Microserviços como alternativa de arquitetura monolítica

Ítalo Andrade de Souza[1], William Roberto Pelissari[1]

[1]Faculdade Cidade Verde (FCV) - Maringá, PR - Brasil

italoasouza@gmail.com, wrpelissari@gmail.com

Resumo

Este trabalho consiste em uma pesquisa bibliográfica e tem por objetivo apresentar uma alternativa ao padrão de arquitetura de desenvolvimento de software conhecido como modelo monolítico. Neste modelo toda a aplicação é desenvolvida a partir de uma única fonte de dados, amarrado a uma tecnologia ou *framework* de desenvolvimento, com módulos fortemente acoplados. Em contrapartida, como alternativa o modelo de arquitetura proposto, também conhecido como modelo baseado em microserviços visa a segregação de serviços como componentes da aplicação, com equipes especialistas em diferentes linguagens de desenvolvimento, sem dependência com tecnologia específica.

Palavras chave: microserviços, monolítica, padrão de arquitetura de software

Abstract

This paper consists in a bibliographic search aiming to demonstrate an alternative for the monolithic software pattern. In this pattern all development is based in a single code base, tied to a specific code language or framework, with all its modules strongly coupled. Meanwhile, as alternative for monolithic software pattern, microservices looks for loosely coupled services as components for the applications, with teams specialized in different code languages, with non technological dependency.

keywords: microservices, monolithic, software architecture patterns

Introdução

O termo microserviços surgiu a poucos anos como uma forma de descrever uma maneira específica de como desenvolver aplicações de software. Ainda que não exista uma definição clara e exata deste padrão arquitetural, existem algumas características comuns em torno da capacidade de negócio, distribuição automatizada, inteligência nos pontos de integração e controle descentralizado das linguagens e dados.

Segundo Martin Fowler (2015), o padrão arquitetural baseado em microserviços corresponde ao modo de desenvolver uma única aplicação como um conjunto de pequenas aplicações, cada uma em seu próprio contexto e com processos independentes que se comunicam através de mecanismos simples, tradicionalmente por requisições a uma API utilizando o protocolo HTTP.

Durante o desenvolvimento de uma aplicação, sua base de código cresce conforme novas funcionalidade são implementadas. Com o passar do tempo torna-se cada vez mais trabalhoso determinar onde uma mudança deve ser realizada devido ao grande volume de código. Ainda que em um sistema monolítico o sistema seja dividido em módulos, é muito comum que as fronteiras de negócio sejam separadas de forma arbitrárias, geralmente resultando em códigos com funcionalidades similares espalhados entre os diversos módulos.

A principal motivação padrão arquitetural microserviços está na ideia de ser pequeno, focado e fazer uma única coisa bem-feita, também conhecido como princípio da responsabilidade única (Single Responsibility Principle) apresentada por Robert C. Martin no qual diz que devemos juntar todas as coisas que mudam pelo mesmo motivo e separando daquelas que mudam por motivos diversos.

Arquitetura Monolítica

Para melhor entendermos as características de uma arquitetura baseada em microserviços precisamos antes entender o modelo atual mais utilizado, as arquiteturas monolíticas. Uma aplicação monolítica geralmente é composta por uma única unidade de software compostas essencialmente por três partes: uma interface de interação com o usuário, uma base de dados e uma aplicação servidor para o processamento de requisições, atualização de dados e lógica de domínio. Construída sobre um único executável lógico, de modo que para qualquer alteração seja necessário a construção e liberação de uma nova versão da aplicação.

A construção de uma aplicação monolítica corresponde ao modelo natural de construção de um sistema. Toda a lógica é manipulada através de um processo singular, possibilitando o uso das características básicas da linguagem utilizada para dividir a aplicação em classes, funções e *namespaces*. Com o devido cuidado é possível rodar toda a aplicação em um computador portátil, utilizar integração contínua para gerenciar as etapas do processo de construção da aplicação, executar rotinas de testes automatizados e por fim publicar o resultado final para o servidor de produção. Aplicações monolíticas podem ser escaladas

horizontalmente através da execução de várias instâncias gerenciadas por balanceador de carga (FOWLER, 2015).

Inicialmente a arquitetura monolítica apresenta uma série de vantagens altamente atrativas para o desenvolvimento, como por exemplo: o desenvolvimento simplificado, uma vez que as principais IDEs de desenvolvimento são totalmente focadas no suporte ao desenvolvimento monolítico; simplicidade na instalação que na grande maioria das vezes consiste em simplesmente publicar um pacote ou estrutura de diretórios; e escalonamento intuitivo onde a performance da aplicação é incrementada através da execução de várias cópias da aplicação por trás de um balanceador de carga.

