Uso de AOP na Migração de Aplicações Monolíticas para *Microservices*

Otávio Medeiros[1], **Américo Sampaio**[1], **Augusto Arraes**[1]

[1]Programa de Pós-Graduação em Informática Aplicada (PPGIA)
Universidade de Fortaleza (UNIFOR)
Av. Washington Soares, 1321, Edson Queiroz, CEP 60811-905 Fortaleza, CE

otaviomd@hotmail.com, americo.sampaio@unifor.br, arraes.augusto@gmail.com

Abstract. *The recent Microservices paradigm is transforming the way to develop applications as a set of small distributed programs that can be deployed and scaled in a completely independent and automated fashion on cloud environments. One challenge that is actually quite common for various organizations that are adopting this paradigm is to migrate their current applications, that might already be in production, from a monolithic structure (containing one single large program) to a Microservices architecture. This paper proposes an approach based on the utilization of Aspect Oriented Programming (AOP) to migrate the monolithic modules using Aspects in a gradual and non-invasive manner. The aspects intercept the calls to the monolithic modules and redirect them to Microservices without the need to change de monolithic code and enabling the roll back to previous versions without the need to rebuild the system.*

Resumo. *O recente paradigma de Microservices vem transformando a maneira de desenvolver aplicações através de um conjunto de pequenos programas distribuídos que podem ser implantados e escalados de maneira completamente independente e automatizada em ambientes de nuvem. Um desafio que atualmente é bastante comum para várias organizações que estão adotando este paradigma é o de migrar suas aplicações existentes, que podem se encontrar em produção, de uma estrutura monolítica (contendo um único programa grande) para a arquitetura de Microservices. Este artigo propõe uma abordagem baseada na utilização de Programação Orientada a Aspectos (AOP) para migrar os módulos do monolítico usando aspectos de uma maneira gradual e não invasiva. Os aspectos conseguem interceptar as chamada as código do monolítico e redirecionar para chamadas a Microservices sem a necessidade de ter que realizar alterações no código do monolítico e possibilitando a volta de versões anteriores sem necessidade de recompilar o sistema.*

1. Introdução

O recente paradigma de *microservices* [Newman 2015] [Richardson and Floyd 2016] [Balalaie et al. 2015a] vem ganhando bastante atenção do mercado e da comunidade acadêmica e transformando a maneira de desenvolver software. Com o objetivo de aumentar a produtividade das equipes de desenvolvimento, este paradigma transforma a maneira de modularizar o software através da distribuição de um conjunto de pequenos programas independentes, chamados *microservices*, os quais eram localizados dentro de

uma única unidade de implantação, chamada de sistema monolítico. Isto muda a maneira de construir software, pois os antigos módulos do monolítico podem ser construídos, implantados, monitorados e escalados, de forma independente, trazendo mais produtividade e desacoplamento, mas também trazendo uma série de desafios, como a gestão mais automatizada da implantação, a escalabilidade independente e a eventual utilização de várias bases de dados. Algumas implementações que são referências baseadas em *microservices*, já estão em pleno funcionamento no mercado, como os sistemas da Amazon e da Netflix, demonstrando os benefícios do paradigma.

Hoje, considera-se uma aplicação monolítica como uma grande aplicação desenvolvida e implantada em uma empresa ao longo de seu processo de informatização. Como uma aplicação de âmbito geral da organização, ela integra os vários departamentos e permite a interação de diversas funcionalidades entre setores da empresa. Desta forma, acaba por englobar os processos e regras de negócios que são automatizados em suas funcionalidades. Essas aplicações normalmente se organizam em uma arquitetura em camadas, onde cada camada procura focar em um conjunto coeso de funcionalidades, por exemplo: interface gráfica (GUI), controle, negócio e acesso aos dados, normalmente baseadas nos padrões *Layers* e *Model-View-Controller - MVC* [Buschmann et al. 1996] [Fowler and Lewis 2014]. Diversas tecnologias surgiram, nos últimos anos, para dar suporte à construção de aplicações baseadas nesses padrões e obtiveram sucesso, principalmente devido ao fato desses padrões darem suporte a uma boa modularização do código, permitindo conseguir reuso e, relativamente, um bom desacoplamento entre os módulos.

No entanto, uma limitação relevante das aplicações monolíticas se deve ao fato de todos os módulos pertencerem a um único e mesmo programa. Isso faz com que à medida que novas funcionalidades vão sendo incorporadas ao software, esse acaba tornando-se muito grande com o passar do tempo, o que impacta nas suas atividades de construção (*build*) e implantação (*deploy*). Sistemas muito grandes tornam-se complexos do ponto de vista operacional, pois seu processo de compilação se torna lento e sua implantação custosa. Além disso, uma aplicação monolítica, normalmente, só pode ser escalada por completo, usando réplicas por trás de um balanceador de carga, mesmo que seus gargalos estejam limitados apenas a uma parte de suas funcionalidades.

Com o intuito de resolver essas limitações foi desenvolvido o paradigma de *microservices*. A ideia é que os módulos do monolítico sejam refatorados em pequenas unidades que executam em programas independentes da maneira mais desacoplada possível, permitindo que seus processos de construção (*build*), implantação (*deploy*) e escalabilidade (*scaling*) sejam independentes. Assim, é possível lançar versões de *microservices* mais rapidamente ou mais lentamente, dependendo do tipo de funcionalidade que o *microservice* endereçar. Além disso, há possibilidade de escalar cada *microservice* individualmente, de forma a atender sua necessidade específica.

