar de forma bloqueante. Ainda é muito comum nos dias de hoje ver linguagens de e abordagens de desenvolvimento que trabalham essencialmente de forma bloqueante, gerando claramente uma barreira para aumentar o throughput da aplicação.

Principais formas para aumentar a eficiência

Reconhecer situações que diminuam a performance da nossa aplicação é algo essencial para nós desenvolvedores. Sabendo como evitá-las, possivelmente conseguiremos aumentar a eficiência do nosso software.

Quando escalamos a capacidade computacional do nosso software, conseguimos perceber qual, e onde, é a "dor" que estamos enfrentando em determinado momento. Por exemplo, se o problema maior estiver na CPU especificamente, significa que o poder de processamento está gerando uma baixa performance.

O disco também pode trazer dificuldades para aumentar a eficiência da nossa aplicação. Caso seja necessário fazer muito I/O (input/output) em nossa aplicação, é provável que o disco esteja muito lento. Possivelmente, caso nosso acesso seja muito grande, nossa própria rede poderá não permitir a chegada de todas as requisições em nossa aplicação. Acredite, isso é muito comum quando trabalhamos com cloud. Dependendo do tipo de máquina que usarmos, veremos que a largura de banda muda completamente.

É essencial lembrarmos que não existe mágica no processo de construção de um software. Para melhorarmos de fato nossa eficiência, precisamos pensar na lógica por trás do software. Assim, é importante aperfeiçoarmos nosso algoritmo, nossas querys e o overhead dos frameworks.

Nós, como devs, temos que conseguir analisar cada um desses tópicos individualmente para sabermos onde está o problema. Não adiantaria fazermos o algoritmo mais performático, se a todo momento estamos fazendo I/O com um disco lento.

Para aumentar a eficiência do nosso software, podemos pensar um pouco sobre o acesso serial. É importante utilizarmos uma linguagem de programação que nos permita trabalhar com concorrência ou paralelismo. Isso, basicamente, vai nos permitir lidar com diversas coisas ao mesmo tempo, isto é, fazer coisas de formas diferentes, porém em conjunto. Precisamos muito disso nos dias de hoje, principalmente quando necessitarmos ter muita performance. A linguagem GO pode ser um bom exemplo. Para cada acesso em um webserver Go, uma nova thread é criada, então nesse momento é possível processarmos de forma simultânea diversas requisições, aumentando assim o throughput.

Grande parte dos softwares são otimizados e passam por diversos procedimentos durante seu desenvolvimento. Mas o obstáculo, na maioria das vezes, está no banco de dados. É importante que saibamos como modelar e usar banco de dados do modo de forma correta. Além disso, precisamos utilizar estratégias para buscarmos por performance intencionalmente. Por exemplo, pensar se o nosso banco de dados está com índice, fazer um “explain” em nossas querys para ver o

tempo de execução. Fora isso, precisamos ter ferramentas de APM que realmente nos mostram, no banco de dados, se nossa query está comprometendo a nossa aplicação.

Outro ponto importante é o uso de caching. É essencial sabermos que isso tem se tornado cada vez menos opcional se quisermos ter alta performance. Muitas vezes, durante o processo de desenvolvimento de uma solução, processamos algo apenas uma única vez e quando precisamos fazer a mesma consulta no banco de dados, ou processar o mesmo template, ou fazer o mesmo algoritmo é possível perceber que a resposta está pronta em um cache. Ele pode estar no disco ou na memória, porém, em um servidor separado de nossa aplicação.

Essas informações podem nos ajudar a desenvolver um software de qualidade, isto é, um sistema altamente performático. Por isso, é crucial que tenhamos familiaridade com esses aspectos. Além disso, devemos ter a boa prática de pensar neles de modo intencional durante todo o processo de desenvolvimento da nossa solução.

Capacidade computacional: Escala vertical vs horizontal

É comum precisarmos aumentar a capacidade computacional do nosso sistema para suportar mais requisições, ou seja, realizar o processo de escala vertical.

Por outro lado, temos a opção de escalar nossos sistemas de forma horizontal,

onde aumentamos também a capacidade computacional, porém, nesse caso, no número de máquinas.

Logo, podemos perceber que performance tem uma relação direta com escalabilidade. Pois a escalabilidade tem um ponto muito claro em relação a aumentar os recursos computacionais, fazendo assim com que a performance do sistema de forma geral seja ampliada.

Diferença entre concorrência e paralelismo

Podemos usar uma citação de Rob Pike para diferenciar esses dois termos: "*Concorrência é sobre lidar com muitas coisas ao mesmo tempo. Paralelismo é fazer muitas coisas ao mesmo tempo*".

Além dessa citação, alguns exemplos práticos podem nos ajudar a compreender o que é concorrência e paralelismo. Imagine que estamos mexendo no teclado de nosso computador, depois passamos a organizar nosso microfone e em seguida falamos com alguém, logo em seguida, após a ligação, ajustamos o teclado e depois o microfone novamente. Podemos perceber, nesta situação, que estamos realizando diversas tarefas, ou seja, um pouquinho por vez, mas diversas tarefas. Isso é denominado como concorrência.

Por outro lado, se estivéssemos gravando um vídeo e falando ao telefone ao mesmo tempo, isto é, fazendo atividades ao mesmo tempo, dizemos que se trata de paralelismo, ou seja, realizar tarefas de forma simultânea.

Exemplo: Vamos imaginar um Webserver

Vamos imaginar que temos um webserver e este tem um worker que trabalha da seguinte forma: recebe cinco requisições, e cada requisição demora 10ms de "response time". Então, se tivermos cinco requisições, demoraremos 50ms para conseguir realizar a tarefa. Nesta situação, temos um acesso serial. Podemos considerar esse processo bloqueante. Pois fará cada ação de uma vez, isto é, cinco requisições de forma serial.

Seria interessante se pudéssemos trabalhar esta mesma situação de forma concorrente ou paralela. Podemos ter cinco threads, ou seja, cinco fios de processamento. Isso significa dizer que teríamos um processo que conseguiria trabalhar essas cinco requisições em conjunto. Podemos atender as cinco requisições em 10ms, pois teremos cinco threads trabalhando de forma paralela.

Isso tudo faz muita diferença. E é importante sabermos que é extremamente comum que isso aconteça em webservers. Por exemplo, o Apache ao ser iniciado temos uma configuração de quantos workers iremos trabalhar. Vamos imaginar que ele tem cinco workers, isso quer dizer que ele só vai poder executar ali de forma paralela cinco requisições ao mesmo tempo. Se temos cinco requisições e nosso programa funciona de uma forma totalmente bloqueante, as outras requisições que forem chegando irão se sobrepor até que saia uma e a outra comece a chegar.

Logo, precisamos pensar em como aumentar essa quantidade de threads. Para

isso, podemos aumentar a quantidade de workers. O grande ponto é que para cada nova thread que chamamos em nosso sistema operacional, gastamos 1 mega. Então, se começarmos gastando 1 mega, somente para liberar uma thread, além da memória adicional utilizada para realizar a requisição, isso pode ser insustentável, fazendo realmente com que os recursos do servidor se esgotem rapidamente.

Se utilizarmos a linguagem Go, por exemplo, esse processo é feito de modo diferente. Essa linguagem abre realmente uma thread para cada chamada no webserver. Por isso ela consegue lidar com várias ao mesmo tempo. Por outro lado, ela trabalha com um esquema chamado de **green threads**, que é uma thread gerenciada pelo próprio runtime da linguagem e que ao invés de custar um mega, custa 2k. Então, ela consegue lidar com muito mais requisições ao mesmo tempo.

De uma forma ou de outra, sempre é importante tentarmos responder o máximo possível de requisições, de uma forma não bloqueante. Para isso, precisamos trabalhar pelo menos de modo concorrente (por esse motivo o node.js e a Sol PHP tem sucesso).

Cache

O cache nos possibilita acessar itens (arquivos, imagens, etc) que já foram processados e utilizá-los para trazer respostas, de maneira mais rápida, ao

usuário final.

Existe um tipo de cache chamado de “cache na borda”. Este fará com que o usuário não precise bater nem mesmo em nosso cloud provider. Para isso trabalhamos com algo chamado de Edge Computing. Quando usamos o Edge Computing, o usuário não bate em nossa máquina, isto é, em nosso kubernetes, etc; porque ele nos trará um cache totalmente processado na borda ou seja, em um servidor que fica antes do seu servidor principal.

A plataforma Full Cycle, por exemplo, trabalha com Edge Computing. Isso significa que trabalhamos com cache de toda nossa plataforma fazendo com que usuário que acessar nossa nosso frontend, antes mesmo de a requisição bater em nosso kubernetes, vai bater no serviço CloudFlare Worker. Desse modo, o browser do usuário fará o download de todo HTML, CSS, Javascript, imagens, etc, do local mais próximo ao usuário; com isso existe a real possibilidade de que os arquivos estejam sendo baixados de uma central telefônica do seu próprio bairro, por exemplo.

Para compreendermos melhor como funciona, é importante falarmos um pouco sobre os dados estáticos. É muito comum que queiramos cachear esses dados na borda. Podemos cachear imagens, css e outras coisas semelhantes a essas, pois, fazendo desse modo, não precisaremos servir esses tipos de coisas o tempo inteiro. Se não fizermos isso, a pessoa vai bater em nosso kubernetes e em nosso servidor web. Então, cachear é algo extremamente barato e fará com que o usuário tenha uma experiência melhor. Podemos pensar em algumas formas

extremamente efetivas para fazermos isso.

Existem muitos tipos de páginas que conseguimos cachear. Um exemplo é exatamente um dos tipos de cache que citamos anteriormente: o HTML, que pode ser cacheado na borda, o usuário somente acessa e pronto. Existem, ainda, alguns caches que conseguem processar toda aquela home do nosso site, da página de contatos até a parte de notícia. Podemos colocar um cache de 5 minutos, assim, todas as vezes que o usuário acessar nosso site, ao invés de processar várias coisas do background, nós já lançamos o HTML e pronto. Não gastamos todo um processamento, pois não bateu no banco de dados, simplesmente devolvemos uma página web.

Vamos imaginar que temos um algoritmo pesado. Esse algoritmo tem muitas variáveis que mudam a cada meia hora. Se todas as vezes que o usuário fizer uma requisição que vai chamar esse algoritmo nós precisarmos processá-lo do zero, será algo extremamente custoso. Então, para evitar isso, podemos cachear esse resultado pronto a cada meia hora.

Outro tipo de cache é o de objetos. Existem objetos que nossa solução terá que criar o tempo inteiro para gerar processamento de alguma forma. Aqui na Full Cycle nós temos um sistema de ORM que é chamado de Doctrine. Ele mapeia classes com a estrutura de banco de dados e, baseado nisso, conseguimos trabalhar com modelo de persistência. O problema é que a todo momento ele precisa fazer essa correlação entre o ID da classe e o ID da tabela do banco de dados. Esse parsing tem um custo e, por esse motivo, podemos cachear o

objeto que tem toda essa relação com banco de dados, pois a estrutura não muda com frequência. Isso quer dizer que sempre que formos trabalhar com o ORM, podemos evitar esse tipo de processamento.

É importante sabermos que os caches podem trabalhar de forma exclusiva ou compartilhada, e aqui vamos conhecer um pouco esses dois tipos de trabalho. É essencial termos esses conceitos claros para sabermos as diferenças entre esses dois modos.

Quando trabalhamos com cache exclusivo, geralmente será de forma local, em uma máquina específica, o que poderá resultar em baixa latência. Isso acontece pois tudo será processado localmente. Por exemplo: vamos imaginar que temos duas máquinas iguais chamadas de “A” e “B”, e os dois sistemas usam o Doctrine. Obviamente teremos uma baixíssima latência, mas o cache será exclusivo em cada máquina, outras máquinas não podem se beneficiar dele. Veja que com a duplicação conseguimos trabalhar com baixa latência. Por outro lado, poderemos ter problemas quando precisarmos trabalhar por sessão. Vamos pensar o seguinte: considerando que eu, Wesley, sou um usuário, e ao acessar um servidor fiz login, no momento que fiz isso os meus dados ficaram cacheados. Desse modo, não será necessário processar todas as informações a todo momento, pois teremos tudo naquele cache. Nessa situação, imaginemos que até a página home está personalizada com meu nome. Se eu acessar outra máquina que tenha meu software duplicado, esta sessão, por ser local, estará na máquina “A”, porém, não existirá na máquina “B”. Quando eu fizer o login,

precisarei cachear tudo novamente. Caso exista uma máquina “C”, terei que fazer o mesmo processo para cachear os dados. Sempre que precisarmos ter uma sessão de usuário, teremos que repetir esse processo, o que provavelmente nos prejudicará. Quando precisarmos que o resultado final para o usuário seja personalizado, se isso estiver espalhado em diversas máquinas, teremos esse tipo de problema.

Por outro lado, temos algo que chamamos de cache compartilhado. Este cache tem uma latência maior, pois trabalha com uma espécie de cache central. Ou seja, os dados estarão centralizados para o uso de todos que precisarem. Mas, por estarem centralizados, haverá uma latência maior. Para chegarmos nesse servidor de cache, apesar de ter uma latência maior, não há duplicação do cache. Vamos imaginar que temos duas máquinas chamadas de “1” e “2”. Pela máquina “1” acessamos um portal e a home deste site é cacheada. Quando acessarmos a máquina “2”, esta ainda não estará cacheada. Então, precisaremos cachear novamente essa home. Tivemos, assim, que fazer o cacheamento duas vezes. Se tivermos o cache compartilhado para 100 máquinas, essas 100 não precisarão gerar cache novamente, pois o cache já estará compartilhado entre todas. Nesse caso, percebemos a possibilidade de ter maior latência, porque precisamos fazer uma consulta externa, porém conseguimos utilizar esse cache em muitas máquinas. Essa é a grande vantagem: não há duplicação do cache. Poderemos, assim, compartilhar sessões, pois sempre que o usuário fizer login, os dados dele estarão no servidor de cache. Então, não importa qual máquina for acessar.

Ao trabalharmos com cache compartilhado, temos um banco de dados de forma externa, ou seja, todos acessam o mesmo banco de dados. E neste, podemos cachear os dados até ele. Podemos ter um cache com resultado dessas consultas em um Redis por exemplo. Esses dados ficam em memória, então temos um cache extremamente rápido.

Cache: Edge computing

Neste tópico veremos, por meio de alguns exemplos práticos, como o Edge Computing pode nos ajudar em relação ao cache.
A “falta” do Edge Computing nos próximos anos pode fazer com que a internet não funcione tão bem quanto esperado, por isso ele está em evidência nos dias atuais e a cada dia se fala mais sobre essa solução.

A Netflix pode ser um bom exemplo para que possamos compreender como o Edge funciona. Imaginemos a quantidade de acessos que a plataforma tem diariamente e a quantidade de tráfego que esses acessos geram. Se esses dados estivessem em um datacenter nos Estados Unidos e os seus usuários estivessem no Brasil, consequentemente seria necessário que o dado saísse dos EUA para bater no Brasil. Isso, provavelmente, faria a rede de internet congestionar, pois simplesmente teríamos uma sobrecarga em todas as máquinas da Netflix para conseguir movimentar esses terabytes de dados. Como resultado, essa sobrecarga se estenderia à internet de forma geral.

O Edge pode nos ajudar fazendo com que a informação do usuário esteja

mais perto. Assim evitamos que a sua requisição trafegue mais tempo pela internet. Fora isso, ele consegue fornecer serviços, além de simplesmente uma CDN (Content Delivery Network), que processam informações mais próximas possível do usuário, evitando assim que ele bata em nosso servidor.

Lembrando que a internet não é ilimitada, isto é, a rede não é ilimitada. Quanto mais pudermos evitar que o usuário fique longe da informação, será melhor em diversos sentidos, tanto para a rede, quanto para o próprio usuário.

Normalmente, arquivos estáticos podem ser colocados imediatamente no Edge por ser algo simples e barato. Em alguns casos pode ser inclusive gratuito como na CloudFlare - em até determinado limite. É importante destacar que não se trata de uma propaganda, mas sim de exemplos práticos utilizados na própria Full Cycle. Utilizamos Cloudflare para diversos serviços como: arquivos estáticos, CSS, imagens, HTML, etc.

Utilizando ainda o exemplo da Netflix, temos algo chamado de CDN (Content Delivery Network). Nesse sistema é criado uma malha de servidores espalhados no mundo. Quando nós, como usuários, subimos um vídeo da plataforma, o pegamos de um datacenter mais próximo possível.

Hoje, trabalhamos com CDN na Full Cycle. E, mais uma vez, para exemplo de algo que realmente utilizamos, podemos citar uma empresa que nos oferece esses serviços: a Akamai. Ela é uma das maiores empresas que trabalham com Edge Computing. Para termos uma ideia, a Akamai possui mais de 500 pontos

espalhados por todo Brasil. Então, inicialmente subimos nossos vídeos em um Bucket da Amazon S3 na Virgínia. Quando o usuário acessa esses vídeos pela primeira vez, a Akamai faz o processo de baixar esse vídeo e o joga em uma malha de servidores, baseado em contratos e parcerias com provedores de internet brasileira. Assim, é muito comum que, caso você esteja em São Paulo, seu vídeo esteja sendo baixado de um servidor de São Paulo mesmo. Se você estivesse em Portugal, seria baixado de um servidor em Portugal do mesmo modo. Dessa maneira, as coisas começam a funcionar de maneira mais fluída.

Logicamente, tudo tem um custo. Esse custo é dividido em duas partes: a primeira é custo da CDN e o outro é o de "transfer out" da S3 (como é no nosso caso).

Quanto maior o vídeo, quando mais longo e mais longe estiver, provavelmente teremos que fazer download o tempo inteiro. Isso fará com que nossa latência seja maior e a chance desse vídeo começar a travar em nosso computador, consequentemente, será maior. Quanto mais próximo esse vídeo estiver, menor vai ser a latência e teremos mais chances de conseguir baixar esse vídeo e gerar um cache local em nosso computador e assisti-lo de uma forma muito mais tranquila. A CDN permite isso. É um preço que precisamos pagar, mas que facilitará muito a experiência que teremos com nossos usuários.

Cloudflare Workers

Cloudflare é uma plataforma de Edge computing que começou fazendo proxy e cacheando informações de sites para carregamento mais rápido. Fora isso, também possui gerenciamento de DNS.

Hoje em dia, além dela trazer diversos serviços - inclusive de WAF (Web Application Firewall), também podemos encontrar na plataforma os Workers. Este é um serviço que permite que façamos deploy de aplicações. Normalmente essas aplicações são executadas em javascript. Eles conseguiram isolar cada requisição em um "container" utilizando a Engine V8, que é a mesma usada pelo Google Chrome e também no Node.js. Resumidamente, podemos dizer que a Cloudflare conseguiu criar um mini container que consegue executar de uma forma muito rápida as requisições de forma mais próxima do usuário.

A plataforma Full Cycle, atualmente, roda na Cloudflare Workers. Isso significa que, quando um usuário acessa a plataforma, o conteúdo é baixado mais próximo dele. Fazendo com que tudo rode de forma extremamente rápida, processando somente o javascript que está sendo chamado e cacheado do computador dessa pessoa. Toda vez que fazemos o deploy de uma nova versão, ela vai distribuí-las para todos os seus datacenters espalhados pelo mundo. Assim, vamos gastar menos banda na hora que chegar ao usuário final. Além disso, vale lembrar que o preço é extremamente acessível na maioria das vezes.

Escalabilidade

Em seu livro, Elemar Jr. nos traz a definição de escalabilidade. Ele nos diz que: *"É a capacidade de sistemas suportarem o aumento (ou redução) dos workloads, incrementando (ou reduzindo) o custo em menor ou igual proporção."* Ou seja, dizer que um software é escalável significa que temos o "poder" de aumentar ou diminuir o throughput, adicionando ou removendo a capacidade computacional. É essencial termos essa definição bem clara, pois é muito comum que exista uma mistura de conceitos, principalmente em relação a performance. Enquanto performance tem o foco em reduzir a latência e aumentar o throughput; a escalabilidade visa termos a possibilidade de aumentar ou diminuir o throughput, adicionando ou removendo a capacidade computacional.

Nesse sentido, é possível termos um software performático, diminuindo a latência e aumentando o throughput. Mas isso não significa que teremos necessariamente uma solução escalável. Assim, percebemos que apesar de existir certa ligação entre ser escalável e ser performático, existem muitas diferenças.

Escalando software: vertical vs horizontal

Existem duas maneiras de escalar um software: podemos fazer isso de modo vertical ou horizontal.

Quando aumentamos o poder computacional de uma máquina através de

seus os recursos computacionais como memória, CPU, disco, etc, temos o que chamamos de escala vertical. Por outro lado, se aumentarmos esses recursos através do aumento de máquinas em si, temos uma escala horizontal.

Por exemplo: ao invés de usarmos uma máquina de 64gb de ram, usamos quatro máquinas de 16gb cada. Então, nesse exemplo, aumentamos as máquinas para facilitar nosso dia a dia. Logicamente, será necessário colocar um proxy reverso ou um load balancer para rotear as requisições feitas por essas máquinas.

Conforme aumentamos os recursos computacionais, seja de modo vertical ou horizontal, teremos mais escalabilidade. Porém, é importante sabermos que algumas dificuldades podem surgir se trabalharmos com escalabilidade vertical. Isso porque a quantidade de limite em uma máquina poderá dificultar nosso trabalho. Chegará um momento em que uma máquina não terá todo poder computacional para resolver nossos problemas. Além disso, se eventualmente houver uma queda do sistema, todo nosso sistema cairá, pois está tudo concentrado em uma máquina. Em uma escala horizontal isso não acontece, porque teríamos várias máquinas. E se uma cair, logicamente nem todas ficarão fora do ar. Isso facilita a manutenção da solução no ar. Assim, hoje em dia, é mais comum que trabalhemos com escala horizontal. Porém, perceberemos que para que isso seja possível existem algumas práticas que teremos que fazer em nosso software.

Nos próximos tópicos, falaremos sobre os pontos que devemos observar para garantir que tenhamos uma solução escalável horizontalmente.

Escalando software: descentralização

Vimos que, de certo modo, é impossível permanecer escalando uma solução verticalmente. Por isso, é necessário sabermos o que fazer para permitir que nossa aplicação escale de modo horizontal. E, para que isto seja possível, precisamos ter atenção em alguns pontos relacionados a descentralização de dados, de estrutura, de arquitetura, etc. Pois o modo como nosso software foi desenvolvido afeta diretamente se conseguiremos ou não escalar horizontalmente.
Precisamos desenvolver nosso software de modo que, caso precisemos escalá-lo, possamos aumentar a quantidade de máquinas a qualquer momento. E, caso precisemos desescalá-lo, possamos remover essas máquinas. Então podemos perceber que as máquinas devem ser algo "descartável", ou seja, não devemos ter apego a uma máquina específica. Isso quer dizer que ela precisa ser facilmente criada e removida sempre que for necessário e sem medo algum. Precisa ser algo natural. Assim, devemos seguir alguns guidelines.

Primeiramente, precisamos partir do princípio de que nosso disco é efêmero. Isso significa que tudo que salvamos em disco, na nossa máquina, tem que poder ser apagado na hora sempre que for preciso. Como exemplo, vamos imaginar o wordpress. Este é um sistema que administra blogs, páginas, etc. Sabendo disso, pensemos na seguinte situação hipotética: criamos um artigo para um blog e fizemos um upload de imagens. Depois disso, fizemos outros uploads de vídeos, etc. Provavelmente, essas imagens ficaram salvas em nosso disco.

O problema, neste caso, é que se perdermos essa máquina, consequentemente perderemos todas as imagens deste blog. Isso é algo muito complexo, pois não teremos o poder de eliminar essa máquina quando quisermos. Além disso, se criarmos outra máquina rodando o wordpress para balancear a carga, essas imagens só estarão naquela primeira máquina, não estarão na segunda. Assim, se o usuário cair na outra máquina pelo Load Balancer, essas imagens não vão existir. Portanto, toda vez que subirmos nossa aplicação, precisamos partir do princípio que tudo que está no disco poderá ser perdido. Isso nos traz, de fato, uma mudança de paradigma. Podemos pensar, então, para que usar o disco? A resposta é: para gravar arquivos temporários e que irão auxiliar qualquer tipo de processamento. Mas de forma geral, é um disco efêmero. Uma solução possível para esse caso do WordPress seria fazermos o upload das imagens em um bucket da S3, por exemplo. Dessa maneira, todas as máquinas que acessem esse software não olharão para o próprio disco, mas sim para o bucket.

Outro ponto fundamental para conseguirmos fazer escala de modo horizontal é pensarmos no cache centralizado, ou seja, o cache não deve ficar em nossa própria máquina. Podemos nos lembrar, neste momento, do cache exclusivo e compartilhado. Neste caso, devemos ter um cache compartilhado. Então, se vamos fazer um cache de uma consulta em um banco de dados, esse cache não deverá mais ficar em nosso servidor. Ele deve ficar em um servidor externo, específico para cache, onde todas as máquinas que precisarem dele poderão acessá-lo. Lembrando que a ideia principal que devemos ter em mente é: tudo que está em nossa máquina poderá ser descartado para que essa máquina possa

ser destruída ou criada a qualquer momento.

Quando falamos em cache, no tópico anterior, abordamos um pouco a necessidade de se termos sessões centralizadas. O usuário não deve precisar fazer todo login novamente sempre que cair em um servidor diferente. Mas isso só é possível se as sessões estiverem centralizadas. Normalmente, fazemos isso através de um servidor específico para armazenamento de cache. Relembramos esse conceito, para afirmar que o nosso software deve ser efêmero, isto é, ele não deve armazenar estado. Todo estado que existir em nossa aplicação deverá ficar gravado de forma externa.

Então, podemos dizer que escalar um software significa descentralizar. Tirar tudo o que está focando em uma única máquina e jogar para um servidor externo. Fazendo assim, permitimos que essa máquina seja criada e removida sempre que quisermos, sem perder informações. Tudo com um único objetivo: poder aumentar nosso throughput.

Escalando banco de dados

Falar sobre banco de dados é sempre muito complexo, pois é algo que muitos de nós desenvolvedores temos dificuldade em trabalhar. Muitas vezes exige que tenhamos um arquiteto tecnológico - uma pessoa especialista que seja um DBA para nos auxiliar em alguns aspectos mais técnicos. Neste tópico, abordaremos o mínimo que precisamos saber sobre escala de banco de dados e alguns pontos fundamentais que precisamos levar em consideração sobre esse tema.

Primeiramente, devemos aumentar nossos recursos computacionais, pois quanto mais recursos temos, quanto mais disco, mais memória, mais CPU, consequentemente, mais escalamos. Porém, precisamos nos lembrar que escalar recursos computacionais, normalmente, tem um limite.
Por conta disso, precisamos começar a fazer segregações em nossos bancos de dados. Por exemplo: distribuir responsabilidades. Então, se um banco de dados está tendo muita escrita e leitura, podemos criar um banco específico para leitura e um para escrita. Isso ajudará muito a distribuir responsabilidades nesse ambiente. Normalmente, quando precisamos ter gargalos com banco de dados, começamos pensando em fazer coisas semelhantes a essa, isto é, começamos fazendo essas segregações.

Fora isso, podemos pensar também em como escalar nossa solução de forma horizontal desde seu início. Às vezes, a quantidade de leitura está tão grande que necessitamos, desde o início, adicionar várias máquinas de leitura ou, eventualmente, até mudar o formato do banco de dados. Começamos a trabalhar com diversos shards por exemplo.

Hoje em dia temos diversas opções de bancos de dados, por isso vale muito a pena compreendermos qual tipo de aplicação vamos trabalhar. Isso possibilitará uma escolha adequada, permitindo que possamos trabalhar bastante com determinado banco de dados. Por exemplo: um banco que nos possibilite relacionar e fazer consultas muito pesadas para gerar relações, um outro que nos auxilie no trabalho com grafos. Caso precise pegar dados que nos ajudem a evitar milhares

de consultas de muitos relacionamentos, temos o MongoDb para trabalharmos com documentos. Mas, se o caso for a necessidade de muita escrita, poderíamos pegar o Cassandra. De forma geral, é muito comum separarmos leitura e escrita para começarmos a criar máquinas e fazer essa escala de forma horizontal.

Hoje em dia muitas pessoas estão trabalhando de forma serverless, isso significa que elas estão trabalhando de modo que, basicamente, não se "preocupam" mais com o lado de servidores. Isto é, deixam seu cloud provider trabalhar por conta própria, porque esses sistemas, normalmente, trabalham e criam bancos de dados específicos para escalar de uma forma muito mais tranquila. Assim, podemos ler documentos, mas não nos preocupamos de modo geral. Vale dizer que, quando falamos em não nos preocuparmos, estamos nos referindo a esses desafios que são bem complicados de enfrentar. Lembrando também que serverless não significa apenas a lambda functions da AWS.

Muitas pessoas iniciam essa escala de qualquer forma, assim, deixam de olhar os principais gargalos. Para evitar isso, é importante termos uma APM (Application Performance Monitoring, pois com isso conseguiremos entender todas as queries que estão rodando. É comum ouvirmos que o banco de dados está lento, mas será que a pessoa está trabalhando com índice da forma correta? Ou está com medo do banco de dados ficar mais lento por causa do índice?

Fora isso, podemos começar a modificar alguns padrões na hora de desenvolver nosso software. Existe um padrão muito comum chamado CQRS (Command Query Responsibility Segregation) e, neste padrão, podemos separar o comando

que é uma intenção do usuário da query que é para fazer leitura de dados. Ou seja, separa a leitura da escrita. Falamos um pouco sobre isso em um tópico anteriormente.

Proxy Reverso

Ao longo dos nossos estudos sobre desenvolvimento de soluções, é importante termos algumas informações sobre o Proxy Reverso. Isso porque este é um recurso que eventualmente poderemos precisar.

O nome proxy, traduzido do inglês, significa 'procurador', isto é, uma pessoa que pode falar em nosso nome. Por este significado podemos construir a ideia de como o proxy "normal" funciona. Normalmente, as empresas que utilizam um proxy têm vários usuários e, geralmente, essas pessoas precisam frequentemente acessar diversos sites. O proxy poderá direcioná-la para os sites que elas desejam. Além disso, o recurso possui um filtro contra sites maliciosos. Resumidamente, ele pega a requisição e redireciona para o site que o usuário solicita.

Por outro lado, segundo o site da Cloudflare, proxy reverso é um servidor que fica na frente dos servidores de web e encaminha as solicitações do cliente (por exemplo, navegador web) para esses servidores. Ou seja, quando tentamos acessar um site, iremos bater num proxy que terá regras. Este irá nos encaminhar para um servidor que esteja configurado para conseguir responder essa requisição. Assim, precisamos entender que o proxy reverso é um servidor que fica na frente de todos os outros. Ele tem regras e por isso nos encaminha

corretamente para determinados servidores atrás dele. Por exemplo, vamos imaginar que estamos acessando um site chamado “a.com.br”. Este site está atrás de um proxy reverso. Então, quando o acessamos, o proxy reverso percebe isso e nos direciona para o servidor 1, 2 ou 3. Caso tentemos acessar “b.com.br”, ele fará outro direcionamento para o IP 2, 3 ou 4. Ou seja, ele vai fazer adequadamente os roteamentos. O proxy recebe todas as requisições, lê os dados, e, baseado neles, toma uma ação para redistribuir essas ações.

Existem, atualmente, três soluções em proxy reverso que podem ser consideradas mais populares e por isso, vale a pena conhecermos. O Nginx, o HAProxy (HA = High Availabillity) e o Traefik. Definitivamente, desses três, o mais conhecido é o Nginx. Então é interessante que saibamos configurá-lo, assim, conseguiremos dominar diversas ferramentas que são baseadas nele.

Introdução à resiliência

O conceito de resiliência pode nos ajudar a ter uma ideia inicial do que significa desenvolver uma solução capaz de se adaptar em diversas situações do nosso dia a dia. Podemos dizer que resiliência é um conjunto de estratégias adotadas intencionalmente para a adaptação de um sistema quando uma falha ocorre. Fora isso, existe uma frase popular que também nos ajuda a compreender melhor o conceito do que significa ter uma aplicação resiliente: “ou você dobra, ou você quebra”.

Quando estamos trabalhando em um software e acontece uma situação de erro, podemos ter duas respostas: primeiro, ele pode simplesmente explodir uma exception, danificando a requisição, ou ele pode perceber que tem algo errado e ter uma estratégia, um plano "B", para, ainda assim, atender a requisição do cliente, mesmo que de modo parcial. Então, podemos dizer que resiliência é esse poder que temos de nos adaptar quando algo inesperado acontece.

É importante sabermos que a resiliência precisa ser feita de modo intencional, porque, se não for feita assim, não saberemos qual será o comportamento da nossa aplicação quando algo der errado. E, hoje em dia, não podemos desenvolver um software jogando culpa em terceiros pelos seus "bugs". Por exemplo, se precisamos acessar um CEP para finalizar a requisição, mas o site está fora do ar, não é interessante que o usuário deixe de finalizar seu pedido por não termos esse dado. Devemos pensar em uma solução para conseguirmos criar um usuário mesmo que o site fornecedor do CEP não funcione. Além desse exemplo, temos também o da gateway de pagamento. Se o site estiver fora do ar, precisamos pensar em alternativas para não perder a venda pela falha nesse site, que deveria processar o pagamento. Precisamos adaptar nosso software para que ele consiga ter um plano "B", um plano "C", e até um plano "D" se for preciso, pois podemos ter absoluta certeza que nosso software vai falhar; ou por nossa culpa, que desenvolvemos errado, ou por culpa de soluções em volta dele, que farão com que ele dê erro. É essencial pensarmos o quão resiliente ele vai estar para entregar a melhor experiência possível ao cliente.

Assim, ter estratégias de resiliência nos possibilita minimizar os riscos de perda de dados e transações importantes para o negócio. Então, é essencial conhecermos algumas estratégias que nos possibilitem ter uma solução, de fato, resiliente.

Proteger e ser protegido

Não devemos criar planos de solução de maneira aleatória. Precisamos ter estratégias de resiliência que nos ajudem a resolver os problemas mais comuns que possam surgir em nossa aplicação.

Quando falamos em resiliência, temos que pensar, primeiramente, em proteger e ser protegidos. Isso porque, geralmente, nossa aplicação fará parte de um ecossistema. Nos dias atuais, é muito comum trabalharmos com sistemas distribuídos; por exemplo, os microsserviços, assim, temos vários sistemas que se comunicam entre si. Por esse motivo é importante pensarmos em duas coisas: proteger nossa aplicação e a do vizinho.

Um sistema em uma arquitetura distribuída precisa adotar mecanismos de autopreservação para garantir ao máximo sua operação com qualidade. Vamos imaginar que temos três sistemas… quando formos trabalhar com eles precisamos ao máximo contribuir para que esse ecossistema esteja saudável, que todos, mesmo aqueles sistemas que não fizemos, consigam operar da melhor maneira possível, pois eles nos afetam. Ou seja, precisamos preservar os demais sistemas para que quando precisarmos dele, ele nos responda. Mas, ao mesmo tempo,

temos que nos preservar para que quando eles precisem de nós consigamos responder também.

Então podemos dizer que um sistema não pode ser egoísta a ponto de realizar mais requisições em um outro sistema que está falhando. Se temos um sistema "A" e precisamos de uma informação do sistema "B", mas por algum motivo ele demora a nos responder e, depois disso, precisamos fazer outra pergunta e novamente ele demora, se ao invés de esperar um pouco mais, mandarmos a pergunta por várias vezes seguidas, esse sistema provavelmente sairá do ar. A consequência disso é que, com esse sistema fora do ar, todo ecossistema ficará comprometido. Se estava difícil termos nossa resposta, agora ficou mais difícil ainda. Por isso, um sistema não deve ser egoísta e enviar várias requisições seguidas assim. Em relação a tudo isso, o que queremos destacar aqui é a importância de existir harmonia entre os sistemas. Afinal, sempre vamos depender uns dos outros em algum momento.

Um sistema lento no ar, muitas vezes, é pior do que um sistema fora do ar, pois isso gera algo que chamamos de efeito dominó. Imaginemos a seguinte situação: chamamos o sistema "A", que chama o sistema "B", que chama o sistema "C". Por algum motivo, o sistema "C" está lento, por isso o sistema "B" ficará travado esperando sua resposta. Sabemos que nosso sistema "A" está dependendo do sistema "B". E quanto mais requisições chegam, terá um momento que o sistema "B" não irá aguentar mais recebê-las por causa do "C". E, nessa situação, o "A" também poderá travar. No final, isso pode fazer com que todos os sistemas caiam.

Assim, por vezes, seria melhor que o sistema “C” estivesse fora do ar. É melhor dizer que não está aguentando mais lidar com tantas requisições do que não retornar as respostas.

Resumidamente, quando falamos em se proteger e ser protegido, é melhor utilizarmos táticas de admitir não estar aguentando mais requisições e retorná-las para todos do que ficar impedindo a fila de progredir. Pois, dessa maneira, todos saberão que estamos com problemas. Essa, por incrível que pareça, é uma forma de proteção. E também parametrizar o que faremos em nosso sistema caso percebamos que a solução que estamos tentando nos comunicar está cada vez mais lenta.

Podemos perceber que essa dificuldade não está relacionada a programação. Não é sobre ser um bom programador de java ou de .net. O que estamos querendo repassar são conceitos que nos farão trabalhar com excelência, por exemplo, em um mundo distribuído.

Health Check

O trabalho com health check nos possibilita saber como está a saúde do nosso software. Assim, conseguimos responder aos outros sistemas se temos ou não condições de receber mais requisições. Essa é uma forma de fazer uma checagem de saúde em nossa solução. Por ser um termo muito conhecido, é bem provável que muitos de nós já tenhamos ouvido falar no trabalho com uso das regras de health check. Apesar disso, é importante sempre vermos/revermos as

possibilidades de verificação dos sinais vitais da nossa aplicação. Sem isso não é possível saber como está a saúde de um sistema.

Então, se estamos falando sobre proteger e ser protegido, é essencial que possamos verificar se podemos mandar ou receber mais requisições. Precisamos de algum mecanismo que, de tempos em tempos, faça essa checagem. Por exemplo: se nosso software demorar 5 segundos para responder uma requisição que deveria ser respondida em 500 milissegundos, pode significar que algo está errado. Outro exemplo de um possível bug no sistema seria não conseguirmos acessar o banco de dados.

Baseado nas informações do health check, saberemos se o sistema está saudável o suficiente para receber mais requisições. Caso a resposta seja negativa, teremos que pensar em alguma estratégia de adaptação para nosso software. Um exemplo seria retornar um "erro 500" até que ele se restabeleça. Todos que baterem em nossa solução não perderão tempo, pois o erro estará sinalizado na tela, com isso, ele pode procurar alguma forma criativa para lidar com nossa indisponibilidade ao invés de continuar mandando mais requisições para quem não tem condições de responder.

Um sistema que não está saudável possui uma chance de se recuperar caso o tráfego pare de ser direcionado a ele temporariamente. Vamos imaginar que temos um sistema que tem muito tráfego, quando ele tenta fazer uma consulta ao banco de dados, tem um retorno muito lento. Depois disso, acaba travando. Isso fez com que ele começasse a sobrepor várias requisições. Ele não consegue

mais lidar com todas essas requisições, por isso continua muito lento, até que em certo momento passa a dar timeout. Durante esse processo as requisições continuavam chegando, o que prejudicava ainda mais o funcionamento desse software. Por outro lado, vamos imaginar que os outros sistemas pararam de mandar requisições assim que ele começou a ficar lento. Provavelmente, ele iria pegar todas as requisições travadas, em algumas daria timeout e as que restassem poderia começar a processá-las até ficar 100% novamente. Quando isso acontecesse, passaria a receber novas requisições.

É essencial que tenhamos um health check de qualidade. Quando temos os dados dos sinais vitais corretos, conseguimos verificar se o sistema não está saudável. E, dependendo do sistema, existe a possibilidade dele se recuperar. Chamamos isso de self-healing, ou seja, autocura.

Muitos trabalham por padrão com health check da seguinte forma: o dev coloca um "health" e a cada 10 segundos manda um ping acessar aquela URL. Caso esta retorne, o dev chega à conclusão que o software está retornando com qualidade. O problema é que existe uma diferença muito grande entre acessarmos uma URL que retorna somente o HTML, de acessarmos uma que retorna uma URL que pega a média do tempo das últimas requisições e faz uma consulta no banco de dados. Pois toda vez que criamos uma URL para verificar a saúde do nosso sistema, essa saúde não pode ser medida somente pelo arquivo de HTML, já que se tiver um Nginex na frente, sempre terá um retorno incorreto, porque é muito difícil que o Nginex caia. Assim, é importante criarmos esse arquivo de forma

estratégica, incluindo os dados que serão acessados.

Rate limiting

Rate limiting é uma estratégia que protege o sistema de acordo com o que ele foi projetado para suportar. Também é um ponto que se relaciona diretamente com a resiliência da aplicação. Normalmente quando subimos um sistema no ar temos uma ideia de quanto de requisições ele pode aguentar. Caso não tenhamos, é recomendável fazermos um teste de stress. Além disso, podemos ver na empresa quanto de orçamento em relação a quantidade de máquinas está liberado para nossa solução. Mas é importante sabermos esse limite, pois senão teremos complicações em nosso trabalho. Então, é essencial buscarmos essas informações antecipadamente, antes mesmo do problema acontecer. Ao sabermos esse limite, podemos, então, trabalhar com rate limiting. Assim, se o sistema consegue responder 100 requisições por segundo, essa será a regra, esse será o número que vamos trabalhar na estratégia.

No rate limiting podemos dizer que determinado sistema vai aguentar 100 requisições por segundo, passando disso retornará um "erro 500", por exemplo. Então o sistema poderá trafegar com qualidade, ou um nível de qualidade mínima, até o ponto determinado, pois acima desse ponto começará a atrapalhar os outros sistemas.

Para compreendermos como tudo isso funciona, vamos imaginar uma situação hipotética: temos um cliente que utiliza nossa API e esse cliente faz em média

50 requisições por segundo, ou seja, ele tem metade do que normalmente conseguimos prover. Nesse contexto, temos diversos outros sistemas que não possuem tanta prioridade. Por serem sistemas periféricos, não trazem um nível de criticidade ao negócio caso estejam fora do ar. Para completar nosso raciocínio, vamos pensar agora em dois personagens: Pedrinho e Zezinho e os dois são de outro squad. Digamos que Pedrinho tem um sistema que não é tão crítico para a organização quanto o de Zezinho. Ao sair, Pedrinho esqueceu um loop ligado, então esse sistema, que não é tão importante, vai começar a fazer pelo menos 100 requisições em nossa solução. Desse modo, aquele sistema do Zezinho, que é bem mais importante, não vai conseguir acessar porque tem uma rate limiting permitindo somente 100 requisições por segundo. Assim, estaremos dando prioridade a um site que nem é tão crítico para o negócio, deixando de fora um sistema que tem um nível de criticidade muito mais alto para a corporação. Então, para evitarmos isso, precisamos gerar preferências por clientes quando trabalhamos com rate limiting. Podemos programar da seguinte forma: Zezinho tem pelo menos 60 requisições por segundo e o Pedrinho fica com as outras 40. Dessa forma, por mais que outras aplicações tentem "bombardear" nosso sistema, não conseguirão passar do limite e nos afetar. Fora isso, o sistema que realmente precisa utilizar o nosso, vai acessar de forma prioritária.

Então, quando se trata de rate limiting, é importante também determinarmos um limite programado por tipo de cliente. Ou seja, ajustar as preferências e as prioridades que queremos trabalhar. Assim, evitamos que sistemas críticos, que dependem de nós, não fiquem sem acesso.

Circuit breaker

Através do circuit breaker as requisições feitas para um sistema podem ser negadas, por exemplo por meio da devolução instantânea de um “erro 500” para o cliente. Dessa forma conseguimos proteger a nossa aplicação.

Mas antes de explicarmos como isso funciona, é importante apresentarmos três formatos de circuitos possíveis: o circuito fechado, o circuito aberto e o circuito meio aberto. O primeiro acontece quando as requisições estão chegando normalmente, isto é, mandamos as requisições e elas chegam de modo “normal” porque esse circuito está fechado. No segundo temos o contrário disso; as requisições não chegam à aplicação pois passamos instantaneamente um “erro” para o cliente. Então, com o circuito aberto, os que tentam acessar nossa solução deverão pensar em estratégias para que suas aplicações sejam resilientes. Já o terceiro, podemos considerar como um meio termo. Neste caso, é permitida uma quantidade limitada de requisições para que seja feita uma verificação da possibilidade de recebimento destas pelo sistema. É um teste que permite a entrada de certa quantidade de requisições por um determinado tempo. Assim, é possível saber se o sistema tem condições ou não de voltar ao ar integralmente.

Para compreendermos melhor como isso tudo funciona, vamos imaginar que temos um circuito elétrico com um disjuntor em nossa casa e este disjuntor vai servir para abrir o circuito caso venha uma sobrecarga. Assim, ao invés da corrente elétrica continuar passando - o que poderá queimar nossos eletrodo-

mésticos -, ele abrirá o circuito, fazendo com que essa corrente pare ali.

O circuit breaker funciona de forma semelhante a um disjuntor nos sistemas. Quando o circuito está fechado, significa que uma solução se comunica normalmente com a outra. Mas digamos que exista uma aplicação chamada de "A" que se comunica com outra chamada de "B". A primeira começou a ter dificuldade para responder a segunda. Assim, a solução "A" passou a amontoar as requisições, gerando aquele efeito dominó que comentamos anteriormente. Nesta situação, o circuit breaker vai abrir o circuito e a solução "B" receberá um "erro 500" até que a solução "A" possa começar a se recuperar. Somente após a recuperação o circuito será fechado novamente.

Com uso de códigos é possível implementarmos o circuit breaker em nosso próprio sistema, inclusive existem bibliotecas que fazem isso. Conseguimos, dessa maneira, segurar nossa aplicação e a dos outros. Porém, é importante sabermos que existem recursos mais modernos hoje em dia, um exemplo é quando estamos trabalhando com service mesh. Nesse caso, o circuit breaker é aplicado diretamente na rede, assim o desenvolvedor não precisa se preocupar com isso.