Apesar disso, uma vez que a aplicação assim como a equipe começam a crescer, vários pontos negativos começam a se tornar evidentes e significativos, como por exemplo: o grande volume de código existente por muitas vezes torna-se intimidador, principalmente para os novos integrantes das equipes, deixando o entendimento da aplicação ainda mais difícil; o grande volume de código também ocasiona uma sobrecarga no ambiente de desenvolvimento, deixando a ferramenta lenta e pouco produtiva; na grande maioria das aplicações a sobrecarga do volume do contêiner ou executável da aplicação pode resultar em um tempo maior para sua inicialização, o que significa tempo perdido no desenvolvimento e até mesmo na implantação; aplicações monolíticas apresentam uma barreira natural na implantação de entregas contínuas, uma vez que até mesmo a menor das mudanças obriga que toda a aplicação seja reimplantada, impedindo que a aplicação execute tarefas paralelas nesse período, forçando a aplicação a entrar no famoso estado “Em manutenção”; acima de tudo, aplicações monolíticas geralmente resultam em um casamento com a tecnologia escolhida no início do desenvolvimento, em alguns casos até mesmo com uma versão específica dessa tecnologia, correndo um sério risco de tornar a aplicação obsoleto devido a dependência de módulos e bibliotecas de terceiros (FOWLER, 2015).

Aplicações monolíticas podem naturalmente alcançar grande sucesso, porém cada vez mais nota-se o aumento das frustrações das pessoas envolvidas na produção da aplicação. O ciclo de mudança de funcionalidades torna-se amarrado de tal modo que a realização de uma pequena mudança requer que toda a aplicação seja reconstruída e com o passar do tempo torna-se cada vez mais difícil manter mudanças que afetam um único módulo do sistema. Por conta de sua estrutura, escalar um sistema monolítico requer o escalonamento de toda a aplicação ao invés de partes que realmente requerem mais recursos (RICHARDSON, 2016).

Tais frustrações motivaram a criação do estilo arquitetural baseado em microserviços, desta forma a aplicação é construída como um conjunto de serviços. Uma vez que serviços

podem ser distribuídos de forma independente, além do fato de que serviços possuem fronteiras claras e definidas possibilitando a sua implementação em diferentes linguagens e gerenciadas por diferentes equipes.

Padrão de arquitetura microserviço

Ainda que não exista uma definição formal do padrão arquitetural baseado em microserviços, podemos descrevê-la com base em características comuns identificadas em várias aplicações já distribuídas ao redor do mundo.

Há muito tempo a ideia de desenvolver componentes de software se baseia no mesmo conceito que encontramos no mundo físico, pequenas unidades independentes que podem ser substituídas ou atualizadas.

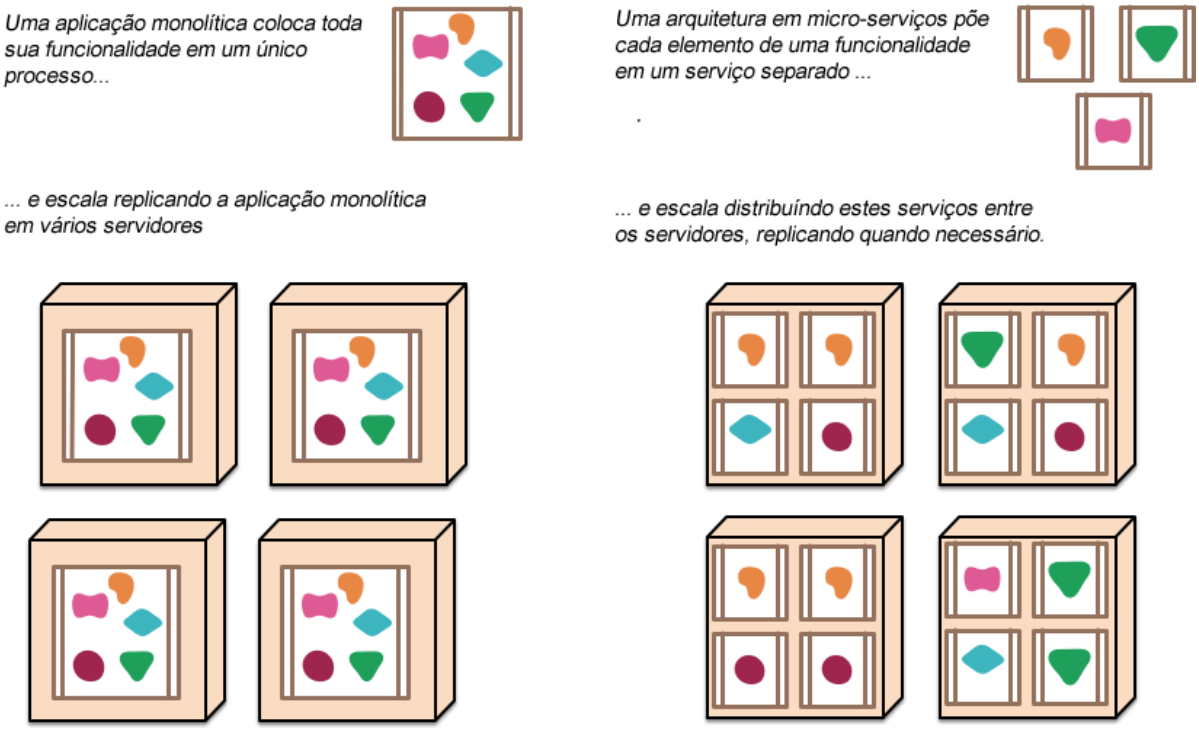
Arquiteturas baseadas em microserviços também utilizam bibliotecas compartilhadas, mas a ideia primária é a de componentização através da segmentação em pequenos serviços. A principal razão para isso é que desta forma os serviços se tornam independentemente distribuíveis. Aplicações que compartilham bibliotecas entre vários componentes necessitam que todos os componentes sejam atualizados ou distribuídos quando ocorrer alguma mudança nesta biblioteca (RICHARDSON, 2016).