Apesar desses benefícios, migrar a arquitetura de uma aplicação monolítica para uma baseada em *microservices* traz uma série de desafios [Balalaie et al. 2016], especialmente se essa aplicação estiver em ambiente de produção, sendo utilizada por clientes. Para que esse problema seja resolvido de maneira adequada várias questões precisam ser abordadas:

- QP1: Qual estratégia de refatoração a ser utilizada? migrar gradualmente o sistema ou refatorar por completo?

- QP2: Como modularizar e reimplementar os módulos do monolítico em *microservices*?
- QP3: Como lidar com o banco de dados do monolítico? manter o banco do monolítico ou refatorar em bancos separados para os *microservices*?

Percebe-se então que começa a se desenhar um problema complexo para ser resolvido pelos atuais interessados em migrar sua arquitetura para *microservices*. Com relação à *QP1* vários trabalhos existentes na literatura [Balalaie et al. 2016] [Yanaga 2017] [Fowler and Lewis 2014] [Newman 2015] apontam no sentido de que é mais prudente se adotar uma estratégia gradual na migração do código monolítico para a arquitetura *microservices* ao invés de uma completa rescrita da aplicação (*big bang rewrite*). Isto permite, principalmente, que aplicações já em produção possam ser refatoradas aos poucos, mantendo o monolítico em produção juntamente com os *microservices*, minimizando a possibilidade de falhas no sistema e de indisponibilidade dos serviços prestados aos clientes.

Durante a realização desta migração *(QP2)*, os módulos do monolítico a serem refatorados precisam ser identificados [Levcovitz et al. 2015] [Richardson 2016b] e sua lógica de implementação refatorada para o *microservice* correspondente que poderá ser inicialmente utilizado em paralelo com o código do monolítico. Em [Richardson 2016b] sugere-se que a responsabilidade a ser migrada seja retirada do módulo do monolítico e reimplementada no *microservice*. Além disso, a abordagem sugere que seja introduzido um componente redirecionador externo que irá redirecionar as chamadas feitas àquele módulo anterior, que estava no monolítico, para o *microservice*. Conforme mencionado em [Yanaga 2017] esta solução é inspirada no padrão *Strangler* [Fowler 2004]. A idéia é que o desenvolvedor possa dividir um aplicativo em diferentes domínios funcionais e substituir esses domínios por um *microservice*. Isso cria dois aplicativos separados que vivem lado a lado e ao longo do tempo, o aplicativo recentemente refatorado "estrangula"ou substitui o aplicativo original até que finalmente o desenvolvedor possa desligar o monolítico por completo.

Um dos maiores desafios da migração para *microservices* encontra-se justamente na solução para a refatoração do banco de dados (*QP3*) conforme [Yanaga 2017] [Balalaie et al. 2015a] [Richardson 2016b]. Há particularmente duas estratégias que podem ser seguidas para solucionar esta questão. A primeira delas, mais simplificada, *"recomenda que você comece sua jornada de microservices de maneira cautelosa. Primeiro, faça funcionar com o seu banco de dados relacional existente. Depois de concluir com sucesso implementar e integrar seu primeiro microservice, você pode decidir se você (ou) seu projeto será melhor servido por outro tipo de tecnologia de banco de dados. [Yanaga 2017] "*. Esta estratégia é também sugerida em [Richardson 2016c] para que os diferentes *microservices* compartilhem o mesmo banco de dados. A segunda estratégia, bem mais complexa, conhecida como *Polyglot Persistence* sugere que cada *microservice* utilize seu próprio banco de dados e que a tecnologia escolhida para o banco pode variar de forma a atender o propósito do serviço da melhor maneira possível [Fowler and Lewis 2014] [Yanaga 2017] [Richardson 2016a].

Este trabalho propõe uma estratégia de migração de aplicações monolíticas para *microservices* baseada no uso da Programação Orientada a Aspectos (AOP) [Kiczales et al. 1997] para permitir uma migração gradual (*QP1*) do monolítico de forma

a preservar totalmente o código do monolítico sem ter que alterá-lo através da introdução de *apects* com mecanismos de *advice around* que permitem sobrescrever as chamadas ao código do monolítico e desviá-las para a execução do código dos *microservices* (*QP2*). Esta abordagem é totalmente nova para a solução deste problema e vantajosa em relação às atuais pois não exige a modificação do código do monolítico e ainda permite, caso assim deseje, que a versão anterior seja restabelecida sem qualquer alteração no código do sistema. Outra vantagem é que o mecanismo de composição dinâmica dos *aspects* pode permitir que esta refatoração ocorra em tempo de execução sem a necessidade de gerar novos *builds* do sistema. Com relação à (*QP3*) adotamos inicialmente a solução mais simplificada de preservar o banco de dados único do monolítico que será depois revista em trabalhos futuros.

O restante deste trabalho é organizado da seguinte maneira. A seção 2 define a fundamentação teórica relevante para a compreensão deste trabalho. A seção 3 explica a abordagem de migração proposta enquanto a seção 4 a avalia comparando a uma abordagem existente. Na seção 5 são descritos os trabalhos relacionados. Finalmente, a seção 6 apresenta a conclusão e os trabalhos futuros.

2. Fundamentação Teórica

Apresentam-se nesta seção alguns elementos de Programação Orientada a Aspectos (*Aspect-Oriented Programming - AOP*) e *microservices*.