API Gateway

Uma API Gateway funciona, basicamente, como uma porta de entrada em nosso sistema. Ela centraliza o recebimento de todas as requisições que estão

acontecendo em nossa aplicação.Ou seja, ela pode aplicar, logo na entrada, regras, políticas, plugins etc. Assim, consegue perceber as necessidades individuais de cada serviço. Baseado nisso, pode rejeitar, ou tomar diversas decisões que favoreçam as aplicações.

Quando falamos em resiliência, esta solução nos ajuda a evitar uma série de situações que poderiam prejudicar nossa aplicação. Por exemplo, imagine que temos um sistema em que o usuário precisa ser autenticado para acessá-la. Digamos que alguém crie um robô para "bater" nessa aplicação repetidamente, nosso servidor tentará fazer a autenticação das requisições feitas por ele. Ou seja, ao detectar usuário e senhas diferentes, nosso software vai retornar um "não". Isso, feito várias vezes, fará com que esse serviço processe inúmeros pedidos, o que provavelmente prejudicará seu funcionamento. Em uma situação dessas, a API Gateway tem condições de validar da seguinte forma: se alguém está batendo em nossa máquina precisa fornecer um token JWT para ser autenticado, por exemplo. Assim, se ele não conseguir ser validado logo na API, não passará nem desta portaria. É como se morássemos em um condomínio fechado. Então, se alguém quiser bater a campainha da nossa porta, ele terá que passar primeiro pela portaria. De modo semelhante, o usuário terá que passar pela API Gateway caso queira acessar nossa aplicação.

Tem sido cada vez mais comum ver empresas utilizando API Gateway. Pois as grandes APIs do mercado tem recursos e plugins que auxiliam muito no dia a dia das aplicações. Um bom exemplo de uma API famosa é a Kong. Ela pode

ser usada tanto standalone, isto é, na frente de diversas aplicações, como em um under control em um Kubernetes. É interessante dizermos também que a Kong usa, por trás dos panos, o próprio Nginex.

Com os diversos plugins que a API Gateway nos oferece, podemos trabalhar com rate limiting e health check. Ou seja, conseguimos dizer para a URL receber até 100 requisições por segundo, sendo que seriam reservadas 50 requisições para os usuários autenticados e os restantes dos usuários ficariam com as outras 50. Assim, conseguimos trabalhar com limites e prioridades. Além disso, é possível fazermos a verificação da saúde de forma ativa para que possamos perceber claramente se aquela aplicação está saudável. Isto é, a própria API Gateway faz o apontamento da saúde da aplicação e retorna um "erro" para quem estiver chegando, caso seja necessário. Por outro lado, se a solução estiver saudável, coloca o usuário dentro do sistema.

Então, podemos perceber que recursos como esses podem facilitar nosso dia a dia. A API Gateway tem tantas funcionalidades que é necessário tomarmos cuidado para que ela não aplique, involuntariamente, regras de negócio. Por exemplo, podemos colocar um plugin para que todas as vezes que recebermos um XML, este seja transformado em um JSON, caso nosso programa não consiga trabalhar com esse XML. Ou, eventualmente, se tivermos uma Lambda Function e queremos que, quando o usuário acesse a "*axpto.com.br/produtos*", seja executada uma Lambda na AWS. Então, a API Gateway consegue fazer esse tipo de tarefa.

Assim, percebemos que essa solução nos oferece diversas funcionalidades, de fato, que contribuem para o bom desenvolvimento dos sistemas. Pois ela consegue trabalhar desde autenticação, autorização, rate limiting, transformações, adição/remoção dos headers, até trabalhar com logs. Mas o que queremos destacar, neste tópico, é que a API Gateway pode nos ajudar a colocar diversas estratégias que auxiliam no processo de resiliência dos nossos softwares.

Service mesh

De modo geral, a service mesh, ou malha de serviços, tem um conjunto de recursos que nos ajudam a controlar o tráfego de rede. Por isso, tem sido muito usada pelas organizações. Atualmente, temos diversas mesh no mercado, mas podemos citar a Istio como um exemplo. Neste tópico, nos dedicaremos a fazer uma relação entre esta solução e a resiliência dos sistemas.

O trabalho com service mesh é feito, basicamente, colocando proxies do lado de cada sistema. Uma vez feito isso, sempre que um serviço se comunicar com outro, isso não será feito diretamente. Por exemplo, em uma situação hipotética, estamos usando uma mesh e temos uma conversa entre o sistema “A” e o “B”. Logicamente, eles pensam que estão tendo essa comunicação diretamente, mas na verdade a solução “A” enviou uma mensagem para um proxy, que chamamos de sidecar. E é esse sidecar que enviará a requisição para o sistema “B”. Caso o serviço “B” também tenha um proxy, este receberá a requisição e depois mandará para o sistema “B”. Então, isso significa que quando trabalhamos com service

mesh toda comunicação de rede é efetuada via proxy. Assim, tudo que estamos passando na rede consegue ser controlado e medido. Conseguimos pegar os dados, saber quem manda/recebe as informações, como e o quanto de tempo essa informação é processada, etc. É extremamente interessante saber isso tudo, pois assim conseguimos entender o comportamento da nossa rede e controlá-la. Ao fazermos isso, podemos dominar tudo o que está acontecendo.

Ainda sobre comunicação entre os sistemas, vamos imaginar algumas ações que precisaremos fazer em nossa aplicação em algum momento. Primeiro vamos nos lembrar do rate limiting. Como poderíamos fazer sua implantação? A resposta é que teríamos que instalar uma biblioteca ou criar uma implementação em nossa aplicação. E, nessa aplicação, guardaremos, no banco de dados, quantas requisições estamos recebendo por segundo. Além disso, teríamos que separar essas requisições por cliente, para ver quando seria necessário negá-las. Vamos nos lembrar também de quando fazemos uma requisição e o sistema não nos retorna. Nesta situação, precisamos fazer um processo chamado de retry, ou seja, tentar novamente para verificar se aquele sistema estava fora do ar. Uma alternativa de como fazer isso seria realizarmos algumas tentativas em nossa biblioteca, uma vez, duas vezes, três vezes… ou quantas vezes forem necessárias. Outra situação para pensarmos é na implementação do circuit breaker em nosso projeto. Primeiro, seria necessário medir mais ou menos a saúde da nossa aplicação. Depois, quantas requests ela está recebendo por segundo. A partir dessas informações podemos começar a negar, ou seja, abrir o circuito. Com esses apontamentos, podemos perceber que existem diversos comportamentos que

são relacionados à comunicação da aplicação. E nós, normalmente, acabamos fazendo isso no código. Sobre tudo isso, podemos dizer que o mais complexo é projetar em qual propriedade nós, desenvolvedores, podemos codificar.

A service mesh provê uma forma de conseguirmos olhar tudo o que está acontecendo de comunicação entre seus sistemas. Além disso, possibilita aplicar alguns comportamentos como rate limiting e circuit breaker direto na rede, porque teremos acesso aos proxies. Então, se quisermos fazer um circuit breaker antes da requisição bater em nossa solução, com uma mesh ela baterá em nosso proxy. O nosso proxy saberá que estamos ruins, assim, abrirá o circuito, impedindo que a requisição bata em nossa aplicação.

Sabemos que atualmente todas as comunicações que acontecem entre os nossos sistemas devem ser criptografadas, pois sempre é possível que tenhamos alguém em nossa rede tentando interceptar essas mensagens, inclusive chamamos isso de ataque man-in-the- middle. Através de uma service mesh podemos trabalhar com uma Mutual TLS (mTLS). Assim, quando um sistema quer se comunicar com o outro nós criamos uma relação criptografada entre eles. Por exemplo, temos o serviço 1 e o serviço 2 tentando se comunicar. E o 2, ao ser chamado pelo 1, quer saber se realmente esse sistema é quem diz ser. Ao trabalhar com mTLS, por meio dos proxies, o serviço 1 consegue mostrar uma chave de criptografia para que o serviço 2 tenha certeza de que pode trafegar as informações. Fazer tudo isso manualmente em uma rede é algo insano, pois teríamos, além de outras coisas, que controlar a geração de certificados. Em sistemas realmente

grandes seria uma tarefa muito complicada, porque trabalhamos com milhares de microsserviços. Com uma service mesh, conseguimos isso imediatamente e com poucas configurações. Então, é essencial considerarmos a importância de conhecer ao menos o básico sobre essa solução. É importante dizermos que nosso objetivo, neste curso, não é infraestrutura, por isso nossa finalidade não é fazer com que os devs compreendam tudo sobre rede. Por outro lado, é essencial que todos nós saibamos que existem tecnologias e soluções que podem suprir, muitas vezes, papéis que em tese nós pensávamos ser da pessoa desenvolvedora, mas que com os recursos que temos hoje não precisa mais ser.

A service mesh nos possibilita trabalhar de forma automática com circuit breaker, com políticas de rate limiting, de retry, de timeout, de fault injection etc. Assim, saberemos qual será o comportamento se metade da rede cair, se é possível fazer testes entre outros. Isso só é possível pelo controle que uma mesh nos proporciona.

Trabalhar de forma assíncrona

O trabalho de forma assíncrona é extremamente antigo, ou seja, as pessoas usam essa forma de trabalhar há bastante tempo. Por exemplo, vamos nos lembrar de quando precisamos fazer um pagamento em um banco ou supermercado. Imagine que, na maioria das vezes, não tem uma quantidade suficiente de caixas para atender os clientes. Logo, é necessário organizar essas pessoas em filas. Pois, já que eles não conseguem absorver a quantidade de requisições que estão

chegando, é necessário que esses clientes fiquem em espera para fazer o seu pagamento. Então, podemos perceber que entrar em uma fila para aguardar uma solução é algo que já fazemos em diversas situações de nossas vidas, em diversas ocasiões não temos nossos problemas resolvidos instantaneamente. Porém, quando estamos trabalhando em sistemas, nós acabamos não pensando dessa forma.

Caso uma aplicação esteja recebendo 100 requisições por minuto, quando aguentaria somente 50 requisições nesse tempo, essas que estão sendo enviadas a mais ficarão travadas e, eventualmente, perderemos uma ou outra requisição. Isso não faz sentido, porque aquela pessoa que nos enviou a requisição pode não estar esperando a resposta exatamente naquela hora. Mesmo que ela tenha enviado logo a requisição, provavelmente poderia esperar. Porém, nós só estamos dando uma opção, ou seja, se alguém quer nos mandar uma requisição vamos responder imediatamente e, como não conseguimos fazer isso, preferimos perder a requisição.

Quando falamos em trabalhar de forma assíncrona, estamos falando em fazer exatamente como os bancos e supermercados: quem quiser nos mandar mensagem, poderá ficar aguardando na fila. Quando chegar a sua vez iremos resolver os seus problemas. Trabalhar desse modo evita perder dados, porque com menos recursos computacionais conseguiremos dar vazão a mais requisições do que poderíamos dar se estivermos sobrecarregados. Isso acontece porque não precisamos entregar as respostas dessas solicitações imediatamente. E, como

conseguimos dar conta de mais requisições, consequentemente não perdemos dados.

Além disso, não há perda de dados no envio de uma transação se o servidor estiver fora do ar. Vamos imaginar que temos um sistema e este estava "rebootando". Isso acontecia quando alguém nos mandava solicitações de pagamento. Isto é, por algum momento teve uma falha, estava usando máquina virtual e subindo. Assim, a pessoa que enviou a requisição percebeu que não havia como fazer o pagamento, logo, deveria tentar em outro momento. Nesta situação seria válido trabalharmos de forma assíncrona. Se esperamos ter a resposta naquela hora, mas não dá, aquele sistema que está "rebootando" perderá os dados. Porém, se trabalharmos de modo assíncrono, permitiremos opções diferentes. Nossa aplicação não precisará, necessariamente, processar tudo o que enviam para ela de modo instantâneo. Então na hora que um usuário enviar uma mensagem, ao invés dessa mensagem ir direto para nosso sistema, ela é enviada para um intermediário que irá armazenar somente a informação. Quando o sistema que está sendo requisitado estiver pronto, ele vai ler desse sistema do meio, para depois processar a informação. Ou seja, o servidor pode processar a transação em seu tempo quando estiver on-line, assim os dados não são perdidos pois estarão no intermediário, que normalmente é chamado de message broker.

Message broker é um sistema específico para receber requisições. Ele funciona da seguinte forma: recebe e guarda a solicitação, sem lê-la ou processá-la. Depois, quando a aplicação estiver disponível, ele entrega essa requisição para que seja

processada. Trabalhando assim, nossa solução consegue receber mais requisições do que poderia, pois essas solicitações são processadas aos poucos. Porém, o mais importante de tudo isso é que mantemos a resiliência porque não perdemos os dados.

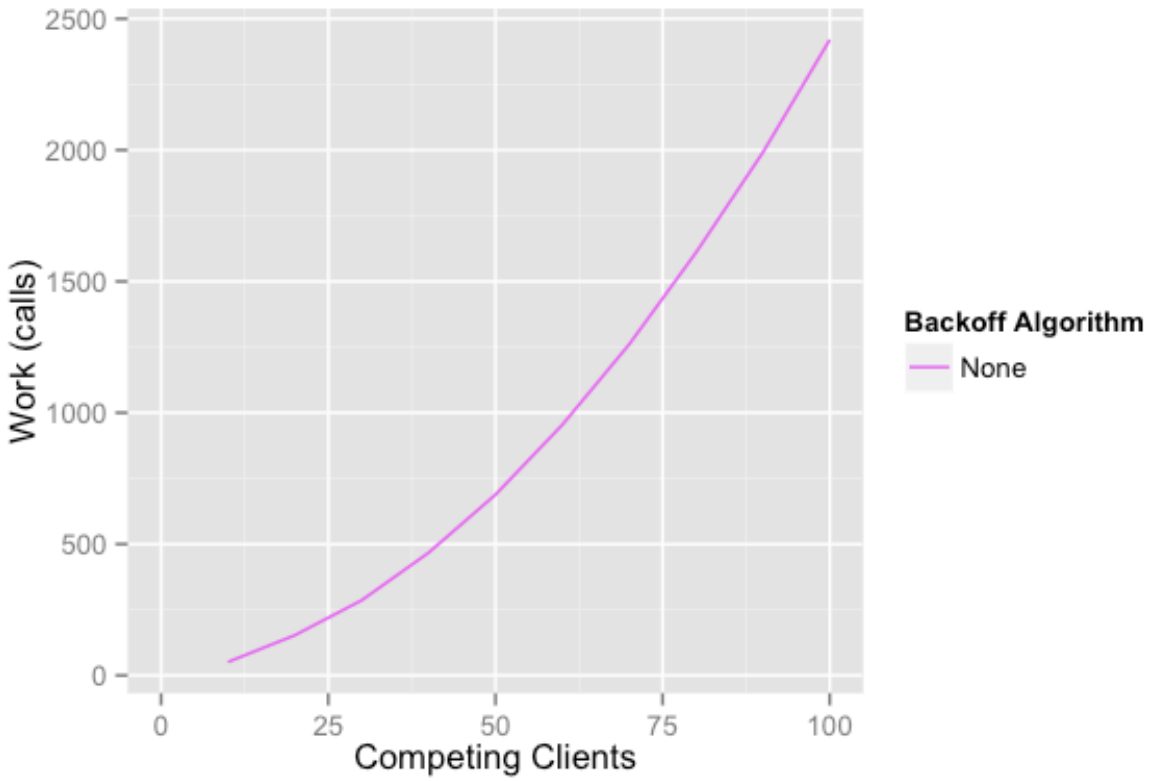
É importante sabermos que, muitas vezes, ao começar o trabalho de forma assíncrona as pessoas desenvolvedoras procuram diversas soluções e ferramentas como webMQ, Kafka, SQS. Fazendo isso, elas têm a ideia de que é só mandar e receber que tudo funciona bem. Mas a nossa sugestão é que as pessoas entendam com profundidade seu message broker. É essencial que se compreenda bem o sistema ao utilizá-lo, pois algumas vezes a forma como estamos usando pode ser inadequada e, ainda assim, perderemos dados. Então, é importante compreendermos bem o seu funcionamento e as garantias de recebimento/entrega para utilizarmos essas soluções da melhor forma possível.

Garantias de entrega com Retry

Quando queremos desenvolver um software resiliente, o primeiro ponto que devemos observar é se temos a garantia de que nossas chamadas serão entregues. Ao realizarmos uma requisição, é essencial termos a garantia de que a mensagem que estamos enviando está chegando ao destino. Mas sabemos que nem sempre isso acontece, pois o outro sistema pode estar lento, fora do ar, etc. Uma alternativa para minimizar esses problemas é trabalhar utilizando políticas de retry.

Essas políticas são basicamente a tentativa de reenvio da mensagem. Ou seja, mandamos uma mensagem e se o outro sistema não respondeu por determinado tempo, mandamos outra. Caso ele não responda novamente, continuamos tentando até o sistema responder.

Então, se observamos o gráfico abaixo, retirado de um artigo da Amazon (https://aws.amazon.com/pt/blogs/architecture/exponential-backoff-and-jitter/), veremos que à direita temos o work (calls) que são as quantidades de chamadas realizadas para um serviço e em sua parte de inferior temos a quantidade de clientes que estão competindo para conseguir realizar aquela chamada.

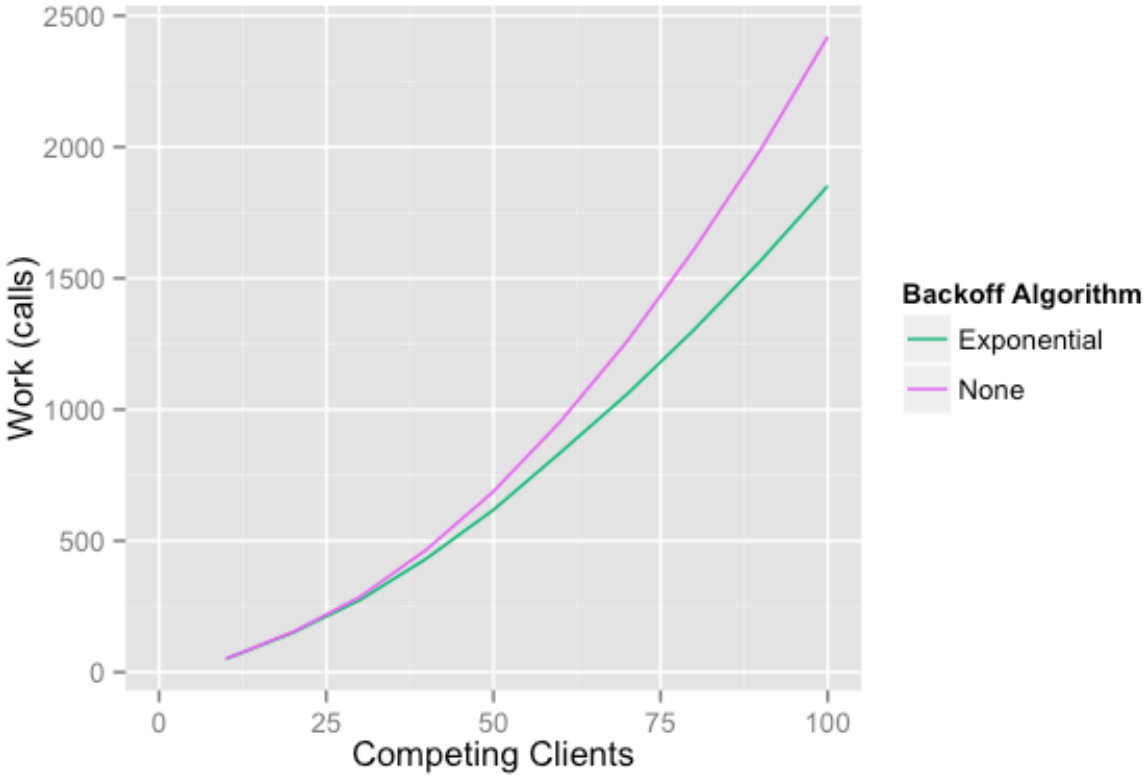


Linear - Sem backoff

Para compreendermos melhor o funcionamento dessa estratégia, vamos imaginar que temos 10 sistemas tentando fazer uma chamada em outro serviço. Digamos que esses 10 clientes resolveram fazer uma chamada ao mesmo tempo. Isso significa que todos vão bater no serviço ao mesmo tempo. Provavelmente, este outro serviço terá dificuldade em lidar com isso e acabar travando. Os sistemas que estão tentando se comunicar com ele poderão enviar a mensagem novamente depois de 2 segundos, e depois de aguardar mais 2 segundos para enviar outra mensagem. Lembrando que os 10 clientes farão isso de maneira simultânea, logo, se o sistema não conseguiu antes, dificilmente conseguirá depois desses intervalos. Isso quer dizer que não adiantaria fazer o retry da chamada de forma linear - a cada 2 segundos, 3 segundos -, pois todos os sistemas estão fazendo retry ao mesmo tempo, o que continuará sobrecarregando o serviço.

Por conta disso, existem algumas técnicas que aumentam a probabilidade de conseguirmos fazer um retry com sucesso, por exemplo: o exponential backoff. Vamos observar que no próximo gráfico a linha rosa representa onde não tínhamos backoff, ou seja, era totalmente linear, e por isso o tempo para conseguirmos ser respondidos na chamada era extremamente longo. Com o exponential backoff, representado pela linha verde, a situação é um pouco diferente. Nesta técnica daremos um tempinho a mais para que o serviço consiga responder. Na prática, se mandamos uma chamada e esta não é respondida,

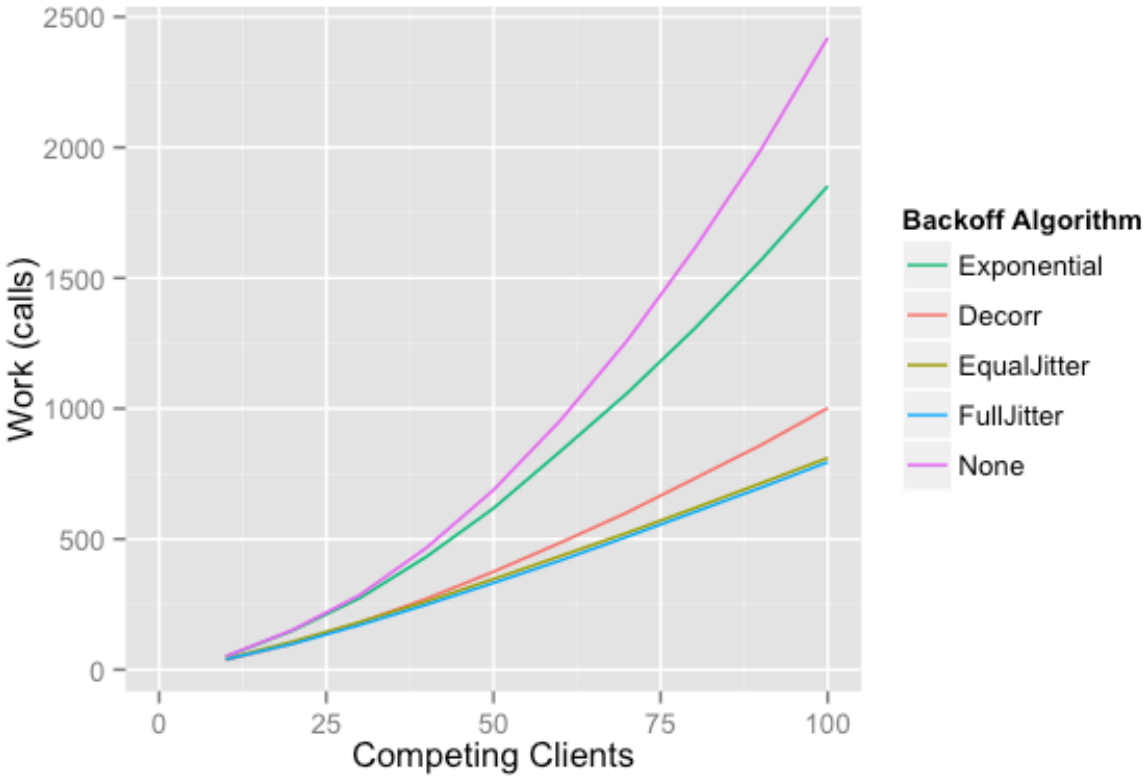
esperamos 1 segundo e mandamos um retry. Depois, caso ele continue não nos enviando a resposta, ao invés de esperarmos 1 segundo novamente, esperamos 2 segundos, depois 4, 8, 16... ou seja, repetimos a tentativa de forma exponencial para que durante esse tempo possamos dar mais espaço para que o sistema consiga se recuperar. Porém, com essa estratégia ainda teríamos bastante dificuldade em relação ao tempo para conseguirmos uma resposta porque apesar dos números terem melhorado um pouco, provavelmente esses 10 clientes também estarão trabalhando de modo exponencial. Os 10 vão esperar 2 segundos, depois 8 segundos, 16 segundos... Isso irá melhorar um pouco pois algumas requisições que estavam travadas serão liberadas, mas ainda assim, como podemos observar no gráfico, não teremos uma melhora significativa. Isso acontece porque um exponential backoff utiliza sempre os mesmos algoritmos para fazer seus processos de retry.



Exponential backoff

Se todos os clientes começarem a fazer chamadas a partir de agora, de modo simultâneo, dificilmente o sistema conseguirá responder a essas solicitações. Uma alternativa para essas dificuldades é utilizar o exponential backoff com jitter. Nessa estratégia colocamos uma espécie de mini algoritmo em nossa requisição para gerar um pequeno ruído no tempo de chamada. Caso tenhamos problema na hora em que formos fazer uma chamada para 1 segundo, e formos fazer a solicitação de 2 segundos, colocaremos 2.1, na outra solicitação 2.05, depois 2.25, em seguida 2.3 até conseguirmos. Podemos perceber que pelo fato de termos um pequeno ruído nesse algoritmo - os números não são mais

exatamente iguais -, conseguiremos fazer com que tentativas simultâneas sejam evitadas. Assim, as tentativas acontecem em momentos diferentes, aumentando as chances da requisição funcionar. Podemos observar no gráfico abaixo como foi possível diminuir a quantidade de chamadas conforme o número de clientes aumentava.



Exponential backoff- Jitter

Isso nos mostra que todas as vezes que trabalhamos com políticas de retry é extremamente válido inserirmos o jitter, ou seja, vamos adicionar algoritmos de ruídos para que nossa requisição não seja exatamente igual a dos outros clientes. Assim, o sistema não terá que lidar com requisições simultaneamente, tendo

mais chances de trabalhar.

Então, caso não sejamos respondidos na primeira vez que solicitarmos, com essa estratégia temos mais chances de alcançarmos o resultado esperado em nossa segunda tentativa. O que queremos destacar, neste tópico, é a necessidade de sermos espertos em nossas novas tentativas. Porém elas precisam ter lógica, os números colocados não são meros palpites.

Garantias de entrega com Kafka

Uma das formas de termos garantia de entrega é utilizando um broker. Neste tópico escolhemos um em específico para usarmos como exemplo, o Kafka. Mas é importante sabermos que a maioria dos brokers trabalha de forma semelhante. Por isso não queremos destacar, necessariamente, o comportamento dessa solução, mas sim o quão longe vai a possibilidade de garantirmos a entrega de uma informação por meio de um broker.

Em relação à resiliência, é importante pensarmos em como garantir que nossas informações não serão perdidas. Além disso, temos que buscar a garantia de que essas informações serão, de fato, processadas pelo sistema que chamamos. Para isso, podemos trabalhar de forma assíncrona, pois essa forma de trabalho pode nos ajudar a lidar com mais informações do que poderíamos. Então, nesse modo de trabalho temos a alternativa de utilizar um message broker, ou seja, um sistema que armazena nossas requisições e depois possibilita que elas sejam lidas em outros sistemas. Mas precisamos ter a certeza de que a informação que

estamos enviando para o kafka, por exemplo, está chegando até ele e ter certeza de que a requisição não será perdida no meio do caminho. Normalmente, gosto de "brincar" que cada transação que envio tem o valor de 1 milhão de dólares, então, não posso perdê-la de forma alguma. Porém, quando se trata de resiliência, tudo tem um lado bom e um lado não tão bom. Veremos isso nos próximos parágrafos.

Partindo do princípio de que não sabemos muito sobre o Apache Kafka, vamos imaginar que mandaremos uma mensagem para que essa solução guarde até que o outro sistema possa processá-la. Quando trabalhamos com alta disponibilidade teremos um cluster, isto é, um conjunto de brokers. Nesse caso, falando especificamente do Kafka, teremos o broker "A", "B" e o "C". A mensagem que enviamos cairá em um broker "A", que é chamado de líder. Sabendo disso, temos alguns pontos para pensarmos em relação à garantia de entrega. Primeiro, podemos optar por não ter uma confirmação de entrega, logo não temos certeza de que nossa mensagem foi recebida. Podemos escolher essa opção dependendo do nível de importância da mensagem.

Por exemplo: se mandarmos uma carta pelos correios e simplesmente aguardamos sua entrega, provavelmente nossa encomenda é algo que não tem tanta importância, por isso escolhemos não ter sua garantia de entrega. Mas se precisarmos ter a certeza de que o correio entregou a encomenda, podemos pagar um valor a mais pela garantia de recebimento. Neste segundo caso, ele nos envia a confirmação de que o enviado foi, realmente, recebido. Semelhantemente,

temos essa opção quando enviamos uma mensagem ao nosso broker: ter ou não a garantia de entrega.

Vamos imaginar, ainda, que estamos trabalhando com o Uber. A cada momento o aplicativo nos mostra a posição do motorista e do passageiro. Mas o que aconteceria se perdêssemos algumas dessas posições? Seria um problema gravíssimo para o negócio? Acreditamos que não. O aplicativo pode nos mostrar o máximo de posições possíveis, ou seja, mesmo que algumas posições sejam perdidas, o mais importante é que sejam mandadas o maior número possível. Como não pedimos uma confirmação, esse uber fica mais rápido. Isso acontece porque só está recebendo, sem necessitar pausar para confirmar que recebeu. Logo, podemos perceber que todas as vezes que não pedimos a confirmação de entrega ganhamos velocidade. Porém, existe a possibilidade de perdermos uma mensagem ou outra. Podemos chamar isso de fire-and-forget, isto é, disparamos, esquecemos e torcemos para que a mensagem seja entregue.

Existem alguns casos em que não podemos trabalhar desse modo, pois precisamos, necessariamente, ter a certeza de que o broker recebeu a mensagem. Isso pode acontecer porque a mensagem é muito importante, por exemplo 1 milhão de dólares. Neste caso, devido à sua importância, precisamos ter a confirmação de que o broker líder realmente recebeu a requisição. Mandamos a solicitação e o líder nos retorna um Ack1, ou seja, a confirmação que precisamos ter.

Porém, vamos supor que temos 3 brokers e essas informações ficam replicadas, assim, caso um caia o outro poderá assumir. Digamos que nós enviamos uma

solicitação para o broker “A” e ele nos deu um retorno de recebimento, neste caso ficamos tranquilos. Mas em seguida, por algum motivo, esse broker caiu. Sem saber da queda, pensaremos que a mensagem está segura, porém, por alguns momentos nossa mensagem não teve alta disponibilidade. Isso pode fazer com que percamos alguma mensagem.

Uma alternativa para evitarmos o problema anterior é trabalharmos com Ack-1 ALL. Neste modo de trabalho, mandamos a mensagem para o broker “A” e antes de nos responder ele manda essa mensagem para o broker “B”. Em seguida, o “B” envia para o broker “C”. Somente depois disso temos um retorno de que a mensagem já foi sincronizada entre os 3 brokers. Dessa maneira, teremos a certeza de que a mensagem, de fato, está segura, pois caso um dos brokers caia, os outros ainda mantêm nossa requisição disponível.

Resumidamente, se queremos alta velocidade para mandar as mensagens, faremos Ack 0, mas sabemos que podemos perder alguma dessas requisições. Se queremos algo moderado, podemos fazer um Ack 1, ou seja, ter a confirmação do líder. Mas se quisermos maior garantia possível, trabalharemos com Ack-1 ALL.

Essas observações não são apenas para o kafka pois existem outros sistemas que trabalham de forma similar. Essa solução foi utilizada apenas para termos um exemplo prático. Diante de tudo isso, o que queremos destacar é que precisamos ter uma noção do nível que as coisas chegam em relação a garantias quando trabalhamos de forma assíncrona. Isso porque não se trata apenas de mandar

uma mensagem. É importante entendermos, com um nível de profundidade, o broker que estamos trabalhando. Assim podemos garantir que a nossa mensagem está chegando ao destino esperado. Então, se pensarmos novamente no exemplo de 1 milhão de dólares, qual seria a melhor opção: usar Ack 0 ou Ack-1? Para garantirmos a resiliência precisamos compreender o custo benefício dessa resiliência. Tanto em relação à performance, quanto em relação à segurança dos dados que queremos ter.

Situações complexas e decisões de alto nível

A resiliência deve ser pensada no dia zero e de forma intencional, pois é algo extremamente importante para a sustentabilidade do software. Por outro lado, existem situações complexas que fazem com que nossos planos para tentar que as soluções sejam mais adaptáveis falhem. Então, neste tópico, nos dedicaremos a fazer alguns questionamentos que podem auxiliar na reflexão de como conseguir isso de fato. Com isso, não buscamos oferecer soluções prontas, mas sim pensamentos capazes de nos fazer analisar sobre aspectos importantes para a tomada de decisões que garantam a resiliência que nosso sistema precisa ter.

A primeira pergunta que devemos fazer é: como ter a resiliência da resiliência? Vamos trazer alguns exemplos práticos que poderão nos ajudar a pensar com mais clareza sobre isso.

Digamos que nós temos um broker que apoia nossa aplicação. O que aconteceria

se este caísse? Se o Kafka caísse? Ou o SQS? Ou o RabbitMQ? Perderíamos mensagens? É possível dizermos como nosso sistema vai se comportar? Devemos pensar nisso, pois existem sistemas que não sobem caso não consigam se comunicar com o RabbitMQ, por exemplo. Precisamos pensar em como garantir a resiliência em situações inusitadas.

Ao observamos bem as aplicações, sempre teremos um single point of failure (SPOF), ou seja, seria como dizer que estamos apoiando toda nossa resiliência em determinada solução. Usando o Kafka como exemplo no Kafka, isso significa que caso o caia, cai tudo em minha solução. Mas como evitar isso? Como preparar meu sistema para que, caso o Kafka caia, não percamos informações e quando ele voltar possamos continuar mandando essas informações? Por mais improvável que as situações pareçam, temos que deixar as aplicações um pouco mais preparadas em diversas situações. Então, quanto mais situações conseguirmos pensar, mais resiliência e disponibilidade teremos, o que nos leva a outra questão, isso será mais caro também.

Pensando, por exemplo, em situações pouco prováveis, qual seria a probabilidade da AWS cair em uma região inteira? As chances de isso acontecer são poucas. E, caso acontecesse, poderíamos trabalhar com multizona, ou seja, zonas de disponibilidade. Configurar nossas máquinas em diferentes zonas de disponibilidades é gratuito, então não existe um motivo para não trabalharmos com A a Z, pois se nossa região inteira cair, precisamos migrar rapidamente para outra região, mesmo que saibamos que a AWS já caiu diversas vezes. Quem dependia apenas

de uma zona de disponibilidade ficou em uma situação complicada, isto é, teve que trabalhar com duas regiões. Tornando a situação mais improvável ainda, precisamos ter a disponibilidade de migrar rapidamente para Edge (Google Distributed Cloud Edge) se a Amazon vier a desaparecer.

Nos dias atuais muitas empresas estão trabalhando com multi-cloud. Muito provavelmente isso não se dá apenas por uma questão de custos. Está relacionado à segurança, resiliência e disponibilidade. Então, o que queremos destacar é que sempre terá um limite para setar nossa resiliência. Ou seja, quanto mais resiliência, mais esforço e mais caro. Todavia, podemos dizer que não é responsabilidade do desenvolvedor definir qual é o nível de resiliência, muitas vezes essa decisão precisa ser estratégica da empresa, pois em níveis elevados, saberão quais riscos a organização está disposta a ter para o negócio. Diferente de decisões que envolvam comunicação entre sistemas, perda de dados, tentativas de retry. Estas são responsabilidades dos devs.

A resiliência é algo extremamente complexo, pois além de envolver um gerenciamento de riscos, a cada 99.999 que colocamos em nosso sistema custará muito mais caro para nossa empresa. Ou seja, teremos que definir e gerenciar riscos para nosso chefe ou com nosso CEO.

Para pensar em resiliência terá que ser definido os custos, a necessidade de mais mão de obra, etc. Isso significa que essas decisões são mais alto nível por envolver dinheiro, especialidades, etc. Assim, será avaliado, como já dissemos, o custo benefício relacionado à necessidade de ter um sistema resiliente.

Sistemas Monolíticos

Sistemas “tradicionais”

Nada melhor de que exemplos concretos para facilitar entendimento quando falamos em desenvolvimento de sistemas, logo, para começarmos esse capítulo, imagine um sistema de gerenciamento de produtos onde tais produtos podem ser categorizados e disponibilizados para venda. Também tal sistema permite que o usuário(a) possua uma área de busca, bem como o recurso de checkout para realizar a compra.

Com isso em mente, já é possível termos ideia de outras áreas que poderiam existir dentro desse sistema, como um catálogo para exibição dos produtos, evoluindo assim para uma loja virtual.

Podemos utilizar plataformas para desenvolver tal loja como um *Magento* por exemplo.

Essa é uma abordagem super comum principalmente devido a evolução de ferramentas, CMSs e frameworks. Todos os recursos fazem parte de uma mesma aplicação.

Esse é o clássico exemplo de um sistema monolítico, onde todas as operações acontecem em sua própria estrutura utilizando normalmente uma única linguagem de programação. Nele, são incorporadas todas as entidades, regras de negócio, disponibilização de APIs e integrações. Obviamente tudo está fortemente acoplado em um único sistema que normalmente possui mais de uma responsabilidade.

Restrições

Pela própria natureza dos sistemas monolíticos, normalmente eles utilizam uma única linguagem de programação. Por exemplo, se nós temos um sistema monolítico feito em Ruby, é improvável tenhamos alguma coisa escrita em PHP ou Python.

Em muitos casos essa possível "limitação" de utilizar uma única tecnologia dentro de um sistema pode fazer com que a organização não tire proveito de tecnologias, que para determinados casos de uso, sejam mais eficientes trazendo assim mais valor para o negócio como um todo.

Além disso, temos que concordar que ao colocarmos, cem, duzentas, ou mesmo mil pessoas para trabalhar na mesma base de código pode ser em determinadas situações algo caótico.

Monolitos não são ruins

Apesar das restrições citadas acima, trabalhar com sistemas monolíticos não é nenhum demérito ou atestado de obsolescência.

Sistemas monolíticos na maioria das vezes sem dúvidas é a melhor opção para grande parte das empresas. Afinal de contas, nem toda empresa possui 6000 devs como Mercado Livre.

Trabalhar com sistemas monolíticos reduz a complexidade e aumenta a eficiência principalmente em situações em que as próprias regras de negócio da empresa estão em constante mudança. Startups, empresas fazendo validações de modelos de negócio, bem como corporações que não fazem o uso "pesado" de tecnologia como principal ponto de sustentação, são grandes candidatas a trabalharem com sistemas monolíticos.

Via de regra, "todo" sistema deve começar com um monolito.

Martin Fowler em seu artigo MonolithFirst[1] faz duas grandes observações logo no início:

1. "Quase todas as histórias de sucesso de sistemas utilizando microsserviços começaram com um monolito que ficou grande e depois foi quebrado em partes."

[1] https://martinfowler.com/bliki/MonolithFirst.html

2. “Todos os casos que ouvi de sistemas que começaram diretamente utilizando microsserviços tiveram grandes problemas”

Para que uma companhia inicie o processo de desenvolvimento de seus sistemas utilizando uma abordagem diferente da monolítica, sem dúvidas os profissionais ali inseridos devem ter muitas cicatrizes com experiências de sucesso anteriores, caso contrário, há uma grande probabilidade do projeto fracassar.

Deploy

Sistemas monolíticos possuem uma característica singular quando precisam ir ao ar. Como tudo encontra-se dentro da mesma estrutura, por mais simples que seja determinada mudança, o deploy de 100% da aplicação precisa ser realizado.

É evidente que quando todas as áreas de uma empresa estão concentradas em um único sistema, o risco de interrupção de todas as áreas aumenta a cada deploy realizado. Em contrapartida, complexidades de comunicação entre diversos sistemas, a necessidade de criação de uma grande quantidade de pipelines de entrega, bem como todos os aspectos comuns em entregar e gerir projetos em produção são reduzidos drasticamente.

Necessidade de escala

Quando trabalhamos com grandes sistemas, invariavelmente teremos a necessidade de escala.

Segundo a definição da Gartner[2]:

"Escalabilidade é a medida da capacidade de um sistema de aumentar ou diminuir o desempenho e o custo em resposta às mudanças nas demandas de seus aplicativos e processamento."

Nesse ponto, é evidente de que quando temos a necessidade de escalar um sistema monolítico, não há a possibilidade de escalarmos apenas as funcionalidades que naquele momento estão exigindo mais recursos computacionais, ou seja, todo sistema precisa escalar.

Por exemplo, se um ecommerce receber uma grande quantidade de acessos em seu catálogo de produtos e por consequência precisar de mais recursos computacionais para segurar a carga, todas as outras funcionalidades, que muitas vezes não precisariam ser escaladas, terão de escalar, pois tudo faz parte de um único conjunto.

Em determinados casos essa necessidade pontual de escala por um recurso pode fazer com que os custos com infraestrutura sejam elevados, pois claramente há uma ineficiência técnica embutida no processo.

[2] https://www.gartner.com/en/information-technology/glossary/scalability

Em contrapartida, em determinadas situações, não é porque um sistema precisa ser escalado por inteiro que seus custos serão mais elevados quando trabalhamos com outro tipo de arquitetura, como a de microsserviços por exemplo. Temos que ter em mente que quando trabalhamos com arquiteturas distribuídas, há também outros custos inerentes a essa modalidade.

Débitos técnicos

É evidente que qualquer tipo de aplicação, independente da arquitetura a ser utilizada em algum momento terá débitos técnicos, ou seja, pequenas melhorias, refatorações, implementações que deveriam ser realizadas e que foram "deixadas para depois".

A medida em que os débitos técnicos se acumulam ao longo do tempo, a instabilidade do sistema como um todo acaba aumentando, gerando assim uma queda considerável na produtividade da manutenção e implementação de novos recursos.

Quando trabalhamos com sistemas monolíticos, isto acaba sendo potencializado, uma vez que há um alto acoplamento em todos os componentes da aplicação.

Por outro lado, isso não significa que pequenos sistemas não possuem ou não possuirão débitos técnicos. Todavia, como a base de código é limitada, tais débitos não terão tanta influência na solução como um todo.

Domain Driven Design

Introdução

O design orientado ao domínio, também conhecido como DDD (Domain Driven Design) é uma abordagem que trabalha com práticas de design e desenvolvimento, oferecendo ferramentas de modelagem tática e estratégica para entregar um software de alta qualidade, acelerando o seu desenvolvimento e garantindo sua sustentabilidade.

Alguns desenvolvedores acreditam que o DDD se resume a apenas uma série de design patterns como agregados e repositórios, criando uma pasta de infra para separar as camadas do nosso projeto, entre outros. Outros já veem o DDD com mais abrangência. Mas, no fim das contas, também não conseguem explicar claramente como isso funciona.

Isso tudo é muito estranho porque de fato o DDD parece complexo quando nós consultamos as principais literaturas a respeito dele. E mesmo quando pesquisamos, isso ainda nos deixa dúvidas sobre como nós podemos colocar isso em prática no nosso dia a dia.

Nesse capítulo não vamos focar apenas nos aspectos práticos porque o DDD vai

muito além disso. O seu foco é conhecer não apenas o ambiente, mas também os contextos e as pessoas que trabalham em um projeto. E ainda, baseado nisso, permitir uma separação que faça sentido para a organização em si.

Não basta começar diversos projetos e dizer que estamos aplicando o DDD em tudo, sendo que no final das contas isso resulta em diversas pastas repetidas em diversos projetos.

O intuito deste capítulo é fazer com que essa filosofia de trabalho mude a sua forma de pensar em software, principalmente no seu trabalho com projetos de grande porte. Normalmente não aplicamos isso em pequenos projetos porque o DDD é fundamentalmente utilizado quando nós não temos clareza total do projeto e suas áreas.

Uma palavra que resume bem o DDD é clareza. A clareza em um projeto minimiza seus riscos em diversas perspectivas, principalmente quando o mesmo deve perdurar por anos. Nós não podemos fazer um software que 6 meses se tornará o famoso “legado” e que nenhuma pessoa desenvolvedora irá querer mantê-lo.

O DDD é sem dúvidas um recurso que pode nos ajudar com esse objetivo e, com a sua devida aplicação, desenvolver software se tornará mais divertido e com menos riscos.

Ponto de partida no DDD

Agora iremos explorar a filosofia e os conceitos teóricos que estão por trás do DDD, considerando que ao termos mais entendimento desses pontos, facilitará o processo de aplicar DDD na prática.

Como o próprio nome sugere, o Domain Driven Design se refere a como podemos desenhar o software guiado ao domínio, que é o coração da aplicação.

Não pense apenas sobre os design patterns, pastas dentro do seu projeto, entre outros, porque o DDD foca muito mais em como modelar o software do que desenvolvê-lo em si.

O DDD é uma forma de desenvolver o software focando no coração da aplicação, o que nós chamamos de domínio. Seu objetivo é entender os contextos e regras do projeto, seus procedimentos e complexidades, separando-as de outros pontos complexos que são adicionados durante o processo de desenvolvimento.

O DDD surgiu de um autor chamado Eric Evans, de seu livro publicado em meados de 2003.

É importante termos isso em mente porque o DDD é um assunto que oscila muito durante os anos. Ora é muito falado, ora não. Atualmente, com a importância dos microsserviços, o DDD também tem destaque porque o grande desafio de trabalhar com microsserviços é modelar o software e os seus contextos.

Quando observamos o livro de Eric Evans, é notável que existe um lado filosófico em torno dele, que chega ser até mais importante do que os padrões de projeto que utilizamos em nosso dia a dia.

Essa filosofia parte de uma visão madura para que o desenvolvedor trabalhe em seu projeto com orientação de trazer soluções para problemas complexos. Não podemos ser inocentes ao ponto de ver um projeto e pensar somente no banco de dados, cadastros, CRUDs, entre outros.

Trabalhar em torno dessa filosofia esclarece o quão é importante é entender e modelar um software baseado em suas complexidades de negócio.