Uma das principais razões para utilizar serviços como componentes, ao invés de bibliotecas de funcionalidades, é que serviços podem ser implantados independentemente uns dos outros. Desta forma, através da decomposição da aplicação em múltiplos serviços, podemos esperar que a alteração em qualquer serviço necessita da implantação somente do serviço afetado, o que geralmente significa um tempo de indisponibilidade da aplicação consideravelmente menor.

Outra consequência do uso de serviços como componentes é a existência de uma interface mais explícita. A maioria das linguagens de desenvolvimento não possuem um bom mecanismo de declaração de interfaces públicas (FOWLER, 2015), ficando sempre a critério da documentação e disciplina dos desenvolvedores a garantia de que dependências entre componentes não serão violadas. O uso de chamadas remotas explícitas nos serviços contribui para que isso não ocorra.

Entretanto, não podemos deixar de considerar que o uso de serviços apresenta uma desvantagem ao utilizar chamadas remotas, uma vez que elas são mais custosas para a aplicação do que chamadas de processos. E caso um dia seja necessário realizar a mudança de responsabilidade entre componentes, como por exemplo mover comportamentos entre

componentes, podemos já nos preparar para uma árdua tarefa, uma vez que significa



extrapolar fronteiras entre processos, conforme podemos observar na figura 1.

Figura 1: Arquitetura monolítica e arquitetura microserviços

Fonte: Martin Fowler (2015)

Um dos possíveis questionamentos ao se pensar neste padrão arquitetural é quão grande um microserviço pode ser. Martin Fowler ressalta que nesse caso o termo micro não necessariamente refere-se ao tamanho do serviço ou de sua equipe. Entre os atuais praticantes deste padrão podemos encontrar uma grande variedade de tamanhos de serviços. Até o momento a definição que corresponde a maior equipe de microserviço foi realizada pela Amazon, que atribuiu o termo *"Two Pizza Team"* (Time de duas pizzas), alegando que duas pizzas devem ser o suficiente para alimentar o time. No outro extremo podemos encontrar equipes com meia dúzia de integrantes mantendo meia dúzia de serviços.

Produtos ao invés de projetos

A maior parte das aplicações desenvolvidas focam em um modelo orientado a projetos, onde o alvo principal é o desenvolvimento de uma parte ou funcionalidade da aplicação até que seja considerada como concluída. Quando finalmente concluída, o

aplicativo é direcionado para a equipe de manutenção, ou correção, e a equipe principal do desenvolvimento é desmembrada, geralmente realocada para outros projetos.

A arquitetura baseada em microserviços tende a evitar este modelo, preferindo então a noção de que o time deve ser dono do produto em todo o seu ciclo de vida. Uma fonte como de inspiração para esse modelo é a definição feita pela Amazon *"You build, you run it"* (SOMMERVILLE, 2003), no qual o time de desenvolvimento assume total responsabilidade na produção da aplicação. Desta forma, os desenvolvedores são envolvidos no dia-a-dia da aplicação e seu comportamento no ambiente de produção, assim como contato com os usuários, uma vez que assumem no mínimo parte do suporte a aplicação.