2.1. Programação Orientada a Aspectos (Aspect-Oriented Programming - AOP)

A filosofia de **dividir para conquistar** é bastante pertinente quando se fala em desenvolvimento de sistemas. No contexto da AOP fala-se em **separação de interesses** ou ***cross-cutting concerns***, como tratado em [Kiczales et al. 1997]. Um dos focos básico da AOP é remover do domínio da aplicação esses **interesses transversais**, por meio de estruturas chamadas de *aspects*.

A utilização de *aspects* acarreta inversão de controle (*Inversion of Control - IoC*), de forma tal que os *aspects* conhecem os componentes do sistema, mas o contrário não é necessário. Conforme menciona [Kiczales et al. 1997], esta propriedade também é chamada de *obliviousness*. Para compor o sistema é necessário um processo chamado combinação (*weaving* - ver **Figura 1**), o qual pode ocorrer de forma estática, aplicado em tempo de *build* do sistema ou dinâmico.

De acordo com [Kiczales et al. 1997], os principais elementos da AOP são:

- *Aspects* - unidade de modularização criada para capturar uma funcionalidade que ”corta”(*crosscuts*) vários módulos da aplicação;
- *Joinpoints* - é um ponto definido na estrutura ou execução do programa (ex: chamadas ou execução de métodos) onde um ou mais *aspects* realizarão interações com a aplicação;
- *Pointcuts* - é a especificação de um conjunto de *joinpoints* onde os *aspects* realizarão interações com a aplicação. Utilizam-se expressões regulares na definição de *pointcuts*;
- *Advice* - é o comportamento adicionado por um *aspect* que é aplicado nos *joinpoints* para realizar uma operação considerada *crosscutting*. Os *advices* podem seguir três comportamentos distintos, quais sejam:

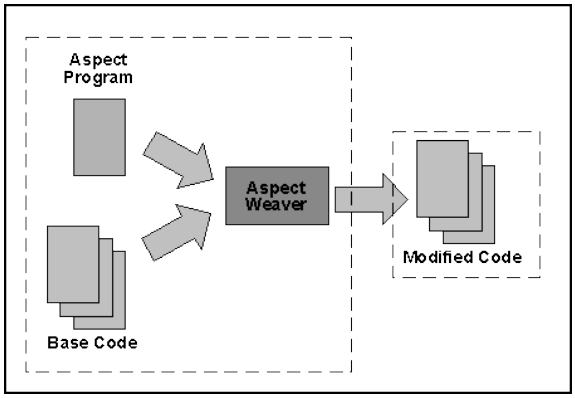

Figura 1. Processo de Weaving - Fonte: [XWeaver 2015]

- *Before* - este comportamento é realizado antes do *joinpoint* a ser executado;
- *After* - este comportamento é realizado após o *joinpoint* ser executado;
- *Around* - este comportamento é realizado ”em torno”do *joinpoint*. Pode subistituir completamente o comportamento associado ao *joinpoint* e é esse tipo de *advice* a ser utilizado ostensivamente neste trabalho.

A **Figura 2** fornece uma ilustração desses conceitos.

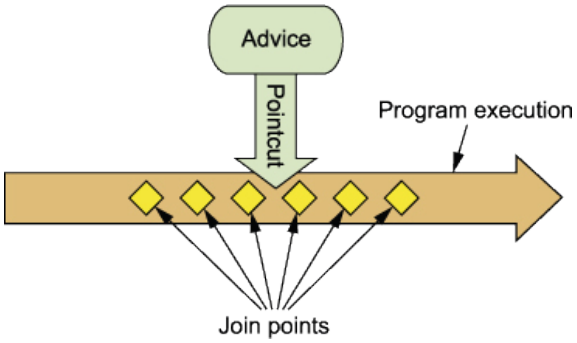

Figura 2. Conceitos de AOP - Fonte: [Sean 2015]

2.2. Monolítico e *Microservices*

Uma aplicação é considerada monolítica, conforme [Carnell 2017], quando é entregue por meio de um único artefato de *software* no qual estão embutidos, em camadas, os códigos da interface do usuário (UI), das regras de negócio e da lógica de acesso ao banco de dados, podendo ser este artefato único implantado (*deployed*) em um servidor de aplicações.

Embora apresente apenas uma unidade de implantação, normalmente são necessários vários grupos de desenvolvedores para implementar e dar manutenção ao mesmo, cada grupo gerenciando uma parte específica de funcionalidades.

Este tipo de arquitetura leva, normalmente, a alguns problemas:

- Os times de desenvolvedores não crescem na mesma proporção que o tamanho e a complexidade das aplicações;

- Apresenta ponto único de falha, que pode ser entendido como a situação na qual se um módulo falhar, a aplicação como um todo tenderá a falhar;
- Sempre que um dos times realiza alguma mudança na aplicação, esta, como um todo, precisa ser reconstruída (*rebuilding*), retestada (*retesting*) e reimplantada (*redeploy*).

A arquitetura *microservice* surgiu para atacar estes e outros problemas. Por meio dela procura-se separar cada funcionalidade em módulos desacoplados a nível tal que possam ser implantados separadamente, um do outro, no servidor de aplicação. Além disso:

- Cada componente é responsável por um pequeno domínio de responsabilidade;
- Pequenos times de desenvolvedores são responsáveis por componentes ou grupos de componentes específicos, facilitando o domínio sobre cada um deles;
- Por serem independentes, cada módulo pode utilizar-se da tecnologia que lhe for mais adequada.