Foram através desses conceitos iniciais de Evans que surgiram entusiastas que tiveram mais clareza nas falhas que existiam em seus grandes projetos.

Depois do lançamento do livro de "capa azul" (como assim é conhecido o livro de Evans), houve uma série de lançamentos de outros livros que foram imprescindíveis para que pudéssemos nos aprofundar na filosofia em torno do DDD.

Entre os mais destacados está o livro de "capa vermelha" de um autor chamado Vaughn Vernon. Este segundo livro é um pouco mais prático em relação ao livro de Eric Evans.

Outro livro de Vernon para se aprofundar é o Domain Driven Design Distilled. Ele é uma obra mais recente em relação as outras contendo um resumo sobre o DDD. É um livro para se ler com cautela pois ele deixa algumas partes

importantes de fora.

A vantagem de ler esse livro são os tópicos que vão direto ao ponto, que desmistificam grande parte do DDD de uma forma menos densa.

As complexidades de um software

Normalmente consideramos aplicar o DDD em projetos de software mais complexos. Não faz sentido usar essa abordagem em um sistema típico de estabelecimentos de pequeno porte, como uma “padaria”, que só vai vender um único produto e receber o troco.

A complexidade de um software como a do exemplo anterior é tão pequena que não há quase o que modelar. É um nível de complexidade tão simples que boa parte dos softwares de empresas desse tipo são softwares de prateleira.

São softwares que podem ser adaptados a qualquer tipo de negócio sem customização.

O DDD é normalmente utilizado quando temos problemas maiores que nos impedem de termos a clareza de como as áreas e pessoas de um projeto se relacionam e se comunicam.

Quando trabalhamos com pessoas de diferentes departamentos, por exemplo, nós percebemos que elas usam termos completamente diferentes entre as suas respectivas áreas.

O DDD nos deixa claro de que em grandes projetos há muitas áreas, regras de negócio e pessoas com diferentes visões da organização que estão situadas em diferentes contextos.

Vamos usar como exemplo uma empresa que seu “core business” é fazer cobranças de contas em aberto em nome de diversas corporações. Essa operação envolve atendentes de telemarketing que usam um software de discagem automática. Se pensarmos bem, com certeza existe um diferencial na automatização desses processos de cobrança em relação ao de uma empresa “tradicional”, que liga para cobrar boletos bancários em aberto de seus clientes.

Isso acontece porque o coração do negócio da empresa de telemarketing é a cobrança. Cobrar é a razão da empresa existir, logo, a modelagem de um problema de cobrança nesse caso torna-se muito mais complexo pois provavelmente envolve seu diferencial competitivo no mercado.

Também, normalmente quando tratamos de empresas e seus diversos departamentos, esses departamentos possuem a sua própria forma de se “expressar” e falar do negócio utilizando certos jargões no dia a dia.

Os bancários, por exemplo, podem usar o termo ‘francesinha’, que é o nome que eles dão para um tipo de relatório de quem realizou pagamentos. Mas quando esse termo é mencionado para funcionários de outro departamento, isso pode não fazer sentido nenhum.

Quando percebemos isso, é possível ter mais clareza para entender que o

software não é apenas uma simples unidade. Ele é feito de contextos, regras, implementações que possuem objetivos diferentes.

Muitas vezes os softwares são independentes, sejam eles microsserviços ou sistemas monolíticos. No fim das contas o software é vivo porque ele é movido a pessoas que atuam em diferentes contextos e entender isso é crucial durante a criação de uma modelagem para cada tipo de contexto.

Não é possível deixar de utilizar técnicas avançadas em projetos de alta complexidade porque não podemos tratar de um software grande, que tem diversos departamentos, complexidades e regras, de uma forma simplória. O software precisa se adaptar a organização e não a organização ao software.

Em torno de uma solução há política, pessoal e cultura. Tudo isso deve ser levado em consideração. Se não levarmos isso em conta, sem dúvidas o projeto já fracassou em seu primeiro dia de desenvolvimento.

Seja qual for a sua experiência trabalhando em grandes projetos, você já deve ter visto um projeto falhar devido a alguns pontos que foram citados até aqui.

É notável que grande parte da complexidade desse tipo de software não vem da tecnologia; mas da comunicação e separação de contextos que envolvem o negócio por diversos ângulos.

Nós perguntamos como as coisas devem ser feitas, seguindo as instruções que nos foram dadas. E de repente o responsável pelo produto diz que quer um resultado diferente.

Frequentemente alguns desenvolvedores são inocentes e recebem o feedback do cliente dizendo que está tudo em ordem com o software, mas em outras reuniões eles levam o que foi solicitado e descobrem que outro membro da equipe não concorda com o que foi feito.

Como o DDD pode te ajudar

De forma geral o Domain Driven Design vai te ajudar a ter uma visão ampla do problema a ser resolvido e a quebra-lo em problemas menores. Também ele te dará técnicas de como minimizar ruídos de comunicação entre todos os envolvidos, bem como trabalharmos com patterns que visam deixar nossas aplicações cada vez mais desacopladas preservando ao máximo suas regras de negócio.

Para que o DDD te ajude a entender os principais desafios de como desenvolver software e suas complexidades geradas em torno de uma organização, é importante que entendamos os pontos abaixo. Lembrando que todos eles serão aprofundados durante nosso capítulo sobre DDD.

Resumindo

Nos tópicos anteriores, falamos sobre Domain Driven Design de maneira introdutória, mas daqui para frente queremos explicar alguns aspectos essenciais

para compreensão do que realmente é o DDD. Porém, antes disso, precisamos alinhar nossas expectativas em relação a este módulo. Digo isso, pois sabemos que muitos devs querem entender o DDD passando direto para a prática, criando várias pastas e organizando sua aplicação. Todavia, este tema é muito mais profundo do que isso. Existem alguns conceitos teóricos que são extremamente relevantes para que possamos realmente trabalhar com DDD.

Antes de explicarmos um pouco mais os aspectos que, de fato, fazem parte do DDD, queremos parafrasear a fala de Vernon. Esse autor nos afirma que falar de DDD diz muito sobre conseguir modelar de forma explícita uma linguagem universal, isto é, uma linguagem ubíqua, dentro de um contexto delimitado. Ou seja, basicamente, antes de usarmos o DDD, temos uma grande confusão em nosso domínio. Então, começamos a entender um pouco melhor o que está acontecendo depois que passamos a escutar as pessoas falando entre seus diversos departamentos. Isso acontece porque conseguimos compreender suas linguagens. Quando percebemos que essa linguagem muda, temos uma sinalização de que o contexto mudou. E ao entendermos o contexto, conseguimos delimitá-lo. Feito isso, podemos desenvolver um software para aquele "pedaço", especificando os jargões, as expressões, os nomes dos relatórios, as entidades e os participantes do projeto. Eventualmente, esse contexto vai poder se comunicar com outro, mas aquela confusão não existirá mais, pois teremos aquela clareza que precisamos.

Assim, quando falamos em DDD é necessário entendermos como modelar as

linguagens, compreendendo seus contextos. Desse modo, trabalharemos cada um dos seus pontos principais.

Domínio

O domínio, ou “domain”, é a razão pelo qual o software existe. Em outras palavras, é o coração do software. E esse conceito por si já define que nós temos um desafio para resolver.

Se não entendermos o real objetivo pelo qual o software será desenvolvido, a batalha já está perdida. Normalmente quando recebemos o “problema” do cliente, tal problema é apenas a ponta do iceberg. Precisamos mergulhar a fundo, entender mais sobre a organização, suas áreas, e as intenções reais por de trás da solução a ser desenvolvida.

Subdomínios

O grande desafio é que normalmente o problema a ser resolvido é muito grande, e nesse ponto, de forma inevitável, precisamos dividi-lo em partes menores. Essas partes são chamadas de subdomínios. Diferente do domínio, que acaba se tornando uma visão geral do problema, os subdomínios acabam cobrindo detalhes finos que nos ajudam a ter um pouco mais de clareza sobre o coração da aplicação.

Linguagem universal

O DDD também estabelece uma linguagem universal entre todos os que estão envolvidos no projeto. A “Ubiquitous Language”, ou Linguagem Ubíqua, é um termo recorrente em qualquer livro sobre DDD.

Vamos pensar no exemplo da francesinha na área bancária (extrato de movimentação de títulos). Não adianta criarmos um sistema com o menu chamado “relatório de contas pagas”, pois todos naquele departamento ficarão procurando por “francesinha”, considerando que ela segue um jargão popular naquele contexto.

Todo esse problema ocorre porque nós não conseguimos ter uma única linguagem universal dentro da empresa. Por mais estranho que pareça, você vai perceber que a empresa é composta por uma cultura que é modificada e adaptada aos poucos dentro de cada departamento.

E como cada área tem o seu próprio jargão, para um funcionário na área de vendas pode existir um cadastro de clientes que fecharam contratos com ele. Já no departamento de de compras também existe uma área para cadastro de clientes, porém nesse caso o cliente é a própria empresa, pois ela possui diversas filiais.

Perceba que em um departamento o cliente é chamado de uma forma, mas dentro de um setor diferente a palavra “cliente” já possui outro significado. Esses

departamentos podem usar a mesma palavra e, caso se comuniquem, o cliente vai representar uma coisa completamente diferente.

Por isso é importante entender, mapear e extrair essa linguagem universal para esclarecer e minimizar os principais ruídos de comunicação.

Design estratégico e tático

Um dos objetivos do DDD é nos ajudar a criarmos o design estratégico e tático para a modelagem de nossas aplicações.

Isso significa que através da clareza da existência de um domínio e seus diversos subdomínios, podemos criar a modelagem estratégica delimitando contextos e seus relacionamentos. Normalmente tal modelagem é chamada de “Context Map” ou mapa de contexto.

Além disso, precisamos de um design tático para mapearmos agregados, entidades, objetos de valor, entre outros, facilitando assim o processo de codificação do sistema.

Agora que tivemos uma idéia geral sobre DDD, a seguir entenderemos os princípios básicos relacionados aos domínios e subdomínios como elementos fundamentais do Domain Driven Design.

Delimitação

O domínio é parte essencial do DDD, considerando que o design guiado ao domínio se refere objetivamente ao coração da aplicação. Entender o domínio é ter entendimento sobre as áreas que envolvem o negócio; e assim que reconhecemos isso nós podemos dividir tal domínio em partes menores, conhecidas como subdomínios.

Delimitar um domínio nos possibilita pensar na solução em torno de toda a complexidade do negócio. Existem, porém, diversos tipos de problemas e complexidades que são partes importantes para o software.

Na própria literatura do DDD existe um exemplo que trata a exploração do nosso domínio como se estivéssemos entrando num quarto escuro segurando apenas uma lanterna. Nós só conseguimos enxergar algumas partes do cômodo quando ligamos a lanterna. O mesmo acontece quando começamos a explorar os nossos domínios e subdomínios porque inicialmente nós não temos a visão de um todo.

E é assim que conseguimos ver, nós percebemos que existem partes que nós podemos separar e é por isso que essas partes são chamadas de subdomínios. Mas na separação dos subdomínios também percebemos que eles possuem graus diferentes de importância para o negócio.

Domínio principal ou “Core Domain”

Quando identificamos e separamos a parte mais importante desse negócio nós temos então o nosso Core Domain, ou seja, o nosso domínio principal. Caso ele não existisse não haveria sentido para todo o restante existir. Seria como a Netflix sem filmes e séries ou uma fábrica de automóveis que não tem carros.

Por outro ângulo, ainda observando através da nossa lanterna, nós também temos alguns pontos importantes para definir.

O Core Domain é o coração do negócio e também o diferencial competitivo da empresa. Normalmente quando pensamos em domínio, isso também compõe o diferencial de toda a concorrência. Se não houvesse diferencial e toda a complexidade fosse banal, nós simplesmente usaríamos softwares de prateleira.

Domínios de Suporte

Diferente do Core Domain, também existe o que chamamos de domínio de suporte. Eles apoiam o domínio principal no dia a dia e apesar de não ser o domínio principal, eles auxiliam o negócio a possuir seus diferenciais competitivos e garantir que tudo funcione plenamente.

Se tivermos um e-commerce, por exemplo, precisaremos de produtos, uma loja e a parte de checkout. Vamos ter também o centro de distribuição.

Não podemos pensar em e-commerce sem um centro de distribuição, porque, nesse contexto, isso pode ser perfeitamente um dos diferenciais da empresa, pois afeta diretamente na velocidade de entrega dos produtos vendidos. Nesse caso, poderíamos considerar o centro de distribuição como um subdomínio de suporte, pois ele viabiliza a operação do domínio principal.

Domínios e Subdomínios Genéricos

Os subdomínios genéricos dão apoio a todo o sistema, mas geralmente não agregam tanto diferencial competitivo para a empresa. E vale mencionar que algumas empresas usam com frequência “softwares de prateleira” como parte de seus subdomínios genéricos.

Contudo, se você observar, esses “softwares de prateleira” ajudam na rotina da empresa, porém normalmente são facilmente substituíveis.

Espaço do problema vs espaço da solução

Sabemos que o DDD está relacionado à tentativa de resolver um problema, isto é, quando utilizamos esse recurso pensamos em um problema e em como tratá-lo, para depois resolvê-lo. Desse modo, podemos dizer que, ao desenvolvermos uma solução, temos um espaço de problema e um de solução. Assim, é importante entendermos a relação entre essas áreas, entre os subdomínios e outros elementos.

Primeiramente, vamos imaginar que temos um quadrado chamado de “espaço problema”. Podemos defini-lo como uma visão geral do domínio e suas complexidades. Logo, quando temos algo para resolver, começamos com uma ideia geral do que precisamos fazer. Em seguida, entendemos as principais complexidades do que precisamos trabalhar. Depois disso, podemos separar esse domínio maior em subdomínios, ainda nesse “espaço problema”. Quando temos um subdomínio e uma ideia geral do que está acontecendo, saberemos qual dificuldade vamos enfrentar.

Por outro lado, vamos imaginar que temos outro quadrado. Neste segundo, poderemos entender o problema e organizá-lo para que possamos encontrar possíveis soluções. Chamaremos este segundo ambiente de espaço da solução. Assim, podemos separar o domínio e suas complexidades para fazermos a modelagem desse domínio. Então, se no espaço do problema temos o domínio de forma geral, no espaço da solução temos tudo para resolvê-lo. Lembrando que, ao falarmos sobre DDD, um dos grandes pilares é conseguir fazer essa modelagem de domínio. Pois o domínio é o problema do negócio e a solução é conseguir modelar esse domínio para desenvolver a aplicação de maneira sustentável.

Assim, quando falamos em “problema vs solução” nos referimos a esse processo todo: de modelar, de separar e de transformar os subdomínios em contextos delimitados. Dessa maneira, conseguimos “atacar” o software bem no seu coração. E partindo disso, começaremos a perceber que grande parte do que fazemos com DDD é trabalhar exatamente nesses contextos delimitados.

Contexto delimitado

Ao separarmos os problemas em subdomínios, para solucioná-los, precisamos delimitá-los. Cada contexto delimitado vira um subproduto que teremos que resolver e trabalhar. Basicamente, veremos um problema e começaremos a modelá-lo, ou seja, iremos separá-lo em partes menores e as delimitaremos em pontos que servirão para começarmos a nos organizar. Feito isso, podemos desenvolver no espaço da solução. Desse modo, temos o domínio modelado e os contextos delimitados. Assim, conseguimos entender o que precisamos fazer, e qual é a prioridade para cada área.

Então, todas as vezes que começamos a falar sobre DDD, estamos fazendo uma exploração do domínio para conseguirmos iniciar sua modelagem. Essa modelagem, no final das contas, sempre será o entendimento do problema do subdomínio. Normalmente esses subdomínios vão se tornar contextos delimitados, isto é, será o local que verificaremos os problemas específicos para resolvê-los.

O que é contexto delimitado?

Antes de prosseguirmos com nossos estudos, é importante sabermos que a grande maioria das literaturas sobre o DDD está presente em livros, e materiais, da língua inglesa. Logo, ao fazermos buscas teremos resultados melhores se usarmos terminologias em inglês. Assim, é essencial sabermos que a terminologia

usada para contexto delimitado, em inglês, é “bounded contexts”. Sabermos isso, como já dissemos, é interessante para que nossas pesquisas sejam mais eficazes.

Para definir o que, de fato, é o contexto delimitado e sua importância dentro do nosso processo, vamos utilizar a definição de Vernon. Ele nos diz que bounded contexts é uma divisão explícita dentro de um domínio, isto é, dentro de um modelo de domínio. Um bounded pode ser considerado uma fronteira, ou seja, um limite. Então, quando falamos em contexto delimitado significa que temos uma divisão explícita de uma parte específica do domínio que estamos modelando. Apesar disso fazer muito sentido, falar pode parecer, de certo modo, muito genérico, por isso Vernon nos fala algumas formas para conseguir fazer essa delimitação. Uma delas é por meio da linguagem de termos e frases que são utilizados como dentro da comunicação de determinados contextos. Nesse caso, usamos os estudos relacionados à linguagem ubíqua, também conhecida como linguagem universal ou onipresente. Assim, tudo que é específico daquele negócio, desde a forma como as pessoas se comunicam, até a forma como os problemas são resolvidos, será considerado para fazermos a delimitação do contexto.

Perceberemos, ao longo de nossos estudos, que dentro de um negócio um dos “ingredientes” mais fortes é a linguagem. Quando notarmos todas as pessoas falando a mesma língua, provavelmente elas farão parte do mesmo contexto. Quando esse linguajar começar a mudar, teremos o indício de que estamos cruzando a fronteira e entrando em outro contexto delimitado. Ou seja, a

linguagem que é utilizada dentro do modelo é um dos grandes indícios que nos possibilita perceber em qual contexto estamos.

Com esses conceitos iniciais, já conseguimos perceber que raramente fará sentido trabalharmos utilizando o DDD em um sistema de "padaria", por exemplo. Isso porque, o contexto desse sistema é tão pequeno que todos já falam a mesma língua. Então, não faz sentido modelarmos o seu domínio por ser algo pequeno. Fora isso, sabemos que esses sistemas não são complexos. Todavia, seria diferente se tivéssemos um negócio de uma "padaria industrial". Neste segundo caso, teríamos diferentes fornecedores e diversas funções, isto é, diversos departamentos com linguagens próprias em cada setor.

Contexto é rei

Quando conseguimos compreender o que é um contexto, é mais fácil entendermos como as delimitações ocorrem. Podemos usar a regra do "contexto é rei" para fazermos essa definição. Tal regra nos diz que o contexto sempre vai determinar os mais diversos aspectos no processo de desenvolvimento de uma solução. Por exemplo, determinará qual área da empresa trabalharemos, o tipo de problema que iremos resolver e, essencialmente, a linguagem que será utilizada naquele contexto.

Para compreendermos como isso funciona de maneira prática, podemos imaginar a seguinte situação: temos dois contextos delimitados chamados de "ticket". Estes têm uma aplicação sendo modelada entre eles, isto é, no meio deles. O

primeiro ticket está em um subdomínio denominado de “venda de ingresso”. E dentro dessa área temos o ticket emitido para venda. Por outro lado, o segundo ticket está em um subdomínio chamado de “suporte ao cliente”. Observamos que, neste caso, temos duas palavras exatamente iguais, mas que estão representando significados diferentes. Isso significa que, claramente, estaremos em contextos delimitados diferentes. Assim, é evidente que no primeiro caso queremos falar sobre vendas e no segundo sobre um suporte dado ao cliente, mas ambos são escritos como “ticket”. Ainda com esse exemplo, vamos imaginar que estamos desenvolvendo um sistema monolítico. Logo, precisamos criar a entidade ticket. Lembre-se que teremos o ticket da venda de ingresso e o de suporte. Desse modo, precisaremos criar entidades diferentes para fazer uma separação e modularizar o sistema (criar áreas diferentes e adaptar o sistema). Isso porque o nosso sistema tem que trabalhar baseado em contextos, pois se não fizermos dessa maneira teremos uma entidade chamada “ticket” com pontos extremamente diferentes. Então, o contexto sempre vai ser rei. E quando temos a mesma palavra com significado diferentes, provavelmente estamos em contextos diferentes.

De modo semelhante, quando temos duas palavras diferentes, porém, com mesmo significado, provavelmente estaremos em contextos diferentes. Por exemplo, a história da “francesinha” do banco. Digamos que a área da contabilidade utiliza a entidade “relatório de boletos pagos”, mas na área dos bancários que estão na agência, chamam isso de “francesinhas”. Sabemos que os dois representam a mesma coisa, assim, é bem provável que essas palavras estejam em contextos diferentes.

É importante sabermos que em algum momento palavras iguais com significados diferentes, ou palavras diferentes com significados iguais, precisam “conversar”. Por isso, no próximo tópico falaremos um pouco sobre como essas relações podem acontecer.

Elementos transversais

É comum que as diferentes áreas de uma solução estabeleçam algum tipo de relação, apesar de existir uma delimitação entre elas. Isso acontece de modo transversal, quando as entidades, de todos os lados, conversam, mesmo tendo perspectivas diferentes.

Por exemplo, vamos imaginar que temos um “cliente” que está em duas áreas: na área de vendas de ingresso e na área de suporte ao cliente. Nesse caso, teremos o mesmo cliente em contextos diferentes. Mas é possível percebermos uma correlação entre essas áreas. Quando trabalhamos com cliente em venda de ingressos, estamos preocupados com o evento, com o ticket local e com o vendedor. Por outro lado, quando trabalhamos com cliente na área de suporte, estamos preocupados com o departamento que vai ser atribuído para dar o suporte, com o responsável pelo retorno ao cliente, com o ticket e com as dúvidas dos clientes. Assim, podemos perceber o quanto a perspectiva muda quando mudamos de contexto. Isso pode gerar uma confusão enorme na cabeça das pessoas que vão desenvolver a solução. Pois nós, desenvolvedores, temos a tendência de pensar que tudo é a mesma coisa, com a mesma perspectiva. Isto é,

geralmente percebemos tudo de modo unilateral.

Ainda com o exemplo da palavra cliente, imagine que precisaremos modelar uma entidade. Primeiro criamos uma classe de clientes, então vamos criar o ID e o nome do cliente. Depois disso, vemos que o cliente vai poder comprar ingressos. Logo, colocamos da seguinte forma no código, “ticket: locais que esse cliente comprou, vendedores que já venderam para ele, os eventos que esse cliente já fez” . Deste modo, modelamos a área de vendas. Porém, ao percebermos que esse cliente pode ter ticket de suporte podemos colocar “ticket: a dúvida que o cliente abriu, departamentos que colaboraram para ele e quais os responsáveis pelo suporte dado ao cliente”. É possível percebermos que a classe de cliente ficou enorme, pois ela está tentando atender diversos contextos onde o cliente existe. Podemos concluir, então, que isso é uma “loucura” já que o cliente pode participar de diversos contextos. Imagine a pessoa desenvolvedora ter que criar apenas um arquivo para modelar tudo isso como uma coisa só.

Assim, quando temos contextos diferentes, mesmo que a entidade seja a mesma, precisamos modelá-la de acordo com aquele contexto. Devemos fazer isso mesmo em um sistema monolítico, pois se não delimitarmos a aplicação, nosso arquivo irá virar um “monstro”. Então, mesmo que tenhamos um único cliente, é extremamente necessário nos atentarmos a essas delimitações, pois caso precise quebrar a solução em microsserviços teremos que reescrever tudo, por termos uma única classe que está servindo para todos os lados do sistema. Ter essa ideia de perspectivas diferentes faz muita diferença ao desenvolvermos um sistema.

Podemos colocar essas ideias, até aqui teóricas, em prática através de um mapeamento de contexto. Com esse recurso, fazemos uma modelagem estratégica das parte do domínio de nossa aplicação. Dedicaremos um tópico para explicarmos como isso funciona.

Visão estratégica

Quando modelamos um software é necessário termos uma visão estratégica de como as coisas estão se encaixando. Podemos fazer isso destacando aspectos de um espaço problema.

Sabemos que buscamos entender o domínio de forma geral. Assim, conseguimos separar os aspectos em partes (áreas) para depois convertê-las em delimitação de contextos. Porém, não podemos nos esquecer que esses contextos invariavelmente irão conversar., ou seja, eles irão se complementar e, eventualmente, um vai servir ao outro. Logo, nesse ponto, precisamos ter uma visão estratégica, ainda que, num primeiro momento, seja de forma superficial. O olhar estratégico é exatamente esse que falamos anteriormente, a visão geral da solução e de como as partes poderão conversar. Dessa maneira, saberemos, inclusive, como organizar o time durante todo o processo de modelagem, organização e desenvolvimento do software. Precisamos entender como criar uma modelagem estratégica para ter uma visão mais de cima.

Por meio de um context map podemos conseguir ter essa visão. Isso acontece pois, utilizando esse recurso, conseguimos mapear nossos contextos e, desse

modo, teremos uma compreensão mais clara de como os relacionamentos entre as áreas de uma aplicação acontecem.

Context mapping na prática

Fazer um context mapping é uma forma de modelar estrategicamente um software. Este é um mapeamento que nos permite visualizar como acontecem as relações entre os contextos de um sistema.

Para entendermos como isso funciona na prática, vamos imaginar que temos um ambiente de um negócio de vendas onde colocaremos todos os nossos contextos. Neste ambiente, teremos a modelagem do nosso domínio com seus contextos delimitados. Podemos deduzir que o "core business" é a área de venda de ingressos online. Mas, além dessa área, teremos também a de suporte ao cliente, a de vendas de ingressos offline (através de parceiros) e a de pagamentos. É importante dizermos, antes de prosseguirmos com o exemplo, que a área de vendas online tem peso diferente da área de vendas offline para o negócio. A primeira seria responsável por 80% do funcionamento do negócio. A segunda seria algo extra, fruto da parceria com shoppings, lojas, casas noturnas, etc. Porém, as duas são partes essenciais para o funcionamento do negócio.

Continuando com nosso exemplo, podemos imaginar esses quatro contextos delimitados bem claros: vendas online, vendas offline, suporte ao cliente e pagamento. E dependendo da organização da empresa, podemos criar um time para cada um desses contextos. Esses times irão falar e se organizar de maneira

específica para que cada um consiga atender as necessidades de seus próprios departamentos. Em alguns casos, é possível que haja a necessidade de termos um domain experts para nos auxiliar na resolução de problemas específicos daquela área.

Tanto a área de vendas de ingressos online quanto a de vendas offline tem um modelo de parceria. Isso porque ambas fazem o que a empresa precisa para se manter no mercado, isto é, elas exercem a principal função da corporação. Obviamente, uma é mais o core business do negócio do que a outra. Porém, mesmo que a venda offline seja uma segunda fonte de receita, ainda assim essas duas áreas precisam vender. Dessa forma, provavelmente, precisam estar integradas ao mesmo sistema. Chamamos a relação que acontece entre esses dois contextos de partnership. O que significa que esses dois times trabalham em conjunto para que o resultado seja satisfatório para os dois.

Então, ao vendermos ingressos online, criamos uma API para que o sistema do shopping possa consumir. Mas o sistema de vendas online também irá consumir a API de quem está vendendo offline. Logo, podemos dizer que, nessa parceria, um contexto consome do outro.

Em algumas situações é possível criarmos uma espécie de núcleo compartilhado. Porém, caso nosso projeto seja grande, isso pode acabar gerando diversos problemas. Por exemplo dificuldades relacionadas ao time ou à manutenção do sistema, pois um contexto afeta diretamente o outro.

Inicialmente a criação de um Sidecar para que a geração de ingressos seja facilitada pode soar como uma boa ideia. Mas na prática é possível que não funcione como esperado. Ainda assim, podemos ter uma relação entre esses dois contextos, pois essas áreas podem dividir ou criar um núcleo de sistema que os dois possam usar. Então, dizemos que essa parceria é uma forma de conexão entre um contexto e outro. Pois é através desse núcleo compartilhado que esses contextos fazem as mesmas ações

Além dessa relação de parceria, temos a relação entre cliente e fornecedor, porque a área de vendas de ingresso online vai precisar realizar pagamentos no momento da emissão da venda. Isto é, uma relação em que um contexto vai fornecer um serviço para o outro. Assim, este outro conseguirá realizar a transação necessária naquele momento. Como vimos neste exemplo, a área de pagamento vai oferecer um serviço para área de vendas de ingressos. Então podemos dizer que, nessa relação, vendas de ingressos é o cliente e a área de pagamentos é o fornecedor. Quando isso acontece, conseguimos criar uma relação de upstream e downstream. O primeiro vai fornecer o serviço e, consequentemente, vai ditar as regras do serviço implantado. Por exemplo: todas as vezes que a área de pagamento fizer uma melhoria, como adicionar uma nova parceria, vai informar para a área de vendas que, em tese, terá que se adaptar para que a área de pagamento funcione adequadamente e, assim, essas áreas consigam trabalhar de forma conjunta. Logo, dizemos que o downstream, que é o cliente, vai se adaptar para conseguir consumir algo do upstream.

O suporte ao cliente também pode ter uma relação de cliente e fornecedor, onde a área de vendas de ingressos (em que os clientes são gerados) pode fornecer informações para que a área de suporte ao cliente funcione. Desse modo, a área de vendas pode ser um upstream e a área de suporte um downstream. Isso vai depender de como a solução está sendo modelada.

É importante dizermos que não há uma regra específica que diga o certo e o errado em relação a esse tema, pois tudo vai depender muito de como a empresa funciona. Por exemplo, se a área de vendas usar a de suporte, teríamos uma situação invertida em relação à que apresentamos anteriormente. Então tudo depende de como vai ser a dinâmica do negócio. O que precisamos compreender é que pode existir relações de parceria, chamada de shared partnership, assim como pode existir também relações cliente e fornecedor entre os contextos.

Além disso, podemos utilizar um módulo para o serviço de pagamento, como uma API Gateway. Por ser um serviço externo, um gateway está em outro contexto, o que é perfeitamente aceitável. Por exemplo: podemos imaginar que essa área de pagamento vai usar uma Gateway chamada XPTO, que tem uma forma própria de trabalhar e sua API definida. Então essa Gateway vai ser o fornecedor e a área de pagamento vai consumir tudo isso. Nesse caso, a Gateway vai impor a maneira que a área de pagamento vai trabalhar, independente do que nós, desenvolvedores, queremos. De forma prática, imagine que teremos que trabalhar com o Banco Itaú, utilizando seus serviços de cartão de crédito para fazer cobranças. Dificilmente pediremos para que esse banco faça alterações

em seu sistema por nossa API trabalhar de modo diferente. O mais provável é que tenhamos uma relação conformista com uma empresa desse porte, a não ser que sejamos tão grandes quanto o Itaú. Neste caso, podemos conversar com o banco para que eles criem algo personalizado que nos atenda. Assim, dessa última maneira, a relação não será conformista.

Outro exemplo de como as relações entre diferentes contextos acontecem é o fornecimento de vídeos em streaming. Não podemos mudar a maneira como um provedor irá distribuir nosso vídeo. Mas podemos dizer que temos algumas exceções, como a Netflix. Empresas desse porte conseguem ter mais personificação por parte do provedor para atender suas necessidades. Então, reforçamos que a maneira como essas relações acontecem depende muito da empresa, do contexto, etc.

É perceptível que, quanto mais conformista é essa relação, temos tendência a nos amarrar em outro sistema. Por exemplo, se usamos um sistema CRM de terceiros, quanto mais o utilizamos, mais informações nossas esse sistema possui. É uma relação conformista e raramente o modo de trabalho poderá ser alterado com facilidade.

Eventualmente, em situações como a desse último exemplo, podemos fazer uma camada anticorrupção. A ACL é uma camada de interface que fica entre o nosso contexto e a gateway de pagamento. Caso precisemos mudar de gateway basta trocar o código da camada de anticorrupção. Assim, não precisamos alterar o código da parte lógica da área de pagamento em nossa aplicação. Então, funciona

como se fosse um adaptador que nos ajuda a minimizar esse problema que vem dos relacionamentos conformistas, pois uma vez que temos uma camada anticorrupção, ficará mais fácil conseguirmos realizar trocas de fornecedores.

Desse modo, o context mapping vai nos auxiliar a entender os relacionamentos entre os contextos e os times de uma solução. Além disso, através dele conseguiremos visualizar qual time fornece para qual time e quais trabalham em parceria. Isso possibilita que tenhamos uma visão mais clara e estratégica da aplicação (do negócio), algo extremamente importante, principalmente quando temos um sistema de grande porte.

Padrões e starter kit

É importante conhecermos diversos padrões de relações entre contextos para trabalharmos com context map. Por isso, neste tópico, nos dedicaremos a mostrar alguns desses padrões e, além disso, falaremos sobre um projeto que alguns desenvolvedores fizeram no Github.

Então, ao mapearmos um software é importante conhecermos os padrões que são utilizados nas relações entre as áreas da aplicação. Por exemplo: o padrão de Partnership, estabelecido por meio de uma parceria entre as áreas; o de Shared kernel, quando mantemos uma biblioteca para algo compartilhado entre os times; o Customer-Supplier Development, uma relação entre cliente e fornecedor (um faz o consumo do outro, ou seja, upstream e downstream); a relação

conformista, em que há uma conformidade de uma das partes e o ACL, uma camada de adaptação de interfaces que evita ficarmos presos em outro sistema.

Fora esses padrões, temos outros exemplos como, o Open host service, um padrão em que um contexto vai fornecer um serviço que estará disponível com determinado protocolo, como um GRPC; o Published language, onde a linguagem faz total diferença na hora que vamos nos comunicar; o separate ways, em que os contextos delimitados não vão mais se comunicar e cada um mantém seu próprio padrão como também o Big Ball of mud, um sistema muito comentado em livros por ter várias coisas misturadas e, por isso, torna-se comum termos que lidar com ele no dia a dia.

Por vezes, é possível que esses nomes tragam algum tipo de complicação nas relações, principalmente quando vamos fazer um contexto mapping. Então, gostaríamos de indicar um projeto no Github chamado de DDD Crew. Você pode acessá-lo aqui: https://github.com/ddd-crew/context-mapping

Dentro desse projeto, na parte de context map, conseguimos ver uma cheat sheet, exemplos de diversos padrões com uma imagem representando cada tipo de relação. Por exemplo, temos uma imagem que nos mostra o resumo do que é o Open Host Service, isto é, um Bounded Context que oferece a definição de uma série de serviços que serão expostos para outros sistemas. Além dessa imagem, vemos uma que resume o customer/supplier, em que o primeiro é o downstream e o segundo é o upstream. Assim, conseguimos ver a explicação de cada um dos padrões.

Podemos ver também, mais abaixo, quais são os tipos de relações, por exemplo se os contextos são mutuamente dependentes, se são free ou upstream e downstream.

Além disso, nesse site temos a possibilidade de acessar uma versão read only para utilizarmos no Miro. Assim, podemos criar novos documentos e utilizar todos os padrões para fazermos o mapeamento de nossos próprios projetos.

Arquitetura Hexagonal

Introdução à Arquitetura Hexagonal

Neste capítulo veremos alguns conceitos relacionados à Arquitetura Hexagonal. Com essas informações teremos uma visão mais clara na hora de programar nossas aplicações. Isso faz com que o processo de desenvolvimento de softwares de qualidade seja realizado de maneira mais fácil.

Por exemplo, imagine que precisamos desenvolver um sistema para um banco, que oferece serviços de empréstimos aos seus clientes. Baseado nos dados das pessoas, ele verifica o Score para calcular qual taxa de juros será cobrada para cada empréstimo realizado. Na hora de desenvolver a solução, é importante sabermos qual a melhor maneira de escrever esse processo. Antes de tudo, precisamos entender o problema que devemos solucionar para a empresa. Neste exemplo, é necessário encontrarmos a maneira mais adequada de calcular a taxa de juros do empréstimo para cada cliente. Tendo o problema claro, seguiremos para os próximos passos de uma programação: escolher uns frameworks, modelar o banco de dados, fazer uma API Rest e, assim, fazer os cálculos. Podemos dizer que esta é a maneira como costumamos desenvolver uma aplicação de

forma geral. Fazemos tudo baseado na frase “nossa função, como dev, é resolver problemas através do código”. De certo modo, podemos afirmar que essa frase é muito “rasa” para descrever nossa função, porque existe muita complexidade quando desenvolvemos um sistema. Inicialmente até teremos uma complexidade de negócio, isto é, relacionada diretamente ao problema que estamos sendo pagos para desenvolver, mas a criação de um sistema envolve também as complexidades técnicas que nós mesmos adicionamos para resolver o negócio. Entretanto, é possível afirmarmos que o problema de negócio é essa complexidade que dizemos ser inevitável, pois estamos sendo pagos especificamente para resolvê-la. Por outro lado, não devemos misturá-la com a complexidade técnica. Dizemos que essa complexidade são os aparatos técnicos que iremos utilizar para resolver o problema do negócio. Então, conseguimos perceber claramente que existem duas complexidades quando vamos desenvolver uma solução. Temos a complexidade de negócio e a técnica. Quando temos clareza sobre a existência desses dois tipos de complexidades, nosso trabalho flui de maneira mais tranquila. Por exemplo, é comum misturarmos regras de negócio, banco de dados, forma de comunicação etc. por não sabermos com qual complexidade estamos trabalhando ao digitar determinado arquivo. Fazendo isso, nós esquecemos que nossa principal função é proteger o negócio. Isso porque, colocar limites muito claros entre as complexidades evita que a complexidade técnica invada o negócio.

Se conseguirmos fazer essa separação, podemos trocar, conforme necessidade, essa complexidade técnica. Por exemplo, um framework por outro ou um sistema

de cache por um protocolo diferente do HTTP. Tal possibilidade nos ajuda a manter nossa aplicação coesa.

A separação de complexidades, faz com que a complexidade técnica seja utilizada apenas para acessar o negócio. Assim, teremos um software totalmente autocontido, com responsabilidades claras e facilmente portável (podemos mudar um framework, um banco de dados e a forma de se comunicar). Com esses conceitos em mente, sempre que formos digitar um código, podemos pensar e raciocinar se naquele momento estamos trabalhando com complexidade técnica (que nós adicionamos), ou se estamos trabalhando com complexidade de negócio (que é o que realmente temos que resolver).

Ainda neste capítulo, nossos estudos serão direcionados para que possamos compreender cada uma dessas complexidades. Pois quando trabalhamos com arquitetura hexagonal, naturalmente separamos o negócio do técnico. Fora isso, veremos também que essa breve introdução foi necessária para que nossa compreensão seja melhor quando falarmos, especificamente, sobre essa arquitetura.

A importância da Arquitetura de Software

A arquitetura hexagonal, também conhecida como “Ports and Adapters”, traz conceitos extremamente importantes para conseguirmos desenvolver softwares de qualidade. Isso porque, as ideias relacionadas a essa arquitetura podem nos auxiliar no processo de desenho da aplicação.

Antes de prosseguirmos nesse assunto, é interessante conhecermos alguns pontos importantes sobre arquitetura de software de modo geral. Quando desenvolvemos um sistema, alguns aspectos fazem com que seja necessário trabalharmos com arquitetura. Um deles é o crescimento sustentável da aplicação. Ou seja, conforme nós desenvolvemos o software, podemos melhorá-lo sem a necessidade de muitos retrabalho ao longo do tempo. Assim, nossa aplicação tem poucos débitos técnicos nesse processo. E, desse modo, ela pode gerar valor e "se pagar" para os seus investidores ao longo do tempo. Sabemos que, hoje em dia, é muito comum precisarmos refazer o software antes mesmo da empresa terminar de pagar por ele. Então, a organização precisa fazer novos investimentos para desenhar e desenvolver "tudo do zero", antes mesmo de ter o retorno financeiro que foi investido inicialmente. Mesmo que isso seja algo comum, devemos estudar meios que evitem prejuízo para os clientes.

A arquitetura de software está diretamente relacionada aos limites que um sistema precisa ter para garantir o seu crescimento sustentável. Ao arquitetar uma solução conseguimos manter a qualidade daquilo que está sendo produzido, para que gere o resultado esperado pela pessoa que investiu nessa aplicação.

Além de nos ajudar com a sustentabilidade da aplicação, os conceitos de arquitetura nos auxiliam a entregar um software que não seja orientado a framework. Isso significa que uma biblioteca não conduzirá a maneira que iremos escrever nosso sistema.

Essa prática é considerada um problema justamente pela dependência que

teremos ao estar “plugados” em um determinado framework. Quando essa dependência acontece não pensamos mais no problema do negócio, mas sim em como o framework vai resolver outras situações técnicas. Então, lembre-se das complexidades que falamos no tópico anterior. A complexidade de negócio não pode estar atrelada à complexidade técnica de nossas bibliotecas/frameworks. Se misturarmos as complexidades fazendo com que o framework resolva o problema do negócio, algo pode estar errado em nosso modo de desenvolver a aplicação.

Uma vez que arquitetamos nosso software, temos como resultado a criação de um “lego”, isso mesmo quando fazemos uma arquitetura mais abrangente, como microsserviços; ou quando trabalhamos pontos mais específicos de design do software. Quando as partes do nosso sistemas são “legos”, conseguimos trocá-las sem quebrar a estrutura desse software. Mas isso só é possível se tivermos uma aplicação bem desenhada.

Então, podemos dizer que a arquitetura está relacionada ao futuro do software. Pois ao arquitetar um sistema pensamos nele funcionando perfeitamente depois de vários anos no ar. Porém, para que isso seja possível não podemos desenvolver apenas um CRUD, algo que qualquer dev faz. Hoje em dia, por termos várias ferramentas capazes de gerar esses CRUDs, é comum que os devs pensem automaticamente em fazer apenas isso. Quando recebemos a solicitação para pensar na modelagem de dados a cultura de criar apenas CRUDs nos impulsiona a pensar em modelarmos o banco de dados antes mesmo de saber como iremos

trabalhar para resolver determinadas complexidades do escopo do software. Todavia, é importante termos em mente que desenvolver um sistema vai muito além de fazer esses CRUDs. Isto é, exige um entendimento do problema e um desenho bem estruturado de toda a aplicação.

Ciclo de vida de um projeto

Um projeto de software tem diversas fases em seu ciclo de vida. Nos próximos tópicos, iremos descrever dez etapas que envolvem um projeto sem sustentabilidade. É importante dizermos que, com esse exemplo, não queremos julgar o trabalho de nenhum desenvolvedor. Ou seja, mesmo que o sistema não seja sustentável, não podemos afirmar que a pessoa desenvolvedora fez necessariamente uma programação ruim. Assim, reforçamos que é muito comum uma aplicação não se sustentar ao longo do tempo, ainda que a pessoa desenvolvedora escreva bem. Por outro lado, isso pode significar que ela tomou más decisões de arquitetura. Então, mesmo que ela escreva bem, por dentro o desenho pode estar mal feito. E conforme o tempo passa é necessário usar uma "borracha" na aplicação, fazendo com que existam várias marcações que geram diversos problemas na estrutura do sistema. Visualizar essas fases pode nos ajudar a compreender como tudo isso acontece na prática. O objetivo é refletirmos sobre nossas decisões, para que possamos desenvolver aplicações totalmente sustentáveis.

Fase 1

Na primeira fase do desenvolvimento de uma aplicação, escolhemos o banco de dados, criamos os cadastros, validações e selecionamos um servidor web. Depois disso, é necessário criar Controllers e Views para mostrar as formas de apresentar os dados. Além disso, é necessário pensar na maneira que faremos a autenticação e os Upload dos arquivos.

Ainda nessa fase, geralmente o cliente pede por determinada quantidade de cadastro, sinaliza o que deve ficar na web e solicita login e senha. Esse é um processo simples, que qualquer framework básico, de qualquer linguagem, pode resolver. Afinal de contas, são vários CRUDs com algumas validações e autenticações.

Fase 2

Na segunda fase, o cliente solicita que façamos algumas mudanças. Normalmente, precisamos colocar regras de negócio importantes para atender os pedidos desse cliente. Como essas regras farão parte de momentos do cadastro, certas ações devem acontecer nesse primeiro acesso dos usuários. Isso irá depender dos objetivos do negócio. Fora isso, criaremos algumas APIs para deixar disponível aos parceiros. E consumimos APIs para pegar alguns dados. Depois disso, precisaremos de autorização, isto é, cada usuário poderá fazer cadastro e login de uma forma diferente. Em seguida, precisaremos de alguns relatórios. Por fim,

é necessário guardar alguns logs (informações para verificar possíveis erros).

Fase 3

Nesta fase, o sistema começa a ter mais acessos, por isso precisamos fazer Upgrade de hardware, ou seja, precisamos realizar uma escala vertical para melhorar o hardware e segurar a quantidade maior de acessos ao software. Depois, começaremos a trabalhar com cache, consumindo API de parceiros. Isso faz com que o sistema fique sujeito a algumas regras externas. Por exemplo, ao fazermos um checkout da empresa “X” de gateway de pagamento precisamos estar prontos para que nossa aplicação esteja sujeita às regras dessa API externa. Além disso, mais relatórios serão necessários para analisarmos os eventos, caso aconteça algum erro na aplicação.

Fase 4

Na fase quatro, o sistema tem ainda mais acessos e precisa de mais Upgrade de hardware. Consequentemente, precisamos de mais relatórios e consultas. Além disso, surgem algumas dificuldades no banco de dados, devido ao aumento dos acessos nele. Então, é preciso gerar alguns comandos, podendo ser via linha de comando mesmo, para conseguir fazer alguns relatórios ou para exportar algumas informações. Durante o processo, alguns problemas podem ter acontecido, ou até mesmo algumas mudanças na empresa, por isso, é necessário criar a versão 2 da API, mas a versão 1 precisa ser mantida.

Fase 5

Na fase cinco, precisamos escalar o software horizontalmente. Assim, ao invés de aumentar o hardware precisamos aumentar a quantidade de máquinas, pois sabemos que não existe hardware infinito. Ao fazer isso, percebemos alguns problemas. Por exemplo, vemos a necessidade de trabalharmos com sessão externa, porque o sistema está rodando em máquinas diferentes.