Não há nenhuma razão pela qual uma arquitetura monolítica não poderia praticar este modelo, entretanto, a granularidade modular de arquiteturas em microserviços favorece a criação deste relacionamento entre desenvolvedor, serviço e usuários.

Governança descentralizada

Uma das grandes consequências da governança centralizada é a tendência de padronizar toda a plataforma de software em torno de uma única tecnologia. Já é de conhecimento comum o fato de que nem todo problema se resolve com a mesma solução. Às vezes somos capazes de obter melhores resultados utilizando as ferramentas adequadas de acordo com o contexto da aplicação. Ainda que seja possível obter vantagens do uso de diferentes linguagens de programação em uma arquitetura monolítica, isso não costuma ser comum (FOWLER,2015).

Ao segmentar uma aplicação monolítica em serviços componentizados há a possibilidade de selecionar a linguagem de programação mais adequada para o serviço a ser construído.

Equipes especializadas na construção de microserviços geralmente apresentam uma abordagem diferenciada em relação aos padrões de desenvolvimento de software. Geralmente ao invés de utilizarem um conjunto de padrões escrito em algum documento, eles preferem a ideia de produzir ferramentas que irão auxiliar outros desenvolvedores a solucionarem problemas similares. Estas ferramentas costumam ser resultado de implementações passadas compartilhadas com uma ampla comunidade, muitas vezes apoiadas pela doutrina do código livre.

Um grande exemplo de organização que segue a filosofia do compartilhamento de ferramentas e lições aprendidas é a Netflix, fortalecendo e motivando seguidores a

solucionarem problemas conhecidos de maneira similar, mas sempre deixando a opção de livre escolha de abordagens diferentes. Estas bibliotecas e soluções compartilhadas geralmente estão focadas em problemas comum de armazenamento de dados, comunicação entre processos e automação de infraestrutura (NEWMAN, 2015).

Podemos considerar que o auge da governança descentralizada reside na ideologia *"You build, you run it"*, com times responsáveis por todos os aspetos da construção da aplicação incluindo a operação 24/7 em produção. Ainda que este nível de envolvimento ainda seja raro, cada vez mais empresas estão adotando este modelo repassando parte das responsabilidades as equipes de desenvolvimento.

Gerenciamento de dados descentralizado

A descentralização do gerenciamento de dados está presente na criação de softwares de diversas maneiras, de um modo geral, isto significa que o modelo conceitual será diferente entre os sistemas. Este é um problema muito comum quando integramos aplicações empresariais em larga escala, por exemplo, as informações visualizadas em um pedido de venda serão diferentes das informações visualizadas pelo suporte, possuindo diferentes atributos, e no pior dos casos, atributos comuns com diferentes significados (FOWLER, 2015).

Este problema é relativamente comum ao integrar diferentes aplicações, mas em alguns casos pode ocorrer na própria aplicação, em especial quando a aplicação é dividida em diferentes componentes. Podemos ter este cenário considerando como exemplo o uso do *Domain-Driven Design* (Projeto Orientado a Domínio). O DDD divide um domínio complexo em múltiplos contextos delimitados com seus relacionamentos devidamente mapeados. Este processo é muito útil tanto em arquiteturas monolíticas quanto arquiteturas em microserviços, porém a correlação entre serviços e contextos delimitados torna o uso de microserviços ainda mais fácil de se compreender, reforçando a ideia de separação entre componentes. (NEWMAN, 2015)

Da mesma forma que ocorre a descentralização de decisões dos modelos conceituais, microserviços também descentralizam as decisões relacionadas ao armazenamento de dados. Enquanto que aplicações monolíticas preferem o uso de uma única unidade lógica para a persistência de dados, algumas soluções corporativas chegam a utilizar um único banco de dados para várias aplicações. A arquitetura em microserviços prefere deixar que cada serviço defina e controle seu banco de dados, seja ele diferentes instâncias de uma mesma solução ou

soluções completamente diferentes, uma abordagem conhecida como *Polyglot Persistence* (Persistência Poliglota) (FOWLER, 2015). Ainda que seja possível utilizar a persistência poliglota em arquiteturas monolíticas, sua presença é mais frequente em arquiteturas em microserviços, conforme podemos observar na Figura 2.