3. Abordagem Proposta

O que é sugerido neste artigo é a utilização de Programação Orientada a Aspectos (*Aspect-Oriented Progrmming - AOP*) de forma tal que as chamadas aos serviços fornecidos por uma aplicação desenvolvida no modelo arquitetural monolítico sejam capturadas e direcionadas para chamadas a serviços equivalentes fornecidos por um conjunto de aplicações desenvolvidas no modelo arquitetural *microservices*, utilizando-se o estilo arquitetural REST (*Representational State Transfer*). Para isso faz-se uso de *advices* do tipo *around*. Esses *advices* não precisam ter necessariamente as mesmas assinaturas que os serviços chamados pelos processos da aplicação monolítica, serviços estes que serão sobrescritos por versões *microservices*, mas precisam ter os mesmos tipos de retorno esperado pelos processos chamadores dos serviços monolíticos, pois, para todos os efeitos, as chamadas estarão sendo realizadas aos serviços originais (monolíticos).

A utilização de *advices* do tipo *around* é um dos pontos chave da abordagem, pois a possibilidade de poder substituir completamente o código original (monolítico) referenciado pelo *joinpoint* por um novo código (*microservice*), dado pelo *advice*, permite que a migração possa ocorrer sem que seja necessário alterar o código fonte original, ou seja, as chamadas a serviços monolíticos selecionados para serem migrados para *microservices* serão capturadas e redirecionadas para os *aspects* / *advices* adequados por meio do processo de *weaver* da AOP. Sendo assim, nenhum código da aplicação monolítica precisa ser alterado.

Desconhece-se solução única que resolva de forma eficaz problemas associados à má qualidade de escrita de código, com exceção de refatoração. Não tem este trabalho, portanto, a intensão de fazê-lo.

Para que tal estratégia tenha sucesso, necessário faz-se que a aplicação monolítica tenha sido desenvolvida respeitando e utilizando elementos da boa prática de programação orientada a objetos. O uso de *design patterns*, em especial os conceitos de alta coesão e baixo acoplamento do GRASP (*General Responsibility Assignment Software Patterns*) e a utilização de *Data Transfer Object - DTO*, determinarão o sucesso ou fracasso da migração e o nível de esforço necessário.

Manteve-se, nesse trabalho, o acesso ao banco de dados relacional da aplicação monolítica, embora não seja essa a melhor abordagem. Como uma forma de aumentar o desacoplamento das aplicações, é interessante que cada aplicação *microservice* mantenha seu próprio banco de dados. Quando assim tratado, o banco de dados costuma ser do tipo NoSQL baseados em *documents*, embora se possa tirar proveito de uma abordagem chamada *polyglot persistence*, como comenta [Fowler and Lewis 2014]. Dessa forma consegue-se encapsular a persistência de dados juntamente com o *microservice* que faz uso deles, favorecendo a descentralização dos dados e a escalabilidade.

É sugerido como trabalho futuro a utilização de bancos de dados NoSQL.

A demonstração da técnica em questão tomou uma aplicação monolítica que realiza operações de CRUD (*Create, Read, Update e Delete*) baseada em regras de negócio para tratar um sistema de gerenciamento de pedidos. Esta aplicação é formada pelos módulos de cadastro de itens de catálogo, cadastro de usuários e cadastro de pedidos.

Tal aplicação persiste seus dados por meio de um sistema de gerenciamento de banco de dados relacional, utiliza o padrão arquitetural MVC (*Model, View, Controller*) e foi desenvolvida com o *framework* Spring (ver **Figura 3**).

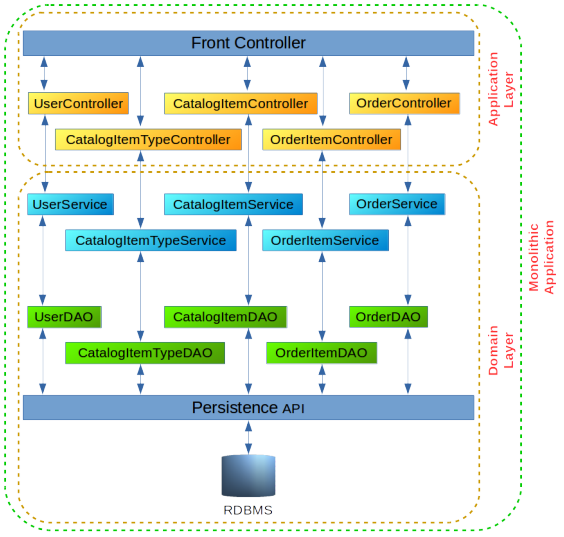

Figura 3. Aplicação Monolítica - Fonte: Elaborada pelo autor

Para capturar as chamadas aos serviços da aplicação monolítica com interesse em serem convertidos para *microservices* foram adicionados à aplicação *aspects* e seus respectivos *advices* do tipo *around*, como pode ser visto na **Figura 4**.

Todas as chamadas aos serviços monolíticos que devam ser capturadas precisam estar sinalizadas em um arquivo do tipo **.properties** externalizado. Cada sinalização é composta de um atributo, cujo nome é o próprio nome da chamada (nome completo), e o valor associado a esse atributo, que pode ser *true* (quando se deseja capturar a chamada) ou *false* (quando não se deseja capturar a chamada). A **Figura 5** mostra parte desse arquivo tipo **.properties** e suas entradas.