Por causa dessas mudanças, precisaremos fazer algumas adaptações importantes. Por exemplo, ter uma sessão no banco de dados e/ou no servidor de cache para que todos os servidores consigam capturar a mesma sessão. Depois disso, veremos outro problema. Dessa vez relacionado ao Upload. Antes esses arquivos ficavam na mesma máquina, mas agora temos várias máquinas e, por isso, é necessário mudar o sistema de Upload. Podemos usar uma nuvem da Amazon S3 para conseguirmos baixar esses arquivos. Além disso, os Uploads antigos devem ser migrados. Conseguimos imaginar como seria trabalhoso fazer tantas mudanças no banco de dados dessa aplicação. Fora isso, não podemos deixar nossos arquivos totalmente públicos. Precisamos de uma regra de assinatura desses arquivos na S3 para que apenas o cliente consiga acessar, quando clicar uma vez no arquivo, por exemplo. Então, teremos que fazer muita refatoração somente para conseguir escala horizontal. Geralmente, quando começamos a desenvolver um software não pensamos nisso, assim, esses problemas começam a surgir e temos dificuldades em resolver tantas situações complexas.

Continuando com o processo de evolução do sistema, devido à quantidade crescente de acesso, precisamos fazer autoscaling. Ou seja, fazer com que os servidores cresçam automaticamente. A situação começa ficar mais complicada, pois existem muitas mudanças no software. Neste momento, temos que colocar um pipeline de integração contínua e um deploy contínuo na solução.

Fase 6

Na fase seis do desenvolvimento da aplicação, surge a necessidade de usarmos GraphQL. Por trabalharmos com uma interface diferente, queremos dar mais força ao front-end. Então, criamos parte da nossa API utilizando GraphQL. Lembrando que a API estava trabalhando de um jeito, e quando colocamos endpoint com GraphQL pode surgir alguns bugs. Pois, afinal de contas mudamos o formato mantendo a versão 1 e 2 da API. Depois disso, é provável que tenhamos problemas com os logs. Por exemplo, nossa aplicação estava rodando e gravando cada log em uma única máquina. Ao trabalharmos com número maior de máquinas, é dificultoso encontrar o problema pois temos que acessar todas as máquinas individualmente para verificar os seus logs e identificar qual delas gerou aquele bug. Por vezes não conseguimos identificar o problema em nenhuma máquina, então descobrimos que, por fazermos o autoscaling, a máquina onde o bug surgiu foi a que escalamos e, consequentemente, foi removida. Assim, perdemos os logs e não conseguimos fazer uma análise do problema. Vemos a necessidade de ter uma ferramenta onde todos os logs fiquem

guardados no mesmo local, tirando essa dependência de analisar as máquinas individualmente para procurar em qual delas o bug teve origem.

Depois disso, precisamos integrar o sistema com o novo CRM. Neste momento, já conseguimos perceber que temos algumas regras de negócio misturadas. Assim, precisamos mudar nossa SPA (frontend) para React e começar a refatoração.

Fase 7

Na fase sete, percebemos que existem certas inconsistências com os dados do CRM. Um dos problemas pode estar relacionado à dificuldade de comunicação. Por exemplo, ao realizarmos um checkout do produto, queremos registrar essa venda no CRM, mas por algum motivo os dados do sistema não chegam até lá. Trabalhamos com Rest, e fazemos um HTTP Request no CRM. Então, na hora que mandamos os dados, simplesmente não são registrados. Nesse caso, alguns motivos possíveis são: o CRM poderia estar fora do ar naquele momento, o sistema não enviou os dados corretamente ou tivemos um problema temporário na rede. Assim, os números do sistema não batem com os dados do CRM.

Sabendo que o mundo está migrando para Docker container kubernetes, vemos o quanto é bom para nossa aplicação colocá-la para rodar em container. Como iremos migrar para container precisamos pensar na maneira que faremos o processo de CI/CD, isto é, a integração contínua e o deploy contínuo. Depois disso, precisamos de um container registry para trabalhar e fazer os processos.

Além disso, nesta fase, pode acontecer um "pico" de processamento, seguido da "morte" do sistema. Isso porque, a aplicação usa muita memória e ao executar uma operação ela "morre", mas o container não. Normalmente, ele tem uma memória pequena, por isso rodamos vários e percebemos que alguns containers começam a "morrer" pois em alguns momentos de fato não foram bem desenvolvidos. Um loop mal feito ou vai gerar algum arquivo que roda muito tempo e "mata" o container, ou perderá algumas operações. Assim, precisamos dos logs dos containers para verificar onde estão os problemas.

Por essas situações, já temos o pensamento de que o sistema está virando um legado. Devido a quantidade de mudanças nos cadastros, nas autenticações etc. está sendo cada vez mais difícil mantê-lo no ar. Porém, ainda conseguimos.

Fase 8

Na fase oito, mesmo percebendo que o software está virando um legado, não queremos refazê-lo. Então, a melhor solução é criar microsserviços em volta desse sistema. Agora, precisamos fazer a comunicação entre esses microsserviços. Para isso, a opção mais simples é fazer com que todos acessem o mesmo banco de dados. Mas com isso começamos a ter alguns problemas de tracing. Acessamos um microsserviço que se comunicou com outro, depois disso, tivemos um problema em nossa aplicação e de repente não conseguimos identificar em qual microsserviço esse problema aconteceu. O que vai dificultar nosso trabalho nesse sistema. Algumas aplicações começam a ficar mais lentas do que eram

antes de trabalharmos com microsserviços. Isso acontece porque agora teremos uma dupla latência, isto é, não bate tudo em um único sistema. Caso isso não esteja bem otimizado, o software fica bem mais lento do que o comum. Consequentemente, o custo financeiro do sistema ficará elevado. Pois temos vários containers, vários CI/CD, muito mais espaço sendo guardado, muitos sistemas de logs e muitos sistemas de cache. Tudo isso, "pesa no bolso da empresa".

Fase 9

Na fase nove, não temos mais condições de trabalhar com containers, pois o custo desses serviços está além do esperado. Assim, vamos para Kubernetes. Se ainda não soubermos trabalhar com essa tecnologia, é necessário bastante esforço para aprendermos a maneira adequada de implementá-la em nossa solução.

Depois disso, precisamos mudar novamente o processo de CI/CD. Isso faz com que os problemas de inconsistência fiquem ainda mais evidentes e o processo de resiliência do software mais difícil. Temos falhas com o CRM, pois os microsserviços não estão com resiliência suficientes para funcionar como esperado. Assim, precisamos separar os bancos de dados e trabalhar com mensageria (filas). Nesse momento, se não tivermos experiência com filas é comum perdermos várias mensagens. Por exemplo, se não soubermos trabalhar com RabbitMQ. Perdemos mensagens e não conseguimos identificar o que aconteceu com elas. Isto é, o Dead Letter não foi criado e, por isso, temos a

necessidade de contratar consultorias para nos ajudar, elevando ainda mais o custo para manter a aplicação rodando.

Fase 10 (fase final)

Finalmente chegamos à fase dez. Nesta fase, precisamos usar a nossa imaginação!

O que foi colocado neste tópico não pode ser considerado um exagero perto do que acontece nos dias de hoje. Percebemos que, conforme essas evoluções acontecem, o sistema vira um "rabisco de borracha". O software aparenta ser um legado, por isso ninguém mais quer "por a mão" nele. Lembrando que os desenvolvedores que estão escrevendo esse sistema não são necessariamente ruins. Provavelmente, eles estão estudando e tentando evoluir, colocando tecnologias novas. Porém, rastros vão sendo deixados para trás. Isto é, vários backlogs e débitos técnicos ficam para trás nesse processo de evolução.

Então, se a arquitetura não estiver definida, ou seja, se não pensarmos na sustentabilidade do software no dia zero, essas dificuldades continuarão acontecendo até chegar o momento que não teremos mais alternativas, a não ser refazer todo sistema. Por isso, é importante arquitetarmos nosso software tendo a clareza do seu funcionamento para que isso não aconteça quando formos desenvolver.

Principais problemas

Neste tópico, faremos algumas reflexões sobre os principais problemas que aconteceram no ciclo do projeto que descrevemos anteriormente. Sabemos que, ao longo do processo de desenvolvimento desse sistema, muito estresse foi gerado para todos os que estavam envolvidos de alguma forma, desde os clientes até os profissionais que ajustaram as dificuldades que surgiram. Por isso, é importante analisarmos as possíveis falhas, para que tenhamos clareza de como poderíamos evitar boa parte desses problemas.

Primeiramente, percebemos a falta de visão de futuro do software. Normalmente, quando iniciamos um projeto temos uma ideia de que será apenas um "sisteminha", ou seja, algo de pequeno porte. Temos meia dúzia de cadastros e, por isso, esquecemos de olhar para um possível crescimento no futuro. Depois, precisamos fazer refatoração e vários processos para manter o sistema crescendo. Mesmo quando criamos nossos primeiros CRUDs, sempre devemos pensar que a solução poderá crescer.

Além disso, é essencial que os limites do projeto sejam bem definidos. Quando pensamos em arquitetura de software, surge a ideia de estabelecer limites. Esses limites são importantes para que uma coisa não "atropele" a outra. Isso significa que o negócio não deve se misturar com o framework. Bem como o framework não deve se misturar com o banco de dados, e assim por diante, para evitarmos que o técnico se torne parte do negócio.

Podemos observar, também, a necessidade de pensarmos na troca e adição de componentes. Quando desenvolvemos um software é importante separarmos a aplicação da complexidade técnica. Fazendo isso, provavelmente conseguiremos criar adaptadores, inclusive para o banco de dados, que é extremamente difícil e raro de substituir. Se não conseguimos "trocar" esse banco de dados, significa que nosso sistema está muito acoplado. Além disso, podem ter outras áreas do software que precisem ser alteradas. Então, essa possibilidade realmente precisa existir.

Fora essas questões, a dificuldade de fazer uma escala é outro problema que ficou bem claro. Começamos a desenvolver um software de uma forma, isto é, com a sessão, o Upload, o banco de dados, os logs e o cache no mesmo servidor. Quando precisamos escalar horizontalmente, temos que refazer muitas coisas. Por exemplo, concentrar os logs no mesmo lugar, colocar upload na nuvem, fazer cache ficar distribuído etc. Assim, mesmo sabendo que fazer um processo de escala sempre vai gerar problemas, é necessário termos uma visão de futuro prevendo escalabilidade. Dizemos que a possibilidade de uma escala deve ser pensado no dia 01 da criação de um sistema.

Fora isso, ao longo do ciclo do software, precisamos realizar várias otimizações. Isso deixou o sistema com muitos débitos técnicos. Sabemos que sempre pode surgir a necessidade de fazer ajustes que deixarão débitos, mas se tivermos limites bem definidos um débito não irá se juntar ao outro. Essa separação faz com que cada problema fique na sua área, evitando o acoplamento da aplicação.

Outra questão importante para nossa reflexão é a necessidade de estarmos preparados para mudanças "bruscas". Por exemplo, a mudança da gateway de pagamento da empresa. Precisamos pensar se realmente conseguiremos chavear a gateway em nosso sistema. Além disso, é importante estarmos preparados para gerar um endpoint GraphQL ou integrar um CRM "X", caso seja necessário. É essencial termos em mente que, ao desenvolvermos um software, essas mudanças bruscas acontecem. Então, se tivermos uma visão de futuro dessa solução, podemos criar camadas anticorrupção para que essas mudanças não afetem o negócio. Neste ponto, podemos lembrar da importância de separarmos as complexidades. Ou seja, precisamos pensar em alternativas para que essas alterações não afetem a complexidade do negócio. Isso porque, na maioria das vezes, essas alterações estão relacionadas diretamente à complexidade técnica do sistema.

Reflexões

Refletir sobre nossas práticas em nosso trabalho como pessoas desenvolvedoras é algo extremamente importante. Essa reflexão, deve ser feita frequentemente, assim, pode contribuir para que nossos softwares sejam cada vez mais sustentáveis. Abaixo, listamos algumas perguntas, relacionadas ao ciclo do projeto que conhecemos no tópico anterior, para nos auxiliar nesse processo de reflexão:

- Está sendo "doloroso" atualizar o software para fazer com que ele continue sendo desenvolvido?
- Os maiores problemas poderiam ser evitados? Já adiantamos que alguns sim, pois são evidentes no dia a dia e acontecem porque perdemos a visão geral do sistema.
- Mesmo com todas as mudanças, que exigem investimentos extras, o software está realmente "se pagando"?
- A relação com o cliente é boa, ou seja, estresses desnecessários foram evitados, levando em consideração todo o ciclo do projeto?
- O cliente teve algum tipo de prejuízo com a mudança brusca arquitetural? Por exemplo, alteração para microsserviços e outras integrações feitas no sistema.

Se fossemos novos na equipe, ajudaríamos os devs que iniciaram o projeto?

Além desses questionamentos, é importante pensarmos em qual ponto esse software se perdeu. Podemos dizer que ele se perdeu principalmente em dois momentos: no dia 01 e um pouco a cada dia. Inicialmente faltou a visão de futuro do software desde o dia 01 do seu desenvolvimento. Depois disso, sabemos que nenhum sistema se perde "do nada", essa perda faz parte de um processo diário. É semelhante ao atraso de um projeto. Isso acontece um dia de cada vez. Então, não é a partir do dia que percebemos o "erro" que o software realmente deu errado. Sem práticas adequadas dia após dia fica mais difícil manter aquele sistema no ar.

Por vezes, o software foi bem feito inicialmente mas se perdeu ao longo dos anos. Então, se depois de 4 anos outra pessoa analisar essa aplicação, poderá concluir que o desenvolvedor é ruim. Pois, só vê os débitos deixados por todas as mudanças realizadas.

Antes de tudo, é importante termos em mente que aquele trabalho exigiu muito esforço e dedicação de quem escreveu. Assim, nós, desenvolvedores, não podemos julgar o trabalho de outras pessoas. Porque não sabemos quais situações permitiram que o sistema se tornasse um "legado". Talvez tenha acontecido situações que fugiram totalmente do controle de quem estava desenvolvendo.

As informações que vimos até aqui nos ajudarão a entender mais adiante o funcionamento da arquitetura hexagonal. Assim, com entendimento desses conceitos e do contexto em que essa arquitetura se encaixa, veremos sua importância no desenvolvimento de sistemas de qualidade e sustentáveis.

Arquitetura vs design de software

Nas comunidades de desenvolvedores, muitos consideram que arquitetura e design são a mesma coisa. Isso pode fazer com que surjam algumas polêmicas quando esse assunto é comentado. Faremos a definição dessas duas áreas para compreendermos se, de fato, são áreas diferentes.

Em seu livro, Elemar Júnior, fala com muita clareza sobre arquitetura e de design: "Atividades relacionadas à arquitetura são sempre de design. Entretanto,

nem todas as atividades de design são de arquitetura. O objetivo primário da arquitetura de software é garantir que os tributos de qualidade, restrições de alto nível e os objetivos do negócio sejam atendidos pelo sistema. Qualquer decisão de design que não tenha relação com esses objetivos não é arquitetural. Todas as decisões de design para um componente que não sejam 'visíveis' fora dele, geralmente, também são."

Conforme essa citação, podemos dizer que arquitetura e design são completamente diferentes. A arquitetura está em um nível mais alto do que o design. Então, na teoria dos conjuntos, o design está dentro da arquitetura. Por isso, quando tomamos decisões arquiteturais algumas vezes elas implicam decisões de design. Ou seja, por vezes tomamos uma decisão arquitetural que indiretamente força uma decisão de design de software.

Por exemplo, vamos imaginar uma visão de alto nível. Nela, todo sistema tem que gerar logs, que ficam armazenados em um único local para que o processo de observabilidade seja mais fácil. Então, os logs precisam trabalhar de forma distribuída. Essa é uma visão arquitetural. Por outro lado, quando precisamos desenhar a saída dos logs por terminal ao invés de deixá-los gravado em arquivo, então, teremos uma decisão de design do software. Perceba que até o momento não estamos falando qual é a linguagem, e nem qual é o componente. Porém, algumas vezes, precisamos definir a linguagem e a biblioteca que o desenvolvedor deverá usar. Isso pode acontecer quando fechamos contrato com um parceiro, por decisão arquitetural. Além disso, podemos definir que o log

utilize determinada biblioteca de uma maneira específica. Essa é uma decisão arquitetural, mas também gera mudança no design. Então, é nesse momento que muitas pessoas consideram arquitetura e design como a mesma coisa, pois uma decisão arquitetural impacta indiretamente na de design. Todavia, é importante termos em mente que muitas decisões de design não têm relação nenhuma com a arquitetura do sistema.

SOLID é um exemplo em que tratamos diretamente do design do software. Pois temos uma visão micro da forma que esse sistema será desenhado. Isso pode acontecer quando fazemos uma classe ter uma única responsabilidade no momento de sua criação. Ou trabalhamos com injeção de dependência. Essas decisões não têm nenhuma relação com a arquitetura do software.

Para que fique claro a diferença entre esses dois tópicos, vamos dar um exemplo mais simples e relacionado com uma atividade prática do dia a dia. Imagine que precisamos fazer a pintura de um pôr do sol em um quadro. As primeiras decisões que tomaremos, serão as de arquitetura. Por exemplo, especificações da pintura, o tipo de tinta a óleo que será usada, os limites que não podem ser ultrapassados na hora de pintar etc. Por outro lado, quando iniciarmos a pintura, teremos que decidir quais tintas não podem ser misturadas, pois sairá uma cor diferente. Analisaremos qual a maneira mais adequada para fazer essa mistura das cores e como iremos desenhar o sol para que tenha o reflexo adequado da sombra, por exemplo. Essas decisões podem ser consideradas relacionadas ao design da pintura. Assim, percebemos que existem diferenças muito claras entre

arquitetura e design. Mesmo que a decisão arquitetural tenha impacto indireto no design não podemos considerar que são a mesma coisa.

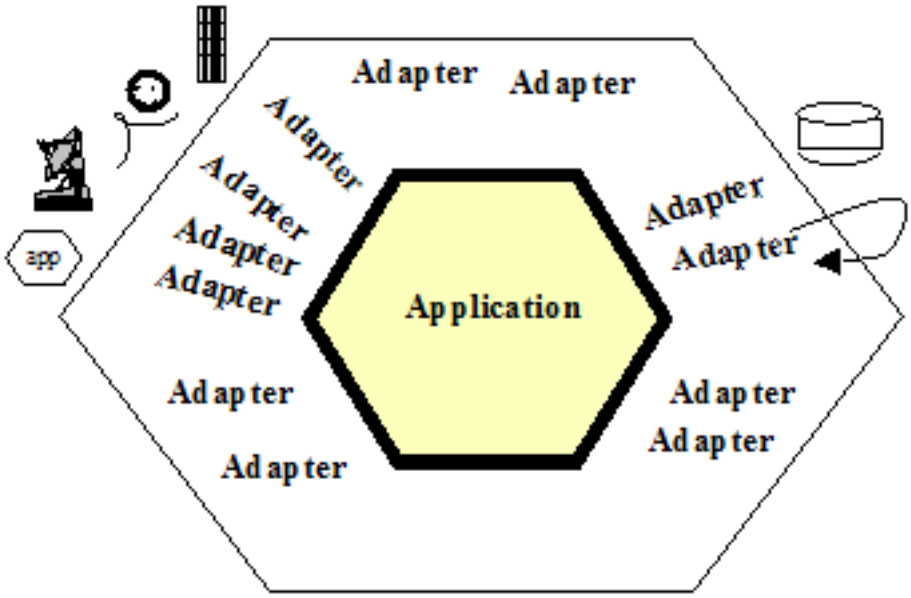
Então, as decisões que ajudam na qualidade do software, na maneira que é desenhado e em como ele é montado são de design. Por exemplo, definir qual biblioteca será utilizada naquele sistema, se trabalharemos com paradigma funcional ou com orientação a objetos. A arquitetura, por outro lado, normalmente vai ter uma visão mais geral. Dependendo do contexto, essa visão pode até ser mais micro, porém terá decisões abstratas. Diferentemente das decisões de design que definem o nome de uma determinada variável como "X" e para o For definirá "Y".

Apresentando Arquitetura Hexagonal

A arquitetura hexagonal, ou "Ports and Adapters", aborda conceitos que nos ajudam a ter uma visão mais clara no processo de desenvolvimento de nossas aplicações. Por isso é um assunto considerado extremamente importante para pessoas desenvolvedoras. Fora isso, será utilizado durante nossa carreira como devs, o que é mais um motivo para estudarmos o funcionamento desta arquitetura. Nos tópicos anteriores, conseguimos observar alguns conceitos que nos ajudam a compreender melhor onde essa arquitetura está inserida. Então, podemos dar continuidade aos nossos estudos e ver especificamente sua definição.

Conforme Cockburn, a arquitetura hexagonal nos permite fazer com que nossa aplicação seja desenvolvida de forma igualitária, focada nos usuários, nos programas que irão acessá-la, nos scripts que serão rodados para que essa solução seja separada para testes e com isolamento em relação a aplicação de acessos. Por exemplo, o isolamento do banco de dados ou de algum device que essa aplicação possa ter. Então, essa arquitetura está diretamente relacionada à possibilidade de criar um software onde podemos isolar a forma como os sistemas, as pessoas, os scripts e os comandos irão acessá-lo. Além disso, faz com que esse software possa acessar sistemas externos de maneira tranquila e organizada.

Particularmente tenho a opinião de que o termo arquitetura hexagonal está muito mais ligado com decisões de design de software do que necessariamente de arquitetura. Mas, como desenvolvedores, podemos ter opiniões diferentes em relação às definições e nomenclaturas existentes.



Fonte: Engenharia de Software, 2023

A imagem acima foi criada por Cockburn para facilitar o entendimento de como essa arquitetura funciona. Percebemos uma aplicação no centro da imagem, dentro de um hexágono, e por fora, diversos adaptadores. Cada um desses adaptadores tem conexão com objetos externos. Assim, é possível que uma solução externa acesse essa aplicação sem interferir em sua estrutura. Fora isso, vários elementos externos podem estar conectados a esses adaptadores. É possível percebemos que nenhum desses que estão de fora acessam diretamente a solução, pois o adaptador faz toda a comunicação entre eles. Neste ponto, é importante lembrarmos das complexidades do negócio e da técnica. Na imagem, conseguimos observar que tudo está bem delimitado. Vemos que a complexidade da aplicação está no centro e não tem relação com banco de dados, cache ou framework. Ou seja, o que é técnico não se mistura com a aplicação. Por outro

lado, como o negócio será acessado, utilizado e persistido tem relação direta com a maneira que iremos adicionar a complexidade técnica.

A nossa maior dificuldade, como desenvolvedores, é misturarmos as complexidades a ponto de parecer que tudo é uma coisa só. Então, muitas vezes, desde o banco de dados, até outras soluções ficam dentro de nossas aplicações. Fazendo essa mistura é muito provável nós nos perdermos no meio do caminho durante o processo de desenvolvimento de nossas soluções. Ao trabalharmos com "Ports and Adapters", criamos portas para usar adaptadores. Assim, permitimos que os adaptadores acessem nossa aplicação.

Para compreendermos melhor como isso funciona na prática, vamos utilizar o exemplo da fiação elétrica de uma casa. Primeiro, precisamos imaginar que não existem tomadas nos cabos elétricos. E por esse motivo precisamos colocar os eletrodomésticos diretamente no cabo do conduíte. Se em determinado momento precisarmos trocar algum desses aparelhos, como a televisão da sala, seria extremamente dificultoso, pois essa troca poderia interferir em toda estrutura elétrica da casa. Por outro lado, se desde o início colocássemos tomadas, seria muito fácil plugar e desplugar qualquer aparelho eletrônico. A arquitetura hexagonal faz exatamente isso, faz com que tenhamos portas e adaptadores, semelhante às tomadas de uma casa, para falar com a aplicação. Neste exemplo, por todos os aparelhos estarem conectados diretamente ao cabo da conduíte, teríamos um tipo de acoplamento na casa. De maneira semelhante, se não tivermos adaptadores em nossa solução, teremos um sistema acoplado.

Então, sempre que olharmos para um software precisamos lembrar que em uma parte temos o problema da aplicação e em outra temos as complexidades técnicas que estamos adicionando. A arquitetura hexagonal traz justamente essa ideia, de criar adaptadores que acessem a aplicação. Com isso não será necessário fazermos mudanças na estrutura da nossa solução sempre que precisarmos fazer alguma alteração técnica. Ao invés disso, conseguimos trocar apenas o adaptador sem necessitar mudar nem uma vírgula em nossa aplicação.

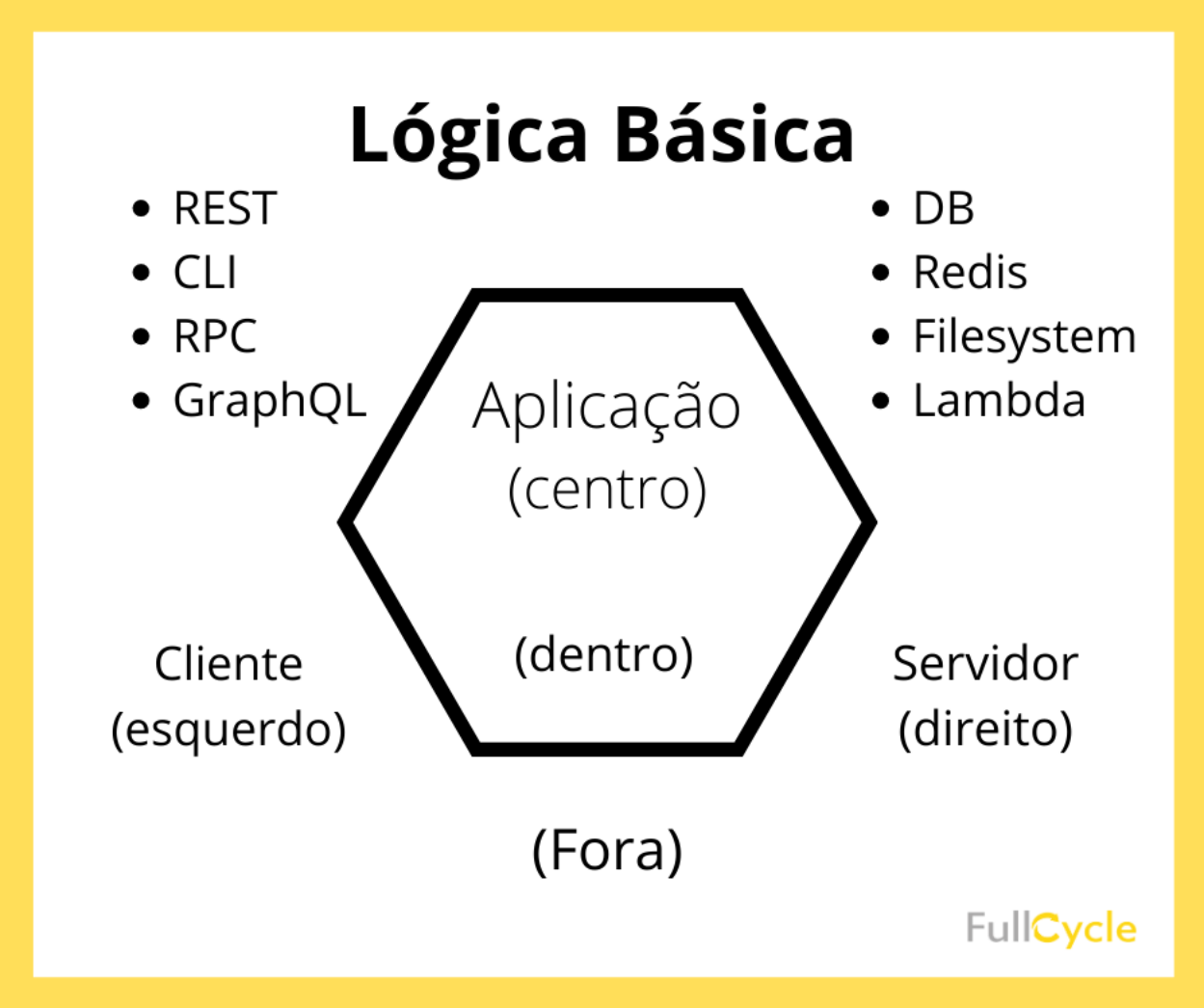
Dinâmica da arquitetura hexagonal

A arquitetura hexagonal tem como dinâmica definir limites e proteção para as regras de uma aplicação. Esses atributos nos auxiliam a desenvolver soluções separando a complexidade de negócio da técnica. Isso porque conseguimos visualizar de maneira clara o que é da aplicação e o que adicionamos ao longo do processo de desenvolvimento dessa solução. Assim, os limites ficam claros em nossa aplicação e sabemos exatamente onde estamos trabalhando, se na complexidade de negócio ou na técnica. Fora isso, ao utilizarmos os conceitos da arquitetura hexagonal, temos a componentização e desacoplação do nosso software. Isto é, conseguimos trabalhar por componentes e nossa aplicação fica desacoplada desses componentes. Por exemplo, imagine que utilizamos a S3 para fazermos o upload, mas queremos trocar para a Google Cloud Storage. Neste caso, simplesmente trocamos de componente sem precisar fazer alterações em nossa linha de comando. Outros exemplos do que podemos desacoplar em nossa

aplicação são: sistemas de logs, de cache, de filas, de upload, o banco de dados, os protocolos de comunicação como: HTTP, APIs, GraphQL, GRPC, comandos externos, etc. Fora isso, separando as complexidades, temos a facilidade para caso você queira mudar para outra arquitetura, como por exemplo microserviços. Por exemplo: se temos um sistema monolítico é muito mais fácil quebrá-lo e criar outros sistemas. Por outro lado, se tivermos tudo misturado, muitas vezes é necessário reescrever muito código para fazer outros microsserviços. Essa facilidade acontece pois separamos a aplicação de todos os elementos externos.

Para que o nosso entendimento fique mais claro e menos abstrato, vamos conhecer a lógica básica dessa arquitetura. Imagine que temos um hexágono, mas não daremos ênfase para a forma geométrica pois poderíamos ter a arquitetura de um octogonal e não faria diferença no objetivo desses conceitos, o mais importante é sabermos que cada ponta do hexágono pode ser um espaço para adaptadores. O termo “Ports and Adapters” , por outro lado, se encaixa perfeitamente nas propostas desse método, por especificar de maneira mais literal o funcionamento desta arquitetura. Porém, vamos usar um hexágono para conseguirmos visualizar de modo concreto como tudo vai funcionar nesse processo. Nossa aplicação ficará bem no centro da forma. No lado direito, teremos o cliente e no esquerdo o servidor. Já sabemos que a aplicação é a complexidade de negócio e ambos os lados serão a parte técnica que adicionamos em nossa solução. Nesse caso, o cliente é quem ou o que vai acessar essa aplicação, por exemplo, um command line interface, um webserver, um gRPC, um Kafka, um RabbitMQ, ou qualquer outro sistema que fará a conexão com

minha aplicação para acessá-la. O servidor é toda parte de infraestrutura que nossa aplicação precisa acessar para funcionar adequadamente, por exemplo, o banco de dados. Lembrando que a aplicação fica dentro do hexágono, o cliente e o servidor ficam do lado de fora, ou seja, são claramente elementos externos à aplicação. Então, no lado do cliente podemos fazer um REST, um gRPC e um UI que são as partes de views com telas para acessar essa aplicação. No lado do servidor, podemos ter um banco de dados, um Redis, um Filesystem para leitura de arquivo externo, chamar uma Lambda function e acessar uma API externa. Abaixo temos uma imagem que ilustra esse exemplo neste parágrafo.



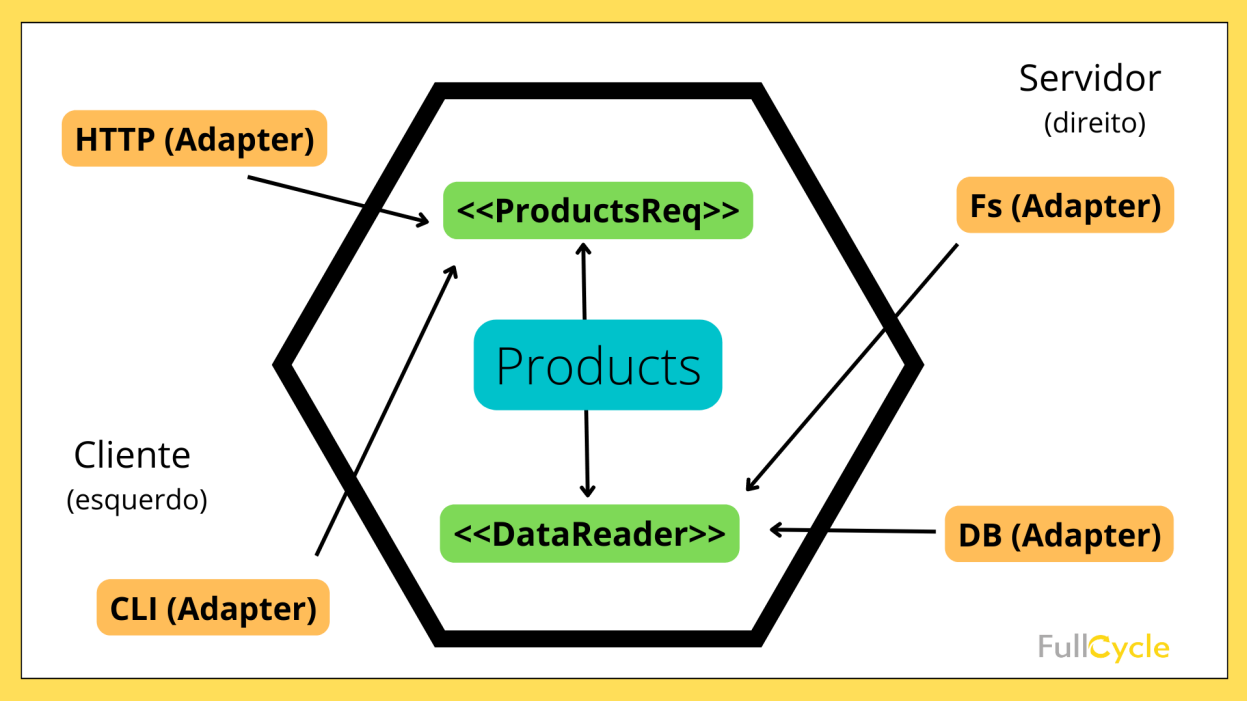
Lógica básica da arquitetura hexagonal

Ao desenvolvermos uma aplicação, vamos imaginar que colocamos essa solução no centro com duas interfaces. Se quisermos que um HTTP, que é um adaptador, se comunique com essa aplicação precisaremos de uma "porta", pois o adaptador não poderá se comunicar diretamente com nossa solução. Normalmente, essa porta é uma espécie de interface. Assim, o HTTP não fica acoplado ao nosso sistema, pois existe a interface fazendo essa separação. Semelhantemente, caso

o command line queira acessar nossa aplicação, isso não acontecerá de maneira direta. Ou seja, eles não estarão acoplados porque a comunicação é feita através de uma interface. Podemos pensar também no outro lado da aplicação, no banco de dados. Ao invés da nossa aplicação estar fortemente conectada ao banco de dados, ela se comunicará por essa interface, que funciona como uma porta, e o banco de dados se torna apenas um adaptador plugável. Logo, por termos mais de um banco de dados, existe a possibilidade de trabalharmos com bancos de dados diferentes.

Para especificar nosso exemplo, vamos imaginar que no centro do hexágono temos um gerenciador de produtos. Então, esse "products" ficará naturalmente na parte de dentro da forma geométrica. No lado esquerdo, teremos um HTTP (adapter) que precisa acessar esses produtos através de um servidor web. Assim, nosso HTTP vai mandar uma requisição para interface e o nosso produto vai acessar essa requisição para conseguir rodar suas regras de negócio e devolver o resultado de volta para o adaptador. Se em algum momento quisermos pegar a lista de produtos através de um command line interface, e ele retorna nossos produtos com as regras de negócio, filtros etc. nosso adaptador manda o acesso para a requisição e, depois disso, acessa os produtos. Perceba que eles não são colocados juntos. Porém, caso nosso produto esteja gravado em um banco de dados, então precisaremos de um DataReader. Isso porque nosso produto vai precisar ler alguns dados externos para conseguir retornar os dados do cliente. Assim, o DataReader vai funcionar como uma interface que vai pegar as informações do banco de dados e mandar para o produto. Esses dados poderiam

estar no Filesystem, ou seja, o DataReader leria esse arquivo TXT e devolveria esses dados para o produto. De qualquer modo, é sempre essa interface que irá se comunicar com o adaptador. Abaixo, podemos podemos observar uma imagem de como essa comunicação acontece através das interfaces:



Exemplo da arquitetura hexagonal

Dependency Inversion Principle

Para desenvolver uma aplicação desacoplada é necessário separar as complexidades, deixando claro o que é adaptador e o que é o negócio. Normalmente, podemos utilizar um dos princípios do SOLID para conseguirmos trabalhar dessa maneira. O dependency inversion principle é um princípio que consideramos ser de design mas contribui para conseguirmos fazer essas separações em nossa

solução. Esse princípio diz que os módulos de alto nível não devem depender dos módulos de baixo nível e ambos devem depender de abstrações. Além disso, abstrações não devem depender de detalhes, mas detalhes devem depender de abstrações. Neste ponto, é essencial termos alguns conhecimentos prévios sobre SOLID para compreendermos como isso funciona. Basicamente, o acoplamento não deve existir de uma forma forte, ou seja, nossa aplicação, que em tese é algo baixo nível, não deve depender do nosso adaptador. Essa aplicação deve depender de uma abstração e essa abstração fará a comunicação com o adaptador. Da mesma maneira, não podemos fazer com que nosso adaptador funcione apenas em nossa aplicação. Por isso, utilizamos as abstrações para realizar essa comunicação, evitando a dependência, isto é, teremos uma solução desacoplada. Dependendo de como estamos desenvolvendo nosso software, podemos chamar essa abstração de diversas maneiras. Em uma orientação objetos, por exemplo, normalmente trabalhamos com interfaces. O essencial é termos a clareza de não misturar o código da aplicação e as regras de negócio com protocolo HTTP, com regras do código 500, ou em como os headers da aplicação devem chegar, ou ainda com a leitura do token JWT, por exemplo. A regra que devemos lembrar é que toda vez que tivermos dentro de uma classe e instanciamos outra classe dentro dela geramos um acoplamento forte. Isso porque quebramos o princípio que aprendemos em SOLID.

Então, normalmente, perceberemos que a inversão que fazemos está relacionada a um controle de classes. Ou seja, a classe recebe a interface em um construtor. E por recebê-la em um construtor, poderá utilizá-la. Mas se essa classe recebesse

outra classe concreta no construtor, novamente teríamos um alto acoplamento, por isso o que vem no construtor é justamente uma interface. Portanto, de modo geral, dizemos que a interface está em todos os lugares.

Todavia, é essencial termos bom senso para não exagerarmos, usando o extremo purismo. Para que nossa aplicação não tenha mais interfaces do que classes. Quando estamos dentro de nossa aplicação podemos trabalhar com interfaces para nos auxiliar mas isso não significa que precisamos criar interfaces para tudo. Sabemos que precisamos para que nossa solução possa trabalhar com objetos externos, porque se não tivermos esse purismo equilibrado provavelmente teremos um alto acoplamento em nossa solução

Se absorvermos esses conhecimentos, nossa forma de desenvolver softwares mudará completamente. Isso não quer dizer que utilizar um framework será algo ruim, mas usaremos com moderação e mantendo nossa solução totalmente desacoplada dessa parte técnica.

Hexagonal vs Clean vs Onion

O princípio fundamental da arquitetura hexagonal é o conceito de observar o que deve estar dentro e o que deve estar fora do nosso software. Além disso, essa arquitetura nos mostra a importância de utilizarmos adaptadores e interfaces, evitando o acoplamento. Ainda assim, é muito comum pessoas desenvolvedoras confundirem essa arquitetura com Onion architecture e Clean architecture. Isso acontece porque essas arquiteturas possuem o mesmo princípio de separar o

coração da aplicação da parte técnica que irá se conectar ao software. Porém, a arquitetura hexagonal não define um padrão de como iremos criar nosso sistema. Ou seja, não há um padrão estabelecido de como o código deve ser organizado. Então, nesta arquitetura não vemos MVC, modo de criar pastas, de trabalhar com agregados, value objects etc. Por outro lado, outras arquiteturas geralmente definem quais são as camadas que devemos utilizar em cada uma dessas partes, tanto de dentro como de fora. A Clean architecture, por exemplo, fala sobre infraestrutura, sobre UI, sobre persistência, entre outras coisas.

Portanto, o que podemos afirmar é que realmente Clean architecture e Onion architecture tem o mesmo princípio que a arquitetura hexagonal, mas vieram depois e são arquiteturas diferentes.

Em seu livro Uncle Bob, fala um pouco sobre como a Clean architecture define de uma forma bem específica o funcionamento da separação do software com o que deve estar na parte fora do sistema. Então, podemos dizer que o modo de trabalhar quando utilizamos essa arquitetura é um pouco “mastigado”. É importante dizermos que não existe certo ou errado, podemos trabalhar com qualquer uma das arquiteturas dependendo de nossos objetivos. O que destacamos é a importância de seguir o princípio da aplicação no meio e os adaptadores na parte de fora, utilizando conectores para evitar o acoplamento da solução.

Então, compreendemos que essas arquiteturas têm diversas semelhanças pois usam o mesmo princípio. Assim, é essencial termos essa informação para que

saibamos diferenciar os conceitos que são mais específicos de cada uma delas. Falar sobre esse assunto chega a ser, de certo modo, até filosófico por ser muito conceitual. Percebemos isso porque fizemos diversas reflexões sobre o tema sem precisar mostrar nem uma linha de código. Acreditamos que, com todos os conhecimentos que tivemos neste capítulo, já conseguimos ter uma ideia mais madura para falarmos e trabalharmos com essas arquiteturas. E ao visualizarmos o desenho da aplicação sabemos se o princípio da arquitetura hexagonal está sendo usado naquele sistema.

Clean Architecture

Introdução

Clean Architecture, ou arquitetura limpa, é um conteúdo extremamente prático, pois no contexto que estamos conseguiremos consolidar o que estudamos sobre fundamentos da arquitetura, Domain Driven Design e arquitetura hexagonal através dessa arquitetura. Então, neste capítulo, nossa ideia é demonstrar um passo a passo de uma sugestão de um boilerplate e do que fazer com nossos softwares. Dizemos isso porque, muitas vezes, compreendemos todos os conceitos, os patterns, os fundamentos etc. mas no momento que precisamos colocar "a mão na massa", é comum travarmos. Assim, as dúvidas começam a aparecer até mesmo na parte de pastas e organizações.

É importante estudarmos com profundidade o que é Clean Architecture, para compreendermos todos os seus conceitos e como as peças se encaixam na hora de desenvolver um sistema. Em nossos estudos veremos para que essa arquitetura serve, por que ela foi feita e quais seus principais pontos. Desse modo, será possível entender seus conceitos e seremos capazes de implementar projetos utilizando as ideias relacionadas à arquitetura limpa.

A origem da clean architecture

Clean Architecture é uma espécie de arquitetura relacionada com o design de software. Ela foi criada por Uncle Bob, o Robert C. Martin, há mais de dez anos. Seus conceitos foram divulgados pela primeira vez no blog do próprio Robert Martin, mas depois de um tempo o autor os transformou em livro sobre o tema. O título desse livro é "Clean Architecture: A Craftsman's Guide to Software Structure and Design". Essa obra é uma leitura muito recomendada para quem deseja reforçar seus conhecimentos sobre o trabalho com arquitetura limpa.

O livro de Uncle Bob, apesar de ser bem extenso, tem um conteúdo que não é considerado denso. Este módulo irá cobrir muita coisa do livro, pois usaremos citações e seus principais conceitos. Nossa ideia é trazer uma orientação, inclusive prática, para que você consiga trabalhar com arquitetura limpa. Mas, caso seja possível, é interessante que faça a leitura diretamente do livro de Uncle. O livro tem cerca de 600 páginas, mas o autor fala especificamente sobre Clean Architecture somente em sete. Essa parte mais específica sobre a arquitetura está literalmente no blog do autor. Ou seja, temos uma "copy and paste" do blog dentro do livro. Então, como a principal ideia não é extensa, entender sobre as camadas dessa arquitetura não é algo especialmente difícil. A maior dificuldade é o embasamento que precisamos ter para que aquilo faça sentido e não viremos apenas ajustadores ou criadores de pastas, que não compreendem de fato a Clean Architecture. A leitura do livro de Uncle pode reforçar nossos conhecimentos

e remover gaps (lacunas) básicas que muitas vezes não percebemos em nosso trabalho. Nele, entenderemos melhor sobre componentes, arquiteturas e limites arquiteturais. Além disso, entenderemos sobre a percepção em relação a parte de regras de negócios. Portanto, recomendamos a leitura do livro para que você beba direto da fonte e consiga reforçar seus conhecimentos.

Podemos considerar a Clean Architecture como uma variação de outras que vieram antes dela. Por isso, percebemos algumas semelhanças entre ela e outras arquiteturas como a hexagonal e a onion. Porém, mesmo com pontos semelhantes, percebemos diferenças claras e precisamos conhecê-las para conseguirmos identificar a arquitetura que estamos de fato trabalhando. Por exemplo, a arquitetura hexagonal trabalha da mesma forma que a arquitetura limpa, com princípio de separar a aplicação do que é um componente técnico, mas a hexagonal deixa em aberto a maneira como o desenvolvedor fará a implementação, diferente da Clean Architecture que possui diversos detalhes. Na arquitetura hexagonal esses detalhes são praticamente inexistentes. Mas as duas usam o mesmo princípio de proteger "o coração" do software. Isto é, tudo o que está em volta do sistema é considerado "resto" e pode se plugar e desplugar facilmente. Então, a Clean Architecture é uma variação da arquitetura hexagonal, trazendo mais detalhes. Por isso, é importante conhecermos primeiramente todos os conceitos de arquitetura hexagonal, inclusive um pouco mais sobre o contexto em que essa arquitetura está inserida. Assim, veremos sentido ao trabalhar utilizando seus termos. Dizemos que tudo é um processo de evolução, e a Clean Architecture é uma evolução da arquitetura hexagonal.

Hoje em dia, é muito comum vermos pessoas desenvolvedoras afirmarem que utilizam a arquitetura limpa em seus projetos. Mas precisamos saber que existe uma diferença entre entender a Clean Architecture e pegar um conjunto de pastas para preencher com diversos arquivos sem ter a capacidade de compreender os detalhes do que precisa ser realizado.

Ao usarmos os conceitos da arquitetura limpa para desenvolver um sistema, trabalharemos com diversas camadas para proteger o "coração" da aplicação. Trabalhar dessa forma significa ter limites arquiteturais no sistema. Assim, sabemos em qual camada estamos e, principalmente, qual limite não pode ser "cruzado", para não interferir nessas camadas. Devemos apenas fazer a comunicação entre elas, caso seja necessário. A melhor forma de conseguirmos fazer isso é através de contratos, isto é, de interfaces para mantermos um baixo acoplamento.