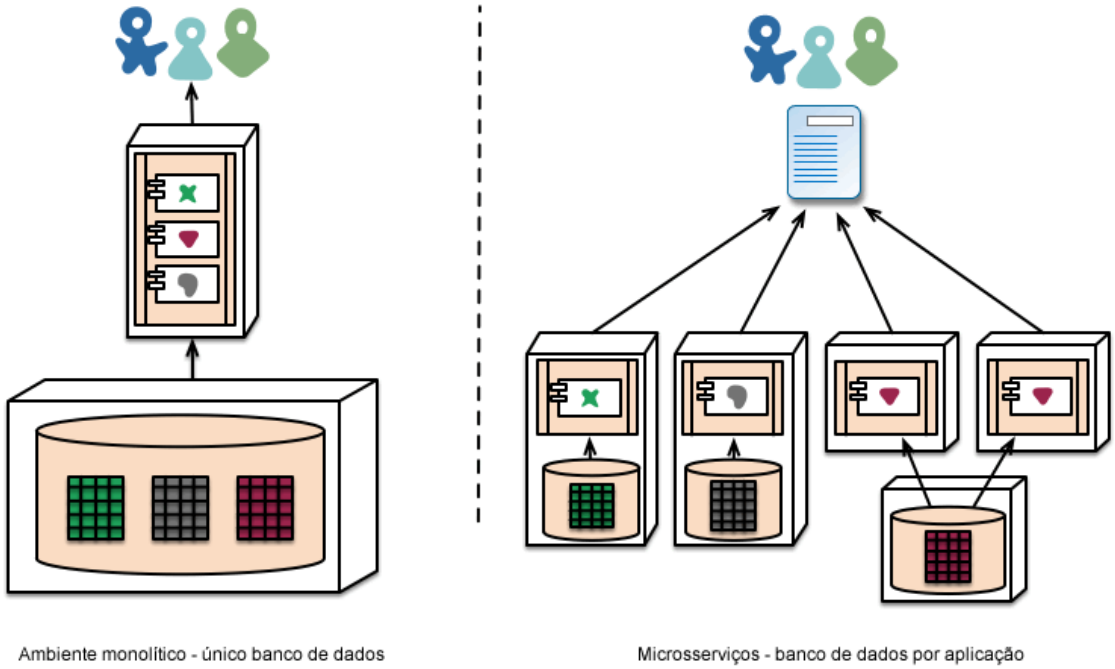


Figura 2: Distribuição de dados entre serviços

Fonte: Martin Fowler (2015)

A descentralização da responsabilidade dos dados entre os vários serviços possui algumas implicações quando falamos do controle de atualização de dados. A abordagem tradicional utilizada em arquiteturas monolíticas para lidar com esta situação é a utilização de transações para assegurar a consistência dos dados quando manipulando múltiplos recursos.

Usar transações ajuda a solucionar o problema de consistência, mas isso também significa a criação de uma acoplamento temporal, o que pode ser um grande problema quando envolvemos múltiplos serviços distribuídos em vários dispositivos. Transações distribuídas são conhecidamente difíceis de se implementar e como consequência arquiteturas em microserviços enfatizam o uso de comunicação sem controle de transação entre os serviços, com o reconhecimento explícito de que a consistência será eventual e que problemas serão compensados por operações feitas pela aplicação (SOMMERVILLE, 2003).

A escolha por gerenciar inconsistências desta forma apresenta novos desafios para a grande maioria das equipes de desenvolvimento, mas é a que está presente na maioria dos modelos praticados. Geralmente as aplicações são projetadas para suportar um certo volume de inconsistência em troca de uma rápida resposta quando sob alta demanda, enquanto algum tipo de processamento paralelo trata os erros ocorridos. Esta troca é válida enquanto o custo de correção destes erros for menor que o custo da operação sob alta demanda. (NEWMAN, 2015)

Automação de infraestrutura

Técnicas de automação de infraestrutura evoluíram consideravelmente nos últimos anos, em especial a evolução de soluções na nuvem que reduziram a complexidade operacional de construção, implantação e operação de microserviços.

Muitos dos produtos ou sistemas sendo construídos com arquitetura em microserviços estão sendo construídos por times com ampla experiência em Entrega Contínua (CD) e seu precursor, Integração Contínua (CI). Times que constroem aplicações desta maneira fazem extenso uso de técnicas de automação de infraestrutura.

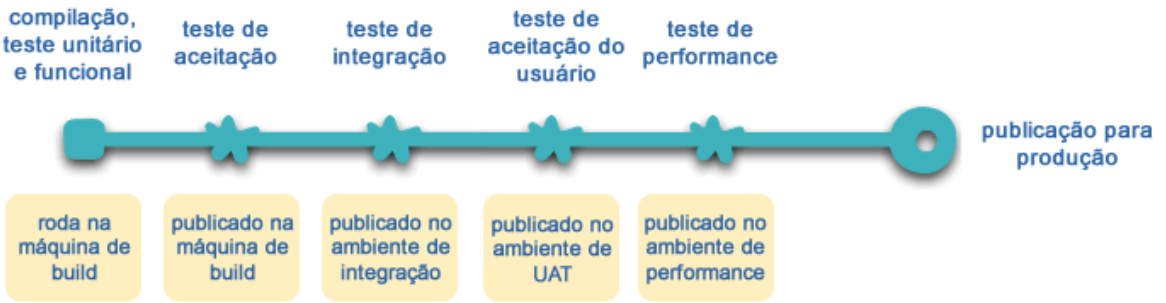


Figura 3: *Pipeline* básico de construção da aplicação.