Essa abordagem permite desfazer a migração ou refazê-la, de acordo com as ponderações e necessidades dos clientes, além de possibilitar uma migração incremen-

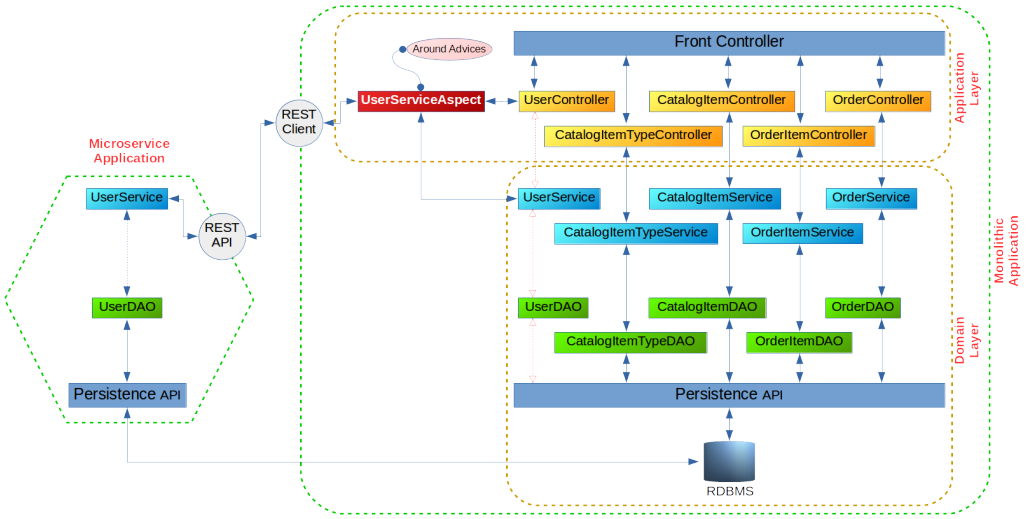

Figura 4. Migração Micro Serviço - Fonte: Elaborada pelo autor

tal, evitando o que Martin Fowler chama de '*Big Bang rewrite*', que, nesse caso, seria a abordagem onde se reescreveria toda a aplicação monolítica para obter um conjunto de aplicações *microservice*.

Uma vez que a configuração de quais chamadas a serviços monolíticos serão capturadas e direcionadas a *microservices* encontra-se em um arquivo externalizado (arquivo do tipo **.properties**), alterações na configuração poderão ser realizadas sem que sejam necessários procedimentos de *build* e *deploy*.

```
34 br.com.oem.everest.monolithic.service.CatalogItemTypeService.findAllSortedByField = false
35 br.com.oem.everest.monolithic.service.CatalogItemTypeService.findAll              = false
36
37 #
38 # User
39 #
40 br.com.oem.everest.monolithic.service.UserService.getIdMicroservice    = true
41 br.com.oem.everest.monolithic.service.UserService.create               = true
42 br.com.oem.everest.monolithic.service.UserService.save                 = true
43 br.com.oem.everest.monolithic.service.UserService.delete               = true
44 br.com.oem.everest.monolithic.service.UserService.findOne              = true
45 br.com.oem.everest.monolithic.service.UserService.findByName           = true
46 br.com.oem.everest.monolithic.service.UserService.findAllSortedByField = true
47 br.com.oem.everest.monolithic.service.UserService.findAll              = true
48
49
50 #
51 # Order
52 #
53 br.com.oem.everest.monolithic.service.OrderService.getIdMicroservice    = false
54 br.com.oem.everest.monolithic.service.OrderService.create               = false
55 br.com.oem.everest.monolithic.service.OrderService.save                 = false
```

Figura 5. Configuração de Captura - Fonte: Elaborada pelo autor

O *aspect* gerenciador de capturas utiliza-se das informações presentes neste arquivo tipo **.properties** para tomar a decisão se desvia (caso *true*) ou não (caso *false*) as chamadas a serviços monolíticos que estão sinalizadas. Caso haja um *advice* do tipo *around* associado à chamada de um serviço monolítico sinalizado com *true*, o fluxo da aplicação é desviado para o *microservice* adequado. Caso a sinalização seja *false*, o fluxo segue seu caminho original (ver **Figura 6** e **Figura 4**).

4. Avaliação

Para fins de avaliação tomou-se algumas das estratégias abordadas por [Richardson 2016b].

```java
@Component
@Aspect
@PropertySource("classpath:static/aop/aop-switch.properties")
public class UserServiceAOP {

    private static final String GET_ID_MICROSERVICE = "br.com.oem.everest.monolithic.service.UserService.getIdMicroservice";
    private static final String CREATE = "br.com.oem.everest.monolithic.service.UserService.create";

    private RestTemplate restTemplate = new RestTemplate();

    @Value("${url.base}")
    private String URL_BASE;

    @Value("${service.name.user}")
    private String SERVICE_NAME;

    @Autowired
    private Environment environment;

    @Around("execution(* " + GET_ID_MICROSERVICE + "(..))")
    public String getIdMicroservice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {

        String idMicroservice;

        if (Boolean.parseBoolean(environment.getProperty(GET_ID_MICROSERVICE))) {

            idMicroservice = (String) restTemplate.getForObject(URL_BASE + SERVICE_NAME + "/id-microservice", String.class);

        } else {

            idMicroservice = (String) proceedingJoinPoint.proceed();

        }

        return idMicroservice;

    }
```