Além disso, essa arquitetura é orientada a casos de uso. Isso significa que a intenção, ou "intent" em inglês, é de conseguir realizar uma ação que gere alguma transformação no código. Por exemplo, uma intenção do usuário em criar uma categoria ou em fazer uma compra.

Uncle Bob fala muito sobre "Screaming Architecture" em seu livro. Esse termo, nos traz a ideia de trabalhar com uma arquitetura que "grite" para a pessoa desenvolvedora. Então, na Clean Architecture vamos documentar para que a aplicação seja "gritante". Os casos de uso são exemplos de como isso pode ser feito, porque conseguimos ver a intenção do sistema de forma clara.

O livro de Uncle Bob sobre o tema, não é considerado denso. Pois apenas sete páginas do livro são dedicadas especificamente a arquiteta limpa. Apesar de ser uma obra extensa, iremos cobrir muita coisa do livro neste capítulo. Para isso usaremos citações e seus principais conceitos. Nossa ideia é trazer uma orientação, inclusive prática, para que você consiga trabalhar com arquitetura limpa. Mas, caso seja possível, é interessante que faça a leitura diretamente do livro de Uncle.

A parte mais específica sobre a arquitetura está literalmente no blog do autor. Ou seja, temos uma "copy and paste" do blog dentro do livro. Então, como a principal ideia não é extensa, entender sobre as camadas dessa arquitetura não é algo especialmente difícil. A maior dificuldade é o embasamento que precisamos ter para que aquilo faça sentido e não viremos apenas ajustadores ou criadores de pastas, que não compreendem de fato a Clean Architecture. A leitura do livro de Uncle pode reforçar nossos conhecimentos e remover gaps (lacunas) básicas que muitas vezes não percebemos em nosso trabalho. Nele, entenderemos melhor sobre componentes, arquiteturas e limites arquiteturais. Além disso, entenderemos sobre a percepção em relação a parte de regras de negócios. Portanto, recomendamos a leitura do livro para que você "beba direto da fonte" e consiga reforçar seus conhecimentos.

Pontos importantes sobre arquitetura

Quando desenvolvemos uma aplicação com Clean Architecture, é importante termos os fundamentos sobre arquitetura claros. Afinal de contas, estamos desenvolvendo com arquitetura limpa. Logo, existem alguns pontos sobre os fundamentos da arquitetura que não podemos esquecer. Isso precisa estar solidificado em nossa mente. O principal deles é que a arquitetura nos auxilia a formatar nosso sistema. Isso porque todo software precisa ter um formato bem específico e desenhado. Além disso, devemos lembrar que a arquitetura divide a aplicação em componentes. Ao longo de nossos estudos entendemos, ainda, o motivo de uma arquitetura limpa de aproximar do design de software.

Podemos comparar o desenvolvimento de um software com o de um prédio. Todo prédio consegue ter um formato, pois antes de construí-lo foi feito um projeto arquitetural dele. Então, externamente conseguimos ver o formato do prédio e internamento vemos suas divisões, que são bem claras, assim como nosso sistema deve ser dividido em componentes claros. Vamos imaginar que esse prédio tem mais do que dois andares. Não é possível que exista metade do primeiro andar no segundo, porque seus limites são muito claros. Semelhantemente acontece na arquitetura, seus componentes devem ter lugares específicos para trabalharmos. Então, pensar em arquitetura significa pensarmos na forma e nos componentes que nosso software vai ter.

Mesmo que cada componente tenha seu lugar é importante termos em mente

que invariavelmente teremos um componente conversando com o outro. E precisamos saber qual melhor maneira de fazer essa comunicação. Ainda no exemplo do prédio, um andar se comunica com outro através de escadas e elevadores. Então, temos diversas formas de ir e vir de um andar para o outro. Dependendo do prédio, conseguimos ter um tipo de forro e um vent de tubulação, ou seja, uma saída de ar que consegue ter acesso ao outro lado do prédio. Da mesma forma, um componente precisa falar com o outro e nós precisamos saber como este componente funciona.

Uma arquitetura bem feita pode nos ajudar a desenvolver. Isso porque, a partir do momento que sabemos o "shape" da solução, ou seja, o formato, conseguimos identificar os componentes e os limites entre eles. Além disso, sabemos o que fazer em cada um deles. Em um prédio funciona da mesma forma, podemos ter o objetivo de trabalhar apenas no segundo andar e determinar que outra pessoa trabalhe no primeiro. Então, o que fizermos no segundo andar não atrapalha em nada o que a outra pessoa estiver fazendo no primeiro. Precisamos apenas que o primeiro fale com o segundo através de uma escada em determinadas situações.

Trabalhando dessa forma conseguimos construir uma arquitetura decente. No prédio, por exemplo, conseguimos pintar perfeitamente cada parede, organizar o edifício em primeiro, segundo e terceiro andar. No software, conseguimos fazer o deploy, ter a operação e fazer a manutenção para melhorar ou adicionar novas features de maneira organizada. Isso só é possível pois o software foi arquitetado desde o início. Por exemplo, quando o sistema é pensado desde o

início no formato de deploy, trabalharemos com variáveis de ambientes. Isto é, não deixamos vários hardcoded. Assim, não temos que pensar em como iria funcionar a sessão caso tivéssemos que duplicar o software, porque já sabemos que vai ser exclusiva ou compartilhada. Fora isso, quando operamos esse sistema no dia a dia sabemos que precisa de observabilidade. Então, desde o início pensamos o que será feito com os logs, se deixamos em arquivo ou se soltamos no console. Além disso, decidimos qual o vendor (fornecedor) de observabilidade vamos utilizar. Talvez, em determinados casos, seja uma boa ideia trabalhar usando open telemetry para deixar nosso sistema mais diagnóstico. Quando essas decisões são pensadas desde o início do desenvolvimento da aplicação fizemos que temos uma arquitetura decente. Ou seja, uma arquitetura que não irá favorecer apenas nosso processo de movimento. Mas também, o processo de deploy, de OPS e de manutenção.

Em seu livro sobre arquitetura limpa, Uncle Bob fala que "a estratégia por trás da facilitação é deixar ao máximo possível portas abertas". Isso significa ter opções na mesa. Pois quanto mais tivermos a possibilidade e a flexibilidade de agregar ao nosso software sem tomar decisões precipitadas, mais temos uma boa arquitetura para nos ajudar.

Keep options open

O tópico anterior foi encerrado com a frase de Uncle Bob sobre a estratégia por trás de uma boa arquitetura. O autor diz que essa estratégia tem o objetivo de

facilitar o processo para que nós possamos postergar decisões. Porque quanto mais conseguimos postergar as decisões, mais claros são os limites arquiteturas em nossa solução. Ou seja, podemos adiar uma decisão pois uma parte não depende diretamente da outra para funcionar. Logo, temos mais portas abertas.

Assim, o principal objetivo de uma boa arquitetura é dar suporte ao ciclo de vida do sistema. Lembrando que o sistema arquitetural precisa ajudar o desenvolvedor. Isso significa ajudar a testar nosso próprio software, a fazer deploy, a observar, a operar e até mesmo a criar a própria documentação do sistema. Desse modo, uma boa arquitetura torna o sistema fácil de entender, de desenvolver, de manter e de implantar. Então, o objetivo final é minimizar o custo de vida útil do sistema e maximizar a produtividade do programador.

Sabemos que é extremamente caro para uma empresa desenvolver um software. E, nos dias atuais, ainda é muito comum surgir a necessidade de refazer um sistema em seis meses ou um ano por causa de uma arquitetura mal feita. Um sistema assim, causa em nós, pessoas desenvolvedoras, a impressão ser muito custoso fazer qualquer alteração, quebrando nossa produtividade. Temos a sensação de estarmos perdendo tempo nesse processo. Por outro lado, se um software for bem arquitetado evitaremos essa sensação e nos prevenimos de perder tempo com coisas que acabaram virando débitos técnicos. Então, a vida do nosso software será mais longa e nosso trabalho mais produtivo. Inclusive, vale a pena mencionar que essa prática facilita até mesmo o upgrade de tecnologias. Assim, Uncle fala sobre “Keep options open”, que traduzido para

o português significa “Mantenha opções abertas”.

Então, ao desenvolvermos um sistema é importante pensarmos em regras e detalhes. Pois, quando falamos em manter portas abertas, estamos pensando exatamente em não perder tempo com detalhes. Por exemplo, imagine que temos duas pessoas desenvolvendo uma aplicação. Nas primeiras reuniões é decidido o funcionamento do software, ou seja, tudo o que o sistema irá realizar, se a comunicação será por fila e se terá uma API REST. Dias depois percebem que o negócio não está acontecendo como esperado. Por esse motivo é feita uma reunião para ajustar as necessidades do sistema. Nessa reunião, é discutido alguns problemas com RabbitMQ. Essas dificuldades parecem não ter um motivo evidente e estão interferindo no funcionamento da aplicação. Sabemos que o RabbitMQ é um detalhe, e tem muito para desenvolvermos em uma solução antes de chegarmos nele. Logo, isso significa que a pessoa está dando mais atenção aos detalhes do que às regras de negócio. Detalhes, simplesmente nos ajudam a suportar as regras. Assim, o RabbitMQ é um detalhe para ajudar com a comunicação do sistema, e por isso pode ser substituído pelo SQS, por exemplo. O que realmente faz o software ter valor são as regras. E os detalhes não devem impactar nessas regras. Se em algum momento nosso detalhe começar a ter impacto nessas regras significa que não conseguimos delimitar uma camada.

O Domain Driven Design é outro exemplo onde vemos que o mais importante do software é o coração. Assim, framework, banco de dados e tipo de API não devem impactar na regra do software, em como ele funciona. Esses são apenas

detalhes que são plugados no sistema e eventualmente podem ser substituídos. Então, quando lembramos de DDD, podemos pensar na frase que é slogan no livro de Eric Evans: “DDD significa atacar a complexidade no coração do software”. Isso quer dizer que o coração do software deve estar separado da parte técnica, que são totalmente plugáveis. Portanto, devemos deixar as opções abertas porque os detalhes são plugáveis e o mais importante desde o início do desenvolvimento de um sistema é conseguirmos trabalhar com as regras de negócio, ou seja, protegendo o coração desse software.

Use cases

A Clean Architecture trabalha orientada a Use Cases, ou seja, a casos de uso. Por isso, faz muito sentido conhecermos essa prática e utilizarmos no nosso dia a dia.

Casos de uso representam uma intenção do código ao desenvolvermos um software. Sabemos que esse sistema terá diversas intenções, mas por vezes elas não ficam claras dentro da nossa solução. Visualizando uma aplicação, vemos códigos, camadas, services, etc. porém, nem sempre conseguimos entender o que esse software faz. Isto é, não identificamos claramente cada caso de uso desse sistema.

Quando conseguimos compreender a ação que o software realiza, apenas visualizando seu código, dizemos que ele tem uma arquitetura gritando “eu faço isso”.

Essa arquitetura tem casos de uso, ou seja, cada ação é uma intenção e cada intenção é um caso de uso nítido. Assim, temos a clareza de cada comportamento que o sistema possui.

É importante lembrar que os detalhes não devem impactar nas regras do negócio. Fazemos isso seguindo a instrução de "Keep options open". Então, trabalhar com arquitetura limpa é a arte de conseguir postergar decisões o máximo possível. Isso significa ter a flexibilidade de não tomar uma decisão naquele momento, podendo pensar nela depois. Logo, se surgir a dúvida de qual será o banco de dados, o sistema de fila ou se usaremos Rest ou GRPC, e quisermos pensar apenas no software naquele momento, decidimos essas questões técnicas em um momento posterior. Por incrível que pareça, quando implementamos Use Cases, ainda nesse momento inicial, não precisamos tomar qualquer decisão técnica. Pois, conseguimos não pensar nessas implementações concretas até a camada de casos de uso. Portanto, reforçamos que frameworks, bancos e API não devem impactar as regras da nossa aplicação.

Use cases vs SRP

SRP (Single Responsibility Principle) é um conceito relacionado ao SOLID, por ser um princípio de responsabilidade única. É muito importante entendermos esse princípio ao trabalharmos com casos de uso, pois temos uma tendência muito grande de reaproveitar códigos parecidos. Mas para utilizarmos Use Cases, cada intenção deve ser escrita de maneira independente.

Sabemos, por exemplo, que alterar e inserir são comandos bem semelhantes. Se precisarmos realizar uma operação de inserir ou alterar um registro no banco de dados, criaremos uma categoria para fazer essa operação. Assim, vamos acessar o banco de dados, verificar se aquele registro já existe e persistir os dados no disco tanto na operação de inserir como na de alterar. Por isso, vemos que é tudo muito parecido. Então, surge a ideia de reaproveitar o código. Obviamente, como pessoas desenvolvedoras, reaproveitamos os códigos, afinal de contas usaremos o princípio de DRY (don't repeat yourself). Ou seja, não repetiremos códigos, se podemos reaproveitá-los. Porém, inserir e alterar, apesar de serem semelhantes possuem intenções diferentes. Portanto, são casos de uso diferentes para nós e, por isso, não podemos reaproveitar o código de inserir para alterar em nossa aplicação.

Ferimos o princípio da responsabilidade única quando a alteração no código é diferente. Ou seja, quando fazemos alterações naquele código por razões diferentes. Essa prática traria complicações futuramente em nosso sistema. Por exemplo, quando inserimos um registro, enviamos a seguinte mensagem ao usuário: "Parabéns! o registro foi inserido." Mas quando houver uma alteração no registro enviamos a seguinte frase: "O registro foi alterado com sucesso". Observamos, então, que as modificações são feitas por intenções diferentes.

A pessoa que tem a intenção de alterar tem um objetivo e a pessoa que deseja inserir tem outro completamente diferente. Então, são razões diferentes que interferem no código. Todas as vezes que um código muda por razões diferentes

e reaproveitamos essa operação o princípio da responsabilidade única é violado.

Logo, dizemos que a primeira coisa que precisamos aprender sobre SRP, quando estamos trabalhando com Use Cases, é resistir a vontade de reaproveitar código. Cada caso de uso deve ser independente, porque no final das contas hoje eles podem ser muito parecidos mas eles vão mudar por razões diferentes e, por isso, precisamos compreender essa ideia.

Uncle Bob fala, em um de seus livros, sobre uma duplicação real vs uma duplicação acidental. Para entendermos o que isso significa precisamos, primeiramente, lembrar de DRY. Isto é, não podemos repetir muitos códigos se pudermos reaproveitá-los. Porém uma duplicação em que copiamos e colamos aquele trecho apenas por ser um código que se repete muito, como as validações, é uma prática que pode não valer a pena, ao pensarmos no futuro da solução. Pois quando acontecem mudanças na regra do software precisamos procurar em todos os códigos para fazer os ajustes necessários. Por outro lado, existem algumas duplicações que são falsas. Percebemos que mesmo duplicando, conforme o sistema muda, esses códigos começam andar por caminhos diferentes. Então, mesmo que no início pareça estar duplicado, no final das contas por essas operações conseguirem seguir caminhos distintos, por esse motivo a decisão de não refatorar foi correta neste caso. Ao invés de reescrever, deixamos tudo o mais pequeno possível. Se tivéssemos refatorado tudo e o código começasse a seguir um rumo diferente, teríamos uma “teia de gato” e faríamos vários if na aplicação. Assim, esse tipo duplicação vale a pena ser feita, pois evita refazer

muitas partes do código.

Para sabermos quando a duplicação deve ser feita, basta pensarmos se aquele código poderá ir para caminhos diferentes quando for necessário, assim não tem problema duplicar. Nessas situações, podemos copiar e colar, pois o DRY entra no jogo. Todavia, quando se trata de casos de uso, a duplicação deve ser evitada, porque estamos trabalhando com intenções.

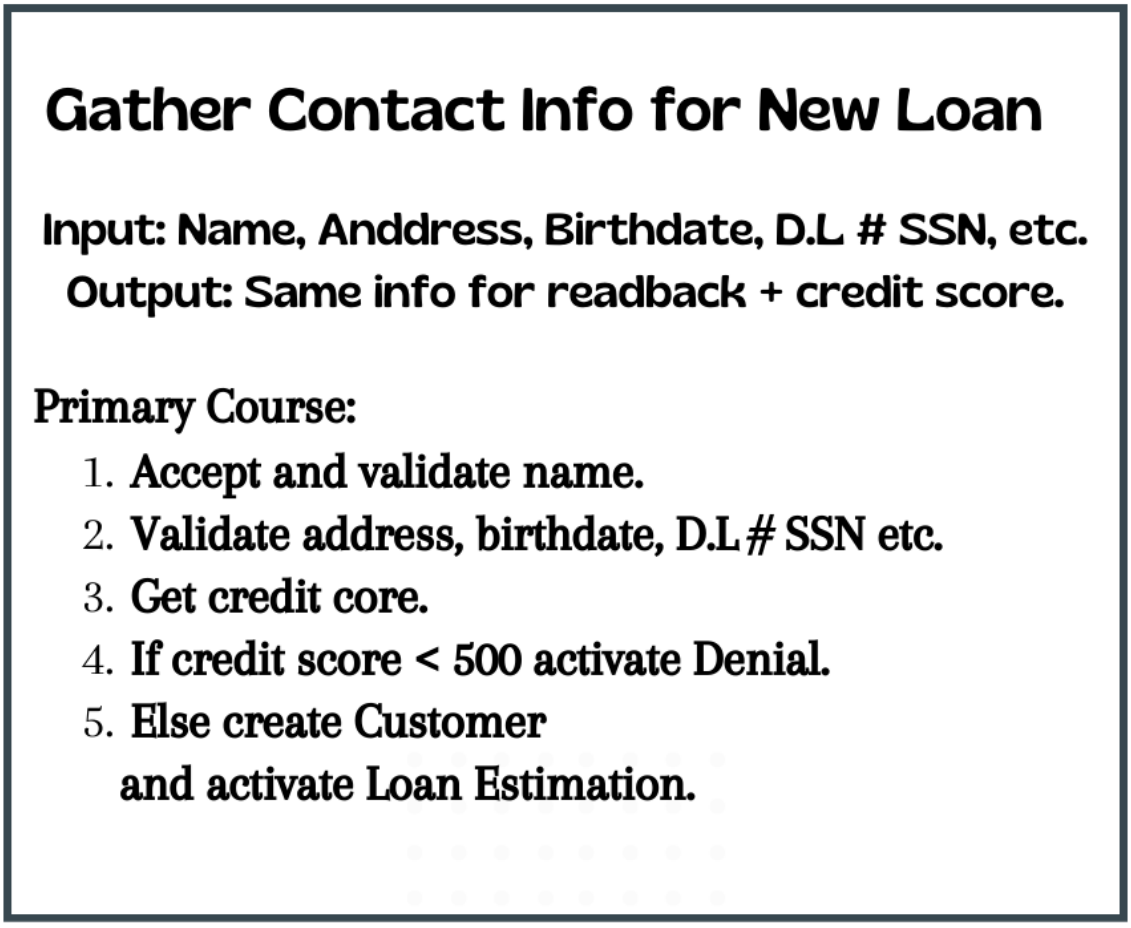
O fluxo dos Use Cases

Falar sobre uma intenção do software é uma das formas mais claras de resumir o que é um Use Cases. Porém, se não compreendermos exatamente o significado de intenção podemos ter certo estranhamento ao ouvir essa palavra no contexto do desenvolvimento de um sistema. Uma operação simples nos ajuda a compreender o que de fato é um caso de uso. Por exemplo, criar uma categoria. Por se tratar apenas de coletar um dado e gravar em um banco de dados dizemos que se trata de uma operação simples. E queremos que operações como essas sejam automatizadas. Afinal de contas, desenvolver um software é automatizar tarefas para facilitar nossa vida. Então, o Use Cases será exatamente a concretização dessa automatização no dia a dia.

Essa automatização depende de regras bem estabelecidas, pois utiliza regras para funcionar. Mas o fluxo, a orquestração e a ordem que cada ação deve acontecer serão representadas através de um caso de uso.

Abaixo, veremos uma imagem retirada do livro Clean Architecture, de Uncle Bob. Nela, podemos observar o que é necessário para conseguirmos realizar uma operação de empréstimo:

Use Cases contam uma história:



Fonte: C., Martin Robert. Clean Architecture (Robert C. Martin Series) (p. 192). Pearson Education. Kindle Edition.

Conforme vimos na imagem, a primeira coisa que precisa ser feita é a validação

do nome e do endereço. Depois disso, devemos acessar o Credit Score da pessoa. Nesse exemplo, a pontuação usada como base para aprovação foi o valor 500. Isso significa que a pontuação deve ser maior do que 500 para que o sistema possa criar e ativar o cliente. Por outro lado, se a pontuação for menor do que o valor base, o empréstimo será negado. Chamamos esse processo de estimativa do empréstimo para uma pessoa. Dessa forma é feito o fluxo de uma operação.

Então, em nossas regras de negócio temos a operação de validar o nome, os endereços, o social, etc., isto é, nesse domínio o componente tem a chamada isolada. Por outro lado, para representar uma intenção do software fazemos chamadas múltiplas, ou seja, múltiplos passos que constroem um sentido capaz de representar essa intenção. Assim, o caso de uso orquestra o fluxo dos acontecimentos na aplicação. Logo, se a pessoa possui Credit Score de 500 ativo, será aprovada pelo sistema. Isso é uma regra de negócio que está sendo chamada a partir de um Use Case.

O banco de dados precisa ser acessado para que essa verificação seja feita, mas isso não acontecerá diretamente. Uma abstração será chamada, por exemplo, um repositório onde se verifica o Credit Score By User para fazer essa validação.

Então, percebemos que o Use cases na prática do dia a dia, faz o fluxo dos acontecimentos. Em outras palavras, é o processo de automação do sistema. Assim, as regras de negócio validam, ou seja, colocam as regras no jogo. Mas o Use Case gera um fluxo que acessa essas regras para que tudo aconteça conforme a empresa precisa. Por isso, podemos dizer que essa automação é a intenção do

software. As regras do jogo ficam no casos de uso, em camadas que chamamos de entidades.

Apesar do Use Cases ser diferente do DDD, se fizermos uma comparação entre eles, entendemos que as regras e o fluxo dos casos de uso não são a mesma coisa. Então, fazendo uma referência disso com Domain Drive Design, pensando em camadas, diríamos que essa seria a camada de aplicação. Porque ela nos diz como a aplicação deve trabalhar. Essa camada, não diz exatamente quais são as regras da aplicação. Isso porque o domínio que vai ditar as regras. Ou seja, nossas entidades serão nossas "Enterprise Business Rules". É importante termos esses conceitos claros para conseguirmos realmente trabalhar com Use Cases.

Limites arquiteturais

Compreender o conceito de limites arquiteturais é muito importante quando trabalhamos com Clean Architecture. Pois com esses limites os nossos componentes são divididos, os contratos são estabelecidos e o software consegue se comportar em fluxos mesmo em momentos diferentes, de acordo com esses componentes. Trabalhar com arquitetura limpa permite fazermos uma separação decente dos limites arquiteturais em nossa solução.

Primeiramente, precisamos entender que tudo o que não impacta diretamente nas regras de negócio, deve estar em um limite arquitetural diferente. Por exemplo, nem o frontend, nem o banco de dados devem mudar as regras da

aplicação. Vamos imaginar que faremos uma solução para calcular os juros de um banco financeiro. Esse banco necessita dessa solução para fazer as análises de empréstimo. A regra para determinar esses juros deve permanecer a mesma independente de usarmos um banco MYSQL ou MongoDB. Além disso, ainda que usemos preto ou branco no frontend ou rótulos diferentes em nosso campo, a regra não deve ser alterada. Logo, sempre que o nosso componente não impactar diretamente as regras de negócio, devem estar em limites arquiteturais diferentes.

A imagem abaixo mostra um exemplo retirado do livro Clean Architecture para compreensão do que são esses limites:

Limites arquiteturais

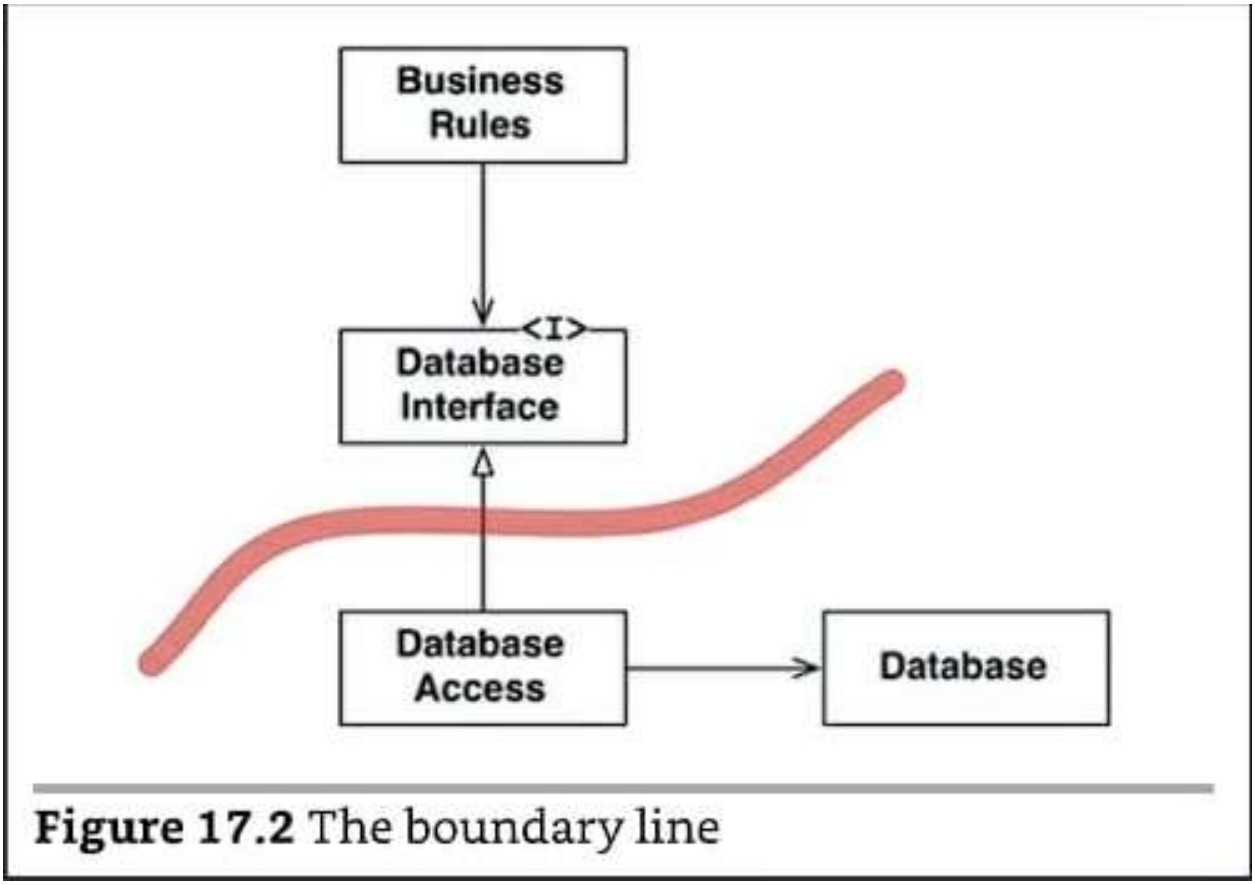


Figure 17.2 The boundary line

Fonte: C., Martin Robert. Clean Architecture (Robert C. Martin Series). Pearson Education. Kindle Edition.

Ao observarmos a direção da seta, percebemos que o banco de dados conhece as regras de negócio. Porém, as regras de negócio não conhecem o banco de dados.

Nesse exemplo, o business rules é a camada que irá trabalhar com as regras de negócio. Por isso, temos os cálculos de juros nessa camada. Sabemos que o banco de dados não deve impactar nessas regras, mas eventualmente essa camada precisa se comunicar com o banco de dados para coletar as informações necessárias. Assim, as regras de negócio chamam uma abstração, ou seja, elas

chamam uma interface que fará esse processo de coleta. Ao implementar essa interface temos uma inversão de controle. Ou seja, ao invés de acoplar nossas regras de negócio com o banco de dados, fazendo com que esse banco de dados tenha impacto sobre as regras, invertemos essa dependência. Então, nossa regra de negócio passa a depender de uma interface. Neste momento, podemos lembrar de boas práticas em desenvolvimento de software, como o princípio utilizado em SOLID.

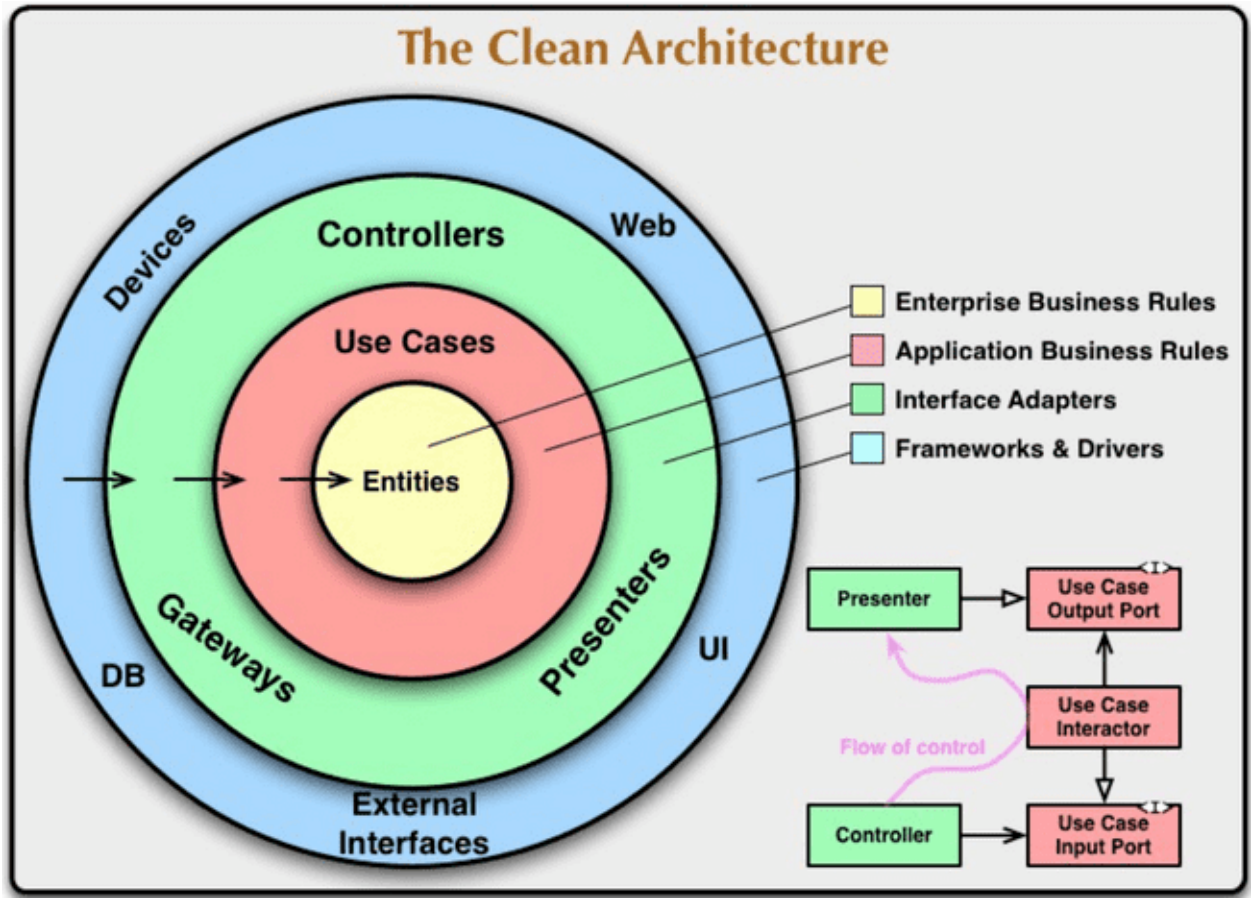
Normalmente, a melhor forma de setar um limite arquitetural é definindo abstrações (geralmente interfaces). Por exemplo, se queremos fazer uma comunicação com um repositório, não podemos grudar o repositório que fala com MYSQL em nossas regras de negócio. Por outro lado, grudamos nossas regras em uma interface de repositório. Assim, sempre que quisermos acessá-lo evitaremos problemas porque é algo agnóstico ao banco de dados. Então, não temos dependência de implementações concretas, pois dependemos de abstrações.

Nosso objetivo não é que a camada de negócios chame o banco de dados, mas sim que o banco de dados chame essa camada de negócios de alguma forma. Assim, ele entra no lugar da interface para que a regra de negócio se comunique diretamente apenas com essa abstração.

Se tivermos alguma experiência com Domain Drive Design, com implementação, criação de repositórios e/ou interface essa definição de limites arquiteturais fica muito clara. Porque, esse limite arquitetural existe claramente no desenvolvi-

mento de uma aplicação.

Abaixo temos um desenho da Clean Architecture. Esse desenho é considerado a imagem mais tradicional para representar a arquitetura limpa. Ao observá-lo, conseguimos compreender ainda mais a definição de limites em um sistema.



Através desse desenho, percebemos que em cada um dos círculos estamos em um limite arquitetural diferente. Assim, temos entidades que não são necessariamente nossos agregados.

Esse conceito é um pouco diferente do que vemos no DDD, pois essas entidades são “enterprises business rules”. Ou seja, nossas entidades são regras de negócio.

Na imagem, fica claro como isso funciona, pois vemos a entidade no centro e em torno dela temos nosso Use Cases. Além disso, vemos a forma de receber as requisições. Por exemplo, se é por web ou por device. Observamos também se cai no controller, se terá presenters para expor a informação; ou, ainda, se teremos gateways. Lembrando que esses gateways são implementações concretas para contratos, ou seja, para conseguirmos nos comunicar com diferentes pontos externos.

Então, percebemos que trabalhar com Clean Architecture, Onion Architecture ou até mesmo com arquitetura hexagonal, significa ter limites arquiteturais bem definidos. Portanto, sempre que criarmos um código podemos nos lembrar da construção de um prédio, onde os limites precisam estar bem estabelecidos, pois não podemos abrigar partes das pessoas em andares diferentes. Assim, precisamos saber exatamente em qual andar estamos durante determinado trabalho. Caso a resposta seja em dois andares diferentes, provavelmente estamos rompendo os limites arquiteturais deste edifício. Semelhantemente, se ultrapassarmos os limites em nossa aplicação teremos algum tipo de problema na arquitetura.

Input vs Output

O conceito mais básico relacionado a input versus output é o de entrada e saída de dados. Dizemos que, no final do dia, temos um input que retorna um output. Por exemplo, o dado para criar um pedido é o input e o retorno com esse pedido

criado é o output. Se raciocinarmos exatamente dessa forma, percebemos que nossa aplicação terá uma entrada de dados de input, com as informações que irão trafegar dentro das camadas da solução. Depois de trafegados, esses dados vão executar uma operação. Ou seja, passar por aqueles casos de uso e bater no domínio para retornar um output.

Então, sempre que desenvolvemos uma solução, é necessário pensarmos em input e output. Precisamos visualizar de maneira clara de onde vem essa entrada e essa saída de dados. O input pode vir de uma API, de um GraphQL, de um GRPC ou de um Command line interface. O output pode retornar dando uma response para web server, para o Command Line Interface ou responses para o GRPC e para o GraphQL. Mas o que precisamos compreender é que se trata de uma entrada versus uma saída de informações.

Quando trabalhamos com Clean Architecture, temos o ciclo exato dessa entrada e saída de dados. Na imagem da arquitetura limpa que vimos anteriormente (Figura. Clean Architecture), podemos observar diversas camadas, a maioria delas conhecidas por nós desenvolvedores. Mas não queremos detalhar essas camadas no momento, queremos, por outro lado, observar as pequenas setas que sinalizam um percurso direcionado à entidade da aplicação. Isso significa que recebemos um dados e que este trafega por um caminho até chegar nessa entidade. Nessa ilustração conseguimos visualizar claramente os limites arquiteturais, porque ela nos mostra que o dado vindo da web não acessa diretamente a entidade, mas existe um percurso que essa informação deve seguir antes de

acessar as regras de negócio da aplicação. Na arquitetura limpa, esse acesso direto às regras não faria sentido, uma vez que a intenção do sistema é feita através do Use Case. Então, o dado vindo da web é tratado antes de chegar até as regras de negócio. Por exemplo, é verificado o endereço que a pessoa acessou e o recurso que usou. Em seguida, a validação é realizada através do controlador, isto é, de um Controller. Continuando esse sistema de rota, o controlador chama o Use Case para determinar a intenção do software. Nesse caso, o controller escolhe um Use Case para chamar, pois não deve chamar diretamente a regra de negócio. Por ter a responsabilidade de realizar a intenção do software, o Use Case sabe exatamente a regra da entidade que deve ser acessada. Assim, ele manda o input para as regras de negócio, que retornam o output. E depois de receber esse retorno, nosso caso de uso pode continuar o fluxo. Quando o Use Case termina o fluxo, envia o output para o controlador, que devolverá uma response. Os dados entram em determinada direção e saem no mesmo sentido que entraram.

No desenho da Clean Architecture, vemos um gráfico demonstrando esse processo, onde um dado é recebido, bate no Controller e entra no Use Case como input. Teremos, então, um Use Case interactor que fará o processamento desse caso de uso, possibilitando o retorno do output para quem o chamou.

Quando o dado retorna para o Controller precisa ser representado da forma ideal. Por exemplo, o output retornado para um Command Line Interface tem uma formatação diferente de um output retornado para web. Portanto, será transformado para ficar com o formato adequado, ou seja, será entregue da

melhor forma possível. Esse processo de formatação é chamado de presenters. Neste tópico, não detalharemos o presenters, pois no momento precisamos saber apenas seu conceito mais básico, isto é, o output será transformado para o tipo de dado que precisamos retornar, de acordo com quem fez a solicitação.

Entendendo DTOs

O DTO (Data Transfer Object) é um padrão que auxilia no tráfego de dados entre os limites arquiteturais. Por exemplo, o dado vindo da web precisa ter uma preparação, pois muitos dados da nossa request vêm junto dele. Essa preparação será feita pelo DTO que vai transportar apenas o dado necessário para o Use Case. Então, ele funciona como um envelope transportando a informação. Em outras palavras, ao recebermos esses dados, criamos um objeto utilizando o DTO. E esse objeto, envia o dado que precisamos ao Use Case. Assim, esse padrão transporta apenas a informação necessária para trabalharmos naquele momento.

O Data Transfer Object é um objeto anêmico e sem comportamento. Ou seja, é um objeto que não possui regras. Por outro lado, precisa possibilitar o acesso aos dados.

A estrutura dos dados são diferentes. Por isso, quando trabalhamos com Clean Architecture, enviamos o dado do input para o nosso Use Case em um DTO e o dado do output em outro. Fazemos isso, justamente, porque nem sempre a estrutura de dados de uma entrada é igual a de uma saída.

Cada intenção do sistema precisa de DTOs diferentes. Por exemplo, imagine a criação de uma categoria. Precisamos passar o nome dessa categoria e jogar em nosso Use Case. Depois disso, criamos um DTO, onde precisamos ter um objeto. Ou seja, temos um campo na classe, que é um atributo chamado nome para que o Use Case possa pegá-lo e retornar seu fluxo, como inserir no banco de dados. Mas se precisamos alterar um dado não podemos usar o mesmo DTO que usamos para criar, porque eles mudam por razões diferentes. Evitamos isso, pois não queremos fazer uma duplicação errada. Por outro lado, queremos fazer uma duplicação correta, para que cada DTO possa ir por um caminho diferente. No caso de alterar, o nome vai passar no mínimo o ID, já que precisamos saber qual o objeto, então, eles são diferentes.

Apesar de não fazer nada com o dado, o DTO tem papel fundamental para conseguirmos movimentar esse dado. Esse é o principal conceito que precisamos compreender para trabalharmos com facilidade utilizando o DTO, pois realmente se trata de um objeto comum. Abaixo vemos uma simulação do caminho percorrido pelo dado:

API -> CONTROLLER -> USE CASE -> ENTITY</

Nesse exemplo, o usuário acessou e mandou os dados através de uma API. O framework, analisa e entende as rotas para mandar esse request ao Controller. Depois disso, jogamos no Use Case apenas os dados que faz sentido utilizarmos. A função do caso de uso é mandar esses dados para a entidade. Ele também faz todo o fluxo e retorna para o Controller como resultado. O controlador

transforma a resposta em um objeto de request e retorna para API. Assim, percebemos que esse processo de devolução acontece via DTO.

Podemos imaginar outro exemplo em que temos o Use Case execute, passando por parâmetro um DTO. Nele, temos todos os dados que o Use Case precisa para ser executado. Então, o Use Case executa o fluxo dele, pega resultado, cria um DTO para colocar os dados de output nele. Nesse exemplo, percebemos que o DTO é um objeto dumb, ou seja, um objeto que não faz nada, mas ele é o dado que normalmente passamos por parâmetro. Portanto, o Controller passa por parâmetro e esse objeto pela execução do DTO.

Quando praticamos o desenvolvimento com arquitetura limpa, as ideias relacionadas aos DTOs ficam mais claras. Pois ao colocarmos "a mão na massa" conseguimos visualizar todo esse processo para entendermos cada um dos conceitos explicados nessa parte mais teórica.

Presenters

Presenters são objetos de transformação. Ou seja, eles transformam um dado que é transportado em um DTO. Assim, o Presenter recebe e adequa o dado ao formato aceito para quem o solicitou.

Um sistema pode ter diversos formatos para entregar os dados. Quando trabalhamos com REST temos como opção um content type. Então, geralmente colocamos o cabeçalho: Application/JSON. Mas ao mandarmos essa requisição

colocamos também Type Object/ XML. Portanto, estamos enviando um dado via JSON, mas queremos o resultado dele via XML. Se não conseguirmos converter esse dado, vamos retornar um erro informando que não é possível ler a informação.

Quando trafegamos um dado, o DTO transforma um output para conseguirmos retornar esse dado no formato adequado. Assim, o resultado pode ser entregue em diversos formatos como, protobuf, JSON, XML, GraphaQL ou Command Line Interface, dependendo do que precisamos. Apesar dos dados de um output serem os mesmo de um input, ou seja, com mesma operação, Use Case e regras de negócio no retorno, o formato que precisamos entregar é diferente. Essa transformação pode ser feita pelos Presenters.

Temos um exemplo que, mesmo sendo grosseiro, pode tangibilizar uma ideia que nos ajudar a compreender o conceito de Presenters. Vamos imaginar o seguinte Input:

input=new CategoryInputDTO(“name”)

Nesse exemplo, estamos criando um DTO dessa categoria e passando o nome dela. Ou seja, temos o input e o objeto com o nome da categoria. Para trafegar esse objeto, precisamos mandar esses dados para o Use Case. Podemos fazer da seguinte forma:

output=CreateUseCase(input)

Dessa forma, passamos o input, trafegando os dados de uma camada para a outra.

Nessa camada, provavelmente estamos em um Controller. Assim, para trafegar o dado do controller para o Use Case, criamos um input, colocamos o dado no formato do objeto e empacotamos em um envelope para que, ao darmos um create category, o Use Case passe esse input. Além disso, podemos dar um ponto execute e o resultado pode ser o ID da categoria. O Use Case retorna um DTO também.
Então, temos um DTO do input e o do output.

O output é um DTO que foi criado e retornado pelo Use Case. Porém, no momento que o dado retorna, precisa ser verificado, inclusive, o tipo de formato que precisamos para responder a solicitação. Portanto, eventualmente, se precisarmos retornar uma informação via JSON podemos fazer assim:

```
jsonResult=CategoryPresenter(output).toJson()
```

Passamos um output com um to Json e ele vai configurar o formato dessa resposta exatamente em um JSON, com assinatura predeterminada como precisamos para entregar o objeto.

Caso a necessidade seja de um dado XML, podemos fazer da seguinte forma:

```
xmlResult=CategoryPresenter(output).toXML()
```

Assim, percebemos que o Presenter vem um pouco depois de termos “a mão” no output, que é um DTO onde conseguimos serializar esse dado no formato que precisamos entregar. Então, temos um dado em determinado formato e o empacotamos em uma response de forma adequada. Ao retornar isso para o

nosso controller, ele pode responder para nossa API. É importante sabermos que o destino do dado não importa para o Presenter, mas sim a transformação desse objeto.

Entities vs DDD

Ao longo dos nossos estudos falamos diversas vezes sobre regras de negócio, domínio da aplicação e entidades. Precisamos compreender bem cada um desses conceitos para conseguirmos trabalhar com arquitetura limpa. Neste tópico, falaremos um pouco mais sobre as entidades, fazendo uma comparação entre as entities da Clean Architecture e a entities do Domain Driven Design.

As entities da Clean Architecture são diferentes das entities de Domain Driven Design. Isso porque na arquitetura limpa, elas são definidas como uma camada e no DDD são a representação de algo único na aplicação. No contexto da arquitetura limpa dizemos que as regras de negócio estão dentro dessa camada. No DDD, podemos dizer que faz parte de um agregado. A palavra agregados nem sequer está presente em Clean Architecture, então, não podemos assumir que o conceito de entities em arquitetura limpa é igual ao conceito no DDD.

A Clean Architecture define que, “entity é igual a camada de regra de negócio”. Assim, para essa arquitetura a entidade é “Enterprises Business Rules”. Ou seja, é uma camada que não muda, por ser invariável e cada vez mais solidificada. Porque independente do que aconteça em nossa aplicação, as entidades são

regras globais dela. A nossa aplicação sempre vai se comportar de determinada forma, pois essa camada possui o coração dela. Então, nessa arquitetura a camada de entidades não varia, pois é sólida e sempre será aplicada de uma forma específica.

No livro "Clean Architecture" de Uncle Bob, o autor não define de forma explícita como devemos criar nossas entities. Ele nos mostra apenas um exemplo em que coloca um diagrama de UML, como uma classe com três métodos, mas não existe uma regra explicando exatamente como isso deve ser feito. Assim, Uncle Bob fala somente para criarmos uma camada, que seja o mais pura possível, contendo todas as regras de negócio.