Fonte: Martin Fowler (2015)

Conforme podemos observar na Figura 3, uma das características e ponto focal da Entrega Contínua é a execução de várias suítes de testes automatizados. O objeto é ter o máximo de confidência possível de que a aplicação está funcionando, ao avançar as etapas no fluxo de construção estamos realizando a implantação automatizada para cada novo ambiente de teste.

Uma aplicação monolítica será construída, testada e passará pelos ambientes tranquilamente. De fato, torna-se perceptível que após o investimento na automação das etapas de publicação para o ambiente de produção, realizar a publicação novamente deixa de ser algo assustador. Na verdade, o principal objetivo da Entrega Contínuo é tornar a publicação uma tarefa entediante (FOWLER, 2015).

Outra área onde é possível notar o extenso uso da automação de infraestrutura é no gerenciamento de microserviços em produção. Em contraste com a definição feita anteriormente, de que não haverá muita diferença entre o processo de publicação de aplicações monolíticas e microserviços, o cenário operacional para cada uma é bem diferente, conforme podemos observar na Figura 4.

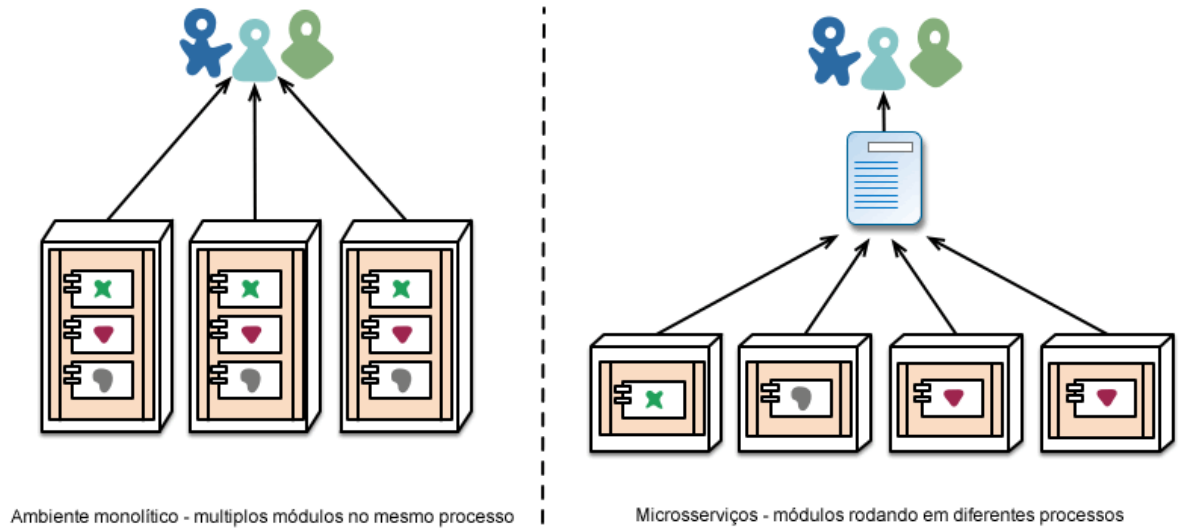


Figura 4: Diferenças oferecidas pela publicação modular.

Fonte: Martin Fowler (2105)

Projetada para falhar

Uma das consequências do uso de serviços como componentes, é que as aplicações devem ser projetadas de tal modo que sejam capazes de tolerar a falha de serviços. Uma chamada a um serviço pode falhar a qualquer momento devido a indisponibilidade do fornecedor do serviço, a aplicação cliente deve sempre responder da forma mais graciosa possível. Isto acaba sendo uma desvantagem em relação a arquiteturas monolíticas, já que isso representa uma complexidade adicional a ser considerada na criação do serviço. A consequência disso é que equipes de desenvolvimento de microserviços estão constantemente refletindo sobre como falhas no serviço podem afetar a experiência do usuário, até mesmo

introduzindo falhas planejadas nos serviços ou datacenters durante sua operação para testar sua capacidade de resiliência e monitoramento (NEWMAN, 2015).