Figura 6. *Aspect* com *Advices Around* - Fonte: Elaborada pelo autor

Uma das estratégias, a de título *'Strategy 3 - Extract Services'* é a mais próxima da abordagem tratada neste trabalho e consiste em transformar os módulos existentes dentro da aplicação monolítica em *microservices* autônomos. Para tal o autor sugere a criação de uma *API* entre a aplicação monolítica e o módulo (serviço) que se deseja separar (transformar em *microservice*). Isso exige alterações no código monolítico que podem ser significativas. Feito isso é necessário, agora, escrever código que permita a aplicação monolítica comunicar-se com o novo módulo (serviço) por meio de *inter-process communication (IPC)*, ou mais especificamente, usando REST (ver **Figura 7**).

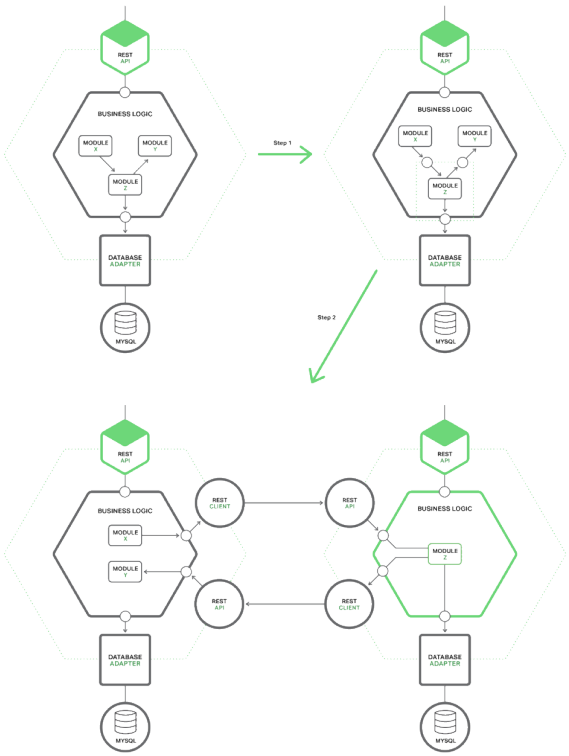

Figura 7. *Strategy 3 - Extract Services* - Fonte: [Richardson 2016b]

A grande diferença existente entre a estratégia supracitada e a sugerida no atual

trabalho é o fato de que no presente trabalho não ser necessária a alteração de absolutamente nenhuma linha de código da aplicação monolítica. Por meio dos *aspects*, seus *around advices* e do arquivo de configuração do tipo **.properties**, as chamadas a serem capturadas na aplicação monolíticas são desviadas aos *microservices* adequados. Isso permite desfazer todo o processo mudando apenas o arquivo de configuração do tipo **.properties**, sem a necessidade de *builds* e *deploys*, como já foi descrito. É necessário, sim, escrever os *apects* e o arquivo de configuração do tipo **.properties**. **Isto também dá ao processo uma condição de permitir que a migração seja realizada de forma incremental, o que adere à *QP1*.**

As dificuldades de isolar (desacoplar) os serviços monolíticos para serem convertidos em *microservices* são absolutamente as mesmas nas duas abordagens. A necessidade de rescrever o código do serviço, agora como *microservice*, de forma a ser acessa do via REST também é a mesma (ver **Figura 8**). **Dito isso, vê-se que a *QP2* foi tratada.**

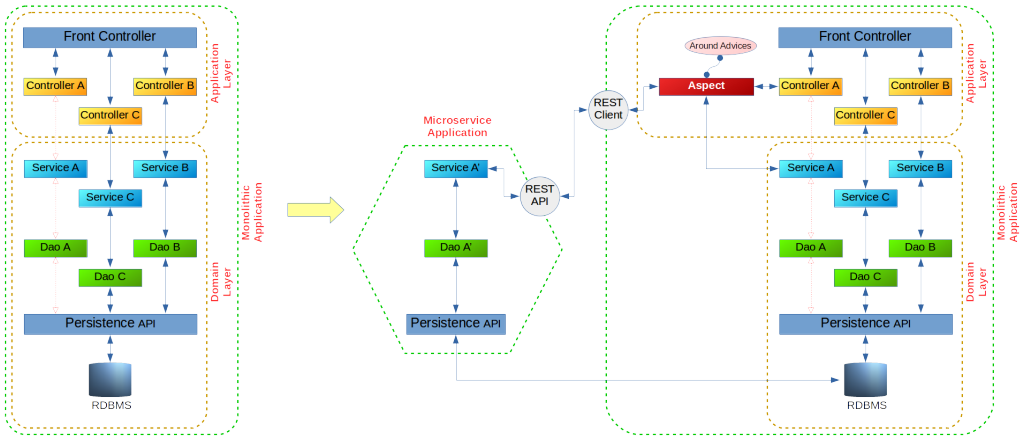

Figura 8. Modelo Monolítico x Modelo Micro Serviço - Fonte: Elaborada pelo autor

Outra estratégia, a de nome *Strategy 1 - Stop Digging*, aborda a situação na qual não está mais a se realizar uma migração, mas a implementação de uma novos serviços. Nesta estratégia [Richardson 2016b] sugere criar esses novos serviços já como *microservices*, adicionando novas rotas ao módulo *request router*, que é neste caso equivale a um *front controller*. Faz-se necessário, ainda, criar estruturas chamadas por ele de *glue code*, que sarão responsáveis por realizar a integração dos novos serviços (*microservices*) com a aplicação monolítica, pois é comum que haja a necessidade de acesso a dados da aplicação monolítica a partir dos *microservices* (ver **Figura 9**).