Mesmo que não exista instruções claras sobre a maneira que devemos trabalhar com essa camada, não será um trabalho totalmente solto. Porque podemos usar alguns conceitos de Domain Driven Design. Então, não ficamos perdidos sobre o que fazer nessa camada, pois esse é um approach muito interessante para criarmos a camada de entity. Assim, usamos as táticas e os padrões definidos pelo DDD na Clean Architecture. Ou seja, nessa camada pode conter os agregados, onde temos nossas entidades, objetos de valor e contratos.

Usar algo do DDD na arquitetura limpa faz muito sentido, porém, não podemos dizer que a entity do Domain Driven Design, mesmo que seja agregado, esteja de um para um, com a entidade da Clean Architecture. Por outro lado, podemos dizer que existe uma correlação para facilitar a forma como as coisas são assimiladas. Assim, partimos do princípio que as entities da Clean Architecture

é uma resultante dos agregados e dos domain services do DDD. Normalmente, fazemos uma separação clara entre o que é agregado e o que é domain services, pois são camadas diferentes do nosso domínio. Então, podemos definir que a entity é o nosso domínio e o nosso domínio é composto de agregados, domain services e eventos.

No DDD criamos camadas e separações muito fortes, assim, podemos dizer que as entities que trabalhamos ali são um conjunto de camadas de domínio que temos para definir nossa aplicação. Logo, se olharmos a imagem da Clean Architecture, percebemos que no centro temos as entities, que aparecem com a seguinte definição de: “Enterprise Business Rule”, ou seja, são definidas como regras de negócio. E nessas regras podemos colocar agregados, contratos, repositórios, domain services, serviços das nossas definições e dos eventos.

Ainda na imagem da arquitetura limpa, vemos a camada de Use Case em volta das entidades. Essa camada tem uma correlação muito forte com o application service do Domain Driven Design. Ou seja, ele chama o application business rules, as regras da aplicação, o fluxo e o playbook. Então, podemos dizer que a camada de caso de usos é como uma receita, que passa as instruções e o fluxo. Porém, dizemos que o Use Case é separado das entities, pois a camada das entidades contém a regra crítica e invariável da aplicação e o caso de usos pode variar de acordo com o fluxo. Lembrando que Web, Devices, External Interfaces e UI são algumas formas do dado chegar, mas esses dados irão bater na entidade e ela vai gerar valor ao negócio, retornando os dados a quem pediu.

Esperamos que os conceitos básicos sobre Clean Architecture tenham ficado claros. Pois trouxemos muitas definições que nos ajudarão na hora de “colocar a mão na massa”. Foi importante explicar cada um desses conceitos para que seja mais fácil internalizá-los quando formos praticar o trabalho com a arquitetura limpa. Se tentássemos aprender essa parte teórica enquanto escrevemos o software, teria um risco de ficar diversos gaps em nosso aprendizado.

Arquitetura baseada em microsserviços

Introdução

Neste módulo, falaremos sobre os microsserviços e especificamente sobre a arquitetura baseada em microsserviços. Estudaremos aspectos teóricos e práticos que nos ajudam a trabalhar com essa arquitetura, para conseguirmos compreender seus prós e contras. Além disso, veremos quando devemos utilizar a opção de microsserviços ao invés de um sistema monolítico.

Conceitos básicos

Antes de falarmos especificamente sobre desenvolver sistemas com arquitetura baseada em microsserviços, é importante compreendermos os conceitos básicos de um micro serviço. Entender esses conceitos e fundamentos evita gaps em nossos estudos. Essas lacunas podem prejudicar nosso trabalho.

Para organizar a explicação desses conceitos, vamos imaginar um quadro onde

colocaremos várias notas mentais que explicam exatamente o que são os microsserviços. Esse quadro funciona como uma maneira de conseguirmos ter essa demonstração esquematizada. O que geralmente facilita nosso entendimento.

A primeira nota seria a seguinte: microsserviços são aplicações comuns. Inicialmente, as pessoas desenvolvedoras têm o pensamento de que os microsserviços podem ser algo diferente ou até muito elaborado. Porém, esses sistemas são aplicações como outra qualquer. Ou seja, não tem nada de diferenciado no aspecto de código de programação. Assim, podemos usar a linguagem que quisermos e fazer o que quisermos quando estamos trabalhando com microsserviços. Por outro lado, nem tudo deve ser colocado na mesma "caixinha". Isso porque os microsserviços têm objetivos bem definidos. E podemos dizer que esse é um diferencial quando trabalhamos com microsserviços ao invés de trabalharmos com um sistema monolítico, por exemplo, pois os sistemas monolíticos geralmente têm diversas responsabilidades.

A segunda nota mental é exatamente o que dissemos no final do parágrafo anterior: microsserviços têm objetivos bem definidos. Mas isso não significa que um microsserviço é uma aplicação pequena, com dez métodos contados. Porque sempre que trabalhamos com essa arquitetura estamos em um processo de evolução. Assim, conforme as soluções e a empresa crescem, os microsserviços tendem a crescer também. Quando isso acontece, percebemos que os objetivos bem definidos começam a quebrar. A partir disso, podemos fazer um microsserviço, que era pequeno e enxuto, com seu objetivo bem definido, se quebrar tornando-

se mais dois ou três microsserviços. Tudo isso tem um nível de maturidade, assim, conforme desenvolvemos o sistema descobrimos que criar aquele microsserviço foi a decisão correta. Atualmente, é muito difícil termos uma precisão cirúrgica do tamanho do escopo de cada coisa. Mas o projeto vai avançando e tudo fica mais claro.

A próxima nota mental do nosso quadro é a seguinte: um microsserviço faz parte de um ecossistema. Dizemos isso porque não faria sentido criar um microsserviço que ficaria isolado. Caso isso aconteça, nem se quer estamos trabalhando com arquitetura baseada em microsserviços. Pois nesta arquitetura, um microsserviço faz parte de um contexto de um domínio muito claro de uma aplicação. Quando criamos uma aplicação que não faz parte de um contexto, isto é, de uma solução maior, provavelmente, criamos um sistema monolítico pequeno que está distribuído.

A quarta nota mental do nosso quadro nos diz que: os microsserviços são independentes ou autônomos. Podemos dizer que essa é a parte mais desafiadora quando trabalhamos com essa arquitetura, pois precisamos fazer com que esses microsserviçoes se comuniquem. Nesse processo de comunicação, uma série de dependências podem ser geradas, fazendo com que o microsserviço não consiga sobreviver sozinho. Isso significa que caso um microsserviço caia o outro simplesmente cai junto. Porque, devido a dependência, não consegue realizar as operações sozinho. Logo, podemos afirmar que, por essência, os microsserviços devem ser independentes. Assim, mesmo que todos os microsserviços ao seu

redor caiam, ele ainda consegue ficar "de pé" e realizar seu trabalho. Obviamente, esse trabalho será mais limitado, pois os outros microsserviços caíram, porém, ainda sim o microsserviço ficará "de pé". Assim, o microsserviço que permanecer funcionando, realiza seu serviço por ter independência, além disso, possui suas políticas de fallback e resiliência.

A nossa última nota sobre o conceito de microsserviço é: eles se comunicam o tempo todo. Então, os microsserviços são vários sistemas que vão se comunicar. Por vezes, essa comunicação não é direta, ou seja, a chamada de um microsserviço não cai diretamente no outro. Podemos trabalhar com esquemas de mensageria, como Message Brokers. Mas de forma geral, o input de um, sai no output de outro microsserviço. Isso faz com que eles se comuniquem mesmo agindo de forma independente.

Caso a aplicação não tenha um objetivo bem definido, não faça parte de um ecossistema, não seja independente e não se comunique com outras soluções, provavelmente é um sistema monolítico bem definido que fica sozinho. Pois microsserviços são aplicações comuns e bem definidas que, mesmo sendo independentes e autônomas, se comunicam o tempo todo. Nesse sentido, é importante sabermos que cada microsserviço tem seu próprio banco de dados. Ou seja, geralmente um microsserviço não compartilha o banco de dados com o outro. Existem algumas exceções em sistemas pequenos que compartilham banco de dados e são considerados microsserviços, mas dizemos que ainda estão em um processo de maturação. Assim, esse sistema está passando por uma

transição de maturidade, porém, em determinado momento precisamos fazer essa migração.

Microsserviços vs. Monolíticos

Neste tópico, vamos entender as principais diferenças entre um microsserviço e um sistema monolítico. Depois de compreendermos os conceitos básicos de um microsserviço é importante sabermos essas diferenças antes de estudarmos especificamente a arquitetura baseada em microsserviços. Isso porque, em nosso trabalho como pessoas desenvolvedoras, é comum surgir a dúvida sobre qual tipo de sistema estamos desenvolvendo.

Uma das principais diferenças entre um sistema monolítico e um microsserviço são os objetivos/domínio bem definidos. Isso significa que ao trabalharmos com um sistema monolítico tendemos a resolver tudo em um único sistema, ou seja, toda aplicação e todos os contextos estão dentro do mesmo sistema. Por outro lado, quando trabalhamos com microsserviços, temos diversos sistemas, onde cada um possui um objetivo separado e bem definido.

Quando falamos de objetivos é comum relacionarmos à linguagem de programação, afinal de contas, em microsserviços podemos utilizar diversas tecnologias. Diferentemente do trabalho com um sistema monolítico, em que utilizamos apenas uma linguagem. Apesar disso, mesmo em um sistema seja monolítico podemos ter algumas exceções como trabalhar com Kotlin e precisar fazer

algumas partes em Java, mas de forma geral se estivermos desenvolvendo um sistema monolítico em C#, não podemos desenvolver um módulo em PHP e depois outro em Kotlin. Pois, priorizamos o uso da mesma tecnologia nesse tipo de sistema. Dizemos que isso é uma vantagem porque sabemos que o time todo vai dominar uma única tecnologia, isso faz com que todo o trabalho se torne mais fácil, até mesmo para contratação profissionais. Por outro lado, em microsserviços temos a vantagem de poder variar em casos que eventualmente surgirem desafios técnicos que exijam uma tecnologia diferente. Por exemplo, se precisarmos de um negócio com altíssima escalabilidade e performance, podemos usar a linguagem GO. Ou se precisarmos de vários CRUDs, podemos usar Java.

Além da diferença em relação aos objetivos, podemos falar também sobre as diferenças no deploy. No trabalho com microsserviços temos uma arquitetura com diversos sistemas, por isso, o deploy é menos arriscado. Isso porque quando caímos um microsserviço não estamos caindo todos os sistemas. Porém, quando fazemos o deploy em um sistema monolítico, podemos ter um risco maior de tudo cair. Mas se tivermos políticas de CI/DC e um ambiente de staging, com todos esses testes automatizados, vamos minimizar muito esses riscos, apesar de estar tudo no mesmo pacote.

Isso não significa que trabalhar com microsserviço é vantagem e com monolítico é desvantagem. Por vezes, trabalhar com um sistema monolítico pode ser uma grande vantagem, pois teremos uma única tecnologia e uma única aplicação que

contém tudo. Assim, não ficamos perdidos em diversos deployments ou várias esteiras de CI/CD etc. Ou seja, mesmo com risco maior, temos a vantagem de ter apenas um deploy para fazer, diferentemente dos microsserviços em que precisamos ter diversas esteiras de CI/CD de processos, de testes entre outros. Então, o sistema monolítico não é desvantagem, pois para determinado contexto ter tudo em um mesmo lugar acaba fazendo mais sentido do que ter diversos sistemas separados por objetivos.

Trabalhar com microsserviços melhora a separação das pessoas, ou seja, o nível organizacional das equipes. Em um sistema monolítico temos todas as equipes trabalhando no mesmo sistema, o que pode gerar alguns problemas. Mas se o sistema monolítico for bem feito, podemos trabalhar por módulos e cada módulo ter uma equipe, por isso não afirmamos ser uma desvantagem, pois vai depender muito de como foi organizado. Por outro lado, se tivermos um sistema monolítico cheio de dependências, com coisas bagunçadas, realmente teremos problemas como conflitos e pessoas mexendo em diversos arquivos etc. Logo, tudo depende da maneira como iremos trabalhar, porque mesmo que seja possível fazer uma separação mais evidente em microsserviços, se essa separação for para que uma equipe cuide de 50 microsserviços, por exemplo, não seria algo vantajoso, pois essas pessoas ficariam perdidas em meio a todo esse processo.

Mesmo observando todas as vantagens do trabalho com microsserviços, nem sempre é necessário criar uma aplicação dessa forma. Por padrão, o sistema monolítico nos ajuda a criar um projeto, uma POC ou uma prova of concept.

Isso porque o sistema monolítico se torna mais simples por existir tudo em um mesmo sistema. E começar um sistema onde você não entende 100% de domínio dos contextos com microsserviços é algo muito complexo.

Quando comparamos o trabalho com microsserviços e um sistema monolítico, ainda existem diversos outros pontos que podemos observar. Porém, o mais importante é ter essa visão de como as coisas funcionam. Lembrando que estamos falando sobre microsserviços de maneira pontual, mais adiante falaremos especificamente sobre arquitetura baseada em microsserviços e teremos uma visão mais ampla desses conceitos mais básicos.

Quando utilizar microsserviços

Não existe uma verdade absoluta para decidirmos se trabalharemos com microsserviços, pois em qualquer regra pode existir uma exceção. Ainda assim, é importante analisarmos alguns pontos e contextos que podem nos ajudar a tomar decisões mais adequadas em nosso trabalho.

Para guiar nossos estudos, partiremos da seguinte pergunta: quando devemos utilizar microsserviços ao invés de criar um sistema monolítico? Geralmente, criar um sistema monolítico no início de um projeto torna tudo bem mais claro. Porque no início de um projeto raramente sabemos tudo o que tem, ou precisa ter, nele. Além disso, alterações inesperadas por causa do mercado ou por pedido do cliente podem ser necessárias. Essas mudanças “bruscas” são

complexas caso a pessoa desenvolvedora não conheça bem todo sistema. Dessa forma, começar com microsserviços, pode ser muito complicado. Pois além de precisarmos ter um domínio muito claro e uma certeza muito grande do mercado, devemos conhecer bem o negócio. Ou seja, ter uma visão ampla do sistema. Logo, esses aspectos fazem com que não seja tão comum iniciarmos uma solução com microsserviços. Obviamente, dependendo da empresa, iniciar com vários sistemas pode ser sim considerada a melhor opção. Isso é possível caso a organização tenha uma ideia de como o mercado funciona e o poder aquisitivo para montar a estrutura necessária com microsserviços. Por exemplo, a Nubank é empresa que iniciou sua arquitetura baseada em microsserviços. Provavelmente, isso aconteceu pois o pessoal do time já dominava o negócio e o mercado. Ela escalou os times desde "pequenininho" , e por esses motivos, foi uma boa decisão ir por esse caminho. Porém, devemos sempre analisar nosso contexto e realidade bem antes de tomarmos essa decisão. Sempre lembrando que por padrão a ideia é iniciar com um sistema monolítico bem organizado. Se queremos iniciar uma Startup com microsserviços, é importante termos o cuidado de saber que se trata de uma exceção. Geralmente, nós temos aquela vontade de utilizar e fazer muitas coisas desde o começo, mas é necessário cautela para que tudo aconteça conforme o esperado.

Normalmente, trabalhamos com microsserviços quando queremos escalar times. Pois quanto mais times temos em microsserviços, mais conseguimos fazer uma boa preparação por times. Em algumas empresas precisamos trabalhar com microsserviços simplesmente pelo fato de que escalar o pessoal é mais viável.

Por exemplo, imagine uma empresa como o Mercado Livre, que possui mais de sete mil programadores, e continua crescendo, seria extremamente difícil colocar todos no mesmo Code Base, em um sistema monolítico. Neste caso, seria totalmente inviável trabalhar ali sem fazer a separação por microsserviços. Então, fazendo dessa forma conseguimos pulverizar mais as equipes, facilitando o trabalho.

Além disso, o trabalho com microsserviços é indicado quando temos contextos e áreas de negócio bem definidas. Porque cada contexto tem sua própria linguagem ubíqua e suas necessidades tecnológicas. Fazendo com que, neste caso, seja mais adequado trabalhar com sistemas separados. Lembrando que é importante observarmos a maturidade nos processos de entrega. Observamos isso se tivermos maturidade para criar esteiras CI, passar por processos de testes, criar esteira de CD, configurar clusters etc. Para isso, existe um pessoal de plataformas que podem ajudar os devs no dia a dia. Logo, essa maturidade significa que estamos preparados para trabalhar com microsserviços. Então, caso a empresa não tenha essa preparação, provavelmente o trabalho com microsserviços não é o mais indicado.

É importante sabermos que se existe uma dificuldade em trabalhar com sistema monolítico, como fazer processo de deploy, rodar testes, ver dificuldades de vulnerabilidades de segurança, dificuldades de escalar, criar novas máquinas, tirar novas máquinas, fazer processos de auto scaling entre outros, pode significar que o não é o momento de trabalhar com microsserviços, pois essas dificuldades

se multiplicam nos sistemas separados. Ao contrário disso, quando dominarmos todas essas dificuldades, podemos trabalhar com arquitetura baseada em microsserviços. Justamente por existir essa maturidade nos processos de entrega.

Além disso, é importante observarmos a maturidade técnica dos times antes de optarmos pelo trabalho com microsserviços. Neste momento, pode surgir a dúvida, se o trabalho com sistema monolítico deve ser feito apenas no início de um projeto. Se nos basearmos somente nessa teoria pode nos faltar fundamentos para trabalhar no dia a dia. Porém, para não misturarmos as ideias, falaremos especificamente sobre o trabalho com sistema monolítico. E, neste tópico, continuaremos mantendo o foco em microsserviços.

Assim, o trabalho com microsserviços não é apenas sobre criar vários sistemas menores. Por exemplo, trabalhamos com microsserviços quando houver uma necessidade de escalar apenas parte do nosso sistema. Isso pode acontecer quando temos muitos acessos no catálogo de produtos, mas a parte de suporte ou área de busca está tranquila. Ou seja, escalamos somente essa parte de catálogo, sem necessitar escalar todo o resto. Portanto, dizemos que esse é um caso em que trabalhar com microsserviço facilitaria todo processo. Além disso, caso precise de um deploy temos menos risco de cair todo sistema.

Fora todos esses aspectos, trabalhar com microsserviço é uma escolha adequada quando precisamos lidar com tecnologias específicas em partes do sistema. Por exemplo, em determinada parte do sistema trabalhamos com GO. Porém, em outra parte precisamos trabalhar com Kotlin ou .Net. Em situações como essa,

nossa arquitetura ser baseada em microsserviços facilita bastante todo processo.

Então, ao observarmos esses pontos, percebemos diversas vantagens no trabalho com microsserviços. Mas precisamos saber que quanto maior o poder, maior se torna a responsabilidade. Isto é, quando trabalhamos com microsserviços temos vantagens, mas ao mesmo tempo precisamos pagar um preço por elas. Assim, conforme "a banda toca" percebemos que não é tão simples utilizar toda essa arquitetura.

Quando utilizar sistemas monolíticos

Sabemos que um sistema monolítico tem toda solução no mesmo código, o que de certa forma é algo bem mais simples de desenvolver. Mas esse não deve ser o único critério que devemos utilizar para escolher essa opção na hora de programar uma aplicação. Assim, veremos neste tópico alguns pontos que precisamos considerar.

O sistema monolítico geralmente é a escolha mais adequada quando temos um POC (provas de conceito). Ou quando iniciamos novos projetos onde não conhecemos todo o domínio. Ou seja, onde não conhecemos todas as áreas e contextos do sistema, pois mudanças podem acontecer inesperadamente e ter tudo no mesmo código nos ajuda a ter uma visão mais ampla da solução, facilitando a resolução de problemas. Além disso, é indicado que o sistema seja monolítico quando queremos garantir a governança sobre as tecnologias.

Se trabalharmos com um sistema monolítico onde podemos especificar que tudo deve rodar em .Net, por exemplo, a parte de governança será bem mais tranquila. Fora isso, a contratação e treinamento do pessoal também é facilitada. Por exemplo, vamos imaginar que contratamos uma desenvolvedora junior com dois anos de experiência. Para essa pessoa, será mais fácil entrar em um sistema monolítico, onde ela irá trabalhar em um módulo específico, do que entrar em um ambiente onde temos diversos sistemas. Pois, com vários sistemas, ela teria que trabalhar com a comunicação por mensagens assíncronas e por mensagens síncronas, como via REST, GRPC e GraphQL. E esse tipo de trabalho exige um domínio técnico por parte de quem está programando. Então, de certo modo, pode existir uma complicação para treinar esse profissional. Com um sistema monolítico esse processo é mais simples por estar tudo em um mesmo código. Assim, caso precise chamar determinado objeto, esse serviço está dentro do próprio sistema. Para a pessoa que trabalha com microsserviços, mesmo com muita experiência, por vezes chamar outro serviço pode ser algo complexo. Então, tanto para quem está começando quanto para empresa é algo mais complicado, por ter que expor todos os conceitos que envolvem o trabalho com diversos sistemas para uma pessoa que está começando como desenvolvedor. No caso do sistema monolítico, esse desenvolvedor não precisa saber de tanta coisa ao mesmo tempo para entregar valor. Obviamente que não se trata de uma verdade absoluta, pois existem empresas que concordam em trabalhar “de pouquinho em pouquinho” em uma situação como esta, de um dev iniciante.

Além disso, com um sistema monolítico temos o compartilhamento claro de libs

(bibliotecas), ou seja, temos um Shared Kernel. Assim, podemos colocar o Shared Kernel para ser mantido dentro do mesmo Code Base.

Se nosso sistema fosse baseado em microsserviço, esse compartilhamento seria bem mais complicado. Porque teremos um repositório onde está essa lib, e todos os microsserviços fazem o clone dela. Caso seja necessário fazer algum tipo de alteração nessa biblioteca, manter a compatibilidade seria algo complexo que poderia até quebrar os outros microsserviços. Por outro lado, dentro de um sistema monolítico se torna mais maleável fazer essas modificações, porque conseguimos ver claramente onde acontece essa quebra, justamente por termos a visão geral do sistema.

Portanto, tanto com sistema monolítico, quanto com microsserviços temos vantagens e desvantagens. Dizemos isso porque não existe "bala de prata" quando estamos desenvolvendo uma solução, o que existe é a necessidade de ter clareza sobre o momento certo para utilizar um ou o outro.

Migração de monolítico para microsserviços

Se tivermos maturidade, separação de contextos etc. pode ser uma boa escolha fazer a migração de um sistema monolítico para microsserviços. As diversas vantagens que vimos nos tópicos anteriores contribuem para termos essa ideia

em mente. Mas é importante sabermos que existem alguns pontos extremamente complexos que fazem parte desse processo.

Inicialmente, temos que fazer a separação dos contextos. Nesse ponto, é recomendável pensarmos e entendermos melhor sobre Domain Driven Design. Uma vez que conseguimos entender melhor os contextos do nosso sistema, percebemos que a tendência é quebrá-los em microsserviços. Porém, devemos sempre observar que não é algo tão simples, pois alguns contextos têm subdomínios muito grandes. Baseado nisso, um subdomínio pode gerar vários microsserviços. Neste ponto, precisamos evitar excesso de granularidade. Ou seja, na hora de fazer essa migração para microsseviços devemos evitar o excesso de "nano serviços", pois esse excesso pode gerar muitas complicações em nosso sistema.

Depois disso, é importante verificarmos as dependências. Se um microsserviço depende claramente de outro teremos algo pior, normalmente chamado de monolítico distribuído. Neste caso, temos toda a complexidade de uma arquitetura baseada em microsserviços, mas perdemos todos os ganhos que um sistema monolítico pode nos dar. Então, acaba não tendo sentido fazer essa migração. E, por vezes, essa prática pode "enrolar" bastante nosso trabalho.

Além disso, é importante pensar e planejar o processo dos bancos de dados. Quando temos um sistema monolítico, utilizamos apenas um banco de dados principal. Ao precisarmos fazer a migração para microsserviços, cada sistema deve ter seu próprio banco de dados. Mas fazer essa migração poderá ser algo complexo. Então, é recomendável fazer a migração enquanto o sistema

monolítico ainda está funcionando, dessa forma, podemos entender quais são as tabelas e os dados que são mais utilizados. Entendendo isso, podemos planejar a migração do banco de dados por partes, até que toda estrutura esteja migrada para banco de dados individuais, cada um pertencendo a um microserviço em específico. E, com isso, fazemos a migração do banco de dados, ou seja, migrar para uma arquitetura baseada em microsserviços é um processo que não acontece de um dia para o outro. Isto é, dizemos que é um processo em que saímos do zero para aos pouquinhos chegar nos 100%. Portanto, mesmo que o banco de dados ainda esteja sendo dividido, podemos considerar que estamos trabalhando com microsserviços, desde que existe um planejamento de como esses dados serão migrados.

Depois que fizermos essa separação dos dados, podemos pensar na criação de eventos para uma migração mais tranquila. Ao trabalharmos com microsserviços, muitas vezes, a ideia principal é a de comunicação assíncrona. Então, quando conseguimos definir claramente quais são os eventos que acontecem ao utilizarmos o sistema monolítico, conseguimos fazer também com que esses eventos sejam disparados para que esse sistema baseado em microsserviços possa utilizá-lo normalmente. Assim, esses eventos vão gerar dados e esses dados vão ser processados por esse microsserviço.

Quando trabalhamos com eventos, não devemos ter “medo” da duplicação de dados. Pois ao fazermos essa separação do sistema monolítico para um microsserviço, eles precisam ter bancos de dados separados. Partindo disso,

podemos perceber que começaremos a ter mais duplicação de dados. Além disso, precisamos pensar na consistência eventual. Ou seja, os dados dessa duplicação não são consistentes. Quando trabalhamos no mesmo sistema, utilizando o mesmo banco de dados não temos eventos e nem chamadas assíncronas. Então, ao alterarmos um nome, essa alteração é feita em todo lugar, assim, em todo lugar que esse nome for utilizado será o mesmo. Mas quando trabalhamos com microsserviços, esse nome estará espalhado em diversos bancos de dados. Por exemplo, vamos imaginar que estamos no Facebook e mudamos o nome do perfil, mas nas discussões e nos grupos nosso nome ainda está da forma antiga. Pois, não houve tempo para refletir no banco de dados essa informação. Mesmo que no primeiro momento isso pareça algo simples, por vezes pode ser algo extremamente complexo e eventualmente até inviabilizar um negócio. Outro exemplo que podemos utilizar para compreendermos as possíveis complicações que essa inconsistência de dados pode gerar é a hospedagem em um hotel. Imagine que alugamos um quarto, mas no catálogo do hotel este quarto ainda aparece como disponível porque os dados não estão consistentes. Ou seja, a consistência não estava correta, por isso, a alteração não foi sinalizada. Pensar nisso é algo muito complexo pois temos que lidar com essa situação de consistência eventual.

Fora todos os pontos que observamos, é importante termos maturidade em processos como CI/CD, em testes e ambientes. Se não conseguirmos fazer esses processos com nosso sistema monolítico, dificilmente conseguiremos trabalhar com microsserviços. Então, essa parte deve estar muito "redonda" para conseguirmos criar, destruir e fazer esses processos rapidamente com os microsserviços.

Além disso, o processo de migração para microsserviços deve ser feito pelas "beiradas". Isso significa que é muito arriscado começar a migração pela principal área do nosso sistema. Assim, é recomendável começar essa migração pelos sistemas periféricos, ou seja, sistemas ou partes do negócio que não tenham tanto impacto no dia a dia. Mesmo que tudo seja importante para o sistema, existem alguns domínios auxiliares que podemos começar a fazer essa mudança. Ao fazermos isso, geramos a necessidade de fazer o padrão Strangler Pattern, isto é, um padrão de estrangulamento. Então, neste momento, podemos fazer a migração para microsserviços das principais partes do sistema. Normalmente, aos poucos, partes desse sistema que inicialmente era monolítico não é mais 100% utilizada, assim, começamos a desabilitar essas partes. Porém, precisamos ter em mente que esse processo quando feito pelas "beiradas" nos possibilita ter práticas que podem nos ajudar quando precisarmos fazer a migração da parte principal da nossa solução.

Portanto, essa migração deve ser feita com cautela e observando esses pontos. Assim, não devemos ir com tanta "sede ao pote" no momento de migrar um sistema monolítico para uma arquitetura baseada em microsserviços. Ao contrário disso, é importante observarmos esta checklist , ou seja, verificar separação de contextos, granularidade, dependências, banco de dados, criação e consistência de eventos, além disso, o padrão de Strangler Pattern. Dessa maneira, nossa migração, apesar de complexa, acontecerá de modo mais tranquilo.

Características

Para trabalharmos com uma arquitetura baseada em microsserviços é importante conhecermos as características desses sistemas. Martin Fowler escreveu um artigo onde ele apresenta nove aspectos que podem caracterizar um microsserviço. É importante entendermos esses conceitos com clareza para que possamos identificar se de fato estamos conseguindo trabalhar com microsserviços, além disso, estudar essas características nos ajudam lidar com as possíveis dificuldades que podem surgir em nosso trabalho. Por isso, preparamos um breve resumo do texto escrito por Martin, onde explicamos essas características com exemplos práticos do nosso dia a dia como desenvolvedores. Porém, reforçamos que a leitura integral do artigo é algo interessante para quem deseja dominar o tema.

Componentização

A primeira característica que Martin Fowler apresenta em seu artigo é a componentização. Esse aspecto é algo que, ao longo da história, todas as pessoas desenvolvedoras almejam para seu software. Componentizar um sistema significa poder trocar um “pedaço” do seu software, ou fazer upgrades desse “pedaço” sem modificar as outras partes do sistema.

Para compreendermos melhor como isso funciona, precisamos entender alguns conceitos relacionados a esse processo. Por exemplo, conforme Martin, na componentização via serviço, um componente é uma unidade independentemente substituível, além disso, é upgradeable por isso também podemos fazer upgrade dela. Precisamos estar atentos porque, mesmo quando trabalhamos no sistema monolítico, podemos ter bibliotecas que conseguimos trocar, fazer upgrades sem mexer no restante do sistema. Então, surge o questionamento se essa biblioteca pode ser considerada como um componente. Neste caso, podemos afirmar que se trata de um componente considerado como um programa in memory, ou seja, está dentro de todo processo da aplicação que está rodando mas é in memory function calls. E os serviços são out of process, ou seja, algo separado do processo principal da aplicação. Logo, neste caso, um componente é um serviço que consideramos ser out of process. Além disso, é independentemente deployavel. Assim, conseguimos fazer um deploy independente que funciona como out of process. Se seguirmos essa regra, temos uma componentização como serviço. Portanto, temos essa componentização via serviços que é um componente substituível, por isso, fazemos o upgrade e o deploy de forma independente.

Capacidades de negócio

A próxima característica dos microsserviços é a maneira como esses sistemas estão organizados. Podemos dizer que os microsserviços são organizados por ca-

pacidades de negócio. Mas antes de entendermos o que isso significa, precisamos compreender o que são as áreas de negócio de um software. Pois antigamente os sistemas eram divididos dessa maneira. Isso não quer dizer que essa maneira de organização deixou de existir. Então, eventualmente algumas empresas ainda mantêm esse tipo de divisão.

Assim, quando uma organização criava seus softwares, inicialmente fazia suas divisões por áreas de funções de desenvolvedores. Neste ponto, podemos lembrar da "Lei de Conway". Em 1968, Melvin Conway citou uma regra que diz o seguinte: "qualquer organização que desenvolve um sistema, de forma geral, vai produzir e replicar a estrutura desse sistema de acordo com a maneira que essa organização se comunica internamente na própria empresa". Isso significa que caso tenhamos, na empresa, um DBA, um designer, um programador back end e um programador front end esses profissionais terão áreas diferentes dentro da organização. Pois, é comum replicarmos a forma como vamos desenvolver de acordo com a divisão organizacional da empresa. Logo, quando falamos de softwares mais antigos, ou até mesmo os de serviços, existe uma tendência para que sejam desenvolvidos dessa forma. Por exemplo, se temos especialistas em UI, em Middle specialist e DBA's desenvolvemos o sistema baseado nessa organização. Isso é levado em consideração inclusive para estruturar os times.

Porém, quando trabalhamos com microsserviços, precisamos pensar nas áreas de negócio da empresa ao invés das divisões por função. Isso porque, se tivermos um especialista em UI, esse profissional irá desenvolver a UI de um sistema

inteiro da organização. Assim, no trabalho com microsserviços vamos focar nas capacidades de negócio, ou seja, criamos tudo junto. Isto é, criamos um time de UI, um de DBA etc. Esse time é Cross funcional, isto é, temos diversos times que vão desenvolver o software focando em uma área de negócio. Diferentemente, da forma antiga em que dividia esses profissionais pela função sem precisar trabalhar desenvolvendo sistemas inteiros.

Então, quando trabalhamos com componentes e microsserviços, normalmente fazemos a separação Cross funcional. Desse modo, a pessoa desenvolvedora pode focar em desenvolver a aplicação sobre uma área de negócio da empresa e não sobre a empresa inteira como na divisão organizacional.

Portanto, em microsserviços é mais interessante fazermos essa divisão por capacidades de negócio. Ou podemos chamar também de divisão por áreas de negócio para facilitar nosso entendimento. Assim, ao invés de focarmos nas funções dos profissionais para fazer todos os sistemas, temos o foco em pequenos sistemas e em áreas de negócio. Assim, temos pequenos times Cross funcionais que conseguem entregar essa aplicação.

Produtos e não projetos

A ideia principal dessa característica é trabalhar com produtos ao invés de projetos. Geralmente, as empresas querem desenvolver projetos para desenvolver seus softwares. Então, criam novas campanhas para fazer algo específico.

Por exemplo, em um novo projeto será necessário montar um time com um DBA, um dev back end, um dev front end, um designer, um product owner entre outros profissionais. E Se soubermos alguns conceitos de gestão podemos dizer que todo projeto tem começo, meio e fim. Dessa forma, depois que esse projeto começa a rodar é muito comum que o time seja desmantelado e, por isso, outros devs precisam manter aquele projeto que foi desenvolvido.

Logo, a ideia que envolve o trabalho com microsserviços é contrária a criação de projetos. Sendo substituída por uma ideia de produtos que envolve o que chamamos de Full Cycle. Ou seja, quem desenvolveu o software também deve mantê-lo. Então, ao invés de tratar esse sistema como um projeto, tratamos como um produto. Porque se ele é um produto, as pessoas que o desenvolvem vão cuidar dele, assim, o mesmo time que desenvolve o sistema, deve manter funcionando.

O Full Cycle acaba cobrindo o software development lifecycle, pois esse time desenvolve, testa, sobe no ar, mantém funcionando e se der algum bug faz a correção. Assim, existe um senso de ownership que é um fator enraizado na visão Full Cycle para que os times possam manter esses produtos.

Smart endpoints dumb pipes

Outra característica sobre microsserviços apresentada por Martin Fowler é a Smart endpoints dumb pipes. Nesse aspecto, vemos conceitos de mensageria,

comunicação assíncrona, aplicações distribuídas e utilização de ESB (Enterprise Service Bus). Pessoas que já trabalharam utilizando esses conceitos provavelmente compreendem com mais facilidade essa característica.

Com essa característica a aplicação tem endpoints que realizam suas operações e regras de negócio dentro dos microsserviços. Por outro lado, como essa comunicação vai chegar dentro de um endpoint tem existe um processo que envolve essa operação. Esse endpoint deve ser simplesmente um canal de comunicação de uma mensagem "crua", ou seja, a mensagem chega do mesmo jeito que saiu através do pipe. Assim, um pipe funciona como um cano para transmitir essa mensagem sem alteração no conteúdo.

Durante muito tempo as pessoas desenvolvedoras usaram Enterprise Service Bus para fazer a comunicação entre as aplicações. No final das contas, era um barramento que possibilitava essa comunicação entre as aplicações. Esse método em si não era um problema, mas trazia dificuldades para quem o utilizava. Porque para conseguir realizar uma boa comunicação entre aplicações em que uma trabalhava com XML e outra com JSON ainda existiam algumas regras. Então, quando uma aplicação se comunicava com a outra, o ESB fazia várias aplicações de regras, conversões etc. Ou seja, vários "malabarismos" para que essa mensagem pudesse ser encaminhada para a outra aplicação. Assim, as aplicações conseguiam se integrar. Isso fazia com que as aplicações dependessem muito do ESB, por ele também ter regras. Pois ao mexer na mensagem original, no final das contas, estamos gerando um acoplamento em nossa aplicação.

Hoje em dia, ao usarmos dumb pipes, podemos utilizar o http como endpoints ou um sistema de mensageria. Ou seja, mandamos uma request e recebemos uma response ou utilizamos sistemas de mensageria como o RabbitMQ, o Apache Kafka entre outros. Assim, mandamos uma mensagem que vai chegar da mesma maneira que enviamos e a aplicação consegue fazer esse processamento.

Para trabalhar com microsserviços é necessário pensarmos em realizar essa operação. Pois a forma como realizamos essa comunicação entre os microsserviços deve ser feita da maneira mais pura possível. Ou seja, sem depender de alguém no meio que realize modificações, principalmente trabalhando com regras de negócios nesses pontos. Portanto, é extremamente importante estudarmos o uso do Smart endpoints dumb pipes, principalmente pessoas que já estavam acostumadas com ESB.

Governança descentralizada

A governança centralizada acontece quando temos uma única forma de tratar todo ecossistema de sua empresa, independente de ser um sistema monolítico ou microsserviços. Então, a forma de trabalhar, de manter as tecnologias e de ditar quais tecnologias serão utilizadas acaba gerando um problema de padronização. Não estamos dizendo que padronizar e criar regras seja algo totalmente ruim. Mas não é necessariamente a melhor forma de trabalhar com softwares. Martin Fowler fala um pouco sobre essa situação para entendermos como isso pode prejudicar certos pontos do nosso trabalho. Ele diz o seguinte: “Nem todo

problema é um prego e nem toda solução é um martelo". Isso significa que, nem sempre para resolver determinado problema precisamos da mesma solução. Às vezes, para ter a solução mais adequada precisamos resolver o problema de maneira diferente do que a empresa está ditando.

Os microsserviços tem a característica de possuir uma governança descentralizada. Ou seja, normalmente temos a opção de criar outras formas para resolver os problemas, ao invés de depender de uma única regra estabelecida para o trabalho de todos na empresa. Isso acontece porque temos gestores diferentes nas diversas áreas da organização. Assim, quando trabalhamos com microsserviços de diversas tecnologias, a governança desses sistemas precisa existir, mas pode acontecer de forma descentralizada.

Apesar dessa governança ser descentralizada, Martin Fowler nos diz que: "Não é porque podemos fazer alguma coisa, que vamos ou devemos fazer". Por exemplo, imagine que temos uma empresa onde 90% dos sistemas são em Java e 10% são em GO por uma razão muito específica. Então, não é porque temos somente 10% em GO que vamos substituir por mais 10% em Java, ou seja, não é porque temos algo que vamos abusar disso.

Além disso, nas gestões descentralizadas com tecnologias diferentes precisamos de contratos para que os sistemas consigam se comunicar e não surjam problemas nessas comunicações. Nesse caso, precisamos trabalhar de forma consumer driven contracts. Isso significa que precisa existir um contrato muito claro para que um microsserviço possa consumir o outro. Se eles estão falando a mesma

língua, ou seja, se eles têm um contrato claro, tudo fica bem definido. Por exemplo, se a comunicação vai ser via http, quais serão os campos, quais serão as regras etc. Dessa forma, com as regras do jogo claras quando um microsserviço for consumir os dados do outro, independente da tecnologia, a comunicação será boa. O sistema que disponibiliza esse contrato não fica preocupado se ele está funcionando ou não. Porque ele sabe que está se monitorando, assim, não está preocupado com o outro sistema que vai consumir, pois as regras do jogo já foram ditadas.

Com o Consumer Driven, quem está desenvolvendo o sistema sempre vai pensar no suporte. Pois essa é a melhor maneira de criar nosso sistema para que outros consigam consumir. Não podemos cair naquele ponto de Development Lifecycle, onde o último bloquinho é o de suporte. Assim, o suporte não fica apenas atendendo o telefone e resolvendo problemas do cliente final. Ele exerce outra função relacionada a deixar nosso sistema da melhor forma possível para ser consumido por outros sistemas. Isso só é possível com a melhor documentação, melhor forma de se comunicar, com protocolos de forma explícita etc. Portanto, conseguimos ter um ambiente funcionando com essa governança descentralizada, independente de tecnologias.

Dados descentralizados

Neste tópico, falaremos sobre a descentralização do gerenciamento de dados. Sabemos que ao trabalharmos com sistemas monolíticos temos um único banco

de dados, onde ficam todas as tabelas e documentos que nosso sistema precisa. Porém, quando quebramos nosso sistema monolítico em microsserviços, cada um desses sistemas precisa que os seus dados sejam somente deles para que possam ser autônomos.

Por exemplo, vamos imaginar que temos um microsserviço que compartilha o banco de dados com outros sistemas. E um desses sistemas tem uma necessidade específica, por isso, muda uma coluna no banco de dados. Caso isso aconteça, provavelmente nosso microsserviço será afetado. Além disso, caso outro sistema tenha uma chamada muito grande de requisições, precise fazer muita leitura ou muita escrita no banco de dados, nosso sistema também poderá ser afetado por essas mudanças. Então, ao tratarmos da descentralização das informações, estamos garantindo que cada sistema tenha seu próprio gerenciamento de dados. Neste caso, temos 1 milhão de bancos de dados espalhados e muita duplicação da informação.

Porém devemos estar atentos para essas duplicações, pois nem todos os dados devem ser duplicados. Por exemplo, imagine que estamos em uma loja virtual, onde temos os dados de um produto. Quando esse produto é comprado para ser vendido na loja, temos um fornecedor manual, porque o produto foi contratado. E no contrato do fornecedor temos várias informações sobre esse produto. Quando vendemos o produto na loja precisamos do nome, da foto e de algumas descrições. Ou seja, não precisamos necessariamente de todas as informações dadas pelo fornecedor no contrato. Assim, podemos ter um banco de dados

contendo somente as informações importantes para o nosso catálogo e outro banco de dados com as informações referentes à parte de compra. Então, existem duplicações do nome do produto, da foto e das descrições, porém, não precisamos duplicar 100% das informações fornecidas no contrato. Normalmente, conseguimos isso pensando na divisão e entendendo como mapear os contextos da aplicação.

Portanto, descentralizar essas informações é muito importante, além de um desafio. Porque ao fazermos isso temos novamente uma consistência eventual. Ou seja, a consistência nas informações não é 100% do tempo. Às vezes, precisamos de um delay para que todas as bases de dados tenham alguma atualização em uma informação em comum. Neste caso, imagine que precisamos mudar o nome do produto. Então, temos que mudar na área de suporte, e em todos os lugares que o nome possa aparecer. Essa sincronização pode demorar um pouco, por isso, dizemos que temos eventualmente um problema de consistência. Logo, esta é uma das características apresentada por Martin Fowler em seu artigo: a necessidade da separação desses dados quando o sistema é quebrado em microsserviços.

Automação de infraestrutura

Quando trabalhamos com um sistema monolítico temos a preocupação apenas com uma esteira de CI, com um processo de teste etc. E, dessa forma, fazemos uma configuração e tudo funciona bem. Mas quando trabalhamos com

microsserviços frequentemente criamos novos sistemas que precisam passar por testes. Ou seja, todos os microsserviços precisam ter procedimentos de segurança, um processo de deploy, verificação de recursos computacionais etc. Assim, nossos microsserviços precisam ter um processo de automação de infraestrutura. Porque caso esse processo não exista, nossa empresa não está madura o suficiente para trabalhar com microsserviços.

Portanto, para trabalhar com microsserviços é necessário pensar em como automatizar essa infraestrutura. Isso pode ser feito com Docker, Kubernetes, VM, Terraform, Ansible etc. Isto é, podemos usar qualquer Cloud Provider para realizar esse processo. Com isso temos um processo consistente para conseguir rodar esses sistemas. Por exemplo, os contínuos delivery, ou seja, entrega e continuous integration. Assim, é feito o teste de aceitação, teste de integração, aceitação, teste de performance, além disso, a parte de produção.

Nos mais diversos exemplos, sempre vamos cair em uma esteira. E daria muito trabalho não ter um processo de automação para fazer esses deploys, testes e, assim, conseguir trabalhar. Hoje em dia, temos diversas ferramentas que nos auxiliam nesse sentido. Por exemplo, Github Actions, CodeBuild e GitLab. Além dessas ferramentas, temos também a Jenkins como uma possibilidade para nos auxiliar, mas não podemos esquecer em qual infraestrutura vamos rodar. Ou seja, devemos pensar se vamos com container, VM, Kubernetes ou só com Docker. Fora isso, precisamos ter em mente como iremos provisionar, como vai funcionar o autoscaling, a quantidade de recurso computacional, como garantir o health

checker, ou seja, garantir que está tudo certinho para colocar esses sistemas no ar.

Então, se não tivermos uma infraestrutura pronta para conseguir lidar com esses múltiplos deploys o dia inteiro, sem que isso seja um fardo dentro da empresa, nosso trabalho com microsserviços será praticamente impossível. Assim, podemos dizer que, antes de tudo, é necessário "arrumar a casa" para depois pensar em trabalhar com microsserviços.

Desenhado para falhar

Quando estamos em um ecossistema de microsserviços, muitas vezes temos certa dependência de outras aplicações. Ou, em alguns casos nossa aplicação simplesmente não consegue pegar dados de outras. Além disso, pode existir situações em que nosso sistema é muito requisitado e precisamos fazer com que ele não caia.

Isso significa que desde o dia zero precisamos pensar na resiliência do nosso software. Ou seja, precisamos ter em mente que em algum momento nosso sistema vai ter algum problema. Então, precisamos pensar em um plano A, um plano B, um plano C, um plano D, etc. para que nosso microsserviço funcione, mesmo que seja minimamente, nas piores condições.

Por exemplo, se nosso sistema de mensageria, ou o microsserviço que fazemos consulta de uma API Rest cair, ou temos várias consultas e estamos quase caindo.

Percebemos que existem “N” situações que preveem uma possível falha mas devemos pensar em como manter nosso microsserviço pelo máximo de tempo possível no ar.

Todos os sistemas estão sujeitos a falhar em algum momento, portanto devemos estar preparados para “quando” isso acontecer. Normalmente, ao criar um sistema pensamos apenas em uma única situação, em que fazemos a chamada e recebemos a resposta. Além disso, pensamos apenas na situação em que recebemos os dados do sistemas de mensageria em um padrão correto, sem que nenhum sistema “detone” nossa requisição. Assim, nem mesmo temos o hábito de prever problemas na rede em que nosso banco de dados está no ar.