Este tipo de teste automatizado em produção é mais do que suficiente para deixar qualquer equipe de apoio com um frio na espinha que precede uma semana de trabalho. Não que isso signifique que arquiteturas monolíticas não sejam capazes de realizar tais configurações de monitoramento, essa prática é apenas menos comum.

Uma vez que um serviço pode falhar a qualquer momento, é importante ser capaz de identificar a falhar com o mais rápido possível, e se possível, restaurar o serviço automaticamente. Arquiteturas em microserviços colocam grande ênfase no monitoramento em tempo real da aplicação, conferindo tanto elementos arquiteturais, como a quantidade de requisições por segundo sendo realizadas a um banco de dados, quanto métricas de negócio, como a quantidade de pedidos feitos por minuto. O monitoramento semântico pode fornecer informações para um aviso antecipado de uma falha iminente, possibilitando os times a investigarem a situação antes do problema ocorrer.

Isto é particularmente importante nas arquiteturas em microserviços devido a sua preferência pela comunicação coreografada e eventos colaborativos, quando múltiplos componentes trabalham juntos comunicando entre sí através de sinais emitidos ao ocorrer uma mudança de estado.

Aplicações monolíticas também podem ser construídas com a mesma transparência que microserviços, de fato, elas realmente deveriam ser construídas assim. A diferença é que ao construir um serviço se faz necessário saber quando um serviço em execução em diferentes processos são desconectados. Quando utilizamos bibliotecas contidas em um único processo essa transparência torna-se menos útil (FOWLER, 2015).

Equipes operando microserviços tradicionalmente possuem sofisticadas ferramentas de monitoramento e auditoria configuradas para cada serviços, agrupadas em painéis contendo informações sobre a situação, uma variedade de métricas operacionais e de negócio como consumo de banda, latência, usuários conectados, entre outros.

Modelo Evolucionário

Praticantes do desenvolvimento de microserviços normalmente possuem em seu histórico uma visão de desenvolvimento seguindo o modelo evolucionário, e veem na decomposição de serviços uma ferramenta que permite aos desenvolvedores controlar as mudanças na aplicação sem reduzir as mudanças. O controle de mudança não

necessariamente significa reduzir a quantidade de mudanças no decorrer do desenvolvimento, mas sim a atitude e uso correto de ferramentas que possibilitam realizar mudanças frequentemente, de modo rápido e bem controlado, impactando o mínimo possível.

Sempre que tentamos separar um sistema em componentes, nos encontramos com a árdua tarefa de decidir como separar e delimitar as partes, normalmente nos baseamos na noção de substituição independente e atualizável. Neste caso devemos procurar por pontos nos quais permitam que o componente seja reescrito sem afetar os demais componentes relacionados. De fato, muitas equipes de desenvolvimento de microserviços vão mais adiante ao considerar explicitamente que muitos serviços devem ser mais como rascunhos do que possuir um envolvimento a longo prazo.

O portal de conteúdo *The Guardian* é um bom exemplo de aplicação inicialmente construída em uma arquitetura monolítica e que posteriormente passou a seguir os caminhos do microserviço. Ainda que o núcleo da aplicação ainda esteja baseada na arquitetura monolítica, toda nova funcionalidade é projetada como serviço que consome dados disponibilizados por uma API contida no núcleo monolítico. Desta forma torna-se fácil a criação de funcionalidades ou conteúdos primeiramente temporários, como por exemplo páginas de eventos esportivos, já que estas páginas podem ser criadas rapidamente através de ferramentas de prototipação e posteriormente removidas após o término do evento.

Esta ênfase na substituibilidade é um caso especial de um princípio um pouco mais amplo da modelagem modular, que é dirigida a modularidade através do Padrão orientado a Mudanças, conforme descrito em *Implementation Patterns* (Padrões de Implementação). Isso significa que devemos manter as coisas que mudam ao mesmo tempo sempre no mesmo módulo. Partes do sistema que raramente sofrem alterações devem estar em serviços diferentes daqueles que geralmente estão sobre grande rotatividade. Se por acaso identificarmos que continuamente estamos sempre alternando dois serviços por conta de uma mesma mudança, talvez seja uma boa ideia unificá-los em apenas um serviço (FOWLER, 2015).

Ao colocarmos componentes em serviços estamos criando a oportunidade de um planejamento de liberação de funcionalidades um pouco mais granular. Em uma arquitetura monolítica, qualquer mudança requer que toda a aplicação seja reconstruída e então sua atualização por completo em produção. Com microserviços será necessário realizar este processo apenas no serviço alterado. Em contrapartida, isso também significa que ao realizar mudanças em um serviços devemos nos atentar se isso não irá interromper a comunicação com outros serviços. Neste caso, a abordagem tradicional é a realização do versionamento da

API de integração, porém quando falamos de microserviços, devemos considerar esta técnica como última alternativa e priorizar a criação de serviços tolerantes o máximo possível a mudanças nos serviços consumidos (NEWMAN, 2015).