A abordagem sugerida neste trabalho também pode adaptar-se à inclusão de novos serviços. Para tanto é necessário a adição de novas rotas ao *controller* adequado, neste caso equivalente ao *request router*, o qual deverá ser criado, caso ainda não exista. Isso feito pode-se chegar a conclusão da necessidade da criação de *glue codes*, como sugerido em [Richardson 2016b]. Percebe-se, pois, que a inclusão de novos serviços como *microservices* não traz nenhuma novidade em relação ao sugerido em *Strategy 1 - Stop Digging*, o que evidencia que a solução sugerida neste trabalho é fortemente direcionada no sentido de migrar serviços monolítico para *microservices*, não conflitando, entretanto, com outras abordagens distintas (ver **Figura 10**).

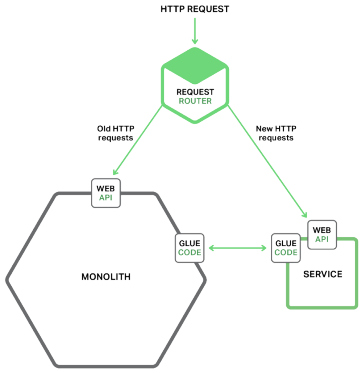

Figura 9. Strategy 1 - Stop Digging - Fonte: [Richardson 2016b]

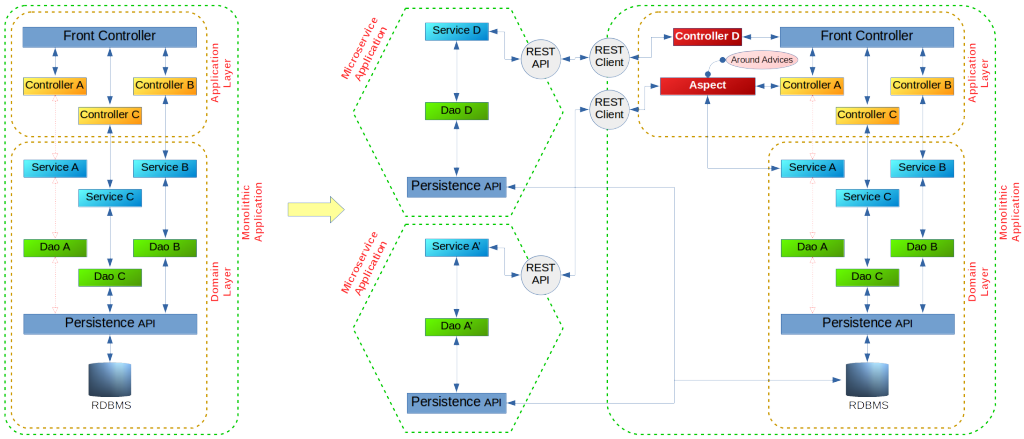

Figura 10. Incluindo Novos Micro Serviços - Fonte: Elaborada pelo autor

Como já foi mencionado, este trabalho parte do princípio que a persistência dos dados permanecerá sob a gerência do RDBMS da aplicação monolítica, sendo o uso de bancos encapsulados com os *microservices*, normalmente NoSQL, tarefa para trabalhos futuros, fato este que responde à QP3.

5. Trabalhos Relacionados

[Balalaie et al. 2015b] define uma abordagem incremental de migração para *microservices*. Este trabalho procura definir uma série de estratégias que podem ser aplicadas na migração, não apenas com relação ao refatoramento do monolítico, mas também relacionadas a outros fatores como integração e contínua e implantação. Com relação mais especificamente ao refatoramento do monolítico este trabalho apenas segue as mesmas estratégias já definidas em outros trabalhos como o de [Newman 2015] e o de [Richardson and Floyd 2016] para procurar as entidades de domínio e migrá-las para microservices. Em seguida, resolve-se as dependências entres os serviços necessários para implantação da aplicação em *microservices* (configurações do serviços, serviços de descoberta, serviços de integração e balanceamento de carga). Nessa abordagem foi possível estabelecer uma espécie de *template* de processo para que outras aplicações pudessem migrar para *microservices*.

[Richardson 2014] e [Fowler and Lewis 2014], sobre migração de monolítico para

microservices, abordam o aspecto de modularização em função de parâmetros convencionais de software como subdomínio da aplicação, ou tamanho da aplicação, ou decomposição de funcionalidades ou de dados. Conforme abordado na seção anterior, o trabalho mais semelhante ao proposto neste artigo é o de [Richardson 2016b]. Conforme foi discutido anteriormente, nosso trabalho consegue não apenas resolver o mesmo problema de migrar facilmente um módulo do monolítico para *microservices*, devido ao uso de AOP, como também permite o desenvolvedor voltar à versão anterior se assim desejar. As modificações, tanto para ir quanto para voltar, são totalmente modularizadas e desacopladas nos aspectos, não exigem mudanças no código do monolítico e podem ser realizadas em tempo de execução sem ter que recompilar o sistema.

A abordagem definida em [Bolar 2018] tem semelhanças com o trabalho em questão. Há, entretanto, algumas diferenças que podem ser consideradas cruciais. A primeira delas é o requisito da aplicação monolítica a ser tratada expor serviços via classes facades utilizando web services baseados em SOAP. Esse não é um requisito para a abordagem sendo apresentada. A segunda, e talvez a mais importante diferença, é a necessidade da alteração do código monolítico por meio da inclusão de uma anotação criada pelo autor que a chamou '@MicroServiceWeaver'. Na abordagem sugerida neste trabalho o código monolítico não sofre nenhuma alteração.

6. Conclusão e Trabalhos Futuros

Por meio da demonstração disponibilizada, percebe-se que pode-se tirar o melhor proveito dos dois mundos (arquitetura monolítica e arquitetura *microservice*)

Não há, necessariamente, a imposição da migração ser apenas em um sentido, ou seja, nada impede que haja o desejo, por parte do cliente, de retornar alguns dos módulos migrados para a versão *microservice* às suas respectivas versões monolíticas. Tal tarefa, entretanto, requer um nível adicional de esforço, caso a escolha por usar bancos de dados NoSQL encapsulados na aplicação *microservice* seja a abordagem utilizada, pois uma vez migrado para a versão *microservice*, o módulo passará a persistir seus dados em um banco NoSQL gerenciado pelo próprio módulo *microservice* e não mais no banco relacional da versão monolítica. Fica o estudo dessa tarefa como sugestão para trabalhos futuros.

Na abordagem seguida por este estudo a migração dos serviços para uma arquitetura ou o seu retorno para a arquitetura anterior transforma-se em uma ação simples de definir valores de propriedades externalizadas para verdadeiro (*true*) ou falso (*false*).

Outro desafio a ser levado quando se utiliza bancos de dados NoSQL encapsulados na aplicação *microservice* é a perda das propriedades transacionais ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Essas propriedades normalmente não são suportadas por bancos de dados NoSQL. Existem estudos que enfrentam esse problema, mas, também, ficam como sugestão para trabalhos futuros.

Como resultado final fica a certeza que a abordagem sugerida é viável e bastante flexível no que diz respeito a tarefa de migrar serviços de aplicações monolíticas para *microservices*. Traz em seu bojo a facilidade de não ser necessário alterar nenhuma linha de código da aplicação monolítica e permitir, com um mínimo de esforço e sem a necessidade de realizar paradas na aplicação, alternar entre as versões monolítica e *microservice* de serviços já migrados. Possibilita, ainda, a utilização de qualquer outro processo de

ampliação no código existente, ou no lado monolítico (não recomendado) ou no lado *microservice*.

Referências

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2015a). Microservices migration patterns. Technical report, Department of Computer Engineering Sharif University of Technology.

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2015b). Migrating to cloud-native architectures using microservices: An experience report. In *Advances in Service-Oriented and Cloud Computing - Workshops of ESOCC 2015, Taormina, Italy, September 15-17, 2015, Revised Selected Papers*, pages 201–215.

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2016). Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52.

Bolar, T. (2018). Integrating microservices with a monolithic application. `https://dzone.com/articles/integrating-micro-service-with-monolith-applicatio`. Acessado: 11/04/2018.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture, Volume 1, A System of Patterns*, volume 1. Wiley.

Carnell, J. (2017). *Spring Microservices in Action*. Manning Publications Co.

Fowler, M. (2004). Strangler application. `https://www.martinfowler.com/bliki/StranglerApplication.html`. Acessado: 27/02/2018.

Fowler, M. and Lewis, J. (2014). Microservices. `https://martinfowler.com/articles/microservices.html`. Acessado: 27/02/2018.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J., and Irwin, J. (1997). Aspect-oriented programming. In *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, pages 220–242.

Levcovitz, A., Terra, R., and Valente, M. T. (2015). Towards a technique for extracting microservices from monolithic enterprise systems. *3rd Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM), p. 97-104, 2015*.

Newman, S. (2015). *Building Microservices Designing Fine-Grained Systems*, volume 2015. O'Reilly Media, Inc.

Richardson, C. (2014). Pattern: Monolithic architecture. `http://microservices.io/patterns/monolithic`. Acessado: 27/02/2018.

Richardson, C. (2016a). Database per service context. `http://microservices.io/patterns/data/database-per-service.html`. Acessado: 27/02/2018.

Richardson, C. (2016b). Refactoring a monolith into microservices. `https://www.nginx.com/blog/refactoring-a-monolith-into-microservices/`. Acessado: 27/02/2018.

Richardson, C. (2016c). Shared database. http://microservices.io/patterns/data/shared-database.html. Acessado: 27/02/2018.

Richardson, C. and Floyd, F. (2016). *Microservices From Design to Deployment*, volume 2016. NGINX, Inc.

Sean (2015). Spring in action - aspect-oriented spring. http://seanzhou1023.blogspot.com.br/2015/10/4-spring-in-action-aspect-oriented.html. Acessado: 27/02/2018.

XWeaver (2015). An introduction to aspect oriented programming. https://www.pnp-software.com/XWeaver/AspectOrientedProgramming.html. Acessado: 27/02/2018.

Yanaga, E. (2017). *Migrating to Microservice Databases From Relational Monolith to Distributed Data*, volume 1. O'Reilly.