Então, nosso microsserviço precisa ter essa característica de ser desenvolvido tendo em mente que várias dificuldades podem surgir. Ou seja, desenhamos nossos microsserviços pensando nessa característica de ser algo que eventualmente vai falhar. Martin Fowler apresenta esse aspecto como: “design for failure”, isto é, desenhado para falhar. Neste ponto, ele nos mostra uma sugestão chamada de Circuit Breaker, uma das formas de garantir resiliência ao nosso software, Essa sugestão serve tanto para ele não acabar com o microsserviço vizinho, quanto para que outros microsserviços não derrubem o nosso.

Portanto, essa é uma das características que um microsserviço precisa ter. Ele é desenhado para falhar em algum momento, por isso, as pessoas que estão desenvolvendo buscam desde o dia zero a resiliência desse sistema. Isso faz muita diferença no dia a dia e garante que nosso sistema permaneça o máximo possível

no ar.

Design evolutivo

A última característica apresentada por Martin Fowler é que os microsserviços têm design evolutivo. Então, “toda vez que eu posso substituir e fazer upgrade do meu serviço sem afetar o restante do meu sistema, significa que eu realmente estou conseguindo trabalhar com esses componentes.” Quando isso acontece conseguimos decidir qual nível de granularidade queremos dar ao nosso sistema. Assim, dizemos que o design pode evoluir da seguinte forma: imagine que temos um sistema monolítico e quebramos uma parte dele. Depois, quebramos essa parte em outras. Tudo isso sem gerar dependências

Nessa característica, podemos refletir também sobre situações em que temos um produto que queremos usar somente em determinado momento mas depois jogamos fora. Isso é bem mais simples quando trabalhamos com microsserviços. Porque subimos uma caixinha ou um novo componente podemos descartar isso sem afetar o Core da aplicação, diferente do que aconteceria em um sistema monolítico. Outra vantagem de um microsserviço é não precisar fazer o Build inteiro da nossa aplicação como precisamos se fosse um sistema monolítico. Isso porque a mudança é feita somente na parte que nos interessa. Portanto, é importante ter essa característica de um design evolutivo, onde trabalhamos com componentes fazendo substituições e upgrades sem afetar os outros usuários

de outros sistemas. Não adiantaria em nada fazer um upgrade e quebrar a compatibilidade com outros consumidores.

Além disso, neste ponto, devemos ter em mente que se repetitivamente mudamos dois serviços para subir uma feature, significa que em algum momento teremos que realizar o merge do código. Por exemplo, imagine que estamos trabalhando com microsserviços para subir uma feature. Mas para isso temos que fazer modificações em outros microsserviços. Ou seja, temos uma dependência muito grande entre esses serviços. Precisamos ter cautela em relação a situações como essa pois pode significar que estamos trabalhando com um sistema monolítico distribuído. E esse é um dos piores ambientes que podemos ter. Então, essas dependências fazem com que precisemos subir outros serviços juntos sempre que precisamos adicionar algo em nosso sistema.

Quando os microsserviços têm a característica de possuir um design evolutivo, conseguimos evoluir outras partes do sistema sem afetar o sistema principal ou sistemas próximos a eles. Por isso, Martin coloca essa característica para um microsserviço como essencial. E é importante compreendermos esses fundamentos antes de desenvolver qualquer tipo de código, assim, temos a certeza de realmente estar trabalhando adequadamente com microsserviços.

Resiliência

Introdução à resiliência

A palavra resiliência é comum para quem trabalha com desenvolvimento de sistemas, principalmente para as pessoas que trabalham com arquitetura baseada em microsserviços.

A resiliência de um sistema sempre vai partir do seguinte princípio: "todo sistema vai falhar". Esse princípio nos faz recordar da necessidade de termos um plano A, B, C, D etc. Neste capítulo, falaremos um pouco sobre quais podem ser esses planos, ou seja, quais estratégias podemos utilizar para mitigar os riscos de falhas, ou trazer o menor impacto possível na hora que essas falhas acontecerem em nosso software. Assim, essas práticas podem nos ajudar a tornar nossos microsserviços cada dia mais resilientes.

O que é resiliência?

Sabemos que em algum momento todo sistema vai falhar. Por esse motivo, precisamos criar sistemas resilientes. Mas, antes de tudo, temos que saber o que

de fato torna um software resiliente. Para fazer uma definição de resiliência, podemos imaginar a imagem de uma árvore inclinada por estar recebendo uma ventania forte. Ao lado dessa árvore, imaginamos a seguinte frase: “Você se dobra ou se quebra?”

Então, a resiliência está diretamente relacionada a essa capacidade de se dobrar para se adaptar ao ambiente em que estamos. Assim, resiliência é o conjunto de estratégias adotadas intencionalmente para a adaptação de um sistema quando uma falha ocorre. Neste sentido, podemos dizer que as duas palavras chaves para definir que um sistema é resiliente são: intencional e adaptação. Ou seja, nosso sistema precisa se adaptar e isso deve ser feito intencionalmente.

Assim, é necessário sabermos quais são os principais pontos de falha e gargalos que podem acontecer para que nosso sistema consiga se adaptar nestes momentos. Dessa maneira, conseguimos criar estratégias de resiliência. E essas estratégias nos possibilitam minimizar os riscos de perda de dados e transações importantes para o negócio.

Portanto, quanto mais resiliente nosso sistema for, menor são os riscos de perdermos dados e transações. Pois se em determinados momento perdermos dados, isso pode tornar as coisas ruins para nossa aplicação. Ou seja, tudo pode ficar caótico pela falta de resiliência do nosso software. Nos próximos tópicos veremos algumas das estratégias para que nosso sistema seja cada vez mais resiliente.

Proteger e ser protegido

A resiliência é uma forma de se adaptar intencionalmente para os pontos de falha que podemos ter em nosso sistema. Além disso, é através da resiliência que conseguimos diminuir os riscos de que essas falhas aconteçam novamente. Então, utilizamos uma série de estratégias para lidar com as mais diversas situações, e assim, conseguimos ter um sistema resiliente.

É importante termos em mente que, ao trabalharmos com microsserviços, estamos dentro de um ecossistema. Assim, vamos receber e enviar requisições a todo momento. Por esse motivo, nossa primeira estratégia deve ser "proteger e ser protegido". Isso significa que um sistema tem uma arquitetura distribuída e, por isso, precisa adotar mecanismos de autopreservação e preservação de outros sistemas para garantir ao máximo que as operações aconteçam com qualidade.

Assim, primeiramente, nosso sistema precisa se autopreservar, ou seja, que ele tem que manter um nível de qualidade. Mas sabemos que o termo qualidade pode ter diversos significados, então, usaremos o exemplo prático da fábrica de cerveja para conseguirmos entender o que significa manter a qualidade nesse contexto. Vamos imaginar que determinada fábrica produz uma cerveja de altíssima qualidade. Mas cada vez que degustamos essa bebida percebemos que o sabor está diferente, piorando aos poucos. Por outro lado, em uma outra fábrica temos uma cerveja que é ruim. E todas as vezes que provamos essa bebida ela continua ruim do mesmo jeito, ou seja, é idêntica todas as vezes

que degustamos. Com isso, vemos que pode existir duas perspectivas para falarmos sobre qualidade. A primeira nos diz que qualidade é algo muito bem feito. A segunda nos mostra que a qualidade é garantir que sempre temos o mesmo comportamento em determinado produto, independente do que esteja acontecendo. Assim, na primeira fábrica a cerveja é muito boa, porém, não consegue manter uma uniformidade, pois cada vez que provamos tem um gosto diferente. Logo, podemos concluir que o controle de qualidade dessa empresa é ruim, porque apesar dela ser boa inicialmente, existe essa variação que faz a cerveja se tornar ruim com o passar do tempo. Na segunda fábrica, temos uma cerveja ruim, mas o controle de qualidade é bom. Quando falamos em manter a operação com qualidade, podemos relacionar isso com o segundo exemplo. Assim, nosso sistema precisa se comportar sempre da mesma forma para garantir que todas as requisições sejam respondidas com qualidade.

Então, se temos um sistema que precisa responder todas as requisições em 500 milissegundos, não importa quantas requisições esse sistema vai receber, precisamos dar essa resposta em 500 milissegundos. Por exemplo, mesmo que existam 1 milhão de requisições temos que garantir a qualidade desse sistema, respondendo essas requisições em 500 milissegundos.

Quando falamos em proteger e ser protegido esse mecanismo de autopreservação deve ser o primeiro ponto observado. Assim, garantir que todas as requisições sejam respondidas exatamente com o mesmo Response Time. Além disso, o sistema funcionará normalmente independente da quantidade de requisições

que ele recebe. Então, caso nosso sistema não consiga funcionar com 1 milhão de requisições e não consiga responder em 500 milissegundos, temos que barrar algumas dessas requisições para garantir que todas vão chegar e ser respondidas no mesmo tempo. Porque, dessa forma, conseguimos manter um padrão de qualidade.

O segundo ponto que devemos observar é não agir de forma egoísta. Isso significa que nosso sistema não pode continuar enviando requisições para outro que está falhando. Assim, é muito comum um serviço ficar lento pelo excesso de requisições ou por qualquer outro motivo. Neste caso, temos duas opções de como agir. A primeira é fazer um um timeout, para esperar que essa aplicação normalize. E a segunda é, mesmo com essa lentidão, enviar um retry atrás do outro. Dizemos que essa segunda atitude seria como "chutar um cachorro morto". Nosso sistema deve evitar esse tipo de atitude egoísta. Ou seja, ao ver que um sistema está falhando, não devemos continuar enviando requisições. Fazendo isso, protegemos outros sistemas, garantindo seu nível de qualidade. Essa prática está diretamente relacionada à resiliência dos sistemas. Pois conseguimos mitigar os riscos de falhas, evitando fazer retry várias vezes seguidas em um sistema que está caindo.

É importante pensarmos em meios de garantir essa proteção, principalmente quando estamos trabalhando com microsserviços. Assim, um sistema lento no ar, muitas vezes, é pior do que um sistema fora do ar. Dizemos isso porque um sistema lento pode causar um efeito dominó muito grande. Ou seja, todos os

sistemas que chamam por ele também ficam lentos. Então, quando chamamos um sistema lento, por tabela ficamos lentos também. Isso pode gerar uma cadeia de problemas para as aplicações. Dessa maneira, muitas vezes é mais indicado retornar um erro 500, sinalizando que aquele serviço está fora do ar porque não está aguentando a quantidade de requisições. Lembrando que isso não é indicado para todos os casos, mas na maioria das vezes é sim a melhor opção.

Portanto, para garantir que vamos proteger e ficar protegidos, precisamos pensar em pontos que garantam a qualidade do nosso software. Ou seja, temos que buscar mecanismos para verificar se o sistema está lento, ter políticas de retry, saber como esperar e criar regras para barrar nossas requisições quando estamos mais lentos do que esperado. Isso pode garantir o padrão de qualidade da nossa aplicação, fazendo com que nosso sistema seja resiliente.

Health Check

O Health Check é um mecanismo utilizado para fazermos a checagem da saúde do sistema e, assim, conseguimos verificar se temos ou não um sistema saudável. Caso não exista uma forma de verificar a saúde da aplicação seria extremamente difícil tomar boas decisões sobre o que esteja acontecendo com esse software. Porém, mesmo sendo um mecanismo que a maioria de nós sabemos da existência, muitas vezes deixamos de lado na hora de desenvolver nossa aplicação.

Então, o Health Check é basicamente uma forma de verificar o quão saudável

está nosso sistema, porque sem sinais vitais não é possível saber a saúde de um sistema, ou seja, precisamos dessas informações a todo momento. E quando dizemos “sistema”, não é somente nosso software mas tudo o que ele utiliza.

É comum vermos pessoas rodando um Health Check para verificar se o seu próprio sistema está em pé, mas deixam de ver outros sistemas que estão conectados a ele, como os bancos de dados. Porém, a checagem de saúde precisa bater no sistema e depois nos bancos de dados para conseguir verificar a saúde, ou seja, o Health Check precisa passar por todas as dependências para garantir que tudo o que o sistema vai utilizar está no ar com saúde. Assim, um sistema que não está saudável possui uma chance maior de se recuperar caso o tráfego pare de ser direcionado a ele temporariamente. Mas essas decisões são possíveis apenas com uma checagem bem feita de tudo que envolve o sistema. Dessa forma, temos um self healing, ou seja, autocura do sistema.

Ao criarmos nosso sistema de maneira organizada, podemos perceber que por vezes ele fica ineficiente, ou com lentidão por está tratando uma quantidade enorme de requisições recebidas. Mas se interrompermos novas requisições por um tempo, ele vai processar o que ficou pendente e voltará ao ar normalmente. Isso pode ser chamado de self healing e deve ser pensado a todo momento quando desenvolvemos um sistema.

Assim, um sistema pode estar ruim, ou até mesmo fora do ar por ter recebido muitas requisições. Nesse caso, continuar recebendo requisições de outro sistema acaba “matando” aquele POD e server. Por isso é necessário subir novos sistemas

para ter mais recursos e conseguir receber as requisições. Porém, as requisições que estavam sendo processadas são perdidas e as novas requisições também são perdidas porque estamos subindo outro sistema enquanto elas chegam. Dessa forma, muitas vezes é melhor parar as novas requisições, processar as requisições que temos até que nossa aplicação esteja curada para conseguir receber novas chamadas. Existem diversas formas de self healing, mas para conseguir fazer a verificação precisamos ter um Health Check. Sem essa checagem não conseguimos tomar decisões eficientes por nosso sistema.

Além disso, nosso Health Check precisa ter qualidade. Por exemplo, não adiantaria em nada testar somente nosso sistema, precisamos testar também as dependências dele. Para compreendermos como isso funciona precisamos saber que existe um Health Check ativo e um passivo.

Dizemos que a checagem é passiva quando alguém consegue verificar a saúde do sistema, ou seja, em um momento que não sabemos como está a saúde do sistema outro serviço consegue ver nossa saúde. Depois disso, sabemos como estão os sinais vitais da nossa aplicação e como vai se comportar, de acordo com as regras. Por outro lado, a checagem é ativa quando o próprio sistema vai se auto verificar, podendo ser até através de outros sistemas, mas nossa aplicação faz essa checagem para verificar se está ativa, ou seja, se está funcionando bem. Caso o sistema não esteja com bom funcionamento, iniciamos os mecanismos de resiliência.

Podemos usar como exemplo o Nginx que por padrão trabalha com Health Check

passivo. Mas na versão paga conseguimos ter um Health Check ativo. Então, se usamos Kubernetes, temos probes. Assim, conseguimos fazer uma checagem de saúde do sistema a todo momento, desde o start up até quando o sistema está no ar. É comum vermos as pessoas não utilizando os probes, por isso acabam sem saber se o sistema está ativo ou não.

Portanto, não devemos subestimar a importância do Health Check ao desenvolvermos nossos sistemas. Precisamos compreender a diferença entre uma checagem ativa e uma passiva. Além disso, é importante entendermos a necessidade de fazer um Health check de qualidade, verificando todas as dependências para garantir que realmente temos um sistema no ar com bom funcionamento.

Rate Limiting

Quando trabalhamos com arquitetura baseada em microsserviços é importante fazermos um check list para revisar se nosso sistema tem alguns pontos sendo aplicados para termos uma aplicação resiliente. Por isso mesmo que algumas dessas estratégias sejam conhecidas por nós desenvolvedores é importante lembrarmos de cada uma delas. Esse é o caso do Rate Limiting. Muitas pessoas já conhecem essa estratégia, mas é importante falarmos um pouco mais sobre esse tema. Assim, o Rate Limiting deve fazer parte do nosso check list para desenvolvermos um sistema com resiliência.

Podemos dizer que a função do Rate Limiting é proteger o sistema baseado no

que ele foi projetado para suportar. Isso nos lembra um pouco da necessidade de manter a qualidade das respostas da aplicação. Quando projetamos um sistema, sabemos quais parâmetros precisamos para conseguir manter essa qualidade, ou seja, acima de determinada escala não conseguimos manter mais a qualidade das respostas. Assim, podemos aplicar as regras de Rate Limiting para limitar a quantidade de requisições que chegam naquele sistema. Isto é, conseguimos barrar imediatamente as novas requisições que nosso sistema não consegue processar. Porque se continuarmos recebendo mais requisições do que realmente podemos, nosso sistema vai "capotar" e perdemos o requisito de qualidade. Então, mesmo que essa estratégia não seja uma novidade, é importante verificarmos se estamos trabalhando com ela.

É importante sabermos que ao trabalharmos com o Rate Limiting temos que fazer preferências por client. Dizemos isso, pois ainda é comum que essa estratégia seja feita com regras genéricas, ou seja, se aguentamos 100 requisições por segundos, apenas barramos as chamadas quando atingimos essa quantidade. Porém, vamos imaginar que temos um cliente importante, por dar um retorno financeiro bom para o negócio. Esse cliente precisa consumir nossa API, e por seu nível de importância, em tese deve ter algum tipo de preferência no acesso ao nosso sistema. Digamos que nesse exemplo ele está conseguindo fazer apenas 10 requisições por segundo. Mas um outro cliente que não dá retorno financeiro ao negócio está conseguindo fazer 90 requisições por segundo. E depois das 100 requisições nosso sistema barra todos que tentam fazer alguma chamada. Percebemos que esse barramento de requisições não está sendo estratégico para

o negócio pois está dando preferência para alguém que não dá retorno à empresa, além disso, tira oportunidade do cliente que realmente seria estratégico para o negócio.

Essa estratégia nos mostra como a área da tecnologia está ligada à área de estratégia de negócios, pois essas decisões têm forte influência no negócio em si. Isto é, muitas vezes essas decisões não ficam apenas na responsabilidade do desenvolvedor, mas provavelmente pessoas da área de gestão da empresa podem contribuir identificando quais clientes são estratégicos ou não.

Portanto, quando trabalhamos com Rate Limiting é essencial lembrarmos que existem diversas regras. Ou seja, existem formas de criarmos grupos de preferências onde podemos determinar quais clientes são estratégicos. Em nosso exemplo, o cliente que proporciona retorno ao negócio poderia fazer 90 requisições por segundo, enquanto o cliente que não é estratégico teria oportunidade de fazer até 10 requisições nesse mesmo tempo. Assim, nosso sistema trabalha com uma preferência estratégica que beneficie a empresa. Ou seja, não podemos fazer o Rate Limiting simplesmente por fazer, precisamos analisar os clientes para criar esses grupos de preferências.

Circuit breaker

Quando desenvolvemos um software podemos usar o circuit breaker como um mecanismo para chegar ao ponto de proteger e ser protegido. Assim, podemos

dizer que esse mecanismo protege o sistema fazendo com que as requisições feitas para ele sejam negadas. Então, sempre que uma requisição é feita para um sistema, acontece uma verificação para sabermos se esse sistema pode ou não responder. Para isso, no circuit breaker, temos um circuito fechado, um circuito aberto e um semi aberto.

No circuito fechado, as requisições chegam normalmente até o sistema. Podemos dizer que é semelhante a um cabo de energia onde um circuito fechado faz a energia chegar até a lâmpada. No circuito aberto, podemos imaginar várias requisições tentando chamar o sistema, mas ele não responde ou retorna um erro 500. Então, ao invés de continuarmos recebendo e/ou mandando requisições, abrimos o circuito e as requisições param de chegar. Dessa maneira, conseguimos proteger o outro sistema e se proteger também, minimizando o tempo gasto com essas tentativas. Além disso, o circuit breaker pode ter o status de meio aberto (half open). Nesse caso, ele permite uma quantidade limitada de requisições para verificar se o sistema tem condições de voltar ao ar integralmente. Funciona da seguinte maneira: uma vez que abrimos o sistema, paramos de mandar requisições. Depois de um certo tempo, mandamos somente algumas requisições para ver se aquele sistema normalizou. Assim, caso o sistema tenha voltado, fechamos o circuito para enviar as requisições normalmente.

Esse mecanismo é extremamente importante e tem uma implementação fácil, porém, existe certa polêmica nos dias atuais em relação a essa e outras estratégias.

Pois quando o desenvolvedor cria uma aplicação, busca por bibliotecas, gasta tempo etc. e fica responsável por escolher todos os critérios para aplicar as regras no circuit breaker, por exemplo, levanta uma reflexão sobre o fato desse profissional ter acesso somente ao microsserviço que está desenvolvendo. Ou seja, ele não tem uma visão geral de todos os microsserviços. Logo, surge essa necessidade de analisar se essa parametrização realmente deve ser feita pela pessoa desenvolvedora desse sistema. Por esse motivo, hoje em dia temos alguns recursos como o Service Mesh. Recursos como esse podem contribuir na implementação de diversas estratégias, incluindo o circuit breaker. Mas nem todas as empresas conseguem fazer a implementação de um Service Mesh, por exemplo, por isso mesmo com essas polêmicas é importante que os desenvolvedores saibam pelo menos o básico de como implementar essas estratégias.

API Gateway

API Gateway é uma ferramenta importante e estratégica que pode contribuir para que nossos sistemas sejam resilientes. Por isso, gera curiosidade na maioria das pessoas desenvolvedoras de softwares. Então, é interessante sabermos um pouco mais sobre a sua implementação, pois ela tem recursos suficientes para tirar algumas responsabilidades dos desenvolvedores, garantir a resiliência e tornar todo processo mais fluido.

Uma API Gateway garante que requisições inapropriadas cheguem até o sistema. Por exemplo, usuário não autenticado. Podemos comparar essa ferramenta com

um portão de um condomínio, onde o usuário precisa passar por esse portão para ter acesso a uma casa. Assim, caso alguém deseje bater no portão da casa, antes disso precisa bater no portão do condomínio. Neste caso, se não queremos receber essa pessoa, o pedido será negado. Então, podemos dizer que este exemplo resume a maneira como uma API Gateway trabalha, isto é, com autenticações para que o sistema não receba requisições indesejadas.

Seria extremamente complicado se um sistema, formado por um conjunto de microsserviços, precisasse fazer a autenticação e verificação de login e senha o tempo todo conforme as requisições fossem chegando. Por isso, hoje em dia, é muito comum que essas validações sejam feitas por uma API Gateway. Depois que essa ferramenta libera a requisição na porta, podemos partir do princípio que tudo o que acontece dentro da rede é seguro e autenticado, pois já foi liberado no portão externo. Mas precisamos ter em mente que isso é uma escolha da empresa. Algumas organizações podem optar por fazer uma segunda validação para garantir que realmente não existe nem um problema com as requisições. Mesmo assim, por padrão, se uma requisição passou pela API Gateway, os sistemas internos podem sim partir do princípio que aquela requisição é verdadeira, não é maliciosa e está autenticada.

Então, ter uma API Gateway pode facilitar o trabalho dos desenvolvedores desses microsserviços, pois estratégias de resiliência como Rate Limiting, Health Check entre outros, que seriam implementadas pelo desenvolvedor são aplicadas pela própria API Gateway. Porém, é importante observarmos que no caso do

Health Check precisamos implementar o código que faz a verificação do banco de dados da aplicação. Normalmente, criamos um "/health" em nossa aplicação e a API Gateway faz a verificação dessa URL de maneira ativa. Assim, caso esteja tudo "Ok", a própria API Gateway garante que as requisições consigam chegar ao sistema. Caso contrário, ela retorna um erro http 500 para garantir que as requisições não batam em nosso sistema. Pois, com a verificação, ela vai identificar que a aplicação está fora do ar e impedindo que as requisições continuem chegando.

Além disso, a API Gateway nos ajuda a organizar nossos microsserviços em contextos. Quando trabalhamos com arquitetura baseada em microsserviços, temos serviços comunicando e chamando o outro frequentemente. Em um primeiro momento, isso pode ser algo confuso. Podemos chamar essa situação de coreografia de microsserviços. E quando temos esse cenário, é muito fácil virar um ambiente cada vez mais caótico. Esta confusão pode ser representada por um desenho chamado de estrela da morte, justamente por ser um ambiente de microsserviços extremamente caótico. Então, as pessoas começam a perceber que para evitar toda essa confusão de comunicação, podemos mapear o acesso aos nossos microsserviços através de múltiplas API Gateways.

Por exemplo, vamos imaginar que temos uma loja virtual. Nesta loja, todos os sistemas mexem com checkout. Então, para que possamos realizar pagamentos, temos 10 microsserviços. Mas ao invés das requisições baterem nesses micros-serviços, batem em outros pontos porque colocamos uma API Gateway na

frente de cada microsserviço. Assim, quem deseja chamar esses sistemas bate primeiramente nessa API Gateway. Neste caso, temos uma API Gateway para cada área, por exemplo, uma para logística, uma para transporte, etc.

Diminuir as chamadas diretas para os microsserviços garante não apenas segurança em relação à autenticação, mas também em relação ao Rate Limiting, redução da estrela da morte etc. pois quando todos falam através da API Gateway o processo é muito mais simples e organizado. Assim, cada time consegue expor suas URL caso seja necessário. E se os microsserviços precisarem fazer algum tipo de comunicação, isso é feito através da API Gateway.

Assim, essa ferramenta faz com que o ambiente seja bem mais organizado, preparado e, inclusive, mais fácil de documentar no dia a dia. Isso acontece justamente por conseguirmos amenizar essa confusão que surge quando criamos diversos microsserviços. Então, ao colocarmos uma API Gateway na frente desses microsserviços todos os processos de resiliência são facilitados.

Service Mesh

Hoje em dia, é muito comum que o Service Mesh seja usado por empresas que trabalham com microsserviço. Isso porque é um recurso que pode ajudar no processo de resiliência de seus sistemas. Porque consegue ter as informações de toda a comunicação que está acontecendo entre todas as aplicações. Então, fazendo esse monitoramento, o Service Mesh consegue controlar a rede, por ter

acesso a todas as comunicações que acontecem entre os sistemas.

O Service Mesh usa proxies para monitorar a comunicação entre os microsserviços. Por exemplo, temos o sistema A e o sistema B. E precisamos fazer a comunicação entre eles. Ao invés de mandarmos uma requisição direta do A para o B, colocamos um proxy antes desses sistemas. Assim, o sistema A envia a requisição para o proxy que está ao seu lado e esse proxy envia a requisição para o proxy do sistema B. Ou seja, toda comunicação acontece indiretamente através dos proxies.

Então, quando controlamos os proxies, conseguimos controlar também a rede. Isso só é possível justamente por conseguirmos visualizar a comunicação que está acontecendo entre os microsserviços. Então, podemos dizer que tendo um proxy na mão, conseguimos trabalhar com a proteção do próprio sistema. Ou seja, conseguimos evitar implementações que seriam bem mais complexas.

Para garantir a segurança dessa comunicação, podemos usar o método mTLS. Assim, em uma comunicação externa utilizamos o TLS, que é o antigo SSL, para termos a certificação dos sistemas. Então, no caso dos sistemas A e B: um sistema solicita ao outro a comprovação de que cada um é verdadeiro. Esse método é muito útil, pois quando temos muitos microsserviços, é complexo rotacionar as chaves de todos eles. E se torna ainda mais difícil quando trabalhamos com Service Mesh, pois esses próprios sistemas têm esquemas de rotacionar as chaves nos proxies, ou seja, o proxy sempre vai interceptar as mensagens para o seu microsserviço.

Isso muda completamente a forma como trabalhamos, facilitando nosso dia a dia. Porém, não estamos afirmando que trabalhar com service mesh é algo muito fácil. Como praticamente tudo no desenvolvimento de um sistema, trabalhar com esse recurso tem seu custo, gera overhead maior no sistema, mas por outro lado ajuda muito, o que de certo modo compensa.

Além disso, no Service Mesh conseguimos implementar o circuit breaker. Logo, ao invés do desenvolvedor ter que fazer rotinas, utilizar bibliotecas para trabalhar com circuit breaker, podemos fazer a configuração dessa estratégia usando o Service Mesh, o que facilita o trabalho do desenvolvedor. Fora isso, podemos usar esse recurso para fazer as políticas de retry, onde mandamos a requisição e estabelecemos outras tentativas no caso de não termos respostas a essas requisições. Mas dificilmente o desenvolvedor sabe quantas vezes deve tentar o retry. Normalmente, por não ter uma visão geral dos sistemas, os desenvolvedores "chutam" a quantidade de retry que deveria ser feita e isso acaba não sendo estratégico para os microsserviços. Então, o serviço de retry também pode ser aplicado, de maneira até mais assertiva, pelo Service Mesh, tirando da responsabilidade do desenvolvedor.

Com o Service Mesh também podemos aplicar outras estratégias como o timeout, onde mandamos uma mensagem e determinamos qual tempo máximo vamos trabalhar, além disso, aplicamos ainda o Fault Injection para conseguirmos fazer testes de resiliência em nosso próprio sistema.

Assim, temos diversas estratégias que podem ser aplicadas pelo Service Mesh.

Isso facilita bastante o trabalho do desenvolvedor pois não é necessário que essas estratégias sejam implementadas internamente. Isso garante economia de tempo e organização do ambiente. Quando um sistema tem diversos microsserviços, cada desenvolvedor pode fazer a implementação das estratégias de resiliência de um jeito diferente, até porque as linguagem desses sistemas são diferentes, então, em algum momento isso pode tornar o ambiente caótico. Mesmo que nem todas as empresas tenham esse recurso, por ser algo relativamente novo no mercado, aos poucos ele vem ganhando espaço, justamente por ajudar muito nesse processo de garantir a resiliência e mitigar problemas.

Trabalhe de forma assíncrona

O trabalho de forma assíncrona muitas vezes é negligenciado pelas pessoas que desenvolvem microsserviços. Apesar disso, é importante estudarmos e aplicarmos essa estratégia para que nosso sistema consiga enviar requisições, garantido a resiliência dos microsserviços.

Trabalhar de forma assíncrona pode ser enviar a informação para um message broker, ou seja, um sistema apropriado de mensagens. Além disso, trabalhar dessa forma significa garantir que essa mensagem esteja disponível para quando o outro sistema tiver interesse nela, isto é, quando o outro sistema estiver disponível possa consumi-la. Então, de maneira assíncrona, conseguimos receber muito mais informações. Porque se tivéssemos que trabalhar de forma síncrona, ou seja, mandando as requisições diretas de A para B em uma quantidade for

alta, esses sistemas podem não aguentar. Mas quando trabalhamos de forma assíncrona conseguimos receber todas as mensagens e mandar para um terceiro que vai armazenar essas mensagens e enviar para o outro sistema conforme ele puder consumir.

Muitos de nós desenvolvedores preferimos trabalhar com REST de um sistema A para B fazendo chamadas diretas, mas as chamadas assíncronas são interessantes, e diríamos até importantes para o bom funcionamento dos sistemas. Assim, trouxemos alguns exemplos de sistemas que nos ajudam trabalhar de forma assíncrona: o Kafka, o Message broker, o RabbitMQ, o Zeromq, o Amazon SQS, além desses, temos também o Redis que consegue trabalhar com pub sub. Então, temos muitas ferramentas hoje em dia para trabalhar dessa forma. É interessante conhecermos ao menos algumas delas pois na medida do possível é melhor trabalharmos com esses sistemas de mensageria numa arquitetura baseada em microsserviços.

Quando buscamos a resiliência de nossos sistemas, falamos muito em mitigação de riscos. Ou seja, queremos evitar a perda de dados e transações. Então, quando trabalhamos de forma síncrona conseguimos mitigar esses riscos porque se um sistema estiver fora do ar perdemos as informações, mas se trabalharmos de forma assíncrona essa informação não é perdida. Logo, sempre que um servidor estiver online consegue processar as informações caso em outro momento tenha precisado reiniciar, ficado fora do ar, feito um deploy etc. Assim, conseguimos evitar a perda de dados.

Além disso, é importante entendermos com profundidade o message broker. Dizemos isso, pois nos dias atuais tem sido muito comum os devs pesquisarem apenas um tutorial de como enviar e consumir as mensagens. E percebemos que um único parâmetro muda completamente a forma como o message broker vai trabalhar, o que pode fazer com que as informações sejam perdidas. Então, estudar para entender o sistema com profundidade é extremamente importante para garantir que tudo funcione corretamente.

Retry

Quando um sistema tenta enviar mensagem para outro e não consegue ter resposta é necessário fazer outras tentativas de envio dessa mensagem até conseguir uma respostas. Chamamos essa estratégia de Retry. Por exemplo, vamos imaginar que eu tenho o sistema “A” enviando mensagem para o sistema “B”. Porém, o sistema “B” não consegue responder a essa mensagem. Nesta situação, o sistema “A” terá que enviar essa mensagem novamente até conseguir ter a resposta do sistema “B”, ou seja, fazer um Retry.

O uso de bibliotecas para implementar o Retry automaticamente é muito comum quando as pessoas desenvolvem sistemas. Então, quando esse sistema faz um get, um post etc. caso não consiga receber a requisição, serão feitas novas tentativas de envio da mensagem. Se imaginarmos uma situação em que o Retry é feito de maneira automática, sem nenhum tipo de estratégia, ou seja, apenas tentando novamente sempre que a requisição não for recebida, a quantidade de envios

de requisições seria alta. E isso pode prejudicar o outro sistema. Mas se em uma segunda situação, o sistema esperar um segundo para enviar a próxima tentativa, as chances de resposta aumentariam. Porém, não somos os únicos a pensar nessa estratégia de aguardar um tempo para fazer outras tentativas de envio das requisições. Ou seja, provavelmente outros sistemas estão esperando um segundo para enviar as mensagens novamente. Então, essa segunda forma de enviar o Retry acaba não resolvendo muito nossa situação, pois também sobrecarrega o sistema que recebe essas requisições. Assim, se esse sistema estiver fora do ar, continuará dessa maneira sem conseguir se recuperar.

Chamamos a estratégia do primeiro exemplo, em que as chamadas são feitas de maneira seguida até outro sistema conseguir responder, de "chamadas lineares sem back off". A estratégia do segundo exemplo, onde aguardamos um segundo para enviar a mensagem é "chamada de exponential backoff". Na chamada exponential backoff esperamos um segundo para reenviar a requisição, mas caso o sistema não retorne esperamos 2 segundos, se ele continuar sem responder aguardamos 4 segundo, depois 8 segundos e assim por diante até conseguirmos a resposta do sistema, ou seja, as mensagens são enviadas exponencialmente. Assim, é uma estratégia diferente da primeira em que as requisições são enviadas de maneira linear. Quando enviamos de maneira exponencial damos mais tempo ainda para que o servidor possa se recuperar. Então, ele consegue responder mais requisições para mais clientes com menos chamadas. Porém, novamente não somos os únicos fazendo essa estratégia no envio das mensagens. Por isso, temos uma terceira estratégia, que inclusive tem sido muito utilizada nos dias

de hoje. O “retry exponential com jitter” é uma opção em que temos um ruído no envio das mensagens. Mandamos uma requisição e caso não seja respondida aguardamos um segundo e enviamos novamente com um número randômico. Dessa maneira, quando enviamos uma requisição com número randômico e outras pessoas enviam com outros números randômicos, a chance de mandarmos a requisição exatamente no mesmo tempo é muito menor. Dizemos que essa é uma alternativa para “embaralhar” os tempos das chamadas, evitando que esses tempos sejam exatamente iguais na hora que diversas pessoas estão fazendo as solicitações. Além disso, com o jitter precisamos fazer uma quantidade bem menor de solicitações e o sistema consegue responder mais clientes.

Garantias de entrega

A garantia de entrega é importante para termos certeza de que nossa mensagem foi recebida pelo nosso sistema de mensageria. Mas dependendo da situação, podemos escolher o quanto teremos certeza de que essa mensagem foi entregue. Ou seja, analisamos os riscos para verificar se naquele momento a prioridade é a certeza de entrega de todas as mensagens ou a performance da nossa aplicação, por exemplo.

Assim, para que possamos fazer essa análise é necessário conhecermos detalhadamente o sistema de mensageria que iremos trabalhar. Porque um parâmetro pode mudar a maneira como esse sistema vai funcionar. Então, quando trabalha-

mos enviando mensagem para um message broker, ou seja, de forma assíncrona, precisamos conhecer esse sistema para evitar a perda de mensagem.

Neste tópico, vamos usar como exemplo o sistema de mensageria Apache Kafka. Porém, o foco não deve ser exatamente o nome desse sistema, isso porque poderíamos usar qualquer outro sistema para fazer essa demonstração. Por outro lado, o foco deve ser em observar que um parâmetro muda toda garantia de entrega das mensagens enviadas para nosso broker. Assim, demonstraremos na prática como um único parâmetro pode afetar completamente a forma como vamos trabalhar. Então, se não pesquisamos e estudamos um pouco mais a fundo o funcionamento do nosso message broker podemos ter complicações em uma transação valiosa. Lembrando que, normalmente, devemos pensar em cada requisição com o valor de 1 milhão de reais, assim, dificilmente alguém vai querer perdê-la.

Quando usamos o Kafka, temos um produtor enviando mensagens. Por isso, precisamos fazer uma configuração onde teremos o parâmetro para fazer um Acknowledge. Isso significa que ao enviar uma mensagem se determinarmos que Ack é zero, esse Ack será none, ou seja, sem nenhum Acknowledgement. Então, enviamos uma mensagem para o Kafka, sem esperar uma resposta sobre essa mensagem ter sido entregue ou não. Ou seja, apenas enviamos várias mensagens para nosso broker aguardar a confirmação de que essa mensagem realmente chegou até ele. Isso porque em determinadas situações podemos até aceitar perder transações, principalmente porque dessa forma conseguimos enviar uma

quantidade bem maior de requisições, justamente por não ter que esperar o outro responder para garantir que a mensagem chegou. Por exemplo, vamos imaginar um serviço de delivery de comida que dispara a localização do entregador a cada 2 segundos. Ao disparar essa localização direto para o Kafka acontece algum erro e perdemos algumas localizações. Ou seja, nosso message broker não garantiu a entrega e perdeu essas informações. Porém, neste caso, a perda de algumas localizações não afeta o sistema como um todo. Mas se essa transação fosse no valor de 1 milhão de reais com certeza teríamos que garantir que essa mensagem chegasse. Assim, quando trabalhamos com Ack zero mandamos a mensagem e esquecemos. Chamamos isso de "Fire and forget", ou seja, "eu disparo e esqueço".

Outro parâmetro que podemos usar no Kafka é o Ack 1. Podemos dizer que no Kafka nós temos um broker que é o líder e os outros são os followers, ou seja, o líder é quem realmente vai receber a mensagem e replicá-las para os followers. Então, no Ack 1 mandamos a mensagem para o líder que ao persistir essa informação dará um retorno ao producer confirmando que a mensagem foi recebida. Desse modo, temos uma garantia de entrega diferente do Ack zero. Ou seja, percebemos que existe uma proteção bem maior nesse caso. Porém, existem algumas situações que, mesmo com essa proteção, podemos perder mensagens. Por exemplo, vamos imaginar que estamos enviando várias mensagens para o Kafka e o líder está confirmando. Então, determinada mensagem foi recebida e confirmada pelo líder, mas por algum motivo o HD "fritou" e esse líder ficou fora do ar antes de conseguir replicar essa informação para os followers. Assim, por mais que o líder tenha confirmado, a mensagem foi perdida. Fora isso, é

importante observarmos que no caso do Ack 1 temos menos performance por esperarmos essa confirmação do líder. Diferentemente do Ack zero onde temos mais performance pois não aguardamos nenhuma confirmação. Por outro lado, temos uma garantia de segurança muito maior da entrega, apesar de termos riscos de perda das informações por algum imprevisto.

Porém, temos um terceiro parâmetro chamado de OL, ou -1 onde temos uma garantia de entrega maior ainda do que no caso do Ack 1. Neste caso, nós enviamos uma mensagem para o líder, esse líder envia a mensagem para os followers. Depois disso, os followers enviam uma confirmação para o líder de que a mensagem foi sincronizada. Somente depois dessa sincronização, o líder manda essa mensagem para o producer. Isso significa que agora não tem erro, pois quando enviamos uma mensagem com -1 e recebemos a confirmação, temos a certeza de que essa mensagem foi replicada em todos os followers. Então, se o líder cair não teremos problemas, porque sabemos que os followers têm a cópia dessa mensagem. Mas é importante termos em mente que esse processo é mais custoso em relação a performance, justamente por termos todas essas confirmações. Por outro lado, a garantia de que a mensagem realmente foi entregue é completa. Diferente do Ack zero que não temos garantia e do Ack 1 que temos uma garantia intermediária.

Portanto, percebemos que um parâmetro muda completamente o comportamento do nosso broker. Mas nem sempre essa escolha é feita pela pessoa desenvolvedora. Isso porque a forma que iremos trabalhar precisa estar alinhada

com a área de riscos da empresa, ou seja, na maioria o gestor deve fazer essa análise. Então, são necessárias várias reuniões com a diretoria da organização para que os riscos sejam explicados e a escolha do parâmetro utilizado seja a mais estratégica possível para o negócio. Ainda que não seja uma escolha especificamente do desenvolvedor, é essencial estudarmos a fundo nosso sistema de mensageria para sabermos como aplicar essas estratégias da melhor forma possível em nossas aplicações.

Situações complexas

Quando pensamos em resiliência é importante imaginarmos todos os cenários possíveis. Então, mesmo que algumas situações sejam complexas e improváveis de acontecer precisam ser refletidas para estarmos preparados caso aconteçam. Dessa forma, podemos analisar cada risco, e mesmo que a nossa escolha seja assumir esses riscos, precisamos estar conscientes de que eles existem mas preferimos não tomar nenhuma decisão naquele momento.

Podemos dizer que existem duas opções relacionadas à resiliência: sabermos da existência do risco, deixando de agir intencionalmente e não sabermos da existência do problema. Na primeira opção, depois de fazermos todas as análises percebemos que podemos correr esse risco por isso de modo intencional não fazemos nada para mitigar esse problema. Porque temos consciência de que esse risco é mínimo e se acontecer uma ou outra vez, estamos dispostos a assumi-lo. Na segunda opção, por não termos consciência do problema, não sabemos se

podemos ou não assumir aquele risco, logo, essa decisão não é feita de maneira intencional. Assim, por não conseguirmos prever esse problema, precisamos "rebolar" para resolvê-lo. Isso acontece justamente porque desconhecemos o risco e não estamos preparados para assumir as consequências, ou seja, não estamos preparados para solucionar aquele problema rapidamente.

Uma situação complexa para refletirmos sobre a resiliência da nossa aplicação é a queda do nosso message broker. Neste caso, inicialmente podemos pensar que temos leaders, followers etc. e por isso a queda do Kafka não traria problemas para nosso sistema. Porém, precisamos pensar no pior cenário possível, isto é, caiu a Confluent, o datacenter da Amazon onde o Kafka estava rodando e realmente ficamos sem nosso Message Broker. Assim, nosso sistema manda frequentemente mensagens para o Kafka, então precisamos saber se nossa aplicação vai travar ou se vai perder mensagens quando o sistema de mensageria estiver fora do ar. Assim, é necessário fazermos testes para sabermos como nossa aplicação vai se comportar sem esse sistema.

Outra situação para pensarmos em resiliência está relacionada à possibilidade de nosso sistema ficar fora do ar. Neste caso, temos que pensar em quais estratégias iremos utilizar caso isso aconteça.

Além disso, é importante termos a consciência de que situações improváveis e inesperadas podem acontecer em nossa solução. Neste caso, podemos aplicar a Lei de Murphy que nos diz: "é exatamente onde não cobrimos o problema que o problema pode acontecer". Por exemplo, a queda do sistema de mensageria.

Geralmente pensamos que esse serviço não pode cair, e ficamos totalmente despreparados para essa queda. Mas pensar em resiliência deve estar relacionado diretamente ao risco dos piores cenários acontecendo com nossa solução, por isso, devemos estar preparados para ser resilientes caso nosso sistema de mensageria caia. Ou seja, devemos garantir que não vamos perder mensagens e que nosso sistema vai funcionar mesmo com essa queda. Então, se enviamos mensagens para nosso broker esperando garantia de entrega, e ele não responde, devemos ter em mente quais decisões vamos tomar e como nossa aplicação vai se comportar nessa situação.

Transactional outbox

Transactional outbox é um padrão que pode ser implementado em nosso software para mitigar os riscos caso nosso Message broker caia. Esse padrão está diretamente relacionado à criação de registros temporários para que os dados não se percam caso nosso sistema de mensageria fique fora do ar.

Por exemplo, vamos imaginar que estamos utilizando o Kafka como sistema de mensageria. Então, temos um serviço com banco de dados normal e uma tabela account, onde podemos inserir um insert, um update, um delete e um Kafka. Um sistema enviou uma mensagem que salvamos em nosso banco de dados para garantir que essa transação será enviada para o Kafka. Assim, digamos que o kafka caiu e como a mensagem que estava no banco de dados já havia sido

enviada, perdemos essa transação justamente por tentarmos enviar para nosso sistema de mensageria.

Porém, quando usamos o Transactional outbox, antes de mandarmos essa mensagem do banco de dados para o Kafka, vamos salvar a mensagem em uma outra tabela, não importa o lugar. Podemos chamar essa tabela de outbox, por exemplo. Nela, iremos fazer um insert. Então, a tabela outbox funciona como uma espécie de registro temporário, onde salvamos as mensagens no Kafka dessa tabela para enviarmos ao nosso sistema de mensageria assim que ele estiver no ar. Assim que o Kafka garantir o recebimento da mensagem, podemos deletar esse registro.

Então, caso o Kafka fique fora do ar, sabemos que teremos várias mensagens para enviar depois. A tabela temporária pode seguir algo como uma id da mensagem, uma key da mensagem, o tópico que será enviado Kafka e o payload. Dessa forma, temos uma tabela pronta para quando o Kafka estiver no ar novamente. Ou seja, esses dados ficam sendo gravados nessa tabela e conforme o Kafka confirmar que recebeu, apagamos os registros. Logo, conseguimos mitigar os riscos, mesmo que o Kafka saia do ar.

Apesar de ser um padrão bem simples, não podemos negligenciá-lo porque pode "salvar" nossa aplicação, garantindo a resiliência caso nosso message broker caia. Sem o Transactional Outbox continuaremos enviando mensagens para o Kafka mesmo ele estando fora do ar. Assim, perdemos as informações, o que pode começar diversas complicações em nosso sistema.

Garantias de recebimento

Nos tópicos anteriores, falamos sobre a garantia de entrega. Vimos que, ao enviarmos uma mensagem, podemos ou não ter a confirmação de que essa mensagem foi entregue. Neste tópico, veremos um pouco sobre a garantia de recebimento, ou seja, quando somos consumidores da mensagem. Assim, é importante sabermos que existem meios de garantir o recebimento das mensagens e inclusive de comunicar isso para o message broker. Essa decisão de forma intencional significa pensar na resiliência da nossa aplicação.

Garantir o recebimento das mensagens é um parâmetro extremamente simples, mas que faz toda a diferença em nosso dia a dia. Por isso, é necessário estudarmos a fundo nosso message broker para sabermos quais decisões tomar. Ou seja, quais parâmetros utilizar em cada situação.

Assim como na garantia de entrega, o nome específico do sistema de mensageria não precisa ser destacado. Mas a partir do momento que escolhemos um message broker para trabalhar com nossa aplicação, temos a necessidade de estudá-lo a fundo para conhecermos todos os seus parâmetros.

Neste tópico, vamos usar como exemplo o RabbitMQ apenas para ilustrar uma situação, mas podíamos trazer o nome de qualquer message broker, no final das contas, mudaria somente a forma como cada um trabalha. Então, imagine que somos um consumidor do RabbitMQ. E no momento que recebemos uma

mensagem aconteceu um erro ou caiu o pod do nosso sistema. Por isso, não conseguimos processar essa mensagem, mas ela saiu do message broker. Assim, a mensagem vai ser perdida, principalmente se estivermos trabalhando com Auto Ack. Isso porque, trabalhando dessa forma o sistema de mensageria pode dar um purge, ou seja, limpará a mensagem após envio. Com isso, percebemos que um único parâmetro pode fazer com que as mensagens sejam perdidas. Porém, se mudarmos esse Auto Ack para false, mesmo que nosso sistema saia do ar, é possível recebermos essa mensagem assim que voltamos ao ar. Isso acontece porque devido o Auto Ack estar false, não temos uma confirmação de recebimento da mensagem para o RabbitMQ. Então, na prática do dia a dia, ao recebermos uma mensagem, primeiro fazemos seu processamento e depois fazemos um commit para que o nosso message broker tenha a confirmação de que recebemos essa mensagem. Somente após esse processo, a mensagem será descartada do sistema de mensageria. E caso nosso sistema caia antes de conseguirmos processá-la, recebemos a mensagem novamente.

Portanto, o trabalho usando Auto Ack pode nos trazer muitos problemas, principalmente relacionados à resiliência da nossa aplicação. Com isso, podemos dizer que a parte que envolve a garantia de recebimento quando somos consumidores está diretamente ligada aos nossos conhecimentos técnicos.

Além disso, precisamos observar como é complicado abrir a comunicação sempre que recebemos uma nova mensagem. Por isso, é mais indicado receber essas requisições em um batch de mensagens, ou seja, estabelecendo um pre-

fetch de acordo com a quantidade de requisições que nosso sistema consegue processar. Por exemplo, se o nosso sistema processa 100 requisições por vez, determinamos que a comunicação vai ser aberta a cada 100 mensagens. Isso é importante pois caso o consumidor não esteja conseguindo processar muito bem as requisições por estar com problemas de lentidão, a comunicação não é aberta frequentemente.

Então, digamos que um sistema de mensageria envia 100 requisições por segundo para uma aplicação. Caso ele continue enviando mais 100 requisições mesmo sem ter um retorno, essa aplicação pode ser prejudicada e até mesmo cair por causa de uma sobrecarga. Assim, podemos fazer testes de stress, verificar o quanto de hardware temos etc. Dessa forma, sabemos o número ideal de prefetch para termos a melhor performance e o menor risco possível de ficar fora do ar por causa das mensagens "encadeadas" que foram mandadas em bach para nossa aplicação. Detalhes como esses podem prejudicar nossa aplicação, por esse motivo é necessário pensarmos com cautela sobre as garantias de recebimento.

Idempotência e políticas de fallback

Nos sistemas monolíticos, as comunicações acontecem dentro do próprio da própria aplicação, através de bibliotecas, módulos, etc. Porém, no caso dos microsserviços, frequentemente temos comunicações que acontecem de maneira externa aos sistemas. Por causa disso, precisamos trabalhar com sistemas de mensageria, com garantias de recebimento etc. Além disso, precisamos trabalhar

com idempotência e políticas de fallback para que nossos microsserviços tenham uma boa comunicação e sejam resilientes.

A idempotência é o ato de conseguirmos lidar com a duplicidade de uma informação. Por exemplo, vamos imaginar que recebemos e processamos uma mensagem de um depósito. Por isso, adicionamos essa mensagem na coluna do banco de dados, informando que o depósito aconteceu. Mas por algum erro no consumidor de quem está publicando a mensagem ou algum erro do sistema de mensageria, essa mensagem é enviada novamente. Então, a mensagem é adicionada novamente em nosso banco de dados. Isso pode gerar um problema para nossa aplicação, pois irá constar dois depósitos quando na verdade só existe um. Assim, temos que ser idempotente para perceber essas duplicidades e conseguir fazer o descarte das mensagens repetidas, evitando problemas que podem ser catastróficos no dia a dia. Ou seja, esse é um tipo de risco que não podemos deixar de mitigar. Existem diversas maneiras de sermos idempotentes, por exemplo através da identificação do ID da mensagem. É importante analisarmos qual melhor maneira de fazermos isso e colocarmos como um ponto a ser lembrado em nosso check list de resiliência.

Além disso, é importante termos políticas claras de fallback para que nosso sistema esteja preparado para lidar com as mais diversas dificuldades no dia a dia. Essas políticas de fallback estão relacionadas a necessidade de termos planos A, B, C etc. para que nosso sistema seja resiliente. Isso significa ter esses planos documentados da melhor forma possível. Então, é necessário pensar, de

maneira antecipada, em quais problemas podem acontecer com nossa aplicação, inclusive em situações que são pouco prováveis de acontecer. Por exemplo, imagine uma empresa que tem muitos servidores trabalhando com WS. Quando trabalhamos com cloud providers, criamos um virtual private network, uma VP e algumas subnets, ou seja, grupos de sub redes para subir as máquinas virtuais. Quando a divisão dessas subnets é feita, a quantidade de máquinas que estimamos para cada uma dessas sub redes também é determinada. Assim, essa empresa sobe milhares de máquinas no ar de acordo com essa divisão. Cada sub rede fica em uma zona de disponibilidade diferente, ou seja, em data centers diferentes. Isso garante que, caso aconteça algum problema em um dos serviços, os outros serviços não fiquem fora do ar. Dessa forma, percebemos que tudo está feito de uma forma adequada. Porém, vamos imaginar que a WS saiu do ar, ou seja, uma zona de disponibilidade caiu e todas as máquinas que estavam naquele grupo de sub rede caíram também. Nesta situação, a empresa tem uma política de drenagem, ou seja, todas as máquinas que estavam naquela zona de disponibilidade são transferidas para outra zona. Isto é, seria somente uma questão de tempo para subir as outras máquinas e tudo estaria “Ok”. Mas o verdadeiro problema surge pois a empresa tinha uma quantidade exorbitante de máquinas que não couberam na outra zona de disponibilidade, pois a subnet não foi configurada para essa quantidade maior de máquinas naquele grupo, ou seja, naquele range de IPs não coube as máquinas vindas da zona que saiu do ar. Assim, como a empresa não tem IPs suficientes, tem um problema muito grande para resolver. Esse exemplo é um problema raro, pois a maioria de nós

não trabalhamos em uma Big Tech. E ainda quem trabalha em uma Big Tech, muitas vezes, não pensa que esse tipo de problema pode acontecer. Por isso, as políticas de fallback precisam ser claras, justamente porque pode acontecer algum tipo de dificuldade que não imaginamos que poderia acontecer. Sendo assim, com políticas de fallback claras conseguimos investigar o problema e dar continuidade ao nosso trabalho.

Assim, desenvolver um sistema monolítico é bem mais simples do que trabalhar com microsserviços. Principalmente quando o trabalho com microsserviços envolve muitas máquinas. Por isso, ao escolher a maneira que irá desenvolver uma solução é importante que a empresa analise a maturidade do time, evitando cair em "ciladas" seja por falta de domínio técnico e principalmente pela imposição de uma alta complexidade se real necessidade.

Observabilidade

A observabilidade é uma estratégia que nos auxilia no processo de criação de métricas para observar se uma aplicação está funcionando conforme esperado. Por isso, dizemos que ter observabilidade é fundamental para que os sistemas sejam resilientes. Então, podemos dizer que subir uma aplicação sem observabilidade é praticamente impossível, pois seria como voar em um voo sem visão, ou colocar uma fita isolante no painel que marca o combustível de um carro para fazer uma viagem longa. Não conseguimos pensar em fazer esse voo ou essa viagem, porque nos faltam as métricas que garantiriam nossa segurança.

De modo semelhante, nosso sistema precisa de métricas para garantir que ficará em ordem.

Quando temos observabilidade entendemos exatamente o motivo dessa segurança ser importante para nossa aplicação. Assim, é essencial conhecermos ao menos os pilares básicos desse tema, e mesmo que neste momento não vejamos o assunto de maneira aprofundada, com esses conceitos iniciais, teremos o conhecimento necessário para compreendermos a necessidade de ter observabilidade em nossos microsserviços.

Podemos dizer que a observabilidade nos ajuda com APM e Application Performance Monitoring. Assim, conseguimos ver o que está acontecendo em nossa aplicação. Ou seja, conseguimos ver exatamente em qual momento o problema aconteceu. Além disso, podemos ver como estavam o banco de dados e a linha código quando o problema aconteceu.

Quando trabalhamos com observabilidade podemos usar alguns recursos para nos auxiliar nesse processo. Por exemplo, o tracing distribuído. Recursos como esse são necessários para conseguirmos trabalhar em um ambiente baseado em microsserviços. Isso porque, ao recebermos uma requisição do serviço A, precisamos chamar o serviço B, que vai chamar o serviço C e assim por diante. Então, temos que pensar no que acontece no meio dessa comunicação. Por exemplo, quando temos um erro na aplicação, precisamos saber em qual microsserviço o erro aconteceu. Assim, trabalhando com tracing distribuído, conseguimos ver o que está acontecendo em nosso sistema através de um

painel. E neste exemplo do jaeger, conseguimos entender onde o erro aconteceu. Mas podemos perceber também como a request caiu. Ou seja, ver esse request atravessando todos os microsserviços. Tudo isso, por meio da observabilidade do sistema.

Além disso, na observabilidade, temos que ter métricas personalizadas. Esse aspecto pode ser frequentemente deixado de lado por muitos devs na hora de desenvolver suas aplicações. Porém, com essas métricas específicas conseguimos garantir que o sistema funcione conforme o planejado. Isso inclui trazer métricas do negócio para o sistema. Por exemplo, imagine que uma loja virtual faz cerca de 100 vendas de produtos por hora. E de repente observamos que naquela hora aconteceu somente 10 vendas. É importante termos esse controle de saber quando a loja vendeu 100 produtos ou somente 10. Ou seja, quando temos métricas, conseguimos essa informação. Assim, podemos setar um alarme, porque utilizamos essas métricas para fazer a detecção de anomalias em nossa aplicação. Inclusive, existem muitos sistemas que usam esse meio de observação e machine learning para disparar alarmes que mostram essas anomalias automaticamente.

Quando trabalhamos com APM temos a opção de auto instrumentação. Assim, pegamos uma lib em nossa API favorita ou do nosso vendor, um New Relic, um datadog, etc, para subir e auto instrumentar trazendo todas as informações do que está acontecendo. Por exemplo, podemos ter um programa em Java que conseguimos auto instrumentar. Por isso, ele nos mostra tudo o que está acontecendo dentro do nosso software. Mas queremos personalizar o que está

acontecendo, ou trazer essas informações deste software de forma mais clara. Assim, ao invés de utilizar auto instrumentação para tudo, conseguimos trazer também spans personalizados. Esses spans são cada etapa do que acontece no nosso sistema. Ou seja, nesse programa temos várias linhas do que acontece, e cada linha dessa pode ser considerada um span. Assim, conseguimos personalizar as linhas do nosso código. Para compreendermos melhor como isso funciona, podemos imaginar que temos uma função que chama uma API e que grava arquivos grandes em disco. Pelo sistema de observabilidade, conseguimos ver que essa função está lenta. Mas em uma situação como esta, dificilmente conseguimos saber onde a lentidão está sendo causada, se é na API ou na gravação de arquivo em disco. Por isso, os spans podem ajudar nessa observação. Porque com eles definimos onde começa e termina cada uma das etapas da aplicação. Logo, quando cai no tracing conseguimos ver quanto tempo demorou a API e quanto tempo demorou a gravação em disco.

Ainda sobre observabilidade, existe um projeto chamado Open Telemetry que pode auxiliar. Esse projeto é uma iniciativa que ajuda com as métricas de observabilidade, contribuindo para que o sistema não dependa diretamente do vendor. Então, mandamos todas as informações para um coletor que é um serviço. Esse serviço, manda essas informações para o seu provider. Por exemplo, caso nosso sistema utilize o New Relic e, por algum motivo, precise mudar isso, podemos fazer essa mudança de coletor mandando as informações para um datadog. Dessa forma, conseguimos trabalhar e filtrar por meio do Open Telemetry. Além disso, o projeto tem SDK para diversas linguagens de

programação. Assim, temos acesso a muitas coisas estáveis para trabalharmos e utilizarmos na produção.

Últimas palavras

As estratégias que estudamos neste capítulo podem ser um check list para desenvolvermos softwares de fato resilientes. Inicialmente, vimos os conceitos dessas estratégias, pois antes de irmos para a parte prática, esses conceitos precisam estar fixados em nossa mente. Dizemos isso porque independente do código, da linguagem de programação ou do message broker esses conceitos são os mesmos. Então, ao dar continuidade aos nossos estudos, podemos nos aprofundar em conhecimentos mais específicos de cada estratégia de maneira mais prática.

Usando essas estratégias, podemos garantir a resiliência do nosso software e evitamos problemas muito sérios no ambiente de nossos microsserviços. Criar um sistema que se comunica com outros é simples e qualquer pessoa desenvolvedora pode fazer, mas dominar o que está por trás de tudo isso é algo mais complexo. Pois para ter esse domínio, precisamos conhecer a fundo os conceitos dessas estratégias de resiliência. Além disso, é importante termos em mente que saber o que está por trás de tudo pode nos “salvar” quando nossa empresa estiver em um momento caótico.

Coreografia e orquestração

Sabemos que no trabalho com microsserviços é fundamental compreendermos como eles se comunicam. Por isso, devemos conhecer as principais formas de comunicação entre os sistemas. É comum imaginarmos duas formas principais de comunicação: a síncrona e a assíncrona. Porém, neste capítulo, vamos estudar algo diferente sobre a forma que os microsserviços se comunicam. Assim, veremos alguns conceitos relacionados à comunicação dos microsserviços como uma coreografia e com uma orquestração.

Quando conseguimos entender os principais conceitos sobre esse assunto, podemos melhorar o processo de comunicação entre os nossos microsserviços. Isso porque, esses conhecimentos nos ajudam a tomar decisões mais adequadas por nossos microsserviços, além disso, conseguimos manejar os ambientes aprendendo algumas técnicas relacionadas ao tema. Por isso, é interessante que as pessoas desenvolvedoras de microsserviços tenham ao menos noções básicas de coreografia e orquestração, principalmente porque esse assunto vai além da comunicação, tratando também do comportamento dos microsserviços. Portanto, nos próximos tópicos explicaremos como a coreografia e a orquestração envolvem a comunicação entre os microsserviços, para entendermos as diferenças entre elas e quando utilizar cada uma.

Como funciona a coreografia

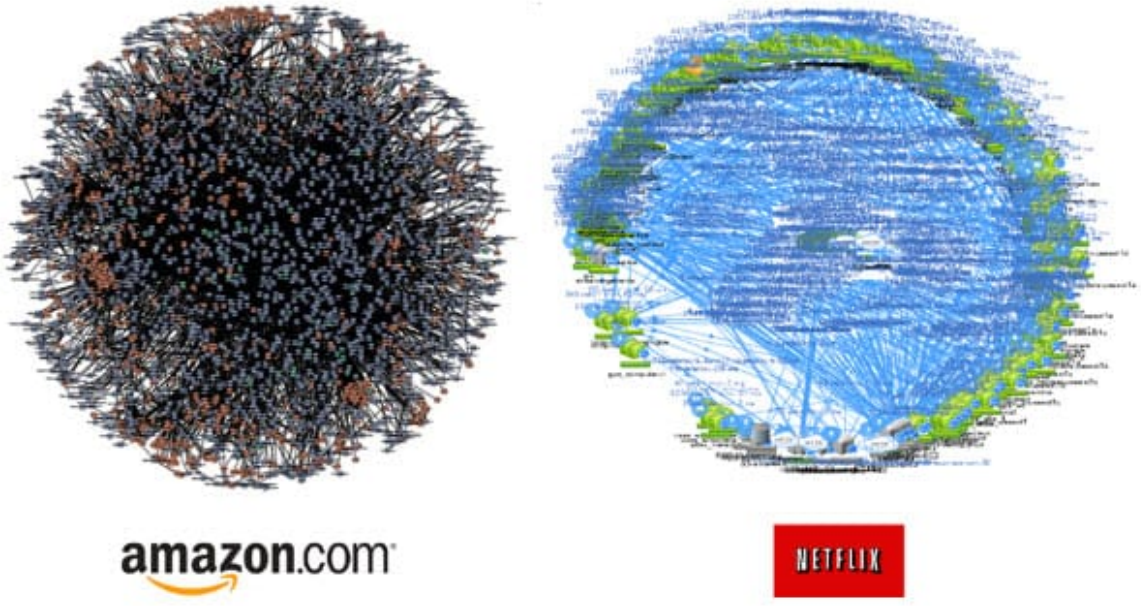
A comunicação orgânica entre os microsserviços pode ser comparada a uma coreografia de dança. Isso porque, em uma apresentação de dança cada participante tem seu papel com movimentos específicos, resultando em uma coreografia que forma a dança. Assim, o resultado acaba sendo cada vez mais descentralizado. A comunicação entre os microsserviços acontece mais ou menos como uma coreografia. Temos o primeiro microserviço e segundo microserviço e o terceiro microserviço, com seus papéis específicos dentro de uma organização. Ou seja, cada sistema é independente, mas todos trabalham em conjunto para alcançar um objetivo.

Quando falamos em coreografia relacionada aos microsserviços, podemos dizer que existe certa "beleza" relacionada ao fato deles se comunicarem de forma independente. Isso significa que o microsserviço 1 pode "bater" no microsserviço 5 para responder uma chamada. Esse microsserviço 5 "bate" no microsserviço 7 para responder suas requisições. Então, o microsserviço 7 vai "bater" no microsserviço 6 e assim por diante. Ou seja, conforme a necessidade, um microsserviço "bate" em outro para responder às chamadas.

Logo, em um primeiro momento observamos que essa comunicação traz certa confusão nesse ambiente. Mas isso acontece organicamente, isto é, não combinamos que a comunicação seria assim. Mas os microsserviços surgiram e começaram a fazer chamadas entre eles para responder suas requisições. Então,

nessa coreografia que acontece de maneira orgânica, mesmo que cada um tenha seu papel não temos um maestro falando como essa comunicação deve acontecer. E em um ecossistema com muitos microsserviços, ou seja, com muito tráfego e muitas chamadas, vamos perceber que essa comunicação acaba sendo confusa.

Para compreendermos melhor como isso funciona, podemos lembrar da "Estrela da Morte" criada pela Netflix e pela Amazon. Se digitarmos "Death Star Netflix versus Amazon" no Google, veremos algumas imagens mostrando um globo com diversos pontos com linhas simbolizando ligações entre esses pontos. Assim, cada ponto representa um microsserviço e as ligações são a comunicação que acontece entre eles. Dessa forma, é possível observar uma confusão tão grande que é como se perdêssemos o "controle" de toda a comunicação da nossa rede.



Estrela da morte Amazon versus Netflix

Na imagem anterior, temos uma noção de como uma coreografia com muitas

chamadas acaba gerando um ambiente confuso e, consequentemente, difícil de administrar. Assim, quando a quantidade de microsserviços é muito grande, cenários como esses da estrela da morte existem, pois está havendo uma coreografia entre os microsserviços. Porém, é importante sabermos que podemos utilizar técnicas para conseguir mitigar um pouco dessa confusão.

Dinâmica de orquestração

Quando trabalhamos em um ambiente de microsserviços, nem sempre a comunicação na forma de uma coreografia orgânica é a melhor opção. Por vezes, esse ambiente se torna tão confuso e, por isso, temos alguns problemas que dificultam o resultado esperado. Neste caso, uma alternativa é trabalharmos com a comunicação de forma orquestrada.

O objetivo da comunicação orquestrada é garantir que os microsserviços sejam organizados com uma ordem específica para que cada etapa seja desfeita em caso de algum erro. Isso pode ser feito de diversas formas, mas inicialmente precisamos compreender como essa orquestração de microsserviços funciona. Podemos imaginar um concerto musical, onde temos um maestro para orquestrar os músicos. Sabemos que esse maestro tem o controle exato do momento que cada instrumento deve ser tocado. Então, cada músico possui sua partitura mas aguarda os comandos desse maestro. E durante esse concerto, em diversos momentos o músico aguarda parado ao invés de tocar o instrumento. Ou seja,

aguarda o momento de entrar e tocar. Assim, tudo acontece sob a coordenação do maestro.

Na orquestração dos microsserviços, também precisamos ter uma coordenação no processo de chamadas dos nossos sistemas. Por exemplo, temos uma solução que trabalha com vendas. Então, temos o microsserviço de checkout, o de payment, o de estoque e o do centro de distribuição. Além disso, temos um microsserviço de shipping que faz o transporte das mercadorias. Percebemos que existe um entendimento simples de como deve ser a comunicação entre esses microsserviços. Assim, fazemos uma compra que chama o payment, caso o payment tenha sido autorizado, ele deduz o estoque e encaminha um desses produtos para o centro de distribuição. O centro de distribuição entra em contato com o shipping, e nesse meio temos a parte de invoice e nota fiscal e, depois disso, o shipping faz a entrega do produto. Lembrando que nem todos os sistemas de vendas funcionam dessa maneira, mas neste exemplo, temos uma comunicação simples de compreender. Porém, imagine que no momento que um microsserviço bateu no centro de distribuição, por um motivo grave, esse microsserviço falhou E por esse motivo o Invoice pode ter sido disparado. Algumas vezes as situações não acontecem uma depois da outra. Assim, no estoque, ao invés de ter 100 produtos, tem apenas 99. Então, o cliente fez o pagamento e espera por seu produto, mas temos um efeito colateral da falha em uma das etapas. Isto é, essas etapas precisam acontecer em determinada ordem. Por exemplo, não podemos devolver o pagamento antes de cancelar a nota fiscal. Então, nessas e em outras situações é necessário garantir a ordem.

Portanto, precisamos de algo que nos ajude a coordenar esse fluxo de trabalho. E é importante dizermos que isso não é somente chamada assíncrona, pode ser também eventos ou qualquer coisa que trabalhamos. Assim, precisamos de um orquestrador, ou seja, um mediador para nossos microsserviços.

E por esse motivo o Invoice pode ter sido disparado. Algumas vezes as situações não acontecem uma depois da outra. Assim, no estoque, ao invés de ter 100 produtos, tem apenas 99. Então, o cliente fez o pagamento e espera por seu produto, mas temos um efeito colateral da falha em uma das etapas. Isto é, essas etapas precisam acontecer em determinada ordem. Por exemplo, não podemos devolver o pagamento antes de cancelar a nota fiscal. Então, nessas e em outras situações é necessário garantir a ordem. Portanto, precisamos de algo que nos ajude a coordenar esse fluxo de trabalho. E é importante dizermos que isso não é somente chamada assíncrona, pode ser também eventos ou qualquer coisa que trabalhamos. Assim, precisamos de um orquestrador, ou seja, um mediador para nossos microsserviços.

Porém, voltando a imaginar um possível erro no centro de distribuição do produto, por uma falha que fez com que o parássemos nessa etapa do processo, nosso mediador entra com o seu plano de fallback. Então, quando algo não responde como esperado, esse maestro vai desfazer o que é necessário para terminar esse fluxo.

Assim, o maestro controla o fluxo de trabalho e com isso consegue garantir que todas as etapas foram concluídas. Quando todas as etapas são finalizadas

corretamente, temos o retorno de que o processo aconteceu como esperado e podemos encerrar esse fluxo. Mas caso alguma das etapas não aconteça como deveria, o maestro desfaz as etapas na ordem correta. No exemplo anterior, vimos que a compra de um produto teve um processo de comunicação simples, por isso conseguimos ter a compreensão de como essa mediação acontece. Porém, precisamos ter em mente que existem eventos mais complexos e, por isso, precisamos de ferramentas mais complexas também.

Em uma empresa, os erros podem acontecer em diferentes etapas de fluxos de diferentes áreas, por isso pode ser necessário trabalharmos com Business Process Management Language, BPMN etc. Porque precisamos mapear todos esses fluxos para sabermos como o erro aconteceu, já que um fluxo pode ter relação direta com outros fluxos. Esse mapeamento auxilia na visualização da rede de microsserviços contribuindo para que a orquestração aconteça de maneira organizada e ordenada. Por exemplo, vamos imaginar que devido uma falha na etapa do centro de distribuição, precisamos entrar em outro fluxo. Isto é, procuramos outro centro de distribuição para resolvermos o problema nessa etapa. Com isso, podemos ir para o Invoice e depois para o shipping. Ou seja, percebemos que um fluxo pode ter um sub fluxo que chama outro sub fluxo. Portanto, as decisões dependem do processo. E esses processos dependem de como as etapas acontecem dentro de cada empresa. Assim, dizemos que não depende somente do desenvolvedor, mas sim das decisões de negócio daquela organização. Neste exemplo, desfazemos o primeiro estoque para procurar em outro centro de distribuição e se quisermos podemos dar continuidade mas caso

contrário todos os outros são cancelados.

As decisões dos exemplos anteriores são feitas de forma orquestrada. Eventualmente acaba sendo complexo lidar com isso, pois trabalhar com um fluxo extremamente predefinido é difícil. Pois quando as etapas acontecem de forma assíncrona, a ordem pode ser alterada inesperadamente, o que dificulta nosso trabalho. Por esse motivo que a orquestração, apesar de ser mais complexa, tem diversas vantagens por ter esses fluxos e estratégias de controle. Ou seja, temos padrões de microsserviços para conseguir fazer diversos tipos de controle.

Então, se compararmos a forma de comunicação orquestrada com a coreografia, veremos que orquestrar pode ser bem mais complexo. Mas é importante conhecermos as formas que a comunicação pode acontecer em um ambiente de microsserviços. Esses conhecimentos nos preparam para lidar com diversas situações. E mesmo que os pontos estudados aqui sejam teóricos, são essenciais para compreendermos como essas comunicações funcionam na prática.

Estratégias de APIs

Quando trabalhamos com coreografia em nossos microsserviços, podemos utilizar estratégias para mitigar o resultado de uma “Estrela da Morte”. Se nosso trabalho for orquestrando os microsserviços, também podemos utilizar dessas estratégias, mas neste tópico falaremos mais especificamente do trabalho com coreografia dos microsserviços.

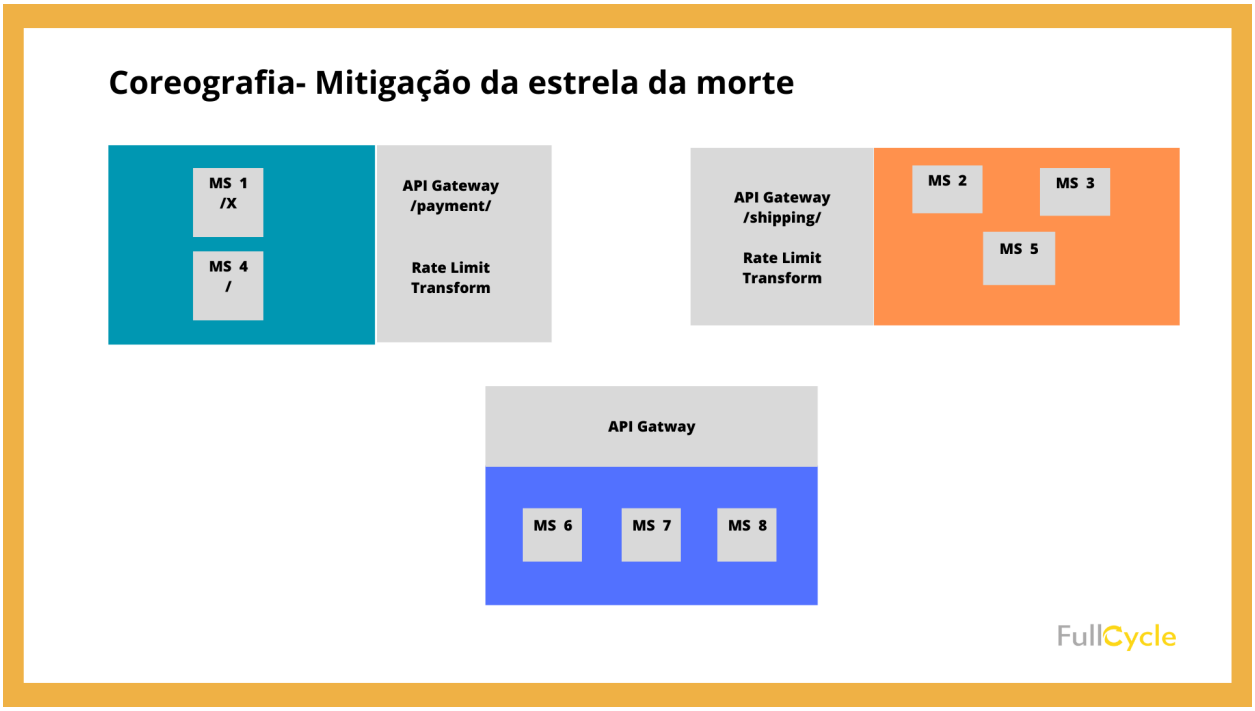
Na comunicação de coreografia temos, basicamente, diversos microsserviços soltos que vão se chamando de maneira exaustiva. O resultado dessa comunicação é um ambiente confuso. Porém, mesmo nesse tipo de comunicação podemos usar estratégias para termos mais controle desse ambiente. Por exemplo, vamos imaginar que temos diversos microsserviços se comunicando. Cada um desses microsserviços têm relação com uma etapa do processo para vender determinado produto. Então, em nosso exemplo, podemos dizer que os microsserviços 1 e 4 estão no contexto de pagamentos. Por outro lado, os microsserviços 2, 3 e 5 estão no contexto de shipping e os microsserviços 6, 7 e 8 estão no contexto de estoque. Isso acontece justamente porque cada um desses microsserviços está relacionado a uma etapa específica dessa venda. Porém, mesmo estando no mesmo contexto, eles são independentes e cada um possui sua URL na API. Assim, o microsserviço 1 tem URL /x, enquanto o microsserviço 4 tem URL /. Quando o microsserviço 2 precisar chamar o microsserviço 1, que faz parte de um contexto diferente, basta fazer a chamada para a URL /x . Na prática, isso não muda o ambiente e teremos uma "Estrela da Morte" difícil de manusear. Por esse motivo, mesmo que nosso trabalho seja com coreografia podemos usar algumas técnicas para mitigar a confusão.

Umas das possíveis técnicas é trabalhar com minis API Gateways. Assim, podemos imaginar que colocaremos uma API Gateway na frente de cada contexto. Por exemplo, na frente do contexto de pagamento colocamos uma API Gateway chamada de /payment. Dessa forma, para que o microsserviço do contexto de distribuição consiga falar com um microsserviço do contexto de pagamento, ele

terá que passar por um Proxy. E esse Proxy tem um mapeamento de URL para conseguir se comunicar com esses microsserviços. Então, quando o microsseviço 2 quiser chamar o micrisserviço 1, ele não faz uma comunicação direta. Neste caso, ele chama /payment /x. Com isso, nenhum desses microsserviços sabe diretamente qual sistema está chamando. Justamente porque ele está batendo em uma API Gateway, e é ela quem faz a redistribuição das nossas URLs. Assim, deixamos nossa rede cada vez mais organizada.

Com essa estratégia de minis API Gateway, ao invés de nos comunicarmos diretamente com cada microsserviço, nos comunicamos primeiro com o contexto. Depois disso, a comunicação acontece dentro do contexto que fizemos a chamada. Assim, em nossa API Gateway conseguimos implementar estratégias como Rate Limit. E determinar que se a chamada vier do contexto de distribuição, por exemplo, podemos receber 100 requisições por segundo. Mas caso essa chamada venha do contexto de estoque recebemos 200 requisições por segundo. Então, conseguimos transformar as requisições e usar os diversos recursos que uma API Gateway possui. Podemos observar que isso faz com que nossa rede fique bem mais organizada, apesar da quantidade de comunicação. E como essa comunicação entre os microsserviços acontece através de uma API Gateway, conseguimos ver os contextos e os microsserviços através desses contextos. Isso nos possibilita limitar e ter um controle bem maior da rede.

A imagem abaixo ilustra a estratégia de colocar colocar APIs em frente dos contextos dos microsserviços:



É importante sabermos que apesar das semelhanças, não se trata de um Service Mesh. Ou seja, não estamos falando que é um sidecar proxy, mas sim especificamente que é uma API Gateway. Isto é, pode ser um Kong configurado em nosso kubernetes, como um Ingress, em que para bater em nossos microsserviços precisa bater na API Gateway. Então, essa é uma forma estratégica de fazer essa comunicação. E conseguimos reduzir a confusão desse ambiente apesar de estarmos trabalhando com uma coreografia de microsserviços.

Assim, com essa estratégia de API conseguimos, basicamente, fazer um agrupamento, uma redistribuição e conseguir subir totalmente stateless, mesmo sem precisarmos ter um banco de dados para fazer isso. Então, acaba sendo algo mais simples, leve e tranquilo. Isso porque conseguimos enxergar a rede de uma forma muito melhor e não "atropelamos" as comunicações, o que evita gerar uma

confusão entre os microsserviços. Porém, sabemos que, conforme a rede cresce, os contextos podem ser maiores, e por isso, precisamos estudar a granularidade. Por isso, dizemos que não existe uma mágica quando temos microsserviços. Mas, pelo menos internamente em nossa empresa, conseguimos ter um olhar diferenciado através dessa estratégia de comunicação, principalmente se essa comunicação forma assíncrona, ou seja, de A para B.

Patterns

Em um ambiente baseado em microsserviços, existem centenas de padrões que podem facilitar o nosso trabalho. Alguns desses padrões são usados com mais frequência do que outros, porque eles suprem necessidades muito claras no nosso dia a dia. Então, podemos dizer que é comum pensarmos em situações onde utilizar esses padrões pode ser útil para nos ajudar na realização de processos que são diários em nossas aplicações. Por exemplo, temos uma situação em que precisamos gerar um único relatório com informações que estão em microsserviços diferentes. Ou seja, é necessário juntar os dados que estão em lugares diferentes para gerar esse relatório. Neste exemplo, percebemos que o trabalho com aplicações distribuídas pode trazer dificuldades que normalmente não temos no trabalho com sistemas monolíticos. Essas situações de dificuldades com microsserviços, geralmente, são conhecidas e, por isso, podemos implementar padrões sem perder tempo pensando no que poderia ser feito para solucionar o problema. Porém, é importante termos em mente que podemos diminuir o tempo para escolher uma solução, mas eventualmente a codificação pode ser trabalhosa e demorada.

API- Composition- Parte-1

Quando precisamos gerar um relatório, e os dados necessários para isso estão em microsserviços diferentes, podemos usar o padrão de API Composition. Com esse padrão conseguimos compor essas informações em um único lugar e assim resolver nosso problema.

Inicialmente criamos um /relatório por precisarmos das informações em um único lugar. Porém, vamos imaginar que metade dos dados estão no microsserviço A e a outra metade no microsserviço B. Logo, é necessário pensarmos em uma solução que junte essas informações. Em determinadas situações resolver esse problema é algo simples. Por exemplo, quando trabalhamos com um Frontend. Neste caso, fazemos uma request get no microsserviço A e outra request get no microsserviço B. Assim, juntamos esses dados para que possam ser exibidos na tela. Isto é, nessa possibilidade colocamos (criamos) um Client- "Frontend" para conseguirmos juntar os dados que precisamos em nosso relatório. Então, podemos dizer que essa seria nossa primeira opção (ver imagem "Opção 1" no final deste tópico).

Mas podemos ter outra situação em que precisamos prestar algum serviço que não vai necessariamente para o Frontend. Nesta outra situação, podemos ter um service que gere o relatório, ou seja, um service específico de relatórios. Então, podemos chamar esse serviço de service composer, pois ele vai fazer a composição das informações que precisamos. Nosso service composer faz uma

chamada para o microsserviço A e outra para o microsserviço B. Com isso conseguimos juntar os dados que precisamos para gerar o relatório. Depois disso, podemos ter um Frontend para chamar esse service. Essa pode ser a nossa segunda opção para resolver nosso problema de dados distribuídos (ver imagem "Opção 3" no final deste tópico). Lembrando que é importante termos em mente que essas soluções não são como uma "pólvora" que simplesmente soluciona todas as dificuldades.

A terceira opção para resolvermos esse problema poderia ser inserir uma API Gateway na frente de cada um dos microsserviços. (ver imagem "Opção 3" no final deste tópico). Então, essa API Gateway faz os redirecionamentos, e as junções dessas informações para retornar uma response ao usuário. Essa estratégia é bem interessante pois as APIs Gateway conseguem fazer bem esse processo de redirecionar as informações que precisamos.

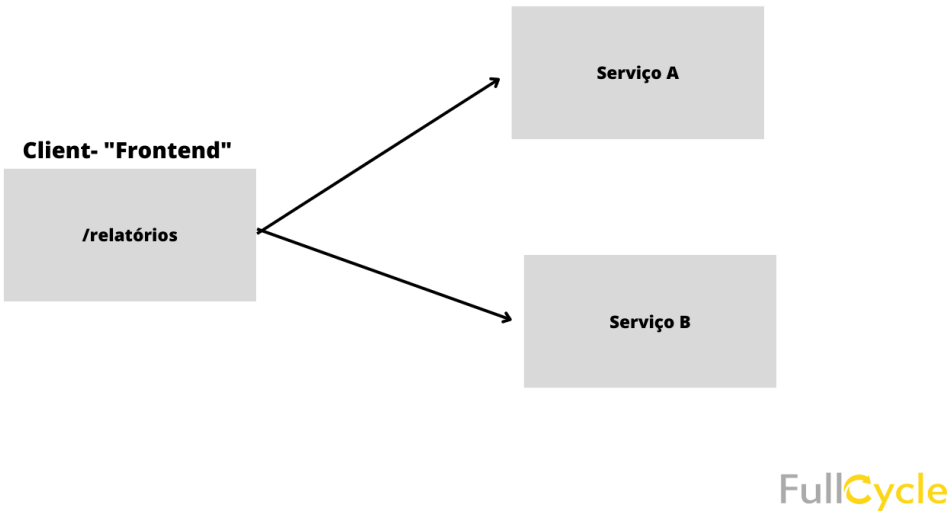
Até o momento estamos falando apenas de dados, ou seja, sem regras de negócio. Nos exemplos que falamos anteriormente, estamos pegando e compondo esses dados, porque precisamos dessas informações. A vantagem que temos com isso é que conseguimos compor os dados de diversos serviços e, além disso, conseguimos retornar um resultado, diminuindo a complexidade do client. Dessa maneira, podemos compor informações de diversas fontes e de diversos serviços. Dificilmente conseguimos trabalhar sem isso em nosso dia a dia. Então, podemos dizer que essa é uma vantagem de trabalharmos com esse padrão de composição de dados.

Por outro lado, temos uma série de desvantagens, por exemplo, a disponibilidade. Pois se um microsserviço cair, os outros microsserviços são prejudicados. Além disso, podemos ter problemas relacionados à consistência nos dados. Porque os microsserviços que fazem chamadas uns para os outros podem não estar exatamente no mesmo passo. Então, pode ser que, ao pegarmos dados de serviços diferentes, a combinação não seja perfeita para o que precisamos naquele momento. Outra desvantagem que podemos ter com o padrão API Composition é um aumento da complexidade. Muitas vezes temos a necessidade de criar um serviço para ler outros serviços, tornando nosso trabalho mais complexo. Fora isso, temos a desvantagem da alta latência que está relacionada à disponibilidade. Isso porque ao chamarmos outros microsserviços precisamos aguardar a combinação desses dados para conseguirmos fazer o retorno para o usuário. Normalmente, isso pode gerar até um problema de dupla ou tripla latência. Além disso, se trabalharmos de forma síncrona essas desvantagens são ainda mais visíveis. Portanto, é importante analisarmos se existe realmente necessidade de usarmos essas soluções. Caso a resposta seja "sim", temos que avaliar todas as possibilidades e suas relações com as vantagens e desvantagens.

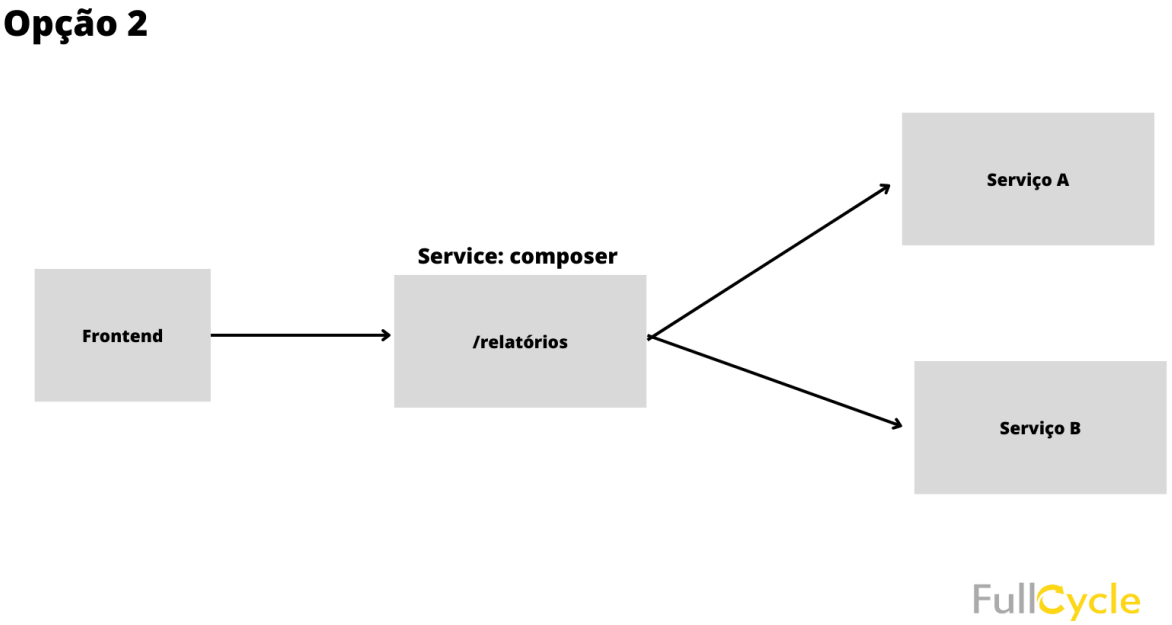
Assim, as opções do padrão de API Composition que vimos neste tópico nos dão uma noção básica de como compor os dados que precisamos sem regras de negócio. Mas é interessante estudarmos também como aplicar esse padrão na parte de serviços, além disso, com regras de negócio, o que pode complicar um pouco mais as situações. Veremos isso no próximo tópico.

Opções de API Composition - DATA “Dados - Sem regras de negócio

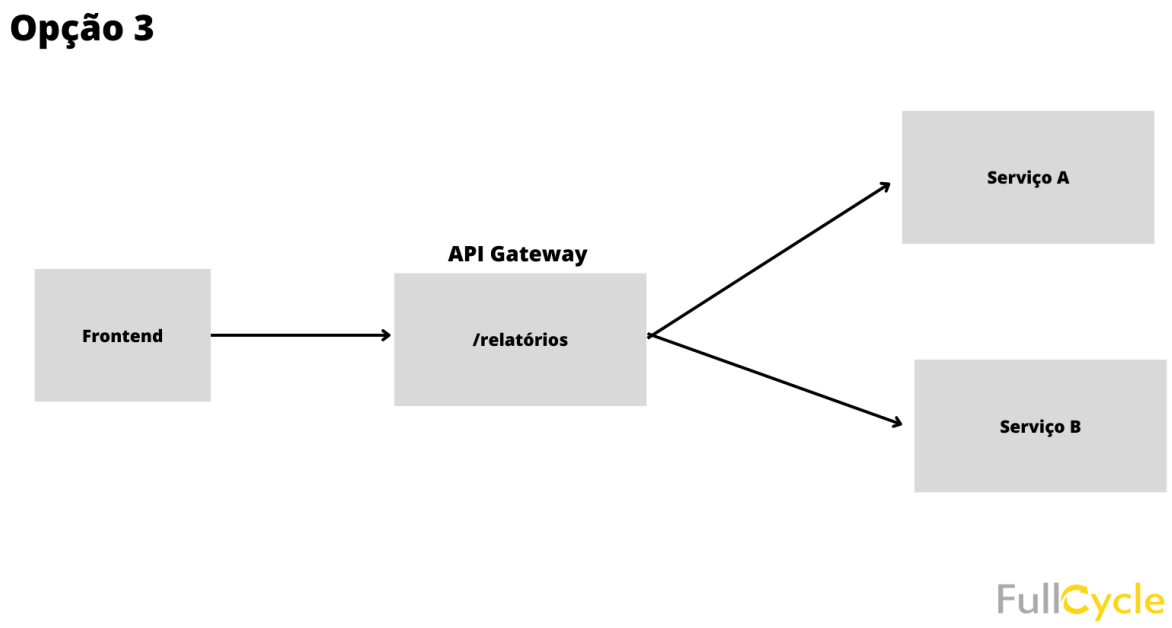
Opção 1:



Opção 2:


Opção 2
Frontend
Service: composer
/relatórios
Serviço A
Serviço B
FullCycle

Opção 3:


Opção 3
Frontend
API Gateway
/relatórios
Serviço A
Serviço B
FullCycle