Considerações finais

Este trabalho teve como objetivo apresentar as ideias e princípios em torno da criação de microserviços. Pudemos notar que este padrão arquitetural é uma importante ideia para a engenharia de software, com grandes vantagens para o desenvolvimento de soluções empresariais.

Até o presente momento temos o conhecimento que empresas como *Amazon, Netflix, The Guardian, UK Governament Digital Services, Realstate.com.au* e *comparethemarket.com* estão entre as pioneiras no uso de microserviços. Segundo Martin Fowler, que esteve presente no circuito de conferências de engenharia de software em 2013, foi possível notar um grande volume de empresas com movimentos de migração ou criação de aplicações muito próximas do que podemos classificar como microserviços. Incluindo a própria companhia que fornece ferramentas para integração contínua na nuvem Travis CI. Inclusive, há uma série de empresas que já desenvolvem aplicação como microserviços mas que até então não haviam atribuído este nome, normalmente nesses casos essas aplicações são generalizadas com soluções SOA, o que possui alguma contradições com o princípio de microserviços (FOWLER, 2015).

Apesar de todos os pontos e experiências positivas apresentados, seria imprudente afirmar que microserviços é a direção definitiva para arquiteturas de softwares. Ainda que tenhamos essas experiências positivas, simplesmente é muito cedo para ter um julgamento completo.

Normalmente, as verdadeiras consequências de decisões arquiteturais se tornam evidentes somente alguns anos após elas terem sido tomadas. Até o momento foi possível observar boas equipes, com forte afinidade por modularidade, construindo aplicações monolíticas que entraram em decadência com o passar dos anos. Muitos acreditam que essa decadência é menos provável de ocorrer ao utilizar microserviços, uma vez que as fronteiras entre os módulos são explícitas e difíceis de serem burlada. Ainda assim, até termos

presenciado uma boa quantia de aplicações com anos de história, não poderemos concluir o modo no qual arquiteturas em microserviços amadurecem.

Por fim, é importante notar que tudo isso depende do nível de habilidade da equipe. Novas técnicas geralmente são adotadas por equipes mais habilidosas, e pode não apresentar o mesmo resultado ou eficiência quando praticado por equipes menos habilidosas, que em muitos casos constroem aplicações monolíticas desorganizadas. Por este motivo não há garantia de que os serviços criados sejam diferentes. Equipes ruins sempre criam sistemas ruins, é pouco provável que o uso de microserviços irá reduzir o volume de desordem e em alguns casos pode tornar as coisas ainda piores.

Um dos argumentos mais comuns é que nestes casos não se deve iniciar o desenvolvimento da aplicação como microserviço, e sim como uma aplicação monolítica, porém com o devido cuidado de mantê-la modular, posteriormente dividindo-a em microserviços. Ainda assim devemos manter um alerta constante pois este formato está longe de ser o ideal, pois não devemos esquecer que uma boa interface de processo nem sempre corresponde a uma boa interface de serviço.

Concluímos este trabalho com um otimismo cauteloso em relação ao uso de microserviços. Este padrão arquitetural apresenta algumas vantagens consideráveis em relação ao modelo monolítico, uma vez que ele é projetado para que cada serviço possa ser substituído causando o menor impacto possível, possui uma delimitação de fronteira entre os domínios da aplicação bem mais claros, e confiando nas experiências compartilhadas por empresas como *Netflix*, *Spotify*, *Amazon* pudemos acreditar que este seja um caminho válido a ser seguido, só não podemos dizer com toda a certeza onde ele irá nos levar, entretanto, um dos desafios do desenvolvimento de software é que geralmente temos que tomar decisões baseadas nas informações quase sempre imperfeitas que temos em mãos.

Referências

FOWLER, Martin. Microservices and the first law of distributed objects.: Thoughworks, 2015. Disponível em: <https://www.thoughtworks.com/insights/microservices>. Acesso em: 22 set. 2016.

MARTIN, Robert C. Clean Code: A Handbook of Agile Software Craftsmanship. 1st Edition, 2008

NEWMAN, Sam. Building Microservices: Designing Fine-grained Systems.: O'reilly Media, 2015.

PRESSMAN, R. Engenharia de Software: Uma abordagem Profissional. 7º edição. Editora Bookman.

RICHARDSON, Chris. Smith, Floyd. Microservices, From Design to Deployment. NGINX 2016

SOMMERVILLE, Ian. Engenharia de software. Addison Wesley, 